STANDARD*

Document Number: 820-0024

Subject: C Coding Standard

Revision: 11

Approvals:

_____	_____
Software Engineering	Corporate Quality

Signatures: (Arena Approvals only)

Issue Date	Revision	Description of Change	Author
Unknown	1	First release	RLS
16 May 2005	2	Added mention of function ordering.	RLS
03 Nov 2005	3	Corrected typos.	RLS
28 Oct 2009	4	Added topics from Netrino “Embedded C Coding Standard,” MISRA, and the code example	TWH
30 Mar 2011	5	New version for SMP project	TWH
30 June 2011	6	Reformatted and updated from review comments.	TWH
26 Jan 2012	7	Adding naming details for callback function, function pointer types, and spacing for prototypes.	CS
29 Mar 2012	8	Re-adding missing rule for Magic numbers (lost in previous iterations), adding naming rules for enum types to provide additional clarity.	CS
26 April 2012	9	Resolving additional naming issues with structures, enumerations and unions. Several rules relating to the naming needed clarification.	CS
09 May 2012	10	Added File check-in comment guidelines and requirements for improved readability of file	RL
28 July 2014	11	Added contents page and fixed numbering if items	RL

1. Introduction

1.1 Purpose

The purpose of this standard is to establish the coding practices, naming convention, and formatting style to be used when developing product software using the C programming language.

1.2 Scope

This standard applies to Delta-Q product software using the C programming language, worldwide.

1.3 Overview

Add stuff about important to have coding guidelines etc

1.4 Change Control

This document is controlled; any changes need to be approved by the SW group lead.

The latest version of this document can be found in SharePoint at the following location: http://dqtsharepoint/Engineering/Shared%20Documents/Process/Standards%20and%20guidelines/DQT%20Coding%20Standard.docx

1.5 References

Ref	Title
[MISRA2004]	MISRA 2004 C Coding guidelines
[Netrino]	Netrino Coding Standard

1.6 Definitions

underscore	The character _.
non-capitalised name	A string of characters containing a number of words (without underscore separators) each starting with a capital letter except the first which starts with a lower-case letter (e.g. nonCapitalisedName).
capitalized name	A string of characters containing a number of words each starting with a capital letter (e.g. CapitalisedName).
upper-case name	A string of upper-case letters (e.g. UPPERCASENAME).
module mnemonic	An uppercase sequence that identifies a software module between 3 and 6 letters (e.g. MOD).
module prefix	A string consisting of a module mnemonic followed by an underscore (e.g. MOD_).
module	A collection of source files that implements a collection of interdependent functionality, in most cases this will be a single source file and header file.
pre-processor constant	A constant defined with a pre-processor *#define* directive.
compile-time constant	A variable defined with a *const* qualifier.
operand	A arguments on which a mathematical or logical operation is performed by an operator.
Operators	Operators are special characters within the language that perform operations on arguments to return a specific result. For example <, > +,...
L-Value	An lvalue is a way of describing the term on the left hand side of an assignment, but doe snot hold for all examples such as consts objects which are l-values but can not appear on the left-handside of expressions.

R-Value	An rvalue is a way of describing the term on the right hand side of an assignment.
	The following example contains instances of l-valuesand r-values:
	#include <string>
	using namespace std;
	int& f();
	
	void func()
	{
	int n;
	char buf[2]; //buf is an r-value
	buf[0]= 'a'; // 'buf[0]' is an l-value, 'a' is an r-value
	n=7; // n is an l-value; the literal 7 is an r-value
	string s1="a", s2="b", s3="c"; // "a", "b", "c" are r-values
	s1=string("a"); // temporaries are r-values
	int* p=new int; // p is an l-value; 'new int' is an r-value

	f()=9; //a function call returning a reference is an l-value
	s1.size(); //otherwise, a function call is an r-value
	}
	
	An *l-value* canappear in a context that requires an *r-value*.In this case, the *l-value* isimplicitly converted to an *r-value*. However,you cannot place an *r-value* in acontext that requires an *l-value*.

declaration	Declares a function or variable, but does not define it. i.e defines the function prototype.
definition	Either defines a function or variable that was declared earlier in the same block or subprogram, or declares and defines a function.
ternary	Composed of three - For example this is how the ternary operator "?:" earns its name because it's the only operator to take three operands.

1.7 Format

Each coding rule shall be formatted as follows:

x.x.x Topic

Rules:
[<Y>1.1] Abbreviations
[<Y>1.2] A table of additional

Examples: See TBD

Reasoning: Rational for the rule, optional rule where an explanation adds value

Exceptions: Project TBD

Enforcement: TBD.

<Y> is defined as:

- N for Naming convention
- S for style
- V for Use of Variables
- P for use of Procedure calls
- M for use of Methods
- C for comments
- E for Expressions and statements

Contents

2. Standard

2.1 General Rules

2.1.1 Magic Numbers

Rules:

[G1.1] Magic numbers shall never be used, they shall be replaced by either a constant variable or #define alias description whichever is more appropriate for the intended use.

Examples:

BAD Example:

```
...
    case DCRLY_eDcRelayOpening:
      if ( ( dc_relay_timer > SECONDS_IN_MS(1u) )  &&
         ( 2.0f > GetBattCurrent ) )
      {
         sm_DCRelayState = 0u;
      }
      dc_relay_timer += 10ul;
      break;
```

GOOD Example:

```
#define SM_TICKRATE_MS          (10ul)
#define SAFE_RELAY_CLOSE_CURRENT    (2.0f)
#define DC_RELAY_CLOSE_TIME (1u)
...
```

```
typedef enum DCRLY_Command_t {
   DCRLY_eRequestRelayOpen = 0u,
  DCRLY_eRequestRelayClose,
  DCRLY_eForceRelayOpen,
  DCRLY_eForceRelayClose,
  DCRLY_eCmdEnd
} DCRLY_Command_t;
...
      case DCRLY_eDcRelayOpening:
         if ( ( dc_relay_timer > SECONDS_IN_MS(DC_RELAY_CLOSE_TIME) )  &&
            ( SAFE_RELAY_CLOSE_CURRENT > GetBattCurrent ) )
         {
            sm_DCRelayState = DCRLY_eDcRelayOpen;
         }
         dc_relay_timer += SM_TICKRATE_MS;
         break;
```

Exceptions: There are only two exceptions to this rules which under some circumstances considered acceptable, 0 and 1. This is to allow basic indexing of adjacent value in an index and for initializing loop counters. These two values are used so commonly it makes sense to allow their appearance in the code as its widely understood the intent for these purposes.

Enforcement: Preventing magic number use will be enforced during code reviews.

2.1.1 Commenting Source Check-In

Rules:

[G2.1] All source files being checked in must have a comment which details the reason for the work. The reason must contain an identifier for the bug it is repairing including number or the code review number for traceability if applicable.

[G2.2] All comments must be bulleted by a * character to indicate a point and the use of a – character for a sub point which is further indented on a subsequent line. This is done to provide easier parsing to the eye when reading module history.

Examples:

The examples below shows the results of comments in a file history. The examples are the same entries in both the good and bad cases so as to highlight the value of inputing good and back comment into the code and its affect on the ability understand was was changed and the scope of the inputs.

BAD Example:

The following is how the comments will appear in the module revision history:

UserA - Tuesday, May 08, 2012 3:14:19 PM

code review correction

UserA - Monday, May 07, 2012 7:15:42 PM

code review correction.

UserB - Monday, May 07, 2012 2:34:30 PM

Add NVS init struct for reset detail structs.

UserC - Friday, May 04, 2012 7:56:17 PM

Clearing up large RAM buffers used. - Freed 0x0993 = 2451 bytes.

