



Firmware Style Guide

- [1 Overview](#)
- [2 READMEs](#)
- [3 Git Repo Setup](#)
- [4 File Structure](#)
- [5 File Line Endings](#)
- [6 Code Organization/Misc Rules](#)
 - [6.1 Include Guards](#)
 - [6.2 Code Sections](#)
 - [6.3 Private Before Public](#)
 - [6.4 Function Names](#)
 - [6.5 Comments](#)
 - [6.5.1 Function Descriptions](#)
 - [6.6 Static keyword](#)
 - [6.7 Variable Names](#)
 - [6.8 Includes](#)
 - [6.9 Tabs](#)
- [7 Autoformat/Code Style](#)
- [8 Use of goto](#)
 - [8.1 General Guidelines](#)
- [9 Unit Tests](#)
 - [9.1 Tip](#)
- [10 CI Pipelines](#)
- [11 Revision History](#)

Overview [🔗](#)

This document serves as a style guide for Ovyl firmware. The guidelines in this document are not hard-and-fast rules, but keeping consistent code style and organization across projects helps greatly with readability and reusability, so following them is encouraged.

READMEs [🔗](#)

Having a readme file for each project is heavily recommended. A README file should be included at the root folder of the project and should usually contain the following items:

- A description of the project
- Links to relevant documentation (requirements docs from client, documents from Ovyl, etc.)
- Description of how to build the project (including instructions to build/run a docker container if needed to build project)
- Steps to flash the project onto the hardware and run it

- Steps to run the project with an attached debugger (if supported)
- Steps to run unit tests
- Other notes that may be useful for development

If a project's README file becomes excessively long, it be broken out into several smaller files (build.md, flash.md, etc.) and those smaller files placed under a "docs" folder. Links to the broken-out readme files should be added to the main README.

Git Repo Setup [🔗](#)

Ovyl firmware projects are typically hosted on our github (<https://github.com/Ovyl>) by default. If the client asks, however, Ovyl typically has no issue with the client hosting the code under their own github account or on another service entirely, so long as we are able to access it as needed for development.

Regardless of where the project is hosted, usual best practices for code repository setup are encouraged. This includes:

- Utilizing .gitignore files to exclude build files and other generated content from being committed
- Using branches (with descriptive names) to keep work separated logically and prevent overwriting others' work
- Using git submodules to include common modules across multiple projects
- Making use of Pull Requests to facilitate code reviews before large changes are merged

File Structure [🔗](#)

Barring any special requirements from clients, vendor code, etc., source files for Ovyl firmware projects are typically organized as follows:

- Source files written by Ovyl are all included in a folder named "src" at the root of the project

"Src" may include any number of subfolders as is convenient

- Vendor code is placed under a "vendor" folder. Vendor code does not need to adhere to any Ovyl style requirements
- Unit tests and related files (mocks, etc.) are placed under a "tests" folder
- .gitignore files, Dockerfiles, Readme files, and other one-off configuration files are under the project root folder
- Support/build scripts are under a folder named "scripts"
- Datasheets and other informational documents, if included in the repository, are placed under a "datasheets" folder

File Line Endings [🔗](#)

Files must use LF for the line ending, not LF CR. Update your git settings if LF CR is being automatically changed.

Code Organization/Misc Rules [🔗](#)

Some miscellaneous rules for code organization:

Include Guards [🔗](#)

Header files should be guarded with #ifndef to prevent accidental multiple inclusion. If you are purposefully omitting the guard so you can include the file multiple times for some special purpose, make note of that fact at the top of the file in a comment.

```
1 #ifndef HEADER_FILE_NAME_H
2 #define HEADER_FILE_NAME_H
3
4 /* Header file contents
```

```

5
6 ...
7
8 */
9
10 #endif

```

The define used for the include guard should be the filename in CAPITAL_SNAKE_CASE, with the '.' replaced with '_'. So a file named spi_driver.h would use an include guard define of SPI_DRIVER_H.

