C Coding Standard

> Author: Kacper Wojciechowski

Modularity

1. Projects should be split into single-responsibility abstract entities. In case of complex logical responsibilities, the modules can internally split into submodules governing specific parts of the overarching responsibility.

> **NOTE**: Single-responsibility relates to a **logical** responsibility, which itself can be abstract and consist of many different aspects, eg. "managing USB peripheral", and an exmaple of a submodules could be eg. "managing data reception", "managing data transmission" and "power control".

1. Each abstract module should expose only its **public** API, using which it intends to interface with other modules and the rest of the system. All other things should remain hidden (**private**).
2. Accessing parts of the internal states of the modules should be done via passing parameters. **Avoid** global access as much as possible, unless it's warranted based on the logic of the project.

Access to data and functions

1. **Private** - entities within code (functions, variables, consts, macros, etc.) that are not meant to be available by everyone, and are meant to be hidden from the outside perspective. Those entities will be exclusively placed in the source files that use them, shall not have any of their signatures exposed in any header files and will be marked using `static` keyword to prevent visibility leaks due to possible `extern` definitions. This scope should be used by default, especially including things like internal state of given module.

> **NOTE**: This pattern aims at ensuring the safety and validity of the private data, and follows a paradigm typically used in Object Oriented Programming, where limiting the visibility of the data protects it from accidental or intentional corruption.

1. **Public** - entities within code that are meant to be accessible by other entities (eg. interface of a module). These entities will require to be visible in the corresponding headers.

> **IMPORTANT**: Some modules may require the ability to have their internal state affected by outside things (eg. setting an error/status flag in an internal register, registering an outside container that will be accessed by the module, etc.). However such entities should not be exposed directly, and instead the module should their own API for affecting this state, so that the module has complete control over how its internal state is managed, to ensure **it is valid at all times**.

Functions

Declarations and definitions

1. The place where you want to put function prototypes, depend highly whether the function is intended to be public or not and will follow the rules described in the Access section.

2. Header files should **not** contain function definitions - only declarations. If a function is the be *inlined*, the header will contain only it's prototype with the `inline` keyword (or `always_inline` attribute).

Parameters

1. The **first thing** a function needs to do is to **ensure all parameters passed to it are valid for its logic** (eg. pointers are not null and point to a valid value, numeric parameters are within expected range, etc.). This should be done either with **asserts** or **early returns** depending on the function's logic.
2. If any of the function parameters has an invalid value according to the function's logic, then depending on the logic the function implements, it will either **early return** a default value, or fail at an `assert_param` if the function is never supposed to be called with the invalid value. Using `assert_param` is highly recommended for both security and optimization purposes: 2.1. **Unit tests** that are run in `Debug` profile will be able to capture the invokation of assert handler, so it drastically increases the **chance of catching whenever something goes wrong** in any of the execution branches. 2.1. Asserts are typically **disabled** in `Release` profile at the preprocessor level, so it **removes the checks made in runtime**.
3. A function can only operate its logic operations when all parameters are deemed valid.
4. If there is some processing required in order to check the validity of a parameter, the function will do only as little processing as possible to make sure all parameters are valid, before it proceeds to its logic.
5. If a function can perform it's logic regardless the value or validity of one or more parameters (**excluding** return parameters), it is a sign that either the parameter is redundant, or the function has more than one responsibility and needs to be split.
6. `Bool` parameters (**excluding** things like register flags) are a sign that a function tries to combine several responsibilities into one. As such, the function should be split into separate functions, and it should be the decision of the upper-level function which calculates the `bool` parameter, which one of those separate functions to invoke. In case where the `bool` parameter in question introduced only partial difference in the function logic and the rest of the logic is shared, the function should be split into one function for the shared logic, and one function per the `bool` branch.
7. `NULL` value can be a valid value, depending on the function's logic. A typical case for this would be an auxiliary return parameter, that might not interest the caller, and which does not disturb the function's logic if absent. Accepting the `NULL` as a valid value in such case allows the simplification of some function calls.
8. When passing a pointer to a buffer, the function must also take a parameter that indicates the size of the buffer, to allow for bounds checks when accessing the memory.
9. You can call a function **only** if you make sure all parameters it takes are **valid according to its logic**.
10. If in some execution branch, a parameter is not used (assuming it's used in at least one of other branches), mark that parameter by casting it to `(void)` or using corresponding compiler attribute.
11. Pointer parameters need to be `const ...*` by default, unless they are to be modified by the logic of this function, or functions called down the line.
12. Do not make primitive type parameters `const`. Pass them by value instead. All primitive types are sub-register-size, so copying them is very efficient and guarantees that regardless of any modification inside the function, the original will not be affected.
13. Avoid making `void*` parameters, unless the data you accept interrests you only in binary byte-by-byte manner, and not as any particular type.

