Coding Style Guide

My Vision: Standardization was designed specifically to reduce the number of programming defects in embedded software. By following this coding standard, firmware developers not only reduce hazards to users and time spent in the debugging stage of their projects but also improve the maintainability and portability of their software. Together these outcomes can greatly lower the cost of developing high-reliability embedded software.

Interactions between electronics and software as well as between inter-connected systems are Clearly, no set of coding rules will be able to eliminate 100% of the defects from embedded systems. Interactions between electronics and software as well as between inter-connected systems are complex by their nature. Even if there existed a team of programmers able to code perfectly and they followed all possible defect minimizing rules, defects in the product could still occur as a result of mistakes in the project requirements; misunderstandings of the requirements by implementers oversights in the architecture of the system and/or software; insufficient handling of hardware failures or other exceptional run-time circumstances... etc.

Other important reasons to adopt this coding standard include increased readability and portability of source code. The result of which is reduced cost of code maintenance and reuse. Adopting the complete set of rules in this coding standard (i.e., not just the defect reducers) benefits a team of developers and its larger organization by helping to reduce the time required by individuals to understand the work of their peers and predecessors.

Guiding Principles:

To focus our attention and eliminate internal conflict over items that are too-often viewed by programmers as personal stylistic preferences, this coding standard was developed in accordance with the following guiding principles:

1- Individual programmers do not own the software they write. All software development is work for hire for an employer or a client and, thus, the end product should be constructed in a workmanlike manner.

2- It is cheaper and easier to prevent a bug from creeping into code than it is to find and kill it after it has entered. A key strategy in this fight is to write code in which the compiler, linker, or a static analysis tool can detect such defects automatically—i.e., before the code is allowed to execute.

3- For better or worse (well, mostly worse), the ISO C Programming Language "Standard" permits a considerable amount of variation between compilers.[2] The ISO C Standard's "implementation-defined," "unspecified," and "undefined" behaviours, along with "locale-specific options", mean that even programs compiled from identical source code but via different "ISO C"-compliant compilers may behave very differently at run-time. Such gray areas in the C language standard greatly reduce the portability of source code that is not carefully crafted.

4- The reliability, readability, efficiency, and sometimes portability of source code is more important than programmer convenience.

5- There are many sources of defects in software programs. The original team of programmers will create some defects. Programmers who later maintain, extend, port, and/or reuse the resulting source code may create additional defects—including as a result of misunderstandings of the original code.

MISRA C:

Guidelines for the use of the C language in critical systems, **MISRA** provide world-leading, best practice guidelines for the safe application of both embedded control systems and standalone software.

A) Data Types

Each Software Module may have its own types that will be described later in each module SRS document.
But now we will the most basic data types that will be used in all modules and any piece of the software, STD_TYPES.h header file that contains definitions for this data types.

u8

This is always defined as the most efficient data type (memory wise)
It is Actual type is unsigned char that means it have positive rang only up to 255,
Because it’s one byte data type.

u16

It is Actual type is unsigned short int that means it have positive rang only up to 65535,
Because it’s two-byte data type independent target architecture.

u32

It is Actual type is unsigned long int that means it have positive rang only up to 4294967296,
Because it’s four-byte data type independent target architecture.

s8

This is always defined as the most efficient data type (memory wise)
It is Actual type is signed char that means it have negative and positive rang, from -128 to +127, Because it’s one byte data type.

s16

It is Actual type is signed short int that means it have negative and positive rang, from -32768 to +32767, Because it’s two-byte data type independent target architecture.

s32

It is Actual type is signed long int that means it have negative and positive rang, from -2147483648 to +2147483647, Because it’s four-byte data type independent target architecture.

B) Files Names and objective:

Any Software module consist of some source and header files, let’s discuss sume of most common files:

1- **MODULE_program.c** ex: **DIO_program.C**
   Source file that contains the module functions implementation.

2- **MODULE_interface.h** ex: **PWM_interface.h**
   Header file that contains the module global functions prototypes and global macros.

3- **MODULE_private.h** ex: **LCD_private.h**
   Header file that contains the module private functions prototypes and private macros.

4- **MODULE_register.h** ex: **TWI_register.h**
   Header file that contains the module registers addresses definitions.

5- **MODULE_config.h** ex: **TMR1_config.h**
   Header file that contains configuration macros (prebuild configuration type).

6- **MODULE_types.h** ex: **EXTI_types.h**
   Header file that contains the module global types.

C) Variables Names:

1- **Global variables** data type **MODULE_typeVariableDescriptiveName;**

Ex:	u16 ICU_u16DutyCycle;
Ex:	u8 DIO_u8PinState;
Ex:	u16* ADC_pu16DigitalValue;

2- **Private global variables** static data type **PRV_typeVariableDescriptiveName;**

Ex:	static void (*PRV_pFunCallBackOVF)(void);
Ex:	static u16* PRV_pu16OnTime;
Ex:	static u8 PRV_u8Counter;

3- **Local variables** data type **local_typeVariableDescriptiveName;**

Ex:	u8 local_u8Counter;
Ex:	u16 local_u16AnalogeValue;
Ex:	u8* local_pu8PinValue;

4- **Function input arguments** data type **copy_typeVariableDescriptiveName;**

Ex:	u8 copy_u8Cmnd;
Ex:	u16 copy_u16Position;
Ex:	u8* copy_pu8String;

D) Macros Names #define **MODULE_MACRO_DESCRIPTIVE_NAME** macroVal

Ex:	#define DIO_PORTA 0
Ex:	#define TWI_START_CONDITION_ACK 0x08
Ex:	#define PWM_CHANNEL_0_NONINVERTING 1

E) Functions Names:

1- **Global functions** return data type **MODULE_reTypeFunctionDescriptiveName**

Ex:	void PWM_voidInitChannel_0(void);
Ex:	u8 EEPROM_u8ReadByte (u16 copy_u16WordAddress);
Ex:	void PWM_voidGenerate_PWM_Channel_0(u8 copy_u8DutyCycle);

2- **Private functions** static return data type **PRV_reTypeFunctionDescriptiveName**

Ex:	static void PRV_voidEnable(void);
Ex:	static void PRV_voidWriteHalfPort(u8 copy_u8Value);
Ex:	static void PRV_voidScheduler(void);