UserA - Friday, May 04, 2012 3:35:20 PM

Swapping the Vishay and Murata temperature curves.

Battery Temp --> Vishay

T1, T2 Temp --> Murata

UserB - Friday, May 03, 2012 3:35:20 PM

adjust flash access wait timing.

GOOD Example:

The following is how the comments will appear in the module revision history:

UserA - Tuesday, May 08, 2012 3:14:19 PM

* code review #23 corrections as follows:

- Variable naming update to conform to coding standard mnemonics

- Added additional comments to the module header block.

UserA - Monday, May 07, 2012 7:15:42 PM

* code review #23 corrections as follows:

- Added static keyword to all local functions.

UserB - Monday, May 07, 2012 2:34:30 PM

* Add NVS init struct for reset detail structs.

UserC - Friday, May 04, 2012 7:56:17 PM

* Clearing up large RAM buffers used. - Freed 0x0993 = 2451 bytes.

- Reduced Stack sizes for Start up task and Monitor Task.

- Changed smpcfg_zVoltModelConfig and SMPCFG_paInit24vModel in module to be static const.

UserA - Friday, May 04, 2012 3:35:20 PM

* Swapping the Vishay and Murata temperature curves as follows:

- Battery Temp --> Vishay

- T1, T2 Temp --> Murata

UserB - Friday, May 03, 2012 3:35:20 PM

* Fix Bugzilla #33 – extended timeout delay for flash write/read to data sheet max.

Exceptions: None.

Enforcement: This shall be enforced through code reviews.

2.2 Naming conventions

2.2.1 Common Abbreviations

Rules:

[N1.1] Abbreviations and acronyms should generally be avoided unless their meanings are widely and consistently understood in the engineering community. The table below contains a list of commonly used abbreviations and their meanings.

[N1.2] A table of additional project-specific abbreviations and acronyms shall be maintained in a version controlled document.

Examples: See **Error! Reference source not found.**Standard Abbreviations Wiki page

At the following loacation:

http://dqtsharepoint/Engineering/Software/SoftwareWiki/Wiki%20Pages/Standard%20Abbreviations.aspx

Exceptions: Project-specific abbreviations that do not conflict with the common abbreviations in this standard may be added in the manner described above.

Enforcement: Consistent use of these abbreviations shall be enforced during code reviews.

2.2.2 Module Names (Source Files)

Rules:

[N2.1] All module names shall consist entirely of lowercase letters, numbers, and underscores.

[N2.2] No spaces shall appear within the file name.

[N2.3] All module mnemonic shall have names between 3 and 6 letters (e.g. mod).

[N2.4] Source file names shall start with the mnemonic of the corresponding module (e.g. mod.c, mod.h).

[N2.5] Each module include file shall be named xxx.h where xxx is the module mnemonic.

[N2.6] If a module has one source file, it shall be named xxx.c.

[N2.7] If a module has more than one source file (this will be done only in exceptional cases), the module's source files shall have names that match the pattern xxx*.c. If the module needs any static variables with a scope of the whole module, they shall be placed into a structure that will be referenced by a single global pointer variable. The global pointer variable shall be given a name that identifies the module. No other modules shall ever reference this global variable.

[N2.8] All module names shall be unique in their first eight characters, with .h and .c used for the suffix for header and source files respectively.

[N2.9] No module name shall share the name of a standard library header file. For example, modules shall not be named "stdio" or "math".

[N2.10] Any module containing a main() function shall have the word "main" in its filename.

Reasoning: Multi-platform work environments (e.g., Linux and Windows) are the norm rather than the exception. To support the widest range, file names should meet the constraints of the least capable platforms. Additionally, mixed case names are error prone due to the possibility of similarly-named but differently-capitalized files becoming confused. The inclusion of "main" in a file name is an aid to the maintainer that has proven useful.[1]

Exceptions: None.

Enforcement: An automated tool shall confirm that all file names used in each build are consistent with these rules.

[1] We have encountered the case of a company with one project having over 200 files containing a function called main().

2.2.3 Types

Rules:

[N3.1] A module's exported types shall use names containing the module prefix and a capitalized name (e.g. MOD_Code_t).

[N3.2] A module's private types shall use names containing the module prefix in lowercase and a capitalized name (e.g. mod_PrivateWidget_t)

[N3.3] Type names shall be suffixed with a "_t".

[N3.4] Structures, unions, and enumeration naming is in accordance with section 2.2.5 Structures and unions.

Examples: See TBD

Exceptions: Project TBD

Enforcement: TBD.

2.2.4 Constants

Rules:

[N4.1] Each local and exported pre-processor constant shall have a name consisting of a module prefix and an upper-case name (e.g. MOD_CONSTANT).

[N4.2] Each global system pre-processor constants or macros, shall have a upper-case name with a application global mnemonic prefix SYS (e.g. SYS_CONSTANT).

[N4.3] Each global system, local or exported pre-processor constant shall indentify in it's name if it is not of integer type.(e,g MOD_CONSTANT_STRING

[N4.4] Each local and exported compile time constant shall be named as per module variables. See section 2.2.7 Variables for details. (e.g. mod_kArraySize).

[N4.5] Each exported pre-processor or compile time constant shall be declared in the corresponding module include file (e.g. mod.h).

[N4.6] Each local pre-processor or compile time constant shall be declared in the source file in which it is used (e.g. mod.c). They should be grouped together, each with a commented description. In the case of multiple source files in a module where constants are used by more than one source file but not externally then they shall share a common private header file (e.g. mod_priv.h).

[N4.7] Compile time constants shall be used in preference to pre-processor constants.

Examples: See

Exceptions: TBD.

Enforcement: Consistent TBD shall be enforced during code reviews.

2.2.5 Structures and unions

Rules:

[N5.1] The names of all new data types, including structures and unions shall consist only of a name that follows the variable naming (see section 2.2.7) (including the type 'z' or 'u' as needed) and end with '_t'.

[N5.2] All new structures and unions shall be named via a typedef.

[N5.3] The name of each field of a structure shall not have a module prefix since its scope is limited to an instance of the structure. Names of fields shall follow the same naming convention as described in section 2.2.7 for other variables, with exception of the module mnemonic.

[N5.4] Any spacer elements within structures shall be prefixed with double underscore (__) characters.

[N5.5] Shall have a tag name which is the same name as the type declaration as many compilers use the tag name in the error messages.

Example:

```
typedef struct TMR_zTimer_t
{
    uint16_t count;
    uint16_t maxCnt;
    uint16_t __unused;
    uint16_t control;
    TMR_zTimer_t* pNextItem;
    TMR_zTimer_t* pPrevItem;
} TMR_zTimer_t;
```

Reasoning: Type names and variable names are often appropriately similar. For example, a set of timer control registers in a peripheral calls out to be named 'timer'. To distinguish the structure definition that defines the register layout, it is valuable to create a new type with a distinct name, such as 'TMR_zTimerReg_t'. If necessary this same type

could then be used to create a shadow copy of the timer registers, say called 'TMR_zTimerRegShadow_t'.

Exceptions: It is not necessary to use typedef with anonymous structures and unions.

Enforcement: An automated tool shall scan new or modified source code prior to each build to ensure that the keywords *struct* and *union* are used only within typedef statements or in anonymous declarations.

2.2.6 Enumerations

Rules:

[N6.1] The names of all new enumerations, shall consist only of a name that follows the variable naming (see section 2.2.7) (including the type 'e') and end with '_t'.

[N6.2] All new enumerations shall be named via a typedef.