Function body

1. Try to keep the function bodies and logic as simple as possible.
2. Limit the length of a function to the extent where they are easily comprehensible. The perfect rule of thumb is that a function body should fit entirely on your screen, which typically translates to roughly **60 lines of code** in standard IDEs. If a function body is larger than that, seriously consider delegating part of its logic to separate, smaller functions.

> **NOTE**: If the delegated logic is not to be seen outside, use the **private** function approach described in Access to data and functions section. **IMPORTANT**: Avoid using macros for visually separating the logic. Macros have a nasty habit of inflating very quickly and can be difficult to analyze, especially when they become either complex expressions, or their expressions are nested more than 2 levels.

1. When making functions that carry out some procedure, divide the procedure into phases and make the function body reflect that split into sections, by either separating the code into visibly separate blocks, or directly carry the section out into a separate function.
2. When your function performs processing that depends on several conditions, prioritize checking the conditions first, doing only the bare minimum of processing. This allows for better error handling and makes using the "early return" pattern significantly easier.
3. Avoid nesting control structures and conditional statements. Instead of this, prefer to invert the conditions and separate them, so they can be checked in an "early return" manner (either using the return or `goto` to handle situations where a condition is not fulfilled). This makes the code easier to analyze and reduces the cyclomatic complexity.

> **NOTE**: A maximum level of nesting that is fine, is two levels, which is having a single level of control structure (`if`/loop) inside an outer control structure (eg. a nested `for`, or `if` checks within a `for`/`if`). Having any more nesting than this prompts the need to evaluate how it can be redesigned to have the complexity simplified (some options are inverting and splitting conditions or extracting the inner control structure into a separate function). The goal is not that the program cannot execute multiple nested control structures - it is to keep the nesting within each function simple for analysis.

Return value

1. If return value of a function varies:
2. If you need to remember a return value, but before you call the `return` you need to do some additional processing, store the value in a variable. Create said variable at the very beginning of the function and initialize it with a default value, and later in the code overwrite that variable as needed.
3. If you can return right away, do so
4. If the return value is based on a conditional expression, and there is no more processing to do after the calculation of the condition, use ternary operator do return arbitrary values, or `return (condition);` if you want to return the condition result itself.
5. If the return value is based on calculation, and there is no more processing to do after the calculation is complete, use `return (expression);`.
6. When returning a value (be it variable/struct, and be it via `return` or a return parameter), you must make sure the returned value is **valid according to the logic of the function**. Such examples are:

- Result of the calculation, which is logically valid;
- Default value;
- Special value indicating an error (typically `0`, `NULL` or `0xFFFFFFFF`)

> **IMPORTANT**: In case of structures, you need to also make sure that values of each fields are valid in terms of their relation with each other. This means that not only each member need to be within its range, but also the combination of the members must be a valid logical result.

1. Avoid returning through pointer if not necessary.
2. Do not return local data in any other way than **by value** or **by writing to a provided buffer**.
3. Avoid returning `void*` unless the function returns a parameter that was also provided as `void*`.
4. Do not return dynamically allocated data, unless the logical scope of the data is wider than the scope of the function (eg. it's a dynamically allocated internal state of an abstract module).

Use of goto

1. `goto` can be used in functions, but only in **forward direction**, and to **move within the same scope** or **exit narrower scope**, similar to the requirements for `goto` described in loops section.
2. You **must not** use `goto` to exit the scope of the function that calls it. The upper limit for escaping a narrower scope is the scope of the function that invokes `goto`. All transitions outside and between functions must be done with regular function invocation and return.
3. Typical application for `goto` is to step over part of the processing in the same scope, and to mimic `try/catch/finally` blocks for things like error handling.

Data entities

Declaration, definition and initialization

1. When you create a variable, array, pointer or a structure, it **always** has to be initialized right away, either with it's target value, or some default value that is considered **valid** for given circumstances.

> **NOTE**: Values that are considered **invalid for processing** but thus allow the function to **detect if something is wrong**, will also be considered a **valid default** value.

