C Firmware Development Standard

|  |  |  |  |
| --- | --- | --- | --- |
| Approvals | |  |  |
|  |  | Date | Signature |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

|  |  |
| --- | --- |
| Document Information | |
| Info | Content |
| Author(s) |  |
| Revision | 1.0 |
| Document Status | Draft |
| Date |  |
| Distribution | [distribution] |
| VPM Number | 723611 |
| Keywords | Coding style, Coding standard |
|  | |

Revision History

| **Rev.** | **Date** | **Author** | **Description** |
| --- | --- | --- | --- |
| 1 | 17-Jun-21 |  | Initial revision |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table of Contents

1 Introduction 4

1.1 Purpose 4

1.2 Scope 4

1.3 Terms, Acronyms and Abbreviations 4

1.4 References 4

2 Files 5

2.1 Naming Convention 5

2.2 File layout 5

2.3 Template files 5

2.3.1 Header files 5

2.3.2 Source files 6

2.4 Tagging 6

2.4.1 SVN 6

2.4.2 Doxygen 6

2.4.3 Module Header Tagging 6

2.4.4 File Header Tagging 6

2.4.5 Function Header Tagging 7

2.4.6 Variable Tagging 8

2.4.7 Structure and Enumeration Tagging 8

3 MISRA Compliance 9

4 General Rules 9

4.1 Build system requirement 10

4.2 RTOS 10

4.3 Code reviews 10

4.4 Coding Conventions 10

4.4.1 Functions 10

4.4.2 Variable Types 11

4.4.2.1 Fixed size 11

4.4.2.2 Floating point 11

4.4.2.3 Forbidden variable modifiers 12

4.4.3 Pointers 12

4.5 Naming Conventions 12

4.5.1 Variable Names 12

4.5.1.1 Systems Hungarian Notation 13

4.5.1.2 Type definitions 14

4.5.1.3 Bit fields 14

4.5.1.4 Defines and constants 14

4.5.2 Function names 15

4.5.3 Function like macros 16

4.5.4 Interrupt Service Routines (ISR) 16

4.5.5 Operator precedence 16

4.6 Braces 16

4.7 Comments 16

4.8 Conditional compilation 17

4.9 Spacing 17

4.10 Conditional statements 17

4.10.1 The IF statement 18

4.10.2 The SWITCH statement 18

4.10.3 Loops 19

4.10.4 The FOR loop 19

4.10.5 The WHILE loop 19

4.10.6 The DO WHILE loop 20

4.10.7 Infinite loop 20

4.10.8 GOTO statement 20

# Introduction

This introduction provides an overview of the coding standard for firmware development. It includes the purpose, scope, terms, acronyms, abbreviations, and references.

## Purpose

The purpose of this document is to give guidance to firmware developers and reviewers to allow them to create code of high quality that is easily readable, understandable and maintainable.

## Scope

.

## Terms, Acronyms and Abbreviations

Below are the terms, acronyms, and abbreviations used within this document. Additional project-specific terms can be found in the project glossary [GLOS].

|  |  |
| --- | --- |
| **Term, Acronym, Abbreviation** | **Definition** |
| API | Application Programming Interface |
| COTS | Commercial-Off-The-Shelf |
| MISRA C | MISRA C is set of software development guidelines for the C programming language developed by MISRA (Motor Industry Software Reliability Association). |
| SAD | Software Architecture Description |
| SysAD | System Architecture Description |
| UML | Unified Modeling Language |
|  |  |

## References

| **Abbreviation** | **Reference Element** | **Description** |
| --- | --- | --- |
| [GLOS] | Project Glossary |  |
| [PRS] | Product Requirements Specification |  |
| [UCS] | Use Case Specification |  |
| [VIS] | Vision |  |
| [SysAD] | System Architecture Description |  |
| [IEC 61508] | IEC 61508-x: 2010 Functional safety of electrical/electronic/programmable electronic safety-related systems. |  |
| [IEEE1471] | ANSI/IEEE-1471-2000, IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, September 2000. |  |
| [Bass03] | L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*, 2nd edition, Addison Wesley Longman, 2003. |  |

# Files

This section specify rules for C-code source and header files design. Naming of files is described and also structure inside file – header, function, variable, etc. tagging with doxygen instructions.