⚠ Do not use `#pragma once` as an include guard

Code Sections [↗](#)

Header and source files should have comments demarking different sections of code within the file. The sections typically marked are as follows:

- Private Defines
- Private Types
- Private Variables
- Private Functions
- Public Defines
- Public Types
- Public Variables
- Public Functions

If a file does not include any code belonging to a section, the section-marking comment can and should be omitted. For instance, header typically contain only public items, while source files often contain a mix of public and private functions, variables, etc.

```

1  #include <stdint.h>
2  ...
3
4  /*****
5  Private Defines
6  *****/
7
8  #define SERIAL_NUMBER_LEN_CHARS  6
9  ...
10
11 /*****
12 Private Variables
13 *****/
14
15 static uint32_t foo;
16 ...
17
18 /*****
19 Private Functions
20 *****/
21
22 bool bar(uint8_t bat)
23 {
24 ...
25
26 }

```

Private Before Public [🔗](#)

Private functions, variables, types, etc. defined in a file should be placed before public items in source files. This cuts down on function declarations that would otherwise be needed if a private function were defined after a public function that made use of it.

Consider this:

```

1  #include <stdint.h>
2
3  /*****
4  Private Functions
5  *****/
6
7  int add(int a, int b)
8  {
9      return a + b;
10 }
11
12 /*****
13 Public Functions
14 *****/
15
16 int math_module_sum(int a, int b)
17 {
18     return sum(a, b);
19 }
```

Versus the following code block, which requires extra boilerplate code:

```

1  #include <stdint.h>
2
3  /*****
4  Private Function Declarations
5  *****/
6
7  int add(int a, int b);
8
9  /*****
10 Public Functions
11 *****/
12
13 int math_module_sum(int a, int b)
14 {
15     return add(a, b);
16 }
17
18 /*****
19 Private Functions
20 *****/
21
22 int add(int a, int b)
23 {
24     return a + b;
25 }
```

Function Names [🔗](#)

Public functions and variables must be prefixed with the name of the module they are a part of. Private functions/variables do not have this requirement.

```
1 //math_module.c
2 #include <stdint.h>
3
4 /*****
5 Private Functions
6 *****/
7
8 static int add(int a, int b)
9 {
10     return a + b;
11 }
12
13 static int sub(int a, int b)
14 {
15     return a - b;
16 }
17
18 /*****
19 Public Functions
20 *****/
21
22 int math_module_sum(int a, int b)
23 {
24     return sum(a, b);
25 }
26
27 int math_module_sum_arr(int* arr, uint8_t arr_len)
28 {
29     int sum = 0;
30     for(int ii = 0; ii < arr_len; ii++) {
31         sum = add(sum, arr[ii]);
32     }
33     return sum;
34 }
35
36 int math_module_sub(int a, int b)
37 {
38     return sub(a, b);
39 }
```

In this module, `math_module_sum()`, `math_module_sum_arr()`, and `math_module_sub()` each have corresponding declarations in a `math_module.h` file and can be used by the rest of the firmware, where as `add()` and `sub()` are used only by `math_module.c`

Comments [🔗](#)

Code must be commented so that an experienced developer who isn't familiar with the code can read the code and comments and get a basic understanding of what the code is trying to do.

Comments should describe why the code is doing what it does, not what the code is doing.

Every line of code does not need to be commented. It is preferred to comment a block of code.

No commented out code is allowed to be merged into mainline code. If code needs to be commented out for testing / development, do it locally.

The exception to this may be reference code, but it must be well documented in the code why it is included, that it is for reference only, and have the approval of the project's FW lead.

Function Descriptions [↗](#)

All functions must be commented in doxygen style comments.

If a function is private, the comment should be attached to the function implementation. This comment should help inform how the function works so that any maintainer can easily understand the abstract of the function.

If a function is public, the comment should be attached to the function definition, usually in the header file, so that anyone using this API/function knows how to call it correctly. If there are any gotchas or side-effects to the function, those must also be documented in the header as well.

Static keyword [↗](#)

Private variables and functions should be marked as such with the “static” keyword when possible

Variable Names [↗](#)

Variable names should be descriptive. Ideally variables should not need a comment next to their declaration describing what they represent, what units the variable is in, etc. There is NO requirement to prefix a variable with an indicator of size, type, or if it is a pointer (modern IDEs track this without cluttering up variable names just fine).

Variable names are typically in snake_case, unless other code (from vendor, client, etc.) requires otherwise.