1. When initializing a pointer, use `NULL` **instead** of `0` or `{0}` as default value, to avoid confusion and make it much more verbose that the pointer is not fit for dereference.
2. When initializing a structure, use the **designated member** notation (`MyStruct_t myStruct = {.field1 = val, .field2 = val}`) so the member values are **explicitly described** in the syntax (thus not considered "magic values").
3. When working with pointers, do not use more than 2 layers of indirection, to avoid complicated chains and multiple dereferences.
4. Keep the structure nesting flat, not deeper than 2 layers (the outer struct containing a flat inner struct).
5. For all numeric literals, use their corresponding specifiers:

- *int* - no specifier
- *unsigned* - `u`
- *long long* - `l`
- *short* - `s`
- *double* - `.0`
- *float* - `.0f`

1. Do not use *magic numbers* - unless the context of what the value means is strictly visible (eg. `setFailureCounterValue(0);` or `SomeStruct myStruct = {.failureCounter = 0};` or

`int failureCounter = 0;`), alias the value or store it in a named variable.

Assignments and expressions

1. If you perform operations during assignment (eg. calculating things), wrap the expression in `( )` for better visibility and protection from operator precedence bugs.
2. In compound expressions (eg. a conditional where one or both sides consist of some calculation, or a conditional that consists of several conditions compounded by logic operators), each of those expressions needs to be wrapped in `( )` for better readability and to protect yourself from the operator precedence bugs.
3. Use definitions from the `<stdbool.h>` standard library header for logic types and literals instead of the `int` with `0` and `1` values, to explicitly show that given variable / expression is to be considered a logical expression, instead of a calculation.
4. If you assign some value based on conditional expression, use ternary operator instead of `if/else` block. If the conditional expression is complicated, split it into several smaller expressions with their result stored in a local variable, and combine those results in the ternary operator condition.

Safe data access

1. When working with dynamic allocation, you must always validate whether the allocation was successful right after the allocation, **before you perform any other operations**.
2. Before dereferencing a pointer, you need to always ensure the pointer has a non-`NULL` value (as long as the pointer value does not change, checking once is sufficient for multiple dereferences).

> **NOTE**: An exception to this rule is when the pointer is assigned the address of other entity directly in the body of the same function that dereferences it (an example would be an iterator for traveling over an array). **IMPORTANT**: The exception mentioned above applies only within the local scope of the exact same function - if this is done by other function that called your function providing the pointer, your function must still validate the pointer before dereferencing it.

1. When iterating over a buffer - either an automatically or dynamically allocated one - it should only be done using the `for` loop which specifies the initial iterator value, and despite any other conditions, is guarded by checking whether the iterator does not violate the upper bounds.
2. If data is not to be modified, it needs to be defined as `const`, to prevent accidental modification.
3. If a struct member is a pointer, and it is intended that all struct objects will be `const`, or that this member will be read-only, this member must be declared as `const ...*`. This combats the side effect of being able to modify dereferenced data using a pointer inside a `const` structure.

> **NOTE**: This side effect stands from the fact, that when you declare a structure as `const`, the `const` qualifier is propagated to all members, but in case of pointers, it makes **the pointer itself** constant, but not the data this pointer leads to.

1. When casting between `signed` and `unsigned` data, before the cast, you need to make sure the conversion will be valid (eg. no data corruption due to re-purposing MSB to sign encoding and no violations of target type boundaries).
2. Prefer `sizeof()` operator over hard-coded sizes.
3. If accessing the data is complicated (eg. it follows down a complex data structure), store address of that data in an unmutable pointer (eg. `... const*`) and make subsequent accesses through that

pointer

4. When using standard library to format a numeric value (eg. `snprintf`, `vsnprintf`), especially floating point values, specify the number of character available for them, and for the decimal part. This allows to statically determine the length of the output string, to ensure it does not overflow the buffer it will be stored in.
5. **Do not** use non-bound string manipulation functions from the standard library (eg. `sprintf`, `vsprintf`, `strcat`, `strcpy`), instead use functions that have a bound parameter which safeguards from buffer overflows (eg. `snprintf`, `vsnprintf`, `strncat`, `strncpy` respectively).
6. **Do not** cast from `void*` to a specified type. Since you cannot determine what type of data is under the received address before the cast, this is a massive security vulnerability.

> **NOTE**: Casting **to** `void*`, eg. for a function that operates in the data in a byte-by-byte basis is fine.

1. Error handling needs to be constructed in a way, that guarantees the program does not crash or HardFault (internal system restart to return to a valid initial state is acceptable). The code must implement a no-crash policy and have recovery mechanisms.

> **NOTE**: Asserts do not violate this rule, because their purpose is to ensure that no branch of the code triggers them at any moment, at the development stage. This rule relates to the release version of the product.

1. If a function obtains external input (eg. from a hardware interface, network, file, stream), unless said function is an ISR, it needs to validate the input and make sure it does not contain corrupted or malicious data.