## Naming Convention - Mandatory

1. Files containing only C, or C specific code shall be named “filename.c” for source files and “filename.h” for header files – name of c and h file is same for module and should describe function.
2. Long filenames are acceptable with the exception that no spaces shall be used (use underscore instead).
3. Filenames shall only contain lowercase letters and underscores.
4. All source files shall be located under a single source directory. The directory may contain subdirectories.
5. The name of the source and header files should be same for the given module.
6. Files from 3rd party source (vendor of chip, drivers, rtos, etc.) should be in separate directories and don’t have to follow our naming convention.

## File layout - Mandatory

File layout should be uniform across all source code: all projects and all developers. This ensures readability during code reviews and maintenance.

## Template files - Mandatory

There are template files, “**template.c**” and “**template.h**” for new \*.c and \*.h source files in the project template repository. Designers should start with copies of these template files and add contents as necessary.

Header files - Mandatory

1. MUST include a version control and copyright header.
2. MUST be guarded from duplicate inclusion by testing for the definition of a value.
3. MUST include the comment blocks, even if the block is empty.
4. Constants, variables, functions, etc. are grouped by functions blocks (where are used).

## Source files - Mandatory

1. MUST include a version control and copyright header.
2. MUST include the comment blocks, even if the block is empty.
3. Every module should have own header file (more header files for shared data, functions, etc.).
4. First include has to be “platform\_stddefs.h” (if it is possible) – standard types header file.
5. Constants, variables, functions, etc. are grouped by functions blocks (where are used).

## Tagging - Mandatory

## SVN

The following file tags shall be present in each source file. The examples shown below are for Tortoise SVN, but may be adapted for whichever source control software is used for the project.

|  |  |
| --- | --- |
| SVN Tag | Description |
| $LastChangedBy$ | Name of the last person to edit this file. |
| $LastChangedRevision$ | The latest revision number of this file. |
| $LastChangedDate$ | The date the last change was made to this file |

## Doxygen

Doxygen is a software package used to document source code. It does this by means of certain tags placed in the code, usually in file headers, function headers and as part of variable and structure definitions. Only the basic tags will be described below, but many more can be found in the Doxygen documentation.

## Module Header Tagging

The following tags should be placed in the module header. This provides deail description about module.

|  |  |  |
| --- | --- | --- |
| Tag | Description | Example |
| @defgroup | Name of the module, source file. | @defgroup uart Uart |
| @ingroup | Name of the module, header file | @ingroup uart |

## File Header Tagging

The following tags should be placed in the file header:

|  |  |  |
| --- | --- | --- |
| Tag | Description | Example |
| @file | Name of the file being documented. | @file uart.c |
| @Copyright | Copyright statement. | @Copyright Copyright 2021 by Affluence Info Systems Pvt Ltd. |
| @brief | Description of file | @brief ADC module |
| @details | A detailed description of the file. | @details This file contains all the functions associated with the setup and reading of ADC values. |

## Function Header Tagging

The following tags should be placed in the function header:

|  |  |  |
| --- | --- | --- |
| Tag | Description | Example |
| @brief | Brief one line function description | @brief Reads the next ADC value from the specified channel |
| @details | A more detailed description of the function. | @details Starts an ADC read of the requested channel, waits for the conversion to complete and then reads the result. This result is then averaged and returned as a UINT32. |
| @pre | Any pre conditions that must be satisfied. | @pre The init\_adc() function must have been called first. |
| @post | Any post conditions including changes to global states or parameters. | @post last\_adc\_reading will be updated with the last reading. |
| @param | Describes function parameters. | @param channel\_number Channel number to be read. |
| @return | Describes return value of function. | @return Latest ADC value read from requested channel. |

1. A function comment header must be included in front of each function definition (one time in source file) according to the following Doxygen format (where are **mandatory** @fn, @brief, @params, @return if exist. Other items compulsory.)

