C - Standards

Last Date Modified: July 21, 2020

Authors: Joshua Crotts

Contents

1 Introduction

N/A

1.1 Purpose

N/A

1.2 Document Conventions

N/A

1.3 Intended Audience

N/A

1.4 Definitions

N/A

1.5 Project Scope

N/A

1.6 Technical Challenges

N/A

1.7 References

N/A

2 Overall Description

N/A

2.1 Project Features

N/A

2.2 User Characteristics/Classes

N/A

2.3 Operating Environment

N/A

2.4 Design & Implementation Constraints

N/A

2.5 Assumptions & Dependencies

N/A

3 Functional Requirements

N/A

3.1 Primary

N/A

3.2 Secondary

N/A

4 Technical Requirements

N/A

4.1 Operating Systems & Compatibility

N/A

4.2 Interface Requirements

N/A

4.2.1 User Interface

N/A

4.2.2 Hardware Interface

N/A

4.2.3 Software Requirements

N/A

4.2.4 Communications Interface

N/A

5 Non-functional Requirements

N/A

5.1 Performance Requirements

N/A

5.2 Safety & Recovery Requirements

N/A

5.3 Security Requirements

N/A

5.4 Policy Requirements

N/A

5.5 Software Quality Attributes

N/A

5.5.1 Availability

N/A

5.5.2 Correctness

N/A

5.5.3 Maintainability

N/A

5.5.4 Reusability

N/A

5.5.5 Portability

N/A

5.6 Process Requirements

N/A

5.6.1 Code Style Guide

To enforce a clean, consistent codebase, I have created a style guide that mimics SDL, but also maintains its own identity. The chosen style guide comes from days of going back and forth between implementations to find the best one.

1. **Structs** should be declared with `type_t`, identical to Linux kernel C code. A `typedef` should also not be used. I understand that typedefs are used all throughout the code right now, but that will change later with updates. This implies that all structs should be declared with the struct keyword (e.g. `struct foo_t f;`). When listing variables inside a struct, do primitives first, then SDL components, then structs from C-Standards, with each category separated by a new line.

2. **Comments** should use the multi-line style `/* */`, even on single lines. Additionally, all `.c` files should start with a file header comment with a description, the license, and who edited it. If you need an example, check out any file in the standards source. All functions in the source file should have javadoc-esque comments, with the purpose of the function listed, parameters with the `@param` tag, and a `@return` tag, even if they are both void. The only time

single-line comments are allowed are on the last line of a header file with the closing header-guard `endif`, and within the file header comments.

3. **Braces** are a controversial topic in most style guides. I prefer the Kernighan and Ritchie version, otherwise known as Egyptian style, but with a twist. Instead of omitting braces in one-line if/loop conditions, they must always exist and start on the same line as the declaration of your condition/loop/function. In essence, if a brace is used, put it on the same line.

4. **Spacing** is another topic where I tend to go astray from common programming projects. When using parenthesis, there must exist a space on the inside of the opening and closing parenthesis, the only exception being if there are no parameters in the function call. For instance: `extern void foo( int32_t x, int32_t y )` is correct, and if we need to *call* `foo`, use `foo( 4, 5 )`. However, if there is a function `extern void bar( void )`, we call it using `bar()`. Brackets `[]` do not follow this convention. **Tabs** follow a strict convention of two spaces per tab. Not four, not eight, but two. **Variables** at the top of a source file should be organized as follows: `structs` that aren't static, followed by SDL variables that aren't static, followed by SDL variables that are static, followed by structs that are static, followed by primitive variables.

5. **Header files** do not have to have the same file header comment that source files do, nor is it necessary that they have javadoc comments. Also, make sure all header files have header guards `#ifndef HNAME_H` followed by `#define HNAME_H`. After all code, put `#endif // HNAME_H`. Structs should be declared at the top of header files and likewise above variables with the `extern` keyword.

6. **Functions** in header files should always have the `extern` keyword, even though it is explicit in C. Functions in the source file should have one line dedicated to the access modifiers/return types, then on the next line, the function name with the appropriate parameters. If the function has no parameters, it must be declared as `void`. Any function declared inside a standard source folder should have a preceding `Stds_` prefix (yes, I know that's redundant). This is to distinguish it from other library code and those inside test files and SDL. Plus, it makes our library stand out! Static function prototypes in a source file should be declared below static and non-static global variables, but the functions themselves should be created below all non-static functions.

7. Pointers and variables should never be explicitly initialized in the header file. They can and should, however, be declared with the extern keyword. Declaring `app_t` and `player_t`, for instance, causes problems on Arch Linux for some reason.

8. When using **types**, try to always stick to length-specifying types found in `stddef.h`. For instance, `int32_t`, and others. Do not declare a `#define TRUE 1` or the equivalent for `false`; this is *not* C89! Use `bool` from `stdbool.h`. It's included for a reason!

9. **Magic numbers** are generally a no-no. If you have to have a magic number for whatever reason, declare it as a preprocessor definition or a static definition. In testing (e.g. in tests/), this is allowed. Magic numbers do *not* include 0, divides by 2, multiplies by 2, bit shifts by 1, or powers of 2 that generally give meaning to something like a color (e.g. 0xff, $x >> 24$, $x >> 16$, $x >> 8$). In summary, if a number doesn't make sense out of context, define it or make it static. Otherwise, it's probably okay.

10. **Never** force a push to GitHub. This will immediately get you removed from the project. Contact me if you have questions about a push, or make a pull request.

11. When using **dynamic memory** (pointers), if you `malloc`, always accompany it with a free, or the respective Stds/SDL clean/destroy function! In addition, immediately after a call to `malloc/calloc/realloc`, check the pointer to see if it is `NULL` to prevent a null pointer from existing. I had a nasty memory leak in `text.c` a few weeks ago because I forgot to free two `SDL_Surface` and `SDL_Texture` pointers that were being created every tick of the game! Thus, memory was chewed through like it was nothing.

5.6.2 Development Process

5.6.3 Time Constraints

5.6.4 Cost & Delivery Date