Embedded C/C++ Coding Standard



# History

|  |  |  |  |
| --- | --- | --- | --- |
| **Author** | **Description** | **Date** | **Rev** |
| J. Taylor | Initial Release | 3/6/2024 | 1.0 |
|  |  |  |  |

# Scope

This standard applies to source code that is developed for in house green field code. Code development for legacy code bases shall follow the coding standards of the legacy code’s existing coding standards. This standard does not apply to third-party code bases.

# Definitions / Abbreviations

| **Term** | **Description** |
| --- | --- |
| **Camel Case** | Camel case is used to eliminate white space in symbol names by using mixed case to separate words. The first word starts with a lower-case letter, e.g.  camelCase |
| **Pascal Case** | Pacal case is used to eliminate white space in symbol names by using mixed case to separate words. The first word starts with a upper-case letter, e.g.  PascalCase |
| **Snake Case** | Snake case is used to eliminate white space in symbol names by using an underscore to separate words. Snake case is all lower case, e.g.  snake\_case |
| **Namespace Case** | Namespace case is variant of Snake Case that is used when prefixing a C symbol with its logical namespace. Namespace case uses an underscore to separate words same as snake\_case, except the first letter of each namespace is upper case, e.g.  Foo\_Bar\_hello\_world, where Foo::Bar is the logical namespace |

# Document References

| **Document #** | **Document name** | **Version** |
| --- | --- | --- |
|  |  |  |
|  |  |  |

# Standards

Standards tagged with a REQ label must always be followed, or any deviation documented in the source code. Standards without the REQ label are strongly recommend, but deviation is allowed (without documenting the deviation). Some required standards explicitly call out exceptions. For these cases – no deviation document in the source is required.

## Language (REQ)

All C++ code shall be compliant with the C++11 standard (ISO/IEC 14882:2011). All C code shall be compliant with C11 standard (ISO/IEC 9899:2011). Language features defined in newer language standard shall not be used. Note: Older standards are intentionally used to facilitate a broader range of tools and legacy platforms.

## No Dynamic Memory Allocation (REQ)

To prevent memory leaks and fragmentation – no dynamic memory allocation is allowed. The application may allocate memory from the heap at start-up. Once the system is “up and running” the application shall not allocate memory from the heap. This practice guarantees the system will not fail over time due to lack of memory.

For objects that must be dynamically created/deleted – the application programmer is required to pre-allocate a memory pool (on a per object type basis) that will be used to construct object at run-time. This requires the developer to use the “placement new” operator to create an object. To delete an object, the developer must explicitly call the object’s destructor and then return the memory back to its associated memory pool.

The malloc(), calloc(), realloc(), reallocarray() and free() functions shall be restricted the same as the new and delete operators.

## No Recursion (REQ)

Recursion shall not be used. Recursion is dynamic memory allocation in disguise, i.e. using the stack instead of the heap as the memory source. Recursion is also more dangerous because instead of a failure indication when memory is exhausted – a stack overrun simply overwrites ‘other’ memory. A stack overrun is even more likely on embedded systems since stack sizes on embedded platforms are typically small (e.g. 0.5K to 4K range).

Note: In some cases, the recursion is bounded (i.e. maximum stack usage can be statically determined). These scenarios are not allowed because it opens up the potential of future failures as the code is maintained and extended.

## Use const Wherever Possible

Every possible modification of data should occur according to explicit relaxation of default read-only policy. Think const. Member functions should be const by default. Only when you have a clear, explicit reason should you omit the const modifier on member functions.

## No Type Casting (REQ)

Take every possible measure to avoid type casting. Errors caused by casts are among the most pernicious, particularly because they are so hard to recognize. *Strict type checking is your friend – take full advantage of it!* The following is list of allowed exception:

* The use of void\* pointers in C code in interfaces in lieu of abstract classes.
* Casting within a class or module is allowed (though strongly discourage) as long as casting is not exposed to consumers/clients of the class or module.

## No Compiler Warnings (REQ)

Code that compiles with warnings will not be accepted for integration. While developing code, eliminate the warnings as-soon-as-possible as the warnings tend to be the source of logic and run-time errors. Build scripts are required to enable the compiler option to treat warnings as errors.

## No Explicit Constants (REQ)

Do not write explicit constants into code, except to establish identifiers to represent them. The exception to the rule is the constant “0”. Always use the keywords true and false for Boolean values (for C code this mean including <stdbool.h>).

## No goto Statements (REQ)

The use of the goto statement is not allowed.

## Use nullptr (REQ)

C++ code must use the nullptr keyword instead of zero the NULL macro when assigning a null value to a pointer.