/\*\* @fn void function(void)

\* @brief A brief one line description (do not exceed 1 line or sentence).

\*

\*

\* @details Purpose of the function. May be excluded if the brief statement is

\* sufficient.

\*

\* @pre Any pre conditions that must be satisfied

\* @post Any post conditions including changes to global states or parameters.

\* @param c1 description (input, include pre and post conditions)

\* @param c2 description (input, include pre and post conditions)

\* @return description (output, include pre and post conditions)

\*/

## Variable Tagging

All global (file scope if necessary) variables should be described using a Doxygen tag like is shown in examples below.

u32 adc\_u32\_val\_read; /\*\*< Example of Doxygen comment for variable in one line \*/

or

/\*\* Example of Doxygen comment for variable with more than one lines (array) \*/ static u32 crc32\_pu32\_table[] =

{

0x48u, 0x96u,

0x19u, 0x48u

}

## Structure and Enumeration Tagging

1. All defined enumerations, structures and their elements should be described using Doxygen tags **\*<** and brief.
2. All enumerations and structures are defined **lowercase** with **underscore**.
3. In case the structure or enumeration is GLOBAL then use prefix “**nameofmodule\_**” like shown below in the example.
4. Enumerations should have prefix “**e\_**”.
5. Structures should have prefix “**st\_**”.

/\*\* @brief Use brief, otherwise the index won't have a brief explanation.

\*

\* Detailed explanation.

\*/

typedef enum

{

ERROR\_MINOR, /\*\*< Comment for enum in doxygen \*/

ERROR\_MAJOR, /\*\*< Comment for enum in doxygen \*/

ERROR\_FATAL /\*\*< Comment for enum in doxygen \*/

} e\_error\_code; /\* this is for using as LOCAL in module \*/

e\_error\_code e\_error\_code\_my\_error1 = ERROR\_MINOR /\* example of INSTANCE \*/

/\*\* @brief Use brief, otherwise the index won't have a brief explanation.

\*

\* Detailed explanation.

\*/

typedef struct

{

u8 u8\_channel\_number; /\*\*< Comment for struct element in doxygen \*/

u32 u32\_last\_reading; /\*\*< Comment for struct element in doxygen \*/

u32 u32\_avg\_reading; /\*\*< Comment for struct element in doxygen \*/

} nameofmodule\_st\_control\_rec; /\* this is for using as GLOBAL in module \*/

# MISRA Compliance - Mandatory

All C code (safety and non-safety) shall follow the MISRA C:2012 rules, both the required and advisory rules.

The static analysis tool Klocwork should be used to check source code for compliance to the MISRA C coding standard and to perform static analysis.

**MISRA rules should take precedence in case of conflict between our guidelines and MISRA.**

Project specific documentation shall document the following:

1. Any deviation from the rules shall be documented in the source code. An example is shown below.
2. All violations of MISRA rules shall be agreed by the system architect/design authority.