[N6.3] The name of each enumeration will carry a pseudo-mnemonic which is made up of the type name (or as much as is reasonable for conveying purpose doesn't include the _t). The pseudo-mnemonic shall be representative of the type appear as the first part of the enumeration and conforms to the same naming convention as described in section 2.2.7 for other variables.

[N6.4] All states of a state machine shall have an enumeration associated with each state and contain the word state as part of the name.

[N6.5] Shall have a tag name which is the same name as the type declaration as many compilers use the tag name in the error messages.

Example:

```
typedef enum sm_eVacStates_t {
    sm_eVacStateOff,
    sm_eVacStateDropout,
    sm_eVacStateLow,
    sm_eVacStateValid,
    sm_eVacStateMax
}sm_eVacStates_t;
```

Reasoning: Enumeration types are a key component of communicating intent and descriptive. As such there use is essential for many code constructs, in particular in error handling and state machines. The use of the pseudo-mnemonic naming helps other developers determine if comparisons between variables of that type are appropriate and

also address the conflicts in naming that can occur with large projects and module reuse. Another useful example is in understanding state transitions in state machines. These examples highlight a few uses for the naming and creation of enumerations according to the above rules.

Enforcement: An automated tool shall scan new or modified source code prior to each build to ensure that the keyword *enum* are used only within typedef statements or in anonymous declarations.

2.2.7 Variables

Rules:

[N7.1] No variable shall have a name that is a keyword of C, C++, or any other well-known extension of the C programming language, including specifically K&R C and C99. Restricted names include static, volatile, extern, pragma, interrupt, inline, restrict, class, true, false, public, private, friend, and protected.

[N7.2] No variable shall have a name that overlaps with a variable name from the C standard library (e.g., errno).

[N7.3] No variable shall have a name that begins with an underscore.

[N7.4] All variable names shall be unique within 31 characters.[2]

[N7.5] No variable name shall be shorter than 3 characters, including loop counters.[3]

[N7.6] No variable name shall contain any numeric value that is called out elsewhere, such as the number of elements in an array or the number of bits in the underlying type.

[N7.7] Each variable's name shall indicate the type of variable, be descriptive of its purpose and include units where appropriate.

I.E <mod>_<type><Descriptive Section>_<Units>.

<mod> is the 3-6 character module mnemonic and is either **UPPERCASE** for exported variables or **lowercase** for local variables or missing for automatic function scope variables.

<Type> values are:

a for array, (e.g. mod_aGpioPins[])

[2] Rule 5.1 (required) of [MISRA2004]

[3] This is because you can't do a meaningful global search for "i".

p for pointer, (e.g. mod_pLedReg)

pp for pointer to pointer, (e.g. mod_ppVectorTable)

f for float (e.g. mod_fOutputCurrent_Amps)

pfn for function point, (e.g static void (*FncPtr)(void) mod_pfnXxYx;

e for enum values (e.g. mod_eInvalidAddrExcpt)

q for string (e.g. mod_qName[])

z for structure (e.g. mod_zEtherMsg)

For other variables the type parameter does not exist. (e.g. mod_localVariable)

<Units> are specified in **Error! Reference source not found.**.

[N7.8] The descriptive section name of all integer variables containing "effectively Boolean" information (i.e., 0 vs. nonzero) shall be phrased as the question they answer[4]. For example, isDoneYet or mod_isBufferFull.

[N7.9] The descriptive section shall always use medial capitals (AKA Camel case)[5] except if a <type> has not been specified. In this case the descriptive section name shall start with a non-capitalised word instead. (e.g. MOD_exportedVariable).

[N7.10] Each local variable or static variable (with scope only within a module) shall have a non-capitalised module name preceded by a lowercase module prefix (e.g. mod_localVariable). In the case of multiple source files in a module where local variables are used by more than one source file but not externally then their declaration they shall share a common private header file (e.g. mod_priv.h), and the STATIC keyword should not be used.

[N7.11] Each automatic variable (with scope only within a specific function) shall have a non-capitalised name without a module prefix (e.g. loopCounter).

Example: The following gives some example for the variable naming conventions.

<type> loopCounter (Function scope variable)

[4] It is unsafe, in C, to define constants such as TRUE and FALSE where TRUE is equal to 1. However, it is safe to compare "effectively Boolean" integer values against 0. For example, the test "if (!gb_buffer_is_full)" is safe and consistent with Rule 13.2

[5] otherwise known as medial capitals or camel case

<type>* mod_pLedReg (File scope pointer to variable)
bool_t MOD_isDoneYet (Exported Boolean variable)
<type> mod_localCurrentVariable_Amps (File scope integer variable)
const <type>* pVarName (Function scope variable)
const <type>* const* ppVarName (Function scope variable)

Reasoning: The base rules are adopted to maximize code portability across c compilers. Many C compilers recognize differences only in the first 31 characters in a variable's name and reserve names beginning with an underscore for internal names. The other rules are meant to highlight risks and ensure consistent proper use of variables.

It's valuable having enumerated types identified since they can change in size depending on compilers. This can be very helpful when porting code.

Exceptions: None.

Enforcement: These variable-naming rules shall be enforced during code reviews.

2.2.8 Functions

Rules:

[N8.1] No procedure shall have a name that is a keyword of C, C++, or any other well-known extension of the C programming language, including specifically K&R C and C99. Restricted names include *interrupt*, *inline*, *class*, *true*, *false*, *public*, private, friend, protected, and many others.

[N8.2] No procedure shall have a name that overlaps a function in the C standard library. Examples of such names include strlen, atoi, and memset.

[N8.3] No procedure shall have a name that begins with an underscore.

[N8.4] All procedure names shall be unique within 31 characters.[6]

[N8.5] Each exported function shall be named with a module prefix and a capitalized name followed by an underscore (e.g. MOD_ExportedFunction).

[N8.6] Each exported function shall be declared with all its parameters in the corresponding module include file. (e.g. EXTERN int MOD_ReceiveWords(MOD_Params_t params, Ptr base, Uns length); in MOD.h)

[6] Rule 5.1 (required) of [MISRA2004]

[N8.7] Each local function within a module shall have a capitalized name with a lowercase module prefix (e.g. mod_LocalFunction).

[N8.8] Each local function within a module shall be declared as static functions so that they are not accessible by other modules. (e.g. static void mod_LocalFunction(Int num); in mod.c). In the case of multiple source files in a module where local functions are used by more than one source file but not externally then their declaration shall share a common private header file (e.g. mod_priv.h) and the static keyword shall not be used.

[N8.9] No function macro name shall contain any lowercase letters.

[N8.10] Underscores shall be used to separate descriptive words from other parameters in procedure names. (e.g mod_LocalFunction_Volts)

[N8.11] Each procedure's name shall be descriptive of its purpose. Note that procedures encapsulate the "actions" of a program and thus benefit from the use of verbs in their names (e.g., adc_ReadChannel(), adc_WriteChannel()); this "nounverb" word ordering is recommended. Alternatively, procedures may be named according to the question they answer (e.g., mod_IsLedOn()). Action verb in the name should be at the end. In short they answer a question.

Reasoning: Good function names make reviewing and maintaining code easier (and thus cheaper). The data (variables) in programs are nouns. Functions manipulate data and are thus verbs. The use of module prefixes is in keeping with the important goal of encapsulation and helps avoid procedure name overlaps.

Exceptions: None.

Enforcement: Compliance with these naming rules shall be established in the detailed design phase and be enforced during code reviews.

2.3 Coding Style

2.3.1 Line Widths

Rules:

[S1.1] The length of all lines in a program shall be limited to a maximum of 90 characters.

