**Embedded C Style – Manual**

**Mohamed Yaqoob**

**November 2021**

**This manual summarises a collection of C programming Style Rules and Good practices known within the embedded community, and specifically the convention that we like to use here at Mutex Embedded. This manual forms an easy-to-use daily reference for developers.**

**COPYRIGHT NOTICE**

This document is the copyright of Mutex Embedded - © Mutex Embedded Solutions 2021 All rights reserved.

You may share this document with friends and colleagues as long as you keep this copyright notice. However, you may not sell this document.

Table of Contents

[1. C Language 5](#_Toc98854616)

[Rule 1.1 – Use C99 Standard 5](#_Toc98854617)

[Rule 1.2 – Make use of the C standard libraries 5](#_Toc98854618)

[2. Code Format 6](#_Toc98854619)

[Indentation 6](#_Toc98854620)

[Braces 6](#_Toc98854621)

[White Space 6](#_Toc98854622)

[Control Statements 6](#_Toc98854623)

[Line Wrapping 6](#_Toc98854624)

[3. Variables 7](#_Toc98854625)

[Rule 3.1 – Use camelCase naming convention 7](#_Toc98854626)

[Rule 3.2 – Use clear and descriptive names 7](#_Toc98854627)

[Rule 3.3 – Declare variables in separate lines 7](#_Toc98854628)

[Rule 3.4 – For Pointer variables, use the letter 'p' as a prefix 7](#_Toc98854629)

[Rule 3.5 – For Double Pointer variables, use "pp" as a prefix 7](#_Toc98854630)

[Rule 3.6 – For Global variables, use the letter 'g' as a prefix 7](#_Toc98854631)

[Rule 3.7 – When a variable is both Global and Pointer, use pointer prefix 8](#_Toc98854632)

[Rule 3.8 – Use Static keyword when a variable or function is to be visible only within a single source file 8](#_Toc98854633)

[Rule 3.9 – When a variable is used by interrupt routine or is a memory-mapped pointer, it must be declared as volatile. 8](#_Toc98854634)

[Rule 3.10 – Typedef your application variable whenever possible 9](#_Toc98854635)

[4. Const function parameters 10](#_Toc98854636)

[Rule 4.1 – For constant value parameter, use the constant keyword for the variable or type only, as follows: 10](#_Toc98854637)

[Rule 4.2 – For constant pointer parameter, use the const keyword to operate on the pointer only, as follows: 10](#_Toc98854638)

[Rule 4.3 – For both constant value and constant pointer parameter, use the const keyword for both the type and the pointer, as follows: 10](#_Toc98854639)

[5. Structure 11](#_Toc98854640)

[Rule 5.1 – Always typedef a structure 11](#_Toc98854641)

[Rule 5.2 – Use PascalCase for Structure name 11](#_Toc98854642)

[Rule 5.3 – Add the suffix \_t to the typedef name 11](#_Toc98854643)

[Rule 5.4 – Use filename as a prefix to the struct typedef name 11](#_Toc98854644)

[Rule 5.5 – Do not memory-map structure bitfields, instead access them directly 11](#_Toc98854645)

[6. Enums 13](#_Toc98854646)

[Rule 6.1 – Always typedef an Enum 13](#_Toc98854647)

[Rule 6.2 – Use PascalCase for Enum Name 13](#_Toc98854648)

[Rule 6.3 – Add the suffix \_e to the Enum name 13](#_Toc98854649)

[Rule 6.4 – Enum elements name shall start with the prefix <TypedefName\_> 13](#_Toc98854650)

[Rule 6.5 – Use filename as a prefix to Enum name and hence the individual elements 13](#_Toc98854651)

[7. If-Condition 14](#_Toc98854652)

[Rule 7.1 – Avoid using single line condition 14](#_Toc98854653)

[Rule 7.2 – When checking for multiple conditions, enclose conditions with parentheses 14](#_Toc98854654)

[Rule 7.3 – When comparing a variable with constant, put constant first on the left-hand side 14](#_Toc98854655)

[8. Switch-Case 15](#_Toc98854656)

[Rule 8.1 – Use Enum as switch case variable, whenever possible 15](#_Toc98854657)

[Rule 8.2 – Do not put a default case when an Enum is used as the switch case variable 15](#_Toc98854658)

[Rule 8.3 – To define a new variable within a case, add a code block explicitly 15](#_Toc98854659)