/\* MISRA VIOLATION: <Rule x.y>:<Brief justification> \*/

# General Rules

1. All code shall conform to **ISO 9899:1990** Programming languages – C, amended and corrected by **ISO/IEC 9899/COR1: 1995**, ISO/IEC 9899/AMD1:1995\*\*, and **ISO/IEC 9899/COR2: 1996**. This is often referred to as [C90].
2. Compiler extensions should be avoided, if possible. Any use of the extensions should be documented and explained.
3. All use of the #pragma directive shall be documented and explained.
4. Lines lengths should not exceed 130 characters.
5. Indentation of code should consist of 4 spaces without Tabs.
6. All source code should place only one C or assembly statement per line.
7. Assembly language shall be encapsulated and isolated (no inline assembly).
8. No storage declaration in header (.h) files should be present.
9. Limit variable scope as much as possible, e.g. variables used within one file / module should be declared as "static" at the top of that module / file. Variables used within one function should be declared at the start of that function (with static modifier if data retention between executions is required).
10. Variables modified by interrupt routines shall be declared as ‘volatile’ - ThreadX variables can't be volatile (processed internally).
11. Do not mix Signed and Unsigned variables in conditional statements or math equations.
12. The code should compile with zero Errors and zero Warnings.
13. Complexity of an individual function shall not exceed a McCabe Complexity Score of 10, and by exception not exceed 25.
14. Size of a module / file shall not exceed 2000 lines. Where this figure is exceeded the module content should be reviewed to see if it all belongs together.
15. Functions shall implement checks on all passed arguments to check for void or unexpected values (for instance a NULL pointer).
16. All source files must end with a new-line character.
17. Wide strings will not be used in the project.
18. The #line directive shall not be used in the project.
19. All results from arithmetic functions shall be checked to have a correct range or an error value before being used. Error values shall be trapped and handled.
20. It is forbidden to use undefined magical numbers in functions, conditions, etc. All values statements should be defined by macros. Exception can create easy number statement for example bit shift one byte but after that is necessary add comment with explanation (see below the two examples).

u32\_number\_out = u32\_number\_in >> 8u; /\* here should be briefly explained the 8u statement \*/

or

/\* you can put this macro in front of command for better clarification \*/

#define BYTE\_SHIFT (8u) /\*\*< 8 bits shifting \*/

u32\_result = (u32\_value1 >> BYTE\_SHIFT) & (u32\_value2 >> BYTE\_SHIFT);

## Build system requirement - Mandatory

1. All tools are kept under version control.
2. All build defines, compiler switches, linker settings, etc. has to be documented – external file
3. Project workspace file should not be dependent on absolute paths.
4. All compiler and RTOS files must be placed under version control.

## RTOS - Mandatory

## Code reviews - Mandatory

## Coding Conventions

## Functions - Mandatory

1. Shall not be recursive (functions may not call themselves directly or indirectly).
2. Shall not be defined with a variable number of arguments (‘…’ notation may not be used).
3. Identifiers shall be given for all of the parameters in a function prototype declaration.
4. Functions limited to file scope shall be declared as ‘**static’**.
5. Not limited to file scope shall be declared as ‘extern’ and must be prefixed by the file name where they are implemented.
6. Should not exceed 200 lines in length.
7. Shall only have a single exit point and it shall be at the end of the function.
8. Returning error information shall have that information tested.
9. Const keyword should be used if function arguments are not modified inside function.
10. A function comment header must be included in front of each function definition. See 2.4.5 for more details.
11. Should be checked that the passed arguments to the function are of the expected size before using them.

extern void uart\_write\_byte(u8 u8\_data); /\* this function is implemented in uart module \*/

static void read\_next\_byte(u8 u8\_data); /\* LOCAL function \*/

## Variable Types - Mandatory

To aid in code portability, defined types shall be used instead of the built-in C or C++ types. These definitions may be changed quickly between platforms, ensuring type compatibility without code modification. The standard types used are described in the following subsections.

## Fixed size

Use only these fixed types. Exceptions are allowed only for the abstraction layer between our code and third party code, RTOS, etc.

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Size | Sign | Use |
| tbool | 8 bits | Unsigned | Boolean values |
| ch | 8 bits | Unsigned | Storage and use of character values |
| u8 | 8 bits | Unsigned | Unsigned numeric values – max 8 bits |
| s8 | 8 bits | Signed | Signed numeric values – max 8 bits |
| u16 | 16 bits | Unsigned | Unsigned numeric values – max 16 bits |
| s16 | 16 bits | Signed | Signed numeric values – max 16 bits |
| u32 | 32 bits | Unsigned | Unsigned numeric values – max 32 bits |
| s32 | 32 bits | Signed | Signed numeric values – max 32 bits |
| u64 | 64 bits | Unsigned | Unsigned numeric values – max 64 bits |
| s64 | 64 bits | Signed | Signed numeric values – max 64 bits |
| f32 | 32 bits | Signed | Floating point values – max 32 bits |
| f64 | 64 bits | Signed | Floating point values – max 64 bits |
| bitfield | 32 bits | Unsigned | Bitfields – max 32 bits |