Variables must include the units as a suffix. If a variable is unitless, a suffix does not need to be added.

```
1 int temp; //Vague, could be "temporary", "temperature", etc
2 int temperature; //Better but missing units
3 int temperature_celsius; //Properly named
4 float rotss; //What?
5 float rotations_since_start; //Better but missing units
6 float rotations_since_start_radians; //Properly named
```

Includes [↗](#)

Includes are typically all grouped at the top of a source or header file. Includes do not have to be sorted in any particular way. Standard library includes, such as string.h, stdint.h, stdio.h, etc are included with `<angle brackets>`, and all other code included with “quotes”. Standard library includes are typically grouped together, either above or below includes of other files.

Headers can be grouped as appropriate (system includes together, OS includes, module includes, etc)

```
1 #include <stdint.h>
2 #include <string.h>
3 #include "math_module.h"
4 #include "other_ovyl_code.h"
5 #include "vendor_module.h"
```

Tabs [↗](#)

Spaces must be used over tabs. 4 spaces = 1 tab.

Autoformat/Code Style [↗](#)

Clang format is a tool to automatically format source code according to a format description file. Many formatting options are supported, such as using tabs or spaces to indent, how far to indent inside of code blocks, how much whitespace to allow

between functions, etc.

Please use the clang format tool and the .clang-format file in this git repository to auto format Ovyl source code

VS Code can be configured to format files automatically on file save

- Ensure Clang-Format extension in VS Code is installed
- In VS Code, open settings through File>Preferences>Settings (or Ctrl+,)
- Search for "Format On Save" and check the box. Make sure to be under the "User" tab and not the "Workspace" tab
- Add the .clang-format file to the top level of your project

**** If formatting on save is not working, it is possible VS Code has multiple formatters that installed and it doesn't know which one to use. To fix this, open a source file, right click and select "Format Document With" and from the drop down select "Configure Default Formatter" Select "Clang-Format"**

Use of goto [↗](#)

In our projects, `goto` should only be used in specific cases for error handling that have multiple error checks and multiple cleanup steps before returning. The use of `goto` should be considered on a per-project basis, as some projects may have specific standards that prohibit its use.

General Guidelines [↗](#)

1. **Error Handling Only:** Use `goto` only for error handling scenarios where it enhances readability or simplifies cleanup within a function.
2. **Single Exit Point:** Limit `goto` usage to a single label at the end of a function.
3. **Avoid in Simple Functions:** In functions with straightforward error handling, avoid `goto`. Prefer using `return` statements directly if the function doesn't require complex cleanup logic.
4. **Follow Project Standards:** `goto` usage is a project-level decision and must adhere to any project-specific coding standards. When project standards are unclear, default to avoiding `goto`.
5. **Descriptive:** Always use descriptive label names like `cleanup` or `error_exit`

Unit Tests [↗](#)

Writing unit tests during firmware development is heavily encouraged. The default framework used in Ovyl firmware projects is Unity (<http://www.throwtheswitch.org/unity>), though if another framework works better for a particular project then it is fine to use a different framework.

Developers should aim for at least 70% line coverage of Ovyl-made firmware modules, but there are many cases where it is reasonable to have less coverage (drivers, for instance, often have little code that is worth unit testing). Vendor code and other code not written by Ovyl does not need to be unit tested.

Tip [↗](#)

When writing unit tests for a source file, you can `#include` the source file itself in your unit test, which lets you access private variables and functions for testing purposes without making them public just for the purposes of testing.

```
1 //Support files
2 #include "unity.h"
3 #include <stdint.h>
4
5 //File under test
6 #include "math_module.c" //NOT .h
```

```
7
8 //Unit tests...
9 void test_add_func(void)
10 {
11     //add() is a static function in math_module.c, but we can still test it
12     TEST_ASSERT_EQUAL(5, add(2, 3));
13 }
```

CI Pipelines [↗](#)

CI Pipelines allow automatic actions to be taken when new code is pushed to a project's repository. Most commonly, this is used to run unit tests on new code, to ensure there are no regressions or new bugs added. It is advisable, but not required, to add a CI pipeline to run unit tests on all new Ovyi projects.

TODO add template for github CI Pipeline config

Revision History [↗](#)

Version	Date	Comment
Current Version (v. 4)	Oct 25, 2024 21:55	Dave Sabol
v. 3	Oct 25, 2024 21:50	Dave Sabol
v. 2	Sep 12, 2024 13:18	Yoan Andreev (Unlicensed) Add function description comments section
v. 1	Sep 12, 2024 13:13	Yoan Andreev (Unlicensed)