[Rule 8.4 – If multiple cases have the same handling, use fall through. 15](#_Toc98854660)

[Rule 8.5 – When using Non-Enum switch case variable, default case must be added 16](#_Toc98854661)

[9. For-Loop 17](#_Toc98854662)

[Rule 9.1 – Avoid using magic numbers for loop counter 17](#_Toc98854663)

[10. Macros 18](#_Toc98854664)

[Rule 10.1 – Use full upper-case letters for Macros 18](#_Toc98854665)

[Rule 10.2 – When defining a series of defines, use double underscore \_\_ to separate prefix text from individual elements name 18](#_Toc98854666)

[Rule 10.3 – Enclose arithmetic operations within Macros with parentheses 18](#_Toc98854667)

[Rule 10.4 – For parametric or function-like Macros, surround any use of Macro parameters with parentheses 18](#_Toc98854668)

[11. X Macro 19](#_Toc98854669)

[Step 1 – Create the LIST Macro 19](#_Toc98854670)

[Step 2 – Create the APPLICATION Macros 19](#_Toc98854671)

[12. Functions 21](#_Toc98854672)

[Rule 12.1 – Function name shall follow the camelCase convention 21](#_Toc98854673)

[Rule 12.2 – Function name shall be clear and concise 21](#_Toc98854674)

[Rule 12.3 – For library functions, use the library name as a prefix 21](#_Toc98854675)

[Rule 12.4 – For callback functions, start the function name with "cb" 21](#_Toc98854676)

[Rule 12.5 – For Boolean conditional functions, start the function name with "is" 21](#_Toc98854677)

[Rule 12.6 – Functions shall have a single exit point 21](#_Toc98854678)

[Rule 12.7 – Do not use standard C library functions names 22](#_Toc98854679)

[13. Header and Source files 23](#_Toc98854680)

[Rule 13.1 – Use lower-case letters for file names 23](#_Toc98854681)

[Rule 13.2 – Make file name unique, clear, and as small as possible 23](#_Toc98854682)

[Rule 13.3 – Make a template for Header and Source files 23](#_Toc98854683)

[Rule 13.4 – Header file shall have a protection against multiple includes 23](#_Toc98854684)

[Rule 13.5 – Header file shall not declare variables or allocate memory space 23](#_Toc98854685)

[Rule 13.6 – Header file shall have minimum includes 23](#_Toc98854686)

[Rule 13.7 – Header and Source file shall have ordered content 23](#_Toc98854687)

[14. Copying code 25](#_Toc98854688)

[Rule 14.1 – Only copy the re-usable part of code 25](#_Toc98854689)

[Rule 14.2 – If your library has many re-usable pieces of code, consider using X-Macro. 25](#_Toc98854690)

[15. Commenting 26](#_Toc98854691)

[Rule 15.1 – Add a comment for non-obvious code or where additional information is necessary 26](#_Toc98854692)

[Rule 15.2 – Use single line comments in general 26](#_Toc98854693)

[Rule 15.3 – Use multiple line comment for commenting a section and single line comment for sub-sections 26](#_Toc98854694)

[Rule 15.4 – To comment out a block of code, use #if 0 macro #endif 26](#_Toc98854695)

[Rule 15.5 – Add TODO comment for any incomplete code 26](#_Toc98854696)

[Rule 15.6 – Use Doxygen for function documentation 27](#_Toc98854697)

# C Language

## Rule 1.1 – Use C99 Standard

* You can use GNU99 mode if you require extended C features such as **atoff().**

## Rule 1.2 – Make use of the C standard libraries

There are many C Standard libraries that you can use instead of writing code from scratch. The following are the most common standard libraries and their brief description:

**<complex.h>** Complex number arithmetic

**<ctype.h>** Characters types checking

**<limits.h>** Sizes of basic types

**<math.h>** Common mathematics functions

**<stdarg.h>** Variable arguments capture

**<stdbool.h>** Boolean type

**<stdint.h>** Fixed-width integer types

**<stdio.h>** Input/output

**<stdlib.h>** General utilities such as: Memory, program, string conversions, random and algorithms

**<string.h>** String handling library

**<time.h>** Time and date utilities

# Code Format

## Indentation

* Avoid using the tab \t character across your entire code.
* Using spaces only for indentation.
* Indentation size is 2 spaces
* Indent statements within **function** body
* Indent statements within **block**
* Indent statements within **switch** body
* Indent statements within **case** body
* Indent **break** statements
* Indent **labels**