## Floating point

The use of floating point variables and floating point arithmetic should be avoided if it is possible. Fixed point math should be used instead. If this is not feasible, the following rules should be followed:

1. Use the defined types shown above i.e. f32 and f64.
2. Assign an ‘f’ to all single precisions constants.
3. If you require double precision math, ensure it is supported by the compiler.
4. Never test for equality or inequality of floating point values.

#define NAMEOFMODULE\_UPPER\_LIMIT (10.245f) /\* GLOBAL in module \*/

#define UPPER\_LIMIT (10.245f) /\* LOCAL in module \*/

f32 f32\_current\_limit = 2.1023f; /\* LOCAL in module \*/

## Forbidden variable modifiers

The following variable modifiers are historical and shall not be used:

1. Auto
2. Register

## Pointers - Good Practice

1. Use pointers sparingly.
2. Subtraction between pointers shall only be applied to pointers that address elements of the same array.
3. The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object.
4. The +, -, += and -= operators should not be applied to an expression of pointer type.
5. Pointer abstraction should be limited. For instance, pointer to pointer may be required in rare circumstances but pointer to pointer to pointer (or worse!) should not be required or used.
6. Casting of pointers from one type to another should be avoided.

## Naming Conventions - Mandatory

## Variable Names

1. May not be a keyword of C/C++, or part of the C standard library.
2. May not start with an underscore.
3. Must be longer than 3 characters (not counting Hungarian notation), but not longer than 31.
4. Should be descriptive of its purpose.
5. Variable names shall be written using **underscore** and **lowercase** (adc\_u8\_read).
6. Variable names shall be **prefixed** **by the module name** **if the variable is defined as** **global in the module**.
7. Variable names shall be prefixed by Hungarian notation.
8. Variable names shall be prefixed by “**e**\_**nameoftypedef\_**” for enums and “**st**\_**nameoftypedef\_**” for structs. For basic datatype use rules in point 4.5.1.1
9. It should be avoided sharing the same name for enum and struct in same module.
10. Names of variables should be unambiguous from each other. (e.g. O and 0, I and 1, I and l (el!), l(el) and 1, S and 5, Z and 2, n and h, B and 8, rn and m)

It is advised that as a minimum identifiers should not differ by any combination of:

* Lowercase / Uppercase difference only
* Presence or absence of “**\_**”

|  |  |  |
| --- | --- | --- |
| Scope | Rule | Example |
| Global in module | The file name where in which it was declared ( acronym if filename is too long) and Hungarian notation | adc\_u8\_current\_sample  adc\_st\_error\_my\_error1 (instance for structure) |
| Local in module | Hungarian notation | u8\_sample |
| SDS Enums | Systems Hungarian Notation - uppercase (see table below). | U32\_ADC\_VALUE |

/\*\* @brief Example of prefix “e\_” for enumeration

\*

\* Detail explanation…

\*/

typedef enum

{

ERROR\_MINOR, /\*\*< Comment for enum in doxygen \*/

ERROR\_MAJOR, /\*\*< Comment for enum in doxygen \*/

ERROR\_FATAL /\*\*< Comment for enum in doxygen \*/

} e\_error\_code; /\* see placement of prefix “e\_” \*/

## Systems Hungarian Notation

These prefixes identify the type of the variable, and are added in the same order as the variable would be described.

1. Table below shows prefixes for basic datatype, pointer to datatype and pointer to pointer for datatype. The prefix for arrays are the same as for pointers.