Lifetime control and memory management

1. Prefer local scope over global scope. Try to limit the lifetime of the variables/structures to as narrow scope as possible. This serves both optimizing RAM usage, aswell as mitigating an entire class of vulnerabilities standing from unintended variable modification/access and leaking data visibility.
2. Prefer automatic data (variables, structures, arrays) instead of dynamically allocated data. This serves not only increasing performance, but most importantly having a significantly better control over lifetime of objects and reducing the memory leaks. Additional alternatives to dynamic memory allocation are provided my mechanisms such as *ring buffer* and *memory pool*, which typically utilize automatic memory inside, and support bounds checking.
3. Prefer allocating a complete chunk of memory you might require right away, instead of iteratively allocating small blocks. This makes such allocations much easier to analyze - both statically and at runtime, ensuring no boundary violations of the allocated block and much easier lifetime and freeing of the memory, thus reducing memory leaks.
4. When working with dynamically allocated data that is shared between functions, only the module that allocated said data should be aware that it's dynamically allocated. This both allows for easy implementation of RAII for the allocated data (via initialization and deinitialization functions of the module), and makes it so the code will not be randomly deallocated in an arbitraty place outside the module.
5. Responsibility for handling the dynamically-allocated data rests on the scope that allocated it (eg. a function, a module). If the alocation in it's logic is only function-scope (is needed only for the time of processing in the function that allocated it), it needs to be freed by the same function once it's not needed anymore (and not later). If the allocation in it's logic is module-scoped, it needs to be handled

by that abstract module logic (typically via initializer/deinitializer function pair). Do not make system-wide allocations - each allocation has to logically operate in a limited scope (the widest of which is module scope).

> **NOTE**: Passing the allocated data outside the module is ok, but as per point 4, for all the other modules it's visible only as regular data. The awareness of how it's managed is limited to the scope that allocated it, and that scope is required to handle the lifetime of that data properly.

1. Do not declare data prematurely (with exception of the retval variable). Local variables must be declared right before you use them, and only in the scope in which you use them. This gives you drastically more control over their lifetime, optimize the RAM usage and protect against accidental modification of this data elsewhere.

Macros

1. Macros' values and parameters used in the expressions the macro expands to, always need to be sorrounded by `( )`, to avoid bugs standing from the evaluation order when the preprocessor expands the macro (namely bugs related to operator precedence).
2. Never put semicolon (`;`) at the end of the value of a macro definition. If a macro serves as a way to name some operation (eg. bitwise operations on registers) emulating an inlined function, place the semicolon at the place of calling the macro, as you would with a function call, and not within the macro expression itself. This allows for potentially using the macro in other expressions (eg. as a condition for an `if` statement) without breaking the syntax.
3. Simple expressions that are hard to read from the language's syntax alone (bitwise operations are a notorious offender here) are a good candidate to be aliased using simple macros.
4. If you want to alias conditions using macros, make one macro per condition.

> **NOTE**: In case of composite conditions, it is acceptable to have each of the conditions aliased as a macro, and then have one macro alias the composing of the conditions, as long as the overall depth of the macro expansions is not bigger than 2 levels.

Loops

1. All loops need to have a fixed uppder bound of iterations that can be proven using static analysis, unless the loop is meant to run indefinitely (`while(true)`).
2. Use `continue` and `break` in loops where applicable, to simplify the loop body
3. `goto` is allowed in loops, but only for moving within a given scope or outside of said scope (**not** inside a scope), and **only in forward direction**. Typical application for `goto` in loops is to exit a nested loop entirely (moving outside of the scope in which the goto is called, in a forward direction).

> **IMPORTANT**: "forward direction" in this case means that the `goto` should not be used for going back to some previous operation, and should only serve to skip a portion of the operations that are further down the line. **IMPORTANT**: "within a given scope or outside of said scope" means that the `goto` can be used to skip several operations inside the same scope (that can include skipping over entering a narrower scope), or to escape the scope to a higher one. `goto` should **not** be used to enter a narrower scope from the outside, as it easily interferes with things such as automatic memory allocation and can lead to an undefined behavior or straight up crash of the program.

Switch cases

1. Each case inside a switch needs to end up either with `break` or `return`, or in case of fall-through mechanisms, be marked as fall-through using a comment in place of `break`. This applies even to cases that are at the very end of a switch block.

2. In case of switching over `enum`, **do not** introduce a `default` case, and provide a separate case for each value (using fall-through if handling is shared for multiple values). This allows to find any new unhandled cases right at compile time, thus preventing logic errors when implementing new feature.