## Braces

* Brace on the **next line** for **function** declaration
* Brace on the **next line** for **blocks**
* Brace on the **next line** for **blocks in case** statements
* Brace on the **next line** for **switch** statement
* Brace on the **next line** for **linkage** (e.g. extern)

## White Space

* Add white space for **Declarator list** after comma and after pointer
* For **functions**, no white space before pointer, add a white space after pointer

e.g. void updateValue (int\* pValue);

* Add a white space before opening parenthesis for **function**
* Add a white space aftersemicolon in **for-loop** parameters
* Add a white space aftercomma in **function arguments**
* Add a white space before and after **assignment and binary operators**
* Add a white space after comma in **Initializer list**

## Control Statements

* Insert new line before **else** in an **if** statement
* Do not insert new line before **while** in a **do** statement
* Keep **else if** on the same line

## Line Wrapping

* A maximum line width of 120 characters is recommended.
* Indent wrapped line 2 indentation levels
* For wrapped functions, wrap all elements each on a new line
* Enum list always wrapped regardless of line width
* For wrapped expressions, wrap only necessary parts
* For wrapped initializer list, wrap only necessary parts

# Variables

## Rule 3.1 – Use camelCase naming convention

* First word starts with a lower case and all following words start with upper case

int16\_t initialVerticalSpeed;

**float** gpsAverageAltitude;

## Rule 3.2 – Use clear and descriptive names

* Avoid using abbreviations
* Make the variable name as short as possible, yet clear.

int16\_t avgAlt; //(Bad)

int16\_t averageAltitude; //(Good)

## Rule 3.3 – Declare variables in separate lines

* This generally improves readability.
* It reduces future potential bugs related to multiple variables in a single line, like delete or comment out.

uint32\_t startTime, finalTime, deltaTime; //(Bad)

uint32\_t startTime; //(Good)

uint32\_t finalTime;

uint32\_t deltaTime;

* Additionally: **Align variable names and values for a group of declarations**, for better readability, as follows:

uint16\_t totalGreenApples = 15;

uint16\_t totalRedApples = 17;

**float** verticalSpeed = 12.56f;

## Rule 3.4 – For Pointer variables, use the letter 'p' as a prefix

* No space shall be placed between type and pointer \* symbol, a space is added to the variable name, as follows:

**char**\* pStudentName;

## Rule 3.5 – For Double Pointer variables, use "pp" as a prefix

**char**\*\* ppWifiNames;

## Rule 3.6 – For Global variables, use the letter 'g' as a prefix

**static** uint8\_t gNextIndex;

## Rule 3.7 – When a variable is both Global and Pointer, use pointer prefix

**Example 1** – Global scope: A Pointer 🡪 Use global prefix

**static** **char**\* gLastName;

**Example 2**– Global scope: Normal variable 🡪 Use global prefix

**static** uint8\_t gNextIndex;

**Notes:**

* Local scope: Referring to when a variable is declared within a function
* Global scope: Referring to when a variable is declared globally in a source file, or a more special case is when declared as static within a function.

## Rule 3.8 – Use Static keyword when a variable or function is to be visible only within a single source file

* It is recommended to keep variables within a source file static if they are not meant to be visible by other files.
* However, it is **not recommended to define a static variable within a function**, this can make your code less readable and more susceptible to bugs.

## Rule 3.9 – When a variable is used by interrupt routine or is a memory-mapped pointer, it must be declared as volatile.

* Tick counters are incremented in an Interrupt routine, thus volatile
* Memory-mapped pointers values are volatile since a register can be changed by hardware.

**Example 1** – Tick counter

**volatile** uint32\_t gCounterTicks = 0;

**void** isrIncrementCounter (**void**)

{

gCounterTicks++;

}

uint32\_t getCounter (**void**)

{

**return** gCounterTicks;

}

**Example 2** – Memory-mapped pointer

**volatile** uint8\_t\* pInputSwitch = (uint8\_t\*)(GPIOA->IDR);

**while**(!(\*pInputSwitch & 0x01)); //Wait indefinitely for switch

printf("Switch is Activated!");

## Rule 3.10 – Typedef your application variable whenever possible

* It is generally recommended to typedef all variables in your application to a user typedef.
* Using the raw c types in application is not recommended as changing the type later is way more complex than using typedef.
* Include unit name as part of variable type name when applicable, as follows:

**typedef** uint32\_t GpsLatitude\_Micro\_t;

**typedef** **float** AverageSpeed\_Kmh\_t;

**typedef** uint32\_t Distance\_Meters\_t;

# Const function parameters

At this section, we are only going to focus on the use of the **Const** keyword for passing parameters to functions.