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Prefix for Type | Prefix for pointer  (or array) | Prefix for pointer to pointer |
| tbool | tbool\_ | ptbool\_ | pptbool\_ |
| ch | ch\_ | pch\_ | ppch\_ |
| u8 | u8\_ | pu8\_ | ppu8\_ |
| s8 | s8\_ | ps8\_ | pps8\_ |
| u16 | u16\_ | pu16\_ | ppu16\_ |
| s16 | s16\_ | ps16\_ | pps16\_ |
| u32 | u32\_ | pu32\_ | ppu32\_ |
| s32 | s32\_ | ps32\_ | pps32\_ |
| u64 | u64\_ | pu64\_ | ppu64\_ |
| s64 | s64\_ | ps64\_ | pps64\_ |
| f32 | f32\_ | pf32\_ | ppf32\_ |
| f64 | f64\_ | pf64\_ | ppf64\_ |

1. Table below shows prefixes for structures

|  |  |  |
| --- | --- | --- |
| typedef struct | Prefix | Example of instance |
| st\_data\_input | st\_data\_input\_ | st\_data\_input\_my\_data |
| st\_dataprepare | st\_dataprepare\_ | st\_dataprepare\_calc1 |

1. Table below shows prefixes for enumerations

|  |  |  |
| --- | --- | --- |
| typedef enum | Prefix | Example of instance |
| e\_err | e\_err\_ | e\_err\_my\_error1 |
| e\_statemachine | e\_statemachine\_ | e\_statemachine\_main |

Example of using prefix for struct and enum is shown in example below.

/\*\* @brief Example of Hungarian notation for enumeration, structure, etc.

\*

\* This is for using **LOCAL** in module

\*/

typedef enum

{

ERROR\_MINOR, /\*\*< Comment for enum in doxygen \*/

ERROR\_MAJOR, /\*\*< Comment for enum in doxygen \*/

ERROR\_FATAL /\*\*< Comment for enum in doxygen \*/

} e\_error\_code; /\* this is for using **LOCAL** in module \*/

/\* there is example of instance of “e\_error\_code” \*/  
e\_error\_code e\_error\_code\_my\_error1 = ERROR\_MINOR /\* example of INSTANCE \*/

e\_error\_code pe\_error\_code\_my\_error2[5]; /\* example of array instance \*/

/\*\* @brief Example of Hungarian notation for enumeration, structure, etc.

\*

\* This is for using **GLOBAL** in module

\*/

typedef enum

{

ERROR\_MINOR, /\*\*< Comment for enum in doxygen \*/

ERROR\_MAJOR, /\*\*< Comment for enum in doxygen \*/

ERROR\_FATAL /\*\*< Comment for enum in doxygen \*/

} **adc\_**e\_error\_code; /\* this is for using **GLOBAL** in module \*/

/\* there is example of instance of “e\_error\_code” \*/  
adc\_e\_error\_code adc\_e\_error\_code\_my\_error1 = ERROR\_MINOR /\* example of INSTANCE \*/

## Type definitions

1. All new data types, structures, unions and enumerations shall be named via a typedef.
2. The name shall be written all **lowercase** with **underscores** separating the words E.g.

typedef struct

{

u8 u8\_temp1;

u8 u8\_temp2;

} st\_temp\_struct;

st\_temp\_struct st\_temp\_struct\_my\_instance1; /\* instance of structure \*/

## Bit fields

1. Bitfield is defined in struct with **bitfield** types (regard to 4.4.2.1 declare unsigned int – 32bits) as shown below.
2. Struct has to be defined only for bit fields, do not combine it with other data.
3. Variable bit field should be prefixed by “**bit\_**”.

struct bit\_field

{

bitfield temp1: 3;

bitfield temp2: 1;

} bit\_data;

## Defines and constants

1. Where define is used as a constant value, its name shall be written all **uppercase** with **underscores** separating the words. E.g.

#define MAX\_COUNT\_VALUE (100u)

1. Where define has external scope, its name shall be prefixed in such a way so as to identify the module to which it belongs. E.g.

#define NAMEOFMODULE\_MAX\_COUNT\_VALUE (100u)

#define NAMEOFMODULE\_NUM\_CHANNELS (10u)

Exception from this rule is only system level defines in preprocessor.h file.

1. Octal constants shall not be used.
2. Escape sequences shall not be used, with following exceptions:
   1. Formfeed (\f)
   2. Newline (\n)
   3. Carriage return (\r)
   4. Backslash (\\)
   5. Single quotation mark (\')
   6. Double quotation mark (\")
   7. Byte interpreted as a hexadecimal number (\xhh, where hh is hexadecimal number) – only one byte per one escape sequence is allowed.
3. Constant values shall be declared with a type specified according to the following table:

|  |  |  |
| --- | --- | --- |
| Type | specified | example |
| Signed integer | N/A | (-100) |
| Unsigned integer | u | (100u) |
| Signed long integer | L | (-70000L) |
| Unsigned long integer | uL | (70000uL) |
| Floating point value | f | (147.56f) |

Note: The uppercase ‘L’ is used in place of a lower case ‘l’ to avoid confusion with ‘1’.

1. All definitions should be encased by parentheses “()”.

## Function names

1. May not be a keyword of C/C++, or part of the C standard library.
2. May not start with an underscore.
3. Must be longer than 3 characters (not counting Hungarian notation), but not longer than 31.
4. Should be descriptive of its purpose.
5. Function names shall be written using **underscore** and **lowercase**
6. If the function is global, it must be **prefixed by the file name where it is implemented**, followed by underscore function name. E.g.:

static void read\_next\_byte(u8 u8\_data); /\* local function in uart.c \*/

extern void uart\_write\_byte(u8 u8\_data); /\* global function in uart.h \*/

## Function like macros

Function like macros should be avoided if an inline function could be used instead.

Replace:

#define MAX(a,b) ((a) > (b) ? (a) : (b))

With:

inline u32 max(u32 u32\_num1, u32 u32\_num2);

If function like macros are used, the following rules apply:

1. The entire macro body must be enclosed in braces.
2. Each parameter must be enclosed in braces.
3. Each argument should be used no more than once to avoid unintended side effects.

## Interrupt Service Routines (ISR)

1. Should be kept as short as possible. ISR must not take more time to execute than 50% of the shortest interval between one same interrupts.
2. Shall NOT use floating point math.
3. All interrupts could be globally disabled only in special cases, it has to be documented and approved by the system architect.
4. Shall not call a function that can also be called in the foreground task unless it has been designed, proved and tested to be re-entrant safe.
5. Any variables modified in an interrupt routine that are also used in the foreground should be declared as volatile.

## Operator precedence

Code shall be written in such a way that order of execution is obvious. Where order of execution can be specified using parenthesis, they should be used.

Avoid implicit or obscure features of the language.

## Braces - Mandatory

1. All control statements must be in braces.
2. All braces shall be aligned horizontally with the preceding construct
3. Opening brace on the same line as the defining construct is NOT acceptable.

if (u8\_cntr < u8\_max)

{

/\* shown vertically aligned braces \*/

}

## Comments - Mandatory

1. Only C style comments are allowed. e.g.

/\* This style is OK \*/

1. C++ style comments are NOT allowed. e.g.

// this style is not allowed.

1. Nested comments are not allowed. e.g.

/\* This is /\*nested comment\*/ forbidden \*/

1. When using trailing comments, if possible, start the comments on the same column.

u16 u16\_count; /\* explanation comment \*/

u16 u16\_upper\_limit; /\* comment \*/

1. Do not comment out sections of code. If required, they can be removed by conditional compilation only for debugging.

#if 0

/\* unused code \*/

#endif

1. Block comments should be used according to the following example.

/\* This is the first line of the block comment

\* middle lines are started with an “\*”

\* the final line should stand alone

\*/

## Conditional compilation - Mandatory

1. Conditional compilation should not be used to remove code in production code.
2. Conditional compilation may be used for product variants and for different hardware but the naming of such compilation constants should be easily understood.
3. Conditional compilation constants should be done in one place (for instance a header file which must be included in each and every file). All conditional constants should be present in this file with an explanation of what they turn on / off and their current state (for instance feature started but not fully tested, or feature complete and deployed).

## Spacing – Good Practice

1. Each of the keywords **if**, **else**, **for**, **while**, **switch** and **return** shall always be followed by a space. If **return** doesn’t return value there is no space before semicolon (“**return;**”).
2. Each of the assignment and logical operators (such as **=, +, ~, &&)** shall always be preceded and followed by a space.
3. Don’t use space after the name of the function.

display\_str(u8\_x\_pos, u8\_y\_pos); /\* after function name without space \*/

1. One space is needed after each comma to separate function arguments:
2. One space is needed after each semicolon:

for (u8\_counter = 0u; u8\_counter < 10u; u8\_counter++)

1. Don’t use trailing whitespaces in code and in empty lines

## Conditional statements – Good Practice

1. Braces “{“and “}” shall be used for ALL conditional statements.
2. If braces are intentionally left empty, a comment shall be added to make this intention clear.

## The IF statement - Mandatory

1. The IF statement should be coded in one of the three ways shown below:

By itself:

if (condition)

{

Statement;

}

With and ‘Else’ clause:

if (condition)

{

Statement;

}

else

{

Statement;

}

With an ‘Else If’ chain:

if (condition)

{

Statement;

}

else if (condition)

{

Statement;

}

else

{

Statement;

}

1. The if statement may contain any number of ‘else if’ statements, but it MUST be terminated by a single ‘else’ clause, even if this clause is empty (in which case a comment should be included stating that this is intentional). This point is required by MISRA.
2. Ternary operator “**?**” shall not be used instead of the if statement.

## The SWITCH statement - Mandatory

1. An unconditional break statement shall terminate every non-empty switch clause.
2. Empty switch clauses that do not terminate (with a break statement) shall have a comments detailing that the fall through is intentional. Empty switch clauses which execute no code shall have a comment detailing why no code execution is required.
3. Every switch statement shall contain a default clause.
4. The default clause shall be the last clause in the switch statement.
5. A switch expression shall not represent a value that is effectively Boolean.
6. Every switch statement shall have at least two case clauses.

Mandatory layout of a switch statement:

switch (input)

{

case cst1:

...

break;

case cst2:

...

break;

case cst3: /\* Fall through to next case \*/

case cst4:

...

break;

default:

...

break;

}

## Loops - Mandatory

1. Braces **{** and **}** shall be used for the body of all loops.
2. If braces are intentionally left empty, a comment shall be added to make this intention clear.
3. Magic numbers shall not be used for initial values or endpoint tests. A constant should be used instead. 0, 1 is not considered a Magic Number and may be used as is.
4. Except for a single loop counter initialization in the first clause of a *FOR* loop, assignments shall not be made in the controlling expressions of any loops.
5. Recommended type of loop is FOR loop, use WHILE loop only if there is no alternative.
6. The CONTINUE statement is not allowed.

## The FOR loop – Good Practice

Recommended layout for a *FOR* loop:

for (u8\_counter = 0u; u8\_counter < MAX\_COUNTER\_VALUE; u8\_counter++)

{

…

…

}

## The WHILE loop – Good Practice

Recommended layout for a *WHILE* loop:

while (condition)

{

…

…

}

## The DO WHILE loop – Good Practice

Recommended layout for a *DO WHILE* loop:

do

{

…

…

} while (condition);

## Infinite loop – Good Practice

Recommended layout for infinite WHILE loop:

while (1u) /\* MISRA VIOLATION: Rule INFINITE\_LOOP.LOCAL: This intentional infinite loop is non-blocking because sleep is used inside (tx\_thread\_sleep) \*/

{

…

}

## GOTO statement – Mandatory

Using the GOTO statement is not allowed.