C Standard and Development Guideline

EMBEDDED SW CODING RULES

ABSTRACT

This document is intended to be the C coding standard for Sofia Technologies Embedded Software team.
This document contains information deemed CONFIDENTIAL and PROPRIETARY. Do not use, duplicate or disclose, even partially, without Sofia Technologies prior consent.

DISTRIBUTION LIST

Name	Function	Organization

REVISION HISTORY

Revision	Changes	Name		Date
1	Creation	**Owner**	**MBR**	**04 Oct 2017**
		Reviewer		
		Approver		
		Owner		
		Reviewer		
		Approver		

TABLE OF CONTENTS

TABLE OF CONTENTS 2

1 INTRODUCTION 3

2 SPIRIT OF THE STANDARD 4

3 NAMING CONVENTIONS 5

3.1 GENERAL 5
3.2 FILE NAMES 5
3.3 CONSTANTS 5
3.4 VARIABLES 5
3.5 FUNCTIONS 6
3.6 TYPES 6
3.7 OTHER NAMES 6

4 C CODING RULES 6

4.1 PREPROCESSOR 6
4.2 CONSTANTS 6
4.3 TYPES 7
4.4 VARIABLES 8
4.5 FUNCTIONS 9
4.6 LOOP STATEMENTS AND CONTROL FLOW 10
4.7 MISCELLANEOUS 11

5 PRESENTATION AND STYLE 12

5.1 SOURCE AND HEADER FILES 12
5.2 C SOURCE CODE 12

6 REFERENCES 17

1 INTRODUCTION

C programming rules and recommendations announced in this document are expected to be adopted for all firmware developments and maintenances within Sofia Technologies Embedded Software team.

Rules starting with **{Req}** are required and must be followed during the development. Variations are permitted if agreed by the project team and should be documented, this is applicable only for advisory rules (those without the prefix **{Req}**).

This standard was defined to fulfill the following goals:

- Maintain consistency through firmware projects developed within Sofia Technologies Embedded Software team;
- Encourage consistent code layout;
- Produce robust and error free code;
- Ease code understanding and usage;
- Increase code portability, reuse and maintainability.

Some of the C coding rules announced in this document:

- are specific to Cortex-M cores (based on ARMv7 architecture)
- based on MISRA C 2004 