## Conditionally Compiled Code is Not Allowed Within a Function Body (REQ)

Do not use the preprocessor #ifdef/#ifndef constructs to support platform, hardware, system dependent code. All platform, hardware, system dependent code must be isolated into individual files that are specific to the dependencies. This means the developer – at the design stage – must identify and plan for handling these dependencies!

## Virtual Destructors (REQ)

If a class has at least one virtual function, declare the destructor virtual, too.

class Foo

{

public:

/// Constructor

Foo( .... );

/// Destructor

virtual ~Foo();

public:

/// Do something

virtual void doSomething( .... );

...

};

## Rule of Five (REQ)

This standard only applies when there has been approved deviation to the No Dynamic Memory Allocation standard.

For classes that utilize dynamic storage, the class is required to implement the following methods. This is to ensure proper and efficient handling of when copying and moving the class’s dynamic data. (<https://en.cppreference.com/w/cpp/language/rule_of_three>)

* Destructor
* Copy constructor
* Copy assignment operator
* Move constructor
* Move assignment operator

class Foo

{

private:

char\* m\_string;

public:

/// Constructor

Foo(const char\* src = "")

: m\_string(nullptr) {

if (src) {

m\_string = new char[strlen(src) + 1];

strcpy(m\_string, src );

}

}

/// Destructor

~Foo() {

delete[] m\_string;

}

/// Copy Constructor

Foo(const Foo& other)

: Foo(other.m\_string) {

}

/// Copy assignment

Foo& operator=(const Foo& other) {

if ( this != &other ) {

delete[] m\_string;

m\_string = new char[strlen(other.m\_string) + 1];

strcpy(m\_string, other.m\_string);

}

return \*this;

}

/// Move constructor

Foo(Foo&& other) noexcept

: m\_string(other.m\_string) {

other.m\_string = nullptr;

}

/// Move assignment

Foo& operator=(Foo&& other) noexcept {

if ( this != &other ) {

delete[] m\_string;

m\_string = other.m\_string;

other.m\_string = nullptr;

}

return \*this;

## Pass and Return Objects by Reference

References provide stricter semantics than raw pointers (i.e. a reference requires that the object it “points to” exists – where a raw pointer does not). If the interface semantics are such that the object must always exist – use references in the interface even if the internal implementation is using pointers. By using references in interfaces whenever possible, the compiler can enforce more of the semantics of the interface.

## Protect Header Files with #ifndef Read-Once Latch (REQ)

Use the following templates for the read-once latch, where Namespace maps to the directory path (relative to the root src/ directory) and ClassName maps to the file name.

#ifndef Namespace\_ClassName\_h\_

#define Namespace\_ClassName\_h\_

…

#endif

#ifndef Cpl\_System\_Win32\_Thread\_h\_

#define Cpl\_System\_Win32\_Thread\_h\_

…

#endif

## Minimize the #include Statements in Your Header Files (REQ)

Headers files should only include those (header) files that are required for *header file* to compile. Do not include files that are only used by the associated .cpp|.c file. The #include statements in your header files define the dependencies of the file –fewer the dependencies the better.

## Avoid the Global namespace Unless Absolutely Necessary.

It should be obvious why you want to avoid polluting the global namespace. If you ever “import” code from a developer, department, third party vendor, etc. there is potential for collision in the global namespace. If a collision occurs – someone’s code must change. Modifying “proven” code is bad thing! The two common options to avoiding polluting the global namespace are:

* Nest enumerations, constants, helper classes, etc. inside of existing classes.
* Use the C++ namespace feature.

## Use Standard Integer Types (REQ)

When using types that need to be specific number of bits, use the language standard integer types, e.g int8\_t, uint32\_t, etc.

## Do Not Use Non-portable Constructs

Avoid any of the following constructs, types, pragmas, etc., since they are not guaranteed by the language standards to be portable across all standard compliant compilers.

* Avoid using data structures/typedefs for defining data structures that are exchanged between processors since the use and or inclusion of pack bytes within a data structure is not guaranteed by the language standard.
* Be aware of data-alignment requirements for addressing multibyte words can/will be different between MCU/CPUs.
* Cannot assume that the endianness of multiple byte integers is the same across MCU/CPU/Platforms.
* Do not use bit fields since the ordering of the bit fields (e.g. MSB 0 is bit field 0 vs MSB 7 is bit field 0 in an octet) is not a language standard.
* Only use C/C++ language defined pragmas and preprocessor symbols, i.e. #pack and \_\_PRETTY\_FUNCTION\_\_ are not language standardized pragmas and preprocessor symbols.