## Rule 4.1 – For constant value parameter, use the constant keyword for the variable or type only, as follows:

**void** setSpeed (**const** uint16\_t speed)

{

gMotorConfig->forwardSpeed = speed;

}

## Rule 4.2 – For constant pointer parameter, use the const keyword to operate on the pointer only, as follows:

**void** updateWifiNameTo4G (WifiName\_t\* **const** pWifi)

{

sprintf(pWifi->buf, "%s - 4G", pWifi->buf);

}

## Rule 4.3 – For both constant value and constant pointer parameter, use the const keyword for both the type and the pointer, as follows:

**void** printWifiName (**const** WifiName\_t\* **const** pWifi)

{

printf("Selected Wifi Name = %s\n", pWifi->buf);

}

# Structure

## Rule 5.1 – Always typedef a structure

**typedef** **struct**

{

…

}

## Rule 5.2 – Use PascalCase for Structure name

**typedef** **struct**

{

…

}AccelData\_t;

* Typedef variable name still uses camelCase

AccelData\_t accelData;

## Rule 5.3 – Add the suffix \_t to the typedef name

**typedef** **struct**

{

int16\_t accelX;

int16\_t accelY;

int16\_t accelZ;

**bool** isValid;

}AccelData\_t;

## Rule 5.4 – Use filename as a prefix to the struct typedef name

**typedef** **struct**

{

int16\_t accelX;

int16\_t accelY;

int16\_t accelZ;

bool isValid;

}Mpu6050\_Accel\_t;

## Rule 5.5 – Do not memory-map structure bitfields, instead access them directly

**typedef** **struct**

{

uint8\_t reserved : 3; //LSb

uint8\_t afs\_sel : 2;

uint8\_t za\_st : 1;

uint8\_t ya\_st : 1;

uint8\_t xa\_st : 1; //MSb

}Mpu6050\_AccelConfigReg\_t;

**int** main (**void**)

{

Mpu6050\_AccelConfigReg\_t hAccelConfig;

hAccelConfig.xa\_st = 0;

hAccelConfig.ya\_st = 1;

hAccelConfig.za\_st = 0;

hAccelConfig.afs\_sel = 2;

hAccelConfig.reserved = 0;

**~~uint8\_t accelConfigBits = \*(uint8\_t \*)&hAccelConfig;~~**

getchar();

**return** 0;

}

# Enums

## Rule 6.1 – Always typedef an Enum

**typedef** **enum**

{

…

};

## Rule 6.2 – Use PascalCase for Enum Name

## Rule 6.3 – Add the suffix \_e to the Enum name

**typedef** **enum**

{

…

}FullScale\_e;

## Rule 6.4 – Enum elements name shall start with the prefix <TypedefName\_>

**typedef** **enum**

{

FullScale\_2g = 0,

FullScale\_4g,

FullScale\_8g,

FullScale\_16g,

}FullScale\_e;

## Rule 6.5 – Use filename as a prefix to Enum name and hence the individual elements

**typedef** **enum**

{

Mpu6050\_FullScale\_2g = 0,

Mpu6050\_FullScale\_4g,

Mpu6050\_FullScale\_8g,

Mpu6050\_FullScale\_16g,

}Mpu6050\_FullScale\_e;

# If-Condition

## Rule 7.1 – Avoid using single line condition

* Always use multiple line body and enclose with braces

**if**(accelZ > 800)

{

printf("Mostly Level, Z-Accel = %d mg\n", accelZ);

}

## Rule 7.2 – When checking for multiple conditions, enclose conditions with parentheses

* Use Boolean conditional operators for multiple conditions relationship such as (OR: ||, AND: &&)

**if**((verticalSpeed > 12) && (horizontalSpeed < 5))

{

}

## Rule 7.3 – When comparing a variable with constant, put constant first on the left-hand side

* When you forget to put double equals =, compiler will detect an assignment to constant error when the constant is on the Left Hand Side, which can be very useful.

**if**(12 == verticalSpeed)

{

}

# Switch-Case

## Rule 8.1 – Use Enum as switch case variable, whenever possible

* Avoid using generic types such as integer because they take on so many values.