Reasoning: Code reviews and other examinations are from time-to-time conducted on printed pages, which must be free of distracting line wraps as well as missing (i.e., past the right margin) characters. Line width rules also ease onscreen side-by-side code differencing.

Exceptions: None.

Enforcement: Violations of this rule shall be detected by an automated scan during each build.

2.3.2 Braces

Rules:

[S2.1] Braces shall always surround the blocks of code (a.k.a., compound statements), following if, else, switch, while, do, and for statements; single statements and empty statements following these keywords shall also always be surrounded by braces.

[S2.2] Each left brace ('{') shall appear by itself on the line below the start of the block it opens. The corresponding right brace ('}') shall appear by itself in the same position the appropriate number of lines later in the file.

Examples:

```
If (.....)
{
    ...
}
```

Reasoning: There is considerable risk associated with the presence of empty statements and single statements that are not surrounded by braces. Code constructs like this are often associated with bugs when nearby code is changed or commented out. This risk is entirely eliminated by the consistent use of braces. The placement of the left brace on the following line allows for easy visual checking for the corresponding right brace.

Exceptions: None.

Enforcement: The appearance of a left brace after each if, else, switch, while, do, and for shall be enforced by an automated tool at build time. The same or another tool shall be used to enforce that all left braces are paired with right braces at the same level of indentation.

2.3.3 Parentheses

Rules:

[S3.1] Do not rely on C's operator precedence rules, as they may not be obvious to those who maintain the code. To aid clarity, use parentheses (and/or break long statements into multiple lines of code) to ensure proper execution order within a sequence of operations.

[S3.2] Unless it is a single identifier or constant, each operand of the logical && and || operators shall be surrounded by parentheses.

Example:

```
if ((a < b) && (b < c))
{
      result = (3 * a) + b;
}
return result;
```

Exceptions: None.

Enforcement: These rules shall be enforced during code reviews.

2.3.4 White Spaces

Rules:

[S4.1] Each of the keywords if, else, while, for, switch, and return shall always be followed by one space.

[S4.2] Each of the assignment operators =, +=, -=, *=, /=, %=, &=, |=, ^=, ~=, and != shall always be preceded and followed by one space.

[S4.3] Each of the binary operators +, -, *, /, %, <, <=, >,>=, ==, !=, <<, >>, &, |, ^, &&, and || shall always be preceded and followed by one space.

[S4.4] Each of the unary operators +, -, ++, --, !, and ~, shall always be written without a space on the operand side and with one space on the other side.

[S4.5] The pointer operators * and & shall be written with white space on each side within declarations but otherwise without a space on the operand side.

[S4.6] The ? and : characters that comprise the ternary operator shall each always be preceded and followed by one space.

[S4.7] The structure pointer and structure member operators (-> and ., respectively) shall always be without surrounding spaces.

[S4.8] The left and right brackets of the array subscript operator ([and]) shall always be without surrounding spaces.

[S4.9] Expressions within parentheses shall always have no spaces adjacent to the left and right parenthesis characters.

[S4.10] The left and right parentheses of the function call operator shall always be without surrounding spaces, except that the function declaration and definition. The function definition shall have one space while the declaration must have one or more spaces between the function name and the left parenthesis to allow that one particular mention of the function name to be easily located while using words search function of IDEs/editors.

[S4.11] Each comma separating function parameters shall always be followed by one space.

[S4.12] Each semicolon separating the elements of a for statement shall always be followed by one space.

[S4.13] Each semicolon shall follow the statement it terminates without a preceding space.

Reasoning: The placement of white space is as important as the placement of the text of a program. Good use of white space reduces eyestrain and increases the ability of the author and reviewers of the code to spot potential bugs.

Exceptions: None.

Enforcement: These rules shall be enforced by an automated tool such as a code beautifier.

Example
```
If (a < b)
{
```

```
    a++;
}
```

2.3.5 Alignment

Rules:

[S5.1] The names of variables within a series of declarations shall have their first characters aligned.

[S5.1] The names of struct and union members shall have their first characters aligned.

[S5.1] The assignment operators within a block of adjacent assignment statements shall be aligned.

[S5.1] The # in a preprocessor directive shall always be located in column 1, except when indenting within a #if or #ifdef sequence.

[S5.1] Closing blocks of preprocessor #endif shall be proceeded by a comment version of the condition that opened the block as shown here: //#if <Condition>

Reasoning: Visual alignment emphasizes similarity. A series of consecutive lines containing variable declarations is easily spotted and understood as a block. Blank lines and unrelated alignments should be used to visually distinguish unrelated blocks of code appearing nearby.

Exceptions: None.

Enforcement: These rules shall be enforced during code reviews.

2.3.6 Blank Lines

Rules:

[S6.1] No line of code shall contain more than one statement.

[S6.1] There shall be a blank line before and after each natural block of code. Examples of natural blocks of code are loops, if-else and switch statements and case groups, and consecutive declarations.

[S6.1] Each source file shall have a blank line at the end.[7]

[7] This is for portability, as some compilers require the blank line.

Reasoning: Appropriate placement of white space provides visual separation and thus makes code easier to read and understand, just as the white space areas between paragraphs of this coding standard aid readability.

Exceptions: None

Enforcement: These rules shall be enforced during code reviews.

2.3.7 Indentation

Rules:

[S7.1] Each indentation level within a module should consist of 4 spaces.

[S7.2] Within a switch statement, each case statement should be indented; the contents of the case block should be indented once more.

[S7.3] Whenever a line of code is too long to fit within the maximum line width, indent the second and any subsequent lines in the most readable manner possible.

Example:

```
sys_error_handler(int err)
{
	switch (err)
	{
		case ERR_THE_FIRST:
			returnErr = firstErr;
		break;

		default:
			returnErr = defaultErr;
		break;
	}

	// Purposefully misaligned indentation; see why?
	if ((first_very_long_comparison_here
	      && second_very_long_comparison_here)
	  || third_very_long_comparison_here)
	{
		...
	}
}
```

Reasoning: Fewer indentation spaces increase the risk of visual confusion while more spaces increases the likelihood of line wraps.

Exceptions: The indentation in legacy code modules that are indented differently shall not be changed unless it is anticipated that a significant amount of the code will be modified. In that case, the indentation across the entire module shall be changed in a distinct version control step. This is because a side effect of changing indentation is the loss of difference tracking capability in the version control system. It is thus valuable to separate the code changes from the indent changes.

Enforcement: An automated tool shall be provided to programmers to convert indentations of other sizes automatically. This tool shall modify all new or changed code prior to each build.

2.3.8 Tabs

Rules:

[S8.1] The tab character shall never appear within any module. This rule is related to [S7.1] which specifies indentation as 4 spaces.

Reasoning: The width of the tab character varies by editor and programmer preference, making consistent visual layout a continual source of headaches during code reviews and maintenance.

Exceptions: Existing tabs in legacy code modules shall not be eliminated unless it is anticipated that a significant amount of the code will be modified. In that case, the tabs shall be eliminated from the entire module in a distinct version control step. This is because a side effect of eliminating tabs is the loss of difference tracking capability in the version control system. It is thus valuable to separate the code changes from the white space changes.

Enforcement: The absence of the tab character in new or modified code shall be confirmed via an automated scan at each build.

2.3.9 Variable Declarations

Rules:

[S2.1] The comma (‘,’) operator shall not be used within variable declarations.

Example:
```
uint8_t * x, y; // Is y supposed to be a pointer?
```

Reasoning: The cost of placing each declaration on a line of its own is low. By contrast, the risk that either the compiler or a maintainer will misunderstand your intentions is high.