## Array Indices (REQ)

Array indices must be guaranteed or explicitly checked to be within the array size prior to use.

## Pointer Check (REQ)

All functions and method shall validate all pointer arguments are non-null before dereferencing the point arguments. The use of the ASSERT() macro (or equivalent) is sufficient to meet this requirement.

# Style

The intent of the style guide is to establish a “style baseline” that all developers are required to follow. This baseline provides consistency across the source code files that aids in reading/maintain the code. In addition, following a common programming style will enable the construction of tools that incorporate knowledge of these standards to help in the development and maintenance of the code.

This guide specifies only basic requirements – the individual programmer is allowed to impose his/her own style/preferences on top of these requirements. There are only two absolute rules to observe when it comes to “creating your own” coding style.

1. The first rule of coding style is consistency:

*Establish a style and stick to it.*

1. The second rule of style is tolerance:

*When you must modify code written by others that has a different style, adopt that style, do not convert the code to your style.*

Style rules tagged with a REQ label must always be followed, or any deviation documented in the source code. Style rules without the REQ label are strongly recommend, but deviation is allowed (without documenting the deviation). Some required style rules explicitly call out exceptions. For these cases – no deviation document in the source is required.

## Comments

### Header Files (REQ)

Header files must be completely documented.  This means every class, method, and data member must have comments.  Header files describe the interfaces of the system, and as such, should contain all the information a developer needs to use/understand the interface.

Header file comments shall not be duplicated-in/copied-to the .c|.cpp files.

### Use Doxygen Comments in Header Files (REQ)

Document your code so that the Doxygen tool (can extract the information stored in your header file comments. Detailed information Doxygen can be found at: [http://www.doxygen.nl](https://www.doxygen.nl/). The following are examples of Doxygen comment styles.

/\*\* This abstract class …

 \*/

class FooBar {

public:

      /// Enables the Widget to begin sending data

      virtual void enableWidget() =0;

};

### C|CPP Files

The quantity/quality of the comments in a CPP file is left to the individual developers to decide.  Comments in the C|CPP files should be limited to implementation details – not the semantics and usage of the functions, methods, classes, etc. (as this are commented in header files).

Add comments whenever you feel that the code is complex, non-standard, and/or clever.  Remember if you want the luxury of other people maintaining your code – they must be able to understand it!

## File Organization

### Organized by namespace (REQ)

Organize your files by component dependencies, not by project. That is, do not create your directory structure to reflect a top-down project decomposition. Rather, organize your code by namespaces where nested namespaces are reflected as subdirectories in their parent namespace directory. By having the dependencies reflected in the directory structure, it is a quick and visual sanity check to avoid undesired and cyclical dependencies.

File names (.h|.c|.cpp files) do not have to be globally unique. The file names only need to be unique within a given directory and namespace. Directories and namespaces are your friends when it comes to naming because they provide a simple mechanism for avoiding future naming collisions.

The C programming language standard does support namespaces. However, the concept of namespace can still be implemented with C by applying the following convention:

* For all functions, types, enums, variable names, preprocessor symbols, etc. that appear in header files, prefix the namespace to the symbol name. For example, for a hypothetical function hello\_world() in the Foo::Bar namespace, the function name would be: Foo\_Bar\_hello\_world(). The hello\_world() function definition would be in a header file in the directory Foo/Bar/. The prefixing rule does not apply to symbols that are exclusive to a .c file (e.g. file static functions and variables).

### README.txt (REQ)

Each namespace directory shall contain a README.txt file that describes the purpose/content of the namespace. The file should provide descriptions, comments, usage semantics, etc. that span multiple files and classes in the namespace.

The file is required to include the following Doxygen formatting (i.e. provides Doxygen with the description of the namespace):

/\*\* @namespace Cpl::System

The System namespace contains platform independent foundation abstractions and classes related to program execution. The interfaces and abstractions are designed to support multi-threaded application.

\*/

### Non-namespace Directories (REQ)

Non-namespace directories can be created for organizational purposes. Non-namespace directories shall start with leading underscore (‘\_’). For example: given the directory path of Cpl/Container/\_0test, the \_0test/ directory would be a non-namespace directory.

### Namespace Directories (REQ)

Namespace directories names shall match (including case) of the namespace they represent/contain.

### Files (REQ)

C++ class (.h and .cpp) file names shall match (including case) of the contained class. If multiple classes are in a single file, the recommendation is name the file after the primary class in the file.