## Rule 8.2 – Do not put a default case when an Enum is used as the switch case variable

* This allows compiler to catch missing or unhandled cases, and throw warning.

Mpu6050\_AccelFullScale\_e fullScaleSelect = Mpu6050\_AccelFullScale\_4g;

**switch**(fullScaleSelect)

{

**case** Mpu6050\_AccelFullScale\_2g:

**break**;

**case** Mpu6050\_AccelFullScale\_4g:

**break**;

**case** Mpu6050\_AccelFullScale\_8g:

**break**;

**case** Mpu6050\_AccelFullScale\_16g:

**break**;

}

## Rule 8.3 – To define a new variable within a case, add a code block explicitly

* Cases are considered labels in C and thus do not support variable definition by default.
* Note: It is recommended to keep the **break** keyword inside the code block, for better readability.

**case** Mpu6050\_FullScale\_4g:

{

**int** value = 4;

printf("FS %dg\n", value);

**break**;

}

## Rule 8.4 – If multiple cases have the same handling, use fall through.

* Fall through is enabled by writing cases in series with just a single break at the end and a common body for all.

**case** Mpu6050\_FullScale\_8g:

**case** Mpu6050\_FullScale\_16g:

**case** Mpu6050\_FullScale\_32g:

printf("FS 8g or higher\n");

**break**;

## Rule 8.5 – When using Non-Enum switch case variable, default case must be added

* This is simply because integer variable can take on many different values, unlike Enum where elements are bounded.

**int** mySelect = 1;

**switch**(mySelect)

{

**case** 1:

**break**;

**case** 2:

**break**;

**default**:

**break**;

}

# For-Loop

## Rule 9.1 – Avoid using magic numbers for loop counter

* Use define constants instead.
* When the constant is changed, all instances will be updated unlike magic numbers where you need to change them individually.

#define ELEMENTS\_COUNT 10

**int** main (**void**)

{

**for**(uint8\_t i = 0; i < ELEMENTS\_COUNT; i++)

{

printf("Counter = %d\n", i);

}

getchar();

**return** 0;

}

# Macros

## Rule 10.1 – Use full upper-case letters for Macros

* Separate words with underscore \_

#define MAX\_HEIGHT 100

## Rule 10.2 – When defining a series of defines, use double underscore \_\_ to separate prefix text from individual elements name

#define MAIN\_TASK\_SIG\_\_SUSPEND os\_SignalNum\_0

#define MAIN\_TASK\_SIG\_\_RESUME os\_SignalNum\_1

#define MAIN\_TASK\_SIG\_\_HEAP\_MONITOR os\_SignalNum\_2

#define MAIN\_TASK\_SIG\_\_RUN\_STATS os\_SignalNum\_3

## Rule 10.3 – Enclose arithmetic operations within Macros with parentheses

#define MAX\_HEIGHT (50 \* 2)

## Rule 10.4 – For parametric or function-like Macros, surround any use of Macro parameters with parentheses

* This will ensure that parameters arithmetic operations are resolved first

#define TEMPERATURE\_CONVERT\_\_C\_TO\_F(X) (((X) \* 1.8f) + 32.0f)

printf("46.30C = %.2fF\n", TEMPERATURE\_CONVERT\_\_C\_TO\_F(45.3 + 1.0f));

Macro Expansion:

---> (((45.3 + 1.0f) \* 1.8f) + 32.0f)

# X Macro

At this section, we will demonstrate how to correctly use X Macros. We will demonstrate this with automatically typing and expanding the following code:

**enum**

{

DatabaseId\_BLDC\_Speed,

DatabaseId\_BLDC\_Status,

DatabaseId\_BLDC\_MotorRpm,

}VariablesIds\_e;

**void** setBldcSpeed (uint16\_t bldcSpeed);

uint16\_t getBldcSpeed (**void**);

**void** setBldcStatus (bool bldcStatus);

bool getBldcStatus (**void**);

**void** setBldcMotorRpm (**float** bldcMotorRpm);

**float** getBldcMotorRpm (**void**);

Where, we are going to create an Enum that populates automatically with the X Macro, and create Set/Get functions for each variable.

## Step 1 – Create the LIST Macro

* This is the only Macro that user can interact with to add and remove elements.