Exceptions: None.

Enforcement: These rules shall be enforced during code reviews.

2.4 Use of Variables, structures or Unions

2.4.1 Keywords to Frequent

Rules:

[V1.1] The static keyword shall be used to declare all functions and variables that do not need to be visible outside of the scope in which they are declared. Static can limit the scope of a variable to be either function or file scope depending on where the variable declaration exists.

[V1.2] The const keyword shall be used whenever appropriate. Examples include:

[V1.3] To declare variables that should not be changed after initialization,

ii. To define call-by-reference function parameters that should not be modified (e.g., char const * param),

iii. To define fields in structs and unions that should not be modified (e.g., in a struct overlay for memory-mapped I/O peripheral registers), and

iv. As a strongly typed alternative to #define for numerical constants.

[V1.4] The volatile keyword shall be used whenever appropriate. Examples include:

ii. To declare a global variable accessible (by current use or scope) by any interrupt service routine,

iii. To declare a global variable accessible (by current use or scope) by two or more tasks,

iv. To declare a pointer to a memory-mapped I/O peripheral register set (e.g., timer_t volatile * const p_timer), and

v. To declare a delay loop counter.

Reasoning: C's static keyword has several meanings. At the module-level, global variables and functions declared static are protected from external use. Heavy-handed use of static in this way thus decreases coupling between modules. The const and volatile keywords are even more important. The upside of using const as much as possible is compiler-enforced protection from unintended writes to data that should be read-only. Proper use of volatile eliminates a whole class of difficult-to-detect bugs by preventing compiler optimizations that would eliminate requested reads or writes to variables or registers.[8]

Exceptions: None.

Enforcement: Appropriate use of these important keywords shall be enforced during code reviews.

2.4.2 Initialization

Rules:

[V2.1] All variables shall be initialized before use.

[V2.2] It is preferable to create variables as you need them, rather than all at the top of a function.[9]

Example:

```
for (int loop = 0; loop < MAX_LOOPS; loop++)
{
...
}
```

[8] Anecdotal evidence suggests that programmers unfamiliar with the volatile keyword think their compiler's optimization feature is more broken than helpful and disable optimization. We believe that the vast majority of embedded systems contain bugs waiting to happen due to missing volatile keywords. Such bugs typically manifest themselves as "glitches" or only after changes are made to a "proven" code base.

[9] Yet another handy feature allowed by [C99] but not in [C90].

Reasoning: Too many programmers assume the C runtime will watch out for them. This is a very bad assumption, which can prove dangerous in a real-time system. It is easier to initialize some variables closer to their use, and this also aids readability of the code.[10]

Exceptions: None.

Enforcement: An automated tool shall scan all of the source code prior to each build, to warn about variables used prior to initialization. Lint is an example of a tool that can do this well.

2.4.3 Globals

Rules:

[V3.1] Global variables shall be used only when absolutely necessary in exceptional circumstances.

For example, modules that do not conveniently fit into a single source file may have a single global variable to reference a structure containing all static variables used by the module.

[V3.2] When the performance of a particular module is critical, the method used to transfer information to time critical routines may be changed from parameter passing to use of global variables. This should only be done when all else fails.

Examples: See TBD

Exceptions: Project TBD

Enforcement: TBD.

2.4.4 Dynamically Allocated Variables

Rules:

[V4.1] Variables whose values need to persist only during the execution of a thread shall be allocated on the thread's private stack.

[10] One study of back-and-forth eye movements during code reviews ([Uwano]) demonstrated the importance of placing variable declarations as close as possible to the code that uses them.

[V4.2] Each software and hardware interrupt routine should minimize the use of automatic variables because they use space on the stack of the thread which they interrupt.

[V4.3] Each task shall allocate a stack large enough to hold all its own automatic variables and all automatic variables that the worst case of nested interrupt service routines shall allocate.

Examples: See TBD

Exceptions: Project TBD

Enforcement: TBD

2.4.5 Static Variables

Rules:

[V5.1] Static variables shall be used with caution, especially in situations where functions may be re-entered.

[V5.2] Each module may use one or more static variables to keep track of its own state between successive calls to its routines by its clients.

[V5.3] Each module that dynamically allocates and frees its own objects shall maintain a linked list of unused such objects. This linked list shall be referenced by a static variable in the module. When this list is empty and another such object is required, a new object shall be allocated from memory.

Examples: See TBD

Exceptions: Project TBD

Enforcement: TBD.

2.4.6 Fixed-Width Integers

Rules:

[V6.1] Whenever the width, in bits or bytes, of an integer value matters in the program, one of the fixed width data types shall be used in place of char, short, int, long,

or long long. The signed and unsigned fixed width integer types shall be as shown in the table below.[11]

[V6.2] The keywords short and long shall not be used.

[V6.3] Use of the keyword char shall be restricted to the declaration of and operations concerning strings.

Table 1 Fixed Width Integer Types

Integer Width	Signed Type	Unsigned Type
8 bits	int8_t	uint8_t
16 bits	int16_t	uint16_t
32 bits	int32_t	uint32_t
64 bits	int64_t	uint64_t

Reasoning: The [C90] standard allows implementation defined widths for *char*, *short*, *int*, *long*, and *long long* types, which leads to portability problems. The [C99] standard did not resolve this, but introduced the uniform type names shown in the table, which are defined in the C99 header file <stdint.h>.

Exceptions: In the absence of a C99-compliant compiler, it is acceptable to define the set of fixed width types in the table above as typedefs based on *char*, *short*, *int*, *long*, and *long long*. If this is done, use compile-time checks (such as static assertions) to have the compiler flag incorrect type definitions. It is acceptable to use the native types when C Standard Library functions are used—just be careful.

Enforcement: At every build an automated tool shall scan for and flag the use of the keywords *short* and *long*, which are not to be used. Compliance with the other rules shall be checked during code reviews.

2.4.7 Signed Integers

Rules:

[V7.1] Bit-fields shall not be defined within signed integer types.

[11] If program execution speed is also a consideration, note that the [C99] standard defines a set of "fastest" fixed-width types. For example, *uint_fast8_t* is the fastest integer type that can hold at least 8 bits of unsigned data.

[V7.2] None of the bit-wise operators (i.e., &, |, ~, ^, <<, and >>) shall be used to manipulate signed integer data.

[V7.3] Signed integers shall not be combined with unsigned integers in comparisons or expressions. In support of this, decimal constants meant to be unsigned should be declared with a 'u' at the end.

Example:

```
uint8_t a = 6u;
int8_t b = -9;
if (a + b < 4)
{
    // This correct path should be executed
    // if -9 + 6 were -3 < 4, as anticipated.
}
else
{
    // This incorrect path is actually executed,
    // as -9 + 6 becomes (0xFF - 9) + 6 = 252.
}
```

Reasoning: Several details of the manipulation of binary data within signed integer containers are implementation defined behaviours of the C standard. Additionally, the results of mixing signed and unsigned data can lead to data-dependent bugs.

Exceptions: None.

Enforcement: Static analysis tools can be used to detect violations of these rules.

2.4.8 Floating Point

Rules:

[V8.1] Avoid the use of floating point constants and variables whenever possible. Fixed-point math may be an alternative. This may be more relaxed in the cases where an FPU is available on chip. If so consideration of using an abstract type to allow portability to MCUs without should be done.

[V8.2] When floating point calculations are necessary:

[V8.3] Use the [C99] type names float32_t, float64_t, and float128_t.

[V8.4] Append an 'f' to all single-precision constants (e.g., pi = 3.1415927f).

[V8.5] Ensure that the compiler supports double precision, if your math depends on it.