### Single Module or Class in a Namespace

Often you will have a case where the natural naming of module of class within a namespace is the same name as the namespace. For this scenario, the recommendation is that the module/class be named api.h|.c (for C) or Api.h|.cpp (for C++). In turn, the module or class name is also api and Api respectively.

### Keep Header and Source Files Together (REQ)

Place header files (.h) and their corresponding .c|.cpp files in the same directory. Do not create separate source/ and include/ directories. This rule does not apply to header files that declare an interface (e.g. a pure abstract class) where there are/will-be multiple implementations of the interface.

### Fully Qualified Header include Statements (REQ)

Reference header files with a full path relative to the root of the source tree. This means that your #include for header files will contain the relative directory path for each header file. For example, given the following file/directory structure:

src

└───Storm

├───Component

│ ├───Equipment

│ │ ├───Stage

├───Dm

│ └───\_0test

├───Thermostat

│ ├───Main

│ │ ├───\_adafruit\_grand\_central\_m4

│ │ └───\_win32

│ ├───SimHouse

│ └───\_file\_logger

├───TShell

├───Type

└───Utils

File: Storm/Component/AirFilterMonitor.h

#include "Storm/Component/Base.h"

#include "Storm/Dm/MpSimpleAlarm.h"

#include "Storm/Dm/MpVirtualOutputs.h"

File: Storm/Component/Equipment/Cooling.h

#include "Storm/Component/Control.h"

#include "Storm/Component/Equipment/StageApi.h"

This rule only applies #include statements in header files. #include statements in .c|.cpp files are not required to use directory paths for files in the **same** directory.

## Naming

See the glossary for definitions of the case terminology.

### Namespaces, Classes, Methods, and Variables (REQ)

* Namespace names are nouns, should be single words, and shall use Pascal Case.
  + Namespace names should be singular, e.g. Use Cpl::Container instead of Cpl::Containers.
* Class names are nouns and use Pascal Case
* Class methods shall contain a verb and use Camel Case
* Class data member names start with a leading ‘m\_’ followed by a Camel Case, e.g. int m\_myCount;
* Static class data member names start with a leading ‘g\_’ followed by Camel Case, e.g. static int g\_instanceCounts;
* Typedefs and/or structs use Pascal Case
* Do not abbreviate names/words; long names are preferred over cryptic abbreviations.

### C functions, variables, typedefs, and struct (REQ)

* Snake Case or Namespace Case shall be used for all C functions, variables, typedefs and structs.
* Function names shall contain a verb.

### Preprocessor and Macros (REQ)

Preprocessor symbols and macros shall be all upper case with underscores to separate words.

### Statically scoped variables (REQ)

Statically scoped variables in .cpp files use Camel Case. Statically scoped variables in .c files use Snake Case. In addition, all statically scoped variable shall have a trailing underscore (‘\_’) appended to their name to indicate that they are statically scoped, e.g. static int foo\_;

### Global variables (REQ)

Globally scoped C++ variables use Camel Case with a leading ‘g\_’ prefixed to the name e.g. int g\_fooBar;. Globally scoped C variables use Snake Case with a leading ‘g\_’ prefixed to the name. e.g. int g\_foo\_bar;. The exception to this rule is when the variable instance is encapsulated inside a namespace (that is not the std namespace).

### Reserved Function and Method Names (REQ)

* Functions/Methods that perform in-thread initialization of an object shall be named start(...)
* Functions/Methods that perform in-thread shutdown of an object shall be named stop(...)
* Functions/Methods that invoke ITC (inter-thread-communication) initialization (e.g. invoke initialization across thread boundaries) shall be named open(...)
* Functions/Methods that perform ITC shutdown (e.g. reclaiming resources across thread bounders) shall be named close(...)

## Formatting

### Indenting Spacing (REQ)

Tabs **stops** will be set to **4** and **spaces** shall be used for indenting.

### Braces (REQ)

The Allman brace style/indenting shall be used (<https://en.wikipedia.org/wiki/Indentation_style>). For example:

while (x == y)

{

something();

somethingElse();

}

finalThing();

### Flow Control Statements (REQ)

The flow control primitives if, else, while, for and do should be followed by a block, even if it is an empty block or a single statement. For example:

while( /\* do something \*/ ) // NOT Allowed

;

while( /\* do something \*/ ) // Good

{

}

if( isOpened() ) // NOT Allowed

foobar();

if( isOpened() ) // Good

{

foobar();

}

### Elimination of Dead Code

All dead code shall be eliminated from the source code. In addition, no blocks of ‘commented out’ code shall be in a source file when the file is merged to a stable branch in the repository.