#define DATABASE\_LIST \

  DATABASE\_DATA(DatabaseId\_BLDC\_Speed, uint16\_t, bldcSpeed) \

  DATABASE\_DATA(DatabaseId\_BLDC\_Status, bool, bldcStatus) \

  DATABASE\_DATA(DatabaseId\_BLDC\_MotorRpm, float, bldcMotorRpm)

## Step 2 – Create the APPLICATION Macros

* APPLICATION macro refers to the user-specific macro.
* At this case, we will add 2 Macros: 1) Enum elements, 2) Set/Get functions
* This makes use of the LIST macro to create dynamically expanding code.

**Enum elements**

**typedef** **enum**

{

#define DATABASE\_DATA(row\_id, type, name) \

  row\_id,

DATABASE\_LIST

#undef DATABASE\_DATA

}VariablesIds\_e;

**Set/Get functions prototypes**

#define DATABASE\_DATA(row\_id, type, name) \

  void set##name (type name); \

  type get##name (void); \

DATABASE\_LIST

#undef DATABASE\_DATA

**Main code**

setbldcSpeed(123);

printf("BLDC Speed = %d\n", getbldcSpeed());

For further explanation of the X Macro, watch my **YouTube** video on X Macro as part of the Embedded C Programming Style tutorial series.

# Functions

## Rule 12.1 – Function name shall follow the camelCase convention

uint16\_t getAccelX (**void**);

## Rule 12.2 – Function name shall be clear and concise

* Name needs to be small, yet descriptive of its purpose.

## Rule 12.3 – For library functions, use the library name as a prefix

* Use an underscore to separate the prefix from function name as follows:

uint16\_t mpu6050\_getAccelX (**void**);

## Rule 12.4 – For callback functions, start the function name with "cb"

**void** mpu6050\_cbBarachuteOpened (**void**)

{

//Do something...

}

## Rule 12.5 – For Boolean conditional functions, start the function name with "is"

* This does not apply to all Boolean returning functions, rather for functions that explicitly perform a Boolean checking operation.

bool osLayer\_isInIrqHandler (**void**);

## Rule 12.6 – Functions shall have a single exit point

* That is, functions shall have a single **return** statement whenever possible.
* Having a **single return** helps reducing code complexity and makes debugging easier.

**Multiple returns (Not preferred)**

**bool** mpu6050\_readAccelX (uint16\_t\* pValue)

{

uint16\_t accelDataRaw;

**if**(!mpu6050\_i2cReadData16(0x00, &accelDataRaw))

{

**return** **false**;

}

**if**(!mpu6050\_isAccelDataRangeValid(accelDataRaw))

{

**return** **false**;

}

//Scale by 1/1000;

\*pValue = accelDataRaw \* 0.001f;

**return** **true**;

}

**Single return (Good)**

**bool** mpu6050\_readAccelX (uint16\_t\* pValue)

{

**bool** isSuccess = **false**;

uint16\_t accelDataRaw;

isSuccess = mpu6050\_i2cReadData16(0x00, &accelDataRaw);

**if**(isSuccess)

{

isSuccess = mpu6050\_isAccelDataRangeValid(accelDataRaw);

}

**if**(isSuccess)

{

//Scale by 1/1000;

\*pValue = accelDataRaw \* 0.001f;

}

**return** isSuccess;

}

## Rule 12.7 – Do not use standard C library functions names

* E.g. printf, scanf, memcpy, etc…
* If you have a function with similar properties, try naming it differently or use a distinct prefix. (e.g. **printf** 🡪 **userPrint()** or **user\_printf()**)

# Header and Source files

## Rule 13.1 – Use lower-case letters for file names

* This is important for case-insensitive file systems.
* For multiple words file name, use underscore "\_" to separate words (e.g1. **usb\_device.h**, e.g. 2 **log\_print\_io.h**)

## Rule 13.2 – Make file name unique, clear, and as small as possible

* File name needs to be distinct and not confused with other files
* It needs to be clear and descriptive of its purpose yet uses minimum characters or words.

For example, I have a file that manages clock events, a good name for it will be as follows:

**clock\_events.h**

Additionally, I have another related file that handles clock events in lower-level, I can name it the same but with the suffix "ll" for low-level to make it unique:

**clock\_events\_ll.h**

## Rule 13.3 – Make a template for Header and Source files

* It is generally a good practise to have a template for both Header and Source files for keeping file information and copyright notices, instead of writing that from scratch each time.