3. In case of very long switch statements, a look-up table can be considered as an alternative, unless the incurred memory/performance penalty prevents that.

Enums

1. Unless each enum value has a non-subsequent value (eg. **one-hot encoding** like in case of registers), specify only the value of the first element - the compiler will automatically calculate the values of all subsequent elements.

conditional expressions and conditional compilation

1. When a conditional compilation serves the purpose of disabling or changing some processing blocks (via compilation flags), instead of preprocessing out that piece of code using preprocessor directives, enclose those code blocks in an `if` statement dependend on a macro, and depending on the presence/absence of the corresponding compilation flag, set the macro to either `true` or `false`. This allows Unit Tests and static analysis tools to test all branches and code interactions that could happen depending on the compilation flags.
2. All possible code paths need to be handled - you cannot have unhandled errors, states or edge cases.

Naming convention

1. Names should be made in a way, that allows to rougly tell what given function does, or what given data represents, without reading it's body.
2. Do not shorten names, except for using standard/commonly known acronyms in the project's domain.
3. Mixing Camel Case and Snake Case is allowed.
4. Macro names should only use capital letters with underscores (`_`) separating words.
5. Names of the functions should start with the module those functions belong to.
6. Do not mark variables with prefixes indicating their types (eg. `pData` for pointer) or scope (eg. `gBuffer`, `lArray` for global and local). It's highly debatable whether marking introduces any benefits, but it tends to add every so slightly more effort.

Codebase structure

Modules

1. Each abstract module within the project should have its own, separate directory, which will contain directories `Inc` and `Src` for header files and source files respectively.

Includes

1. When including a header file, use path notation from some top-level include folder, instead of the header file name alone, to make it easier to see what module the header file belongs to.

> **NOTE**: This sadly cannot be applied to includes in test files due to the limitations of the `Ceedling` and `CMock` frameworks, as it interferes with the frameworks resolving the source files that correspond to the headers.

Headers

1. Header guards should be named after the full path to accessing the header from the project's root directory. This serves avoiding define collisions between two files that have the same name but are placed in different directories.

2. Headers should contain only public data and function prototypes. Functions should have only their prototypes placed in the headers.

3. If a function requires certain data type as their parameter, but does not need to know the size of the data (eg. it's an enum or it's a pointer to this data type), instead of placing the definition / including the header with the definition of said data, use forward declaration. This drastically **reduces the dependency graph** and helps **preventing include cycles**.

4. Headers should contain only the includes that are absolutely necessary, and should include those directly (unless the header that indirectly includes this one is also necessary), to limit the set of symbols present in the header to only those that are required.

5. Limit the number of includes in header as much as possible. The goal is to have headers which include **only** the things **they themselves require** (**not** their corresponding source files) and **nothing more**. Remember to remove redundant includes.

6. In order to fix an include cycle or if you expect an include cycle may occur, extract the declarations that need to be shared with other files into a separate header, and make all the files that require them include that header instead. This decouples the header file that would make a cycle.

7. Extern clauses can only be used at global scope.

Source files

1. Try to expose as little information (function prototypes, constants, macros, variables) to the header as possible.
2. Prefer including in the source file instead of the header file. This makes sure source file has access to all necessary definitions but does not inflate the dependency graph and does not propagate the includes further.
3. Keep macros near the place where you use them.
4. Keep file-scope global definitions at the top of the file, below includes.
5. Keep private functions prototypes at the top of the file, below file-scope global definitions.
6. Keep private functions definitions at the bottom of the file.
7. Extern clauses can only be used at global scope.

Comments

1. If you want to leave information about what needs to be done in the code in the future, make a comment and start it with capital `TODO` so it can easily be found using tools like `grep`.
2. Avoid unnecessary comments. Code should be clear and readable as-is, so consider improving names and making aliases before introducing comments.
3. Comments should explain additional context or reasons for specific implementation decisions, **not how** something is done - this should be the responsibility of the code itself.
4. For multiline comments prefer block comment instead of multiple one-line comments

External dependencies

1. External dependencies (eg. libraries, both in-house and third-party) should be added to the project using the `git submodule` feature into a designated folder in the project's root directory. This designated directory should be the root of the include path for each and every library inside, enforcing using the relative path notation for the include. This allows for easier analysis what library is used by given piece of code, and makes updating the dependencies much easier, by simply using a `git submodule update` command (and resolving any in-code conflicts).

Versioning

1. Binary projects need to be stamped with the hash of the git commit they represent, after the compilation process. This gives a huge insight into the code, when debugging a binary file that has been deployed.