[V8.6] Never test for equality or inequality of floating point values.

[V8.7] If a FPU is available confirm that if it only has float support, that the compiler doesn't automatically promote to double as noted below.

Example:

```
// Ensure the compiler supports double-precision.
#include <limits.h>
#if (DBL_DIG < 10)
#error "Double precision is not available!"
#endif
```

Reasoning: There are a large number of risks of errors stemming from incorrect use of floating point arithmetic; these are outside the scope of this document.[13] By default, C promotes all floating-point constants to double precision, which may be inefficient or unsupported on the target platform. However, many microcontrollers do not have any hardware support for floating point math. The compiler may not warn of these incompatibilities, instead performing the requested numerical operations by linking in a large (typically a few kilobytes of code) and slow (numerous instruction cycles per operation) floating-point emulation library.

Exceptions: None.

Enforcement: These rules shall be enforced during code reviews.

2.4.9 Structures and Unions

Rules:

[V9.1] Appropriate care shall be taken to prevent the compiler from inserting padding bytes within struct or union types used to communicate to or from a peripheral or over a bus or network to another processor.

[V9.1] Appropriate care shall be taken to prevent the compiler from altering the intended order of the bits within bit-fields.[12]

Reasoning: There is a tremendous amount of implementation-defined behaviour in the area of structures and unions. Bit-fields, in particular, suffer from severe portability problems, including the lack of a standard bit ordering and no official support for the fixed-width integer types they so often call out to be used with.

Exceptions: None.

Enforcement: These rules shall be enforced during code reviews.

2.5 Use of Procedure Calls (Functions)

2.5.1 Functions

Rules:

[P10.1] All reasonable effort shall be taken to keep the length of each function limited to one printed page, or about 100 lines.

[P10.2] Whenever possible, all functions shall be made to start at the top of a printed page, except when several small functions can fit onto a single page.

[P10.3] All functions shall have just one exit point and it shall be at the bottom of the function. That is, the keyword return shall appear a maximum of once.[13]

[P10.4] Each function parameter shall be explicitly declared and meaningfully named.

[P10.5] Parameters which are not altered by the function shall be declared as const by the function prototype.

Example:

```
int16_t state_change (int8_t event)
{
    int result = ERROR;
    if (EVENT_A == event)
```

[12] Options include static assertions or other compile-time checks as well as the use of pre-processor directives to select one of two competing struct definitions. This can be accomplished using #pragma pack if support by the compiler.

[13] In fact, [IEC61508] requires it.

```
    {
        result = STATE_A;
        // Don't return here.
    }
    else
    {
        result = STATE_B;
    }
    return (result);
}
```

Reasoning: Code reviews take place at the function level. Each function should be visible on a single printed page, so that flipping back and forth (a distraction) is not necessary. Similarly, multiple exit points are distracting to reviewers and thus do more harm than good to readability.

The function length rule gets better stack utilization, more dense code (smaller footprint) as well as better modularity, albeit at the expense of some cycles.

Exceptions: None.

Enforcement: Compliance with these rules shall be checked during code reviews.

2.5.2 Function-Like Macros

Rules:

[P2.1] Parameterized macros shall not be used if an inline function can be written to accomplish the same task.[14]

[P2.2] If parameterized macros are used for some reason, these rules apply:

i. Surround the entire macro body with parentheses.
ii. Surround each use of a parameter with parentheses.
iii. Use each argument no more than once, to avoid unintended side effects.

Example:

```
// Don't do this ...
#define MAX(A, B) ((A) > (B) ? (A) : (B))

// ... if you can do this instead.
```

[14] Note that individual functions will be needed to support each base type for comparison.

```
inline int max(int a, int b)
```

Reasoning: There are a lot of risks associated with the use of pre-processor # defines, and many of them relate to the creation of parameterized macros. The extensive use of parentheses (as shown in the example) is important, but does not eliminate the unintended double increment possibility of a call such as MAX(i++, j++). Other risks of macro misuse include comparison of signed and unsigned data or any test of floating-point data. Making matters worse, macros are invisible at run-time and thus impossible to step into within the debugger.

Exceptions: In the case of necessary (and tested, and documented) efficiency, a local exception can be approved. That's when the other rules kick in.

Enforcement: The avoidance of and safe use of macros shall be enforced during code reviews.

2.5.3 Tasks

Rules:

[P3.1] All functions that represent tasks (a.k.a., threads) shall be given names ending with "_task" (or "_thread").

Example:

```
Void alarm_task (void * p_data)
{
     alarm_t alarm = ALARM_NONE;
     int8_t  err   = OS_NO_ERR;
     while (1)
     {
          alarm = OSMboxPend(alarm_mbox, &err);
          // Process alarm here.
     }
}
```

Reasoning: Each task in a real-time operating system (RTOS) is like a mini-main(), typically running forever in an infinite loop. It is valuable to easily identify these important functions during code reviews and debugging sessions.

Exceptions: Alternatively, "_thread" may be used.

Enforcement: This naming convention shall be enforced during the detailed design phase and in code reviews.

2.5.4 Interrupt Service Routines

Rules:

[P4.2] Interrupt service routines (ISRs) are not ordinary functions. The compiler must be informed that the function is an ISR by way of a #pragma or compiler specific keyword, such as "__interrupt".

[P4.3] All functions that implement ISRs shall be given names ending with "_isr".

[P4.4] To ensure that ISRs are not inadvertently called from other parts of the software (they may corrupt the CPU and call stack if this happens), each ISR function shall lack a prototype, be declared static, and be located at the end of the associated driver module.[16]

[P4.5] A stub or default ISR shall be installed in the vector table at the location of all unexpected or otherwise unhandled interrupt sources. Each such stub could attempt to disable future interrupts of the same type, say at the interrupt controller, and assert().[17]

Reasoning: An ISR is an extension of the hardware. By definition, it and the straight-line code are asynchronous to each other. If they share global data, that data must be protected with interrupt disables in the straight-line code. The ISR must not get hung up inside the operating system or waiting for a variable or register to change value.

Exceptions: None.

Enforcement: These rules shall be enforced during code reviews.

[16] Be forewarned that a smart static analysis tool, such as lint, will likely complain about this unreachability.

[17] Although this doesn't prevent any bugs, it sure does help find bugs in the hardware and makes the system more robust.

2.5.5 Function Pointers and Callback Functions

Rules:

[P5.1] All functions that implement callbacks shall be given names ending with "Cb". If the function pointer type is for an ISR then the naming rule of both are applied giving an end of "Cb_Isr".

[P5.2] All callback functions shall have a typedef and include the Cb naming convention in the typedefinition so that they can be identified. The typedefining of callback functions makes their use much easier to use the code clean by exposing this type in the header for the API.

[P5.3] Function pointers shall contained the pointer prefix p as noted in variable declarations even when the function pointer type is used and there is no explicit use of the * notation in the declaration. This is to maintain clarity that the item is in fact a pointer.

Reasoning: Explicitly identifying callback functions helps in the readability and intention of the program and its use.

Exceptions: None.

Enforcement: These rules shall be enforced during code reviews.

2.6 Use of Module Files (Source Files)

2.6.1 Header Files

Rules:

[M1.1] Each header file shall contain a preprocessor guard against multiple inclusion, as shown in the example below. This pre-processor guard shall start with an underscore and represent the file name in UPPERCASE format. Pre-Processor

[M1.2] The header file shall identify only the procedures, constants, and data types (via prototypes or macros, #define, and struct/union/enum typedefs, respectively) about which it is strictly necessary for other modules to know about.

i. It is recommended that no variable be defined (via extern) in a header file.
ii. No storage for any variable shall be declared in a header file.

[M1.3] No header file shall contain a #include statement.

Example:

```
#ifndef _ADC_H
#define _ADC_H
...
#endif /* _ADC_H */[18]
```

Reasoning: The C language standard gives all variables and functions global scope by default. The downside of this is unnecessary (and dangerous) coupling between modules. To reduce inter-module coupling, keep as many procedures, constants, data types, and variables as possible hidden within a module's source file.

Exceptions:
Under certain circumstances it may be necessary to share a global variable across tasks within a pre-emptive OS environment. Whenever this is done, such a variable shall be named with the module's prefix, declared volatile, and always protected from race conditions at each location of access.

Enforcement: These header file rules shall be enforced during code reviews.

2.6.2 Source Files

Rules:

[M2.1] Each source file shall include only the behaviours appropriate to control one "entity". Examples of entities include encapsulated data types, active objects, peripheral drivers (e.g., for a UART), and communication protocols or layers (e.g., ARP).

[M2.2] Each source file format shall match the Delta-Q standard Templates for C and H files.

[M2.3] Each source file shall always #include the header file of the same name (e.g., file adc.c should #include "adc.h"), to allow the compiler to confirm that each public function and its prototype match.

[M2.4] Absolute paths shall not be used in include file names.

[18] See [C1.3] for rules regarding pre-processor comments.

[M2.5] Each source file shall be free of unused include files.

[M2.6] No source file shall #include another source file.

Reasoning: The purpose and internal layout of a source file module should be clear to all who maintain it. For example, the public functions are generally of most interest and thus appear ahead of the private functions they call. Of critical importance is that every function declaration be matched by the compiler against its prototype.

Exceptions: None.

Enforcement: Prior to each build, an automated tool shall scan source files to ensure they #include their own header file but not unused header files. Lint is an example of a tool that can be configured to perform the second check automatically.

2.6.3 File Templates

Rules:

[M3.1] A set of templates for header files and source files shall be maintained at the department level. See Surround software branch /software/projects/templates repository for latest templates..

Reasoning: Starting each new file from a template ensure consistency in file header comment blocks and ensure inclusion of appropriate copyright notices.

Exceptions: None.

Enforcement: The consistency of comment block formats shall be enforced during code reviews.

2.7 Comments

2.7.1 Acceptable Formats

Rules:

[C1.1] Single-line comments in the C++ style (i.e., preceded by //) are a useful and acceptable alternative to traditional C style comments (i.e., /* … */).[19]

[C1.2] Comments shall never be nested.

[C1.3] Comments for conditional include pre-processor statement shall clearly identify the scope and hierarchy of the statement. For example

```
#if (X)
....some code...
#else /* X */
....some more  code...
#endif /* X */
```

[C1.4] Comments shall never be used to disable a block of code, even temporarily.

i. To temporarily disable a block of code, use the preprocessor's conditional compilation feature (e.g., #if 0 … #endif /* Comment *\). No block of temporarily disabled code shall remain in the source code of a release candidate.

ii. Any line or block of code that exists specifically to increase the level of debugging output information shall be surrounded by #ifndef NDEBUG … #endif.[20] In this way, useful debug code may be maintained in production code, as the ability to gather additional information is often desirable long after development is done.

Reasoning: Nested comments and commented-out code both run the risk of allowing unexpected snippets of code to be compiled into the final executable. This can happen, for example, in the case of sequences such as /* code-out /* comment */ code-in */.

Exceptions: None.

---

[19] This is a deviation from [MISRA04] Rule 2.2, which we feel will not affect the number or severity of firmware bugs. The C++ "single-line" style makes comments easier to align and maintain. In addition this deviation is consistent with our choice of the [C99] language, which officially added single-line comments to the C language.

[20] Our choice of negative-logic NDEBUG is deliberate, as that constant is associated with disabling the assert() macro. In both cases, the programmer acts to disable the verbose code. It's also good to have just one of these #defines to keep track of.

Enforcement: The use of only acceptable comment formats can be only partially enforced by the compiler or static analysis. The avoidance of commented-out code, for example, must be enforced during code reviews.

2.7.2 Location and Content

Rules:

[C2.1] All comments shall be written in clear and complete sentences, with proper spelling and grammar and appropriate punctuation.

[C2.2] The most useful comments generally precede a block of code that performs one step of a larger algorithm.[21] A blank line shall follow each such code block. The comments in front of the block should be at the same indentation level.

[C2.3] Avoid explaining the obvious. Assume the reader knows the C programming language. For example, end-of-line comments should only be used in exceptional circumstances, where the meaning of that one line of code may be unclear from the variable and function names and operations alone but where a short comment makes it clear. Avoid writing unhelpful and redundant comments such as "shift left 2 bits".

[C2.4] The number and length of individual comment blocks shall be proportional to the complexity of the code they describe.

[C2.5] Whenever an algorithm or technical detail has come from a published source, the comment shall include a sufficient reference to the original source (via book title, website URL, or other details) to allow a reader of the code to find the cited reference material.

[C2.6] Whenever a flow-chart or other diagram is needed to sufficiently document the code, the drawing shall be maintained with the source code under version control and the comments should reference the diagram by file name or title.

[C2.7] All assumptions shall be spelled out in comments.[22]

[C2.8] Each module and function shall be commented in a manner suitable for automatic documentation generation via Doxygen (www.doxygen.org). For the fields to use please see the template C and H files.

---

[21] It is a good practice to write the comment blocks first, as you should not begin the coding until you can explain the logic, algorithm, or sequence of steps in words.

[22] Of course, a set of design-by-contract tests or assertions is even better than comments.

Example:

```
// Step 1: Batten down the hatches.
for (int8_t hatch = 0; hatch < NUM_HATCHES; hatch++)
{
    if (hatch_is_open(hatches[hatch])
    {
       hatch_close(hatches[hatch]);
    }
}
// Step 2: Raise the mizzenmast.
// TODO: Define mizzenmast driver API.
```

Reasoning: Following these rules results in good comments. And good comments result in good code. Unfortunately, it is easy for source code and documentation to drift over time. The best way to prevent this is to keep the documentation as close to the code as possible. Doxygen is a useful tool to regenerate documentation describing the modules, functions, and parameters of a project as that code is changed.

Exceptions: Individual projects may standardize the use of Doxygen features of beyond those in the template files.

Enforcement: The quality of comments shall be evaluated during code reviews. Code reviewers should be on the lookout both that the comments accurately describe the code and that they are clear, concise, and valuable. Rebuilds of Doxygen-generated documentation files, for example in HTML or PDF, shall be automated and made part of the software build process.

2.8 Expressions and Statements

2.8.1 Casts

Rules:

[E1.1] Each cast shall feature an associated comment describing how the code ensures proper behavior across the range of possible values on the right side.

Example:

```
int abs (int arg)
{
   return ((arg < 0) ? -arg : arg);
}
```

```
unsigned int y;
y = adc_read();
z = abs((int) y); // A risky cast.
```

Reasoning: Casting is dangerous. In the example above, unsigned y can take a larger range of positive values than a signed integer. In that case, the absolute value will be incorrect as well. The above cast was likely used to quiet an important warning about possible loss of precision.

Exceptions: None.

Enforcement: These rules shall be enforced during code reviews.

2.8.2 Equivalence Tests

Rules:

[E2.1] When evaluating the equality or inequality of a variable with a constant value, always place the constant value on the left side of the comparison operator, as shown in the *if-else* example below.

Example:

```
If (MOD_CONSTANT < integer_variable)
{
    ....some code....
}
```

Reasoning: It is always desirable to detect possible typos and as many other bugs as possible at compile-time; runtime discovery may be dangerous to the user of the product and require significant effort to localize. By following this rule, the compiler can detect erroneous attempts to assign (i.e., = instead of ==) a new value to a constant.

Exceptions: None.

Enforcement: A static analysis tool shall be configured to raise an error or warning about all assignment statements where comparisons are ordinarily expected.

2.8.3 If-Else Statements

Rules:

[E3.1] The shortest (measured in lines of code) of the if and else if clauses should be placed first.[23]

[E3.2] Nested if-else statements shall not be deeper than two levels. Use function calls or switch statements to reduce complexity and aid understanding.

[E3.3] Assignments shall not be made within an if or else if expression.

[E3.4] Any if statement with an else if clause shall end with an else clause.[24]

Example:

```
if (NULL == p_object)
{
    result = ERR_NULL_PTR;
}
else if (p_object = malloc(sizeof(object_t))) // No!
{
    ...
}
else
{
    // Normal processing steps,
    // which require many lines of code.
    ...
}
```

Reasoning: Long clauses can distract the human eye from the decision-path logic. By putting the shorter clause earlier, the decision path becomes easier to follow. (And easier to follow is always good for reducing bugs.) Deeply nested *if* blocks are a sure sign of a complex and fragile state machine implementation; there is always a safer and more readable way to do the same thing.

Exceptions: For efficiency purposes, it may be desirable to reorder the sequence of *if-else* clauses to ensure the most frequent or most critical case is always found the fastest. Of course, *if-else* statements are typically not as efficient as tables of function pointers in terms of worst-case analysis.

---

[23] Thanks to [Holub] for putting this "formatting for readability" idea into words.

[24] This is the equivalent of requiring a default case in every switch, as we do below.

Enforcement: These rules shall be enforced during code reviews, when reviewers feel it may aid readability.

2.8.4 Switch Statements

Rules:

[E4.1] The break for each case shall be indented to align with the associated case, rather than with the contents of the case code block.

[E4.2] All switch statements shall contain a default block.

[E4.3] There shall be a blank line between the break statements of a block and the next case.

Example:

```
switch (err)
{
      case ERR_A:
            ...
      break;

      case ERR_B:
            ...
      break;

      default:
            ...
      break;
}
```

Reasoning: Switch statements are powerful, but prone to errors such as missed break statements and unhandled cases. By aligning the *case* and *break* keywords it is possible to spot missing *breaks*.

Exceptions: None.

Enforcement: These rules can be enforced by an automated scan of all new or modified code during each build. Alternatively, they shall be enforced in code reviews.

2.8.5 Loops

Rules:

[E5.1] Magic numbers shall not be used as the initial value or in the endpoint test of a while or for loop.[25]

[E5.2] Except for a single loop counter initialization in the first clause of a for statement, assignments shall not be made in any loop's controlling expression.

[E5.3] Infinite loops shall be implemented via the controlling expression "while (1)".

[E5.4] Each loop with an empty body shall feature a set of braces enclosing a comment to explain why nothing needs to be done until after the loop terminates.

Example:

```
// Don't use a magic number ...
for (int8_t row = 0; row < 100; row++)
{
    // ... when you mean a constant.
    for (int8_t col = 0; col < MAX_COL; col++)
    {
            ...
    }
}
```

Reasoning: It is always important to synchronize the number of loop iterations to the size of the underlying data structure. Doing this through a single constant prevents a whole class of bugs that can result when changes in one part of the code, such as the dimension of an array, are not matched by changes in other areas of the code, such as a loop iterator that operates on the array. The use of named constants also makes the code easier to read and maintain.

Exceptions: It is acceptable to start or end a loop with an integer value of 0 (considered less magical by most), such as when iterating from or toward the base of an array.

Enforcement: These rules shall be enforced during code reviews.

---

[25] Note that sizeof() is a theoretically handy way to dimension an array but that this method does not work when you pass a pointer to the array instead of the array itself. Thus the most portable method is a constant shared between the array declaration and the loop. (Google Mike Barr and N_ELEMENT macro for how this works.)

2.8.6 Unconditional Jumps

Rules:

[E6.1] As stated earlier, the keywords *goto*, *continue*, and *break* shall not be used to create unconditional jumps.

Reasoning: Algorithms that utilize unconditional jumps to move the instruction pointer can be rewritten in a manner that is more readable and thus easier to maintain.

Exceptions: None.

Enforcement: These rules shall be enforced by an automated scan of all modified or new modules for inappropriate use of these tokens.

2.9 MISRA 2004

The 121 ***required*** rules of the Motor Industry Software Reliability Association 2004 C language subset shall be followed in all embedded application software except the following:

Rule 2.2 Source code shall only use /* ...*/ style comments

This is not required as the compliers we are using going forward all support the C99 standard and it is felt that this commenting style is easier and more efficient to use.

Rule 5.7 No identifier name should be re-used (Partial non compliance)

This rule may be broken for loop counters only. Which do not offer much value in having different names, but should still be descriptive especially where there are loops within loops.

Rule 11.3 A cast should not be performed between a pointer type and an integral type. (Partial non compliance)

This rule may be broken when addressing memory mapped registers or other HW specific features.

3. Coding Style Quick Reference

tabs	Shall be set to 4 spaces. Tabs should not be used ever.
braces	Shall appear on a line by themselves. Opening and closing braces shall be aligned. Braces shall be mandatory for all for, while, do, if, and structure declaration statements.
indents	Statements within braces shall be indented one tab stop to the right of the braces.
line length	Shall not exceed 90 characters. Code statements which exceed 80 characters should either be re-written into two statements, or else the single statement should be continued onto the next line with an indentation of one tab stop from the start of the statement.
blank lines	Shall be put after a function header block, before a block comment, a case block, and any logical group of statements.
comments	Shall be used liberally, but not blindly. Comment all variables, constants, and #defines. Comments documenting sections of code should be at the same indentation level as its code. Use Doxygen comments
variables aligned	Wherever possible, within reason, for groups of variable declarations or definitions.
spaces	Shall be used after commas, before and after arithmetic and Boolean operators. No spaces before opening parentheses for functions. No spaces before semicolons. Spaces shall be inserted between nested parenthesis.
parentheses	Shall be explicit for arithmetic and Boolean operations.
extern	Shall precede every variable and function (except in-line functions) with the XXX_ prefix in the header file.
static	Shall precede file scope variables and functions with the lowercase xxx_ prefix.
const	Shall be used wherever possible.
volatile	Shall be used for all variables which are hardware modifiable registers, or shared globals used by different tasks where mutexes are not appropriate.
???	Shall be placed within comments describing sections of code not yet implemented.
return value	Shall be documented in the appropriate section of the function header.
XXX_	All exported functions, variables, constants, and #defines shall be prefixed by an uppercase three letter module mnemonic and an underscore.
#defines	Shall be all capitals separated by underscores.
function names	The first letter after the module mnemonic shall be uppercase.
variable names	The first letter, or the first letter after the module mnemonic shall be lowercase, otherwise capitalised case should be used.
type names	The first letter after the module mnemonic shall be uppercase.
enum names	Shall be in mixed case as in any other name with a “e” prefix.
C file names	Lowercase and shall start with the module mnemonic. If the file consists of a single C file, the name of the file is the module mnemonic.
header file names	Lowercase and Shall correspond to that of the C file name, except for the extension. A module shall have only one public header file that needs to be included by other modules. The public header file name shall be the module mnemonic.
templates	Shall be used for all function and file headers. Template sections which are not used shall not be deleted.