## Rule 13.4 – Header file shall have a protection against multiple includes

* The entire Header file code shall be enclosed withing the following ifndef and endif:

#ifndef DATABASE\_TEST1\_H\_

#define DATABASE\_TEST1\_H\_

//Code goes here

#endif

* The #define NAME need to be unique to each header file, usually it uses the actual file name in upper-case letters (e.g. clock\_events.h 🡪 #define CLOCK\_EVENTS\_H\_).

## Rule 13.5 – Header file shall not declare variables or allocate memory space

## Rule 13.6 – Header file shall have minimum includes

* The rest of includes apart from those required by the header file, shall go into the source file.

## Rule 13.7 – Header and Source file shall have ordered content

**Header file**

Header file content shall be ordered as follows:

* Header comment
* Multiple includes protection
* Includes
* Types
* Functions prototypes
* Callback prototypes (callback functions prototypes)

**Source file**

Source file content shall be ordered as follows:

* Header comment
* Includes
* Macros / Defines
* Variables
* Private functions prototypes
* Private functions definitions
* Public functions definitions
* Callback definitions

# Copying code

## Rule 14.1 – Only copy the re-usable part of code

* Never copy the full code with the intention to change it, rather copy only parts of the code that you need fully.

Here is an example of copying re-usable code safely. I have some Database library that has a set function for each variable. Currently it has Tank Level set function. Set Battery function needs to be added.

**void** databaseApp\_gui\_setTank (Database\_GUI\_TankLevel\_t level)

{

database\_setUint8(DatabaseId\_GUI\_MAIN\_Tank, level);

}

Step (1): Function name

**void** databaseApp\_gui\_set

Step (2): Put the right name

**void** databaseApp\_gui\_setBattery ()

Step (3): Copy parameter part

**void** databaseApp\_gui\_setBattery (Database\_GUI\_)

Step (4): Manually fill in name

**void** databaseApp\_gui\_setBattery (Database\_GUI\_Battery\_t battery)

Step (5): Copy body line upto re-usable part

{

database\_set

}

Step (6): Manually fill in the remaining parts to get to this final form

**void** databaseApp\_gui\_setBattery (Database\_GUI\_Battery\_t battery)

{

database\_setFloat(DatabaseId\_GUI\_MAIN\_Battery, battery);

}

## Rule 14.2 – If your library has many re-usable pieces of code, consider using X-Macro.

* Use of X-Macro can help you automate your code safely.
* Use of X-Macro is recommended when many parts of code are repeated.

# Commenting

## Rule 15.1 – Add a comment for non-obvious code or where additional information is necessary

* Avoid adding trivial comments, to make your code tidy and easy to read

//Accelerometer full scale, read page 15 of the Registers map manual

**typedef** **enum**

{

Mpu6050\_AccelFullScale\_2g = 0,

Mpu6050\_AccelFullScale\_4g,

## Rule 15.2 – Use single line comments in general

## Rule 15.3 – Use multiple line comment for commenting a section and single line comment for sub-sections

/\* MPU6050 Initialisation sequence \*/

//Allocate memory for the object

Mpu6050\_Config\_t\* pConfig = malloc(**sizeof**(Mpu6050\_Config\_t));

//Set the I2C address

pConfig->i2cAddrs = addrs;

//Intialise the full scale value

pConfig->accelFullScale = fullScale;

## Rule 15.4 – To comment out a block of code, use #if 0 macro #endif

* Do not use actual comment to disable part of code, it is much safer to use #if 0 macro
* It generally improves readability

#if 0

void mpu6050\_cbHighAltitudeDetected (uint16\_t altitude);

bool mpu6050\_isFreeFallDetected (void);

bool mpu6050\_readLoadCurrent (float\* pCurrent);

#endif

## Rule 15.5 – Add TODO comment for any incomplete code

* Include developer name in the comment to help tracing changes "TODO (Mohamed Yaqoob): ..."

**void** mpu6050\_updateAccelData (Mpu6050\_Config\_t\* pConfig)

{

//TODO(Mohamed Yaqoob): Link to I2C data read function

pConfig->accelX = 20;

pConfig->accelY = 40;

pConfig->accelZ = 940;

}

## Rule 15.6 – Use Doxygen for function documentation

* Doxygen can be used to generate documentation, especially for functions
* Put function comments or documentation in header file only

/\*\*

 \* @brief Get Accelerometer X-Axis data

 \* @param pConfig

 \* @return acceleration in milli-g unit

 \*/

int16\_t mpu6050\_getAccelX (Mpu6050\_Config\_t\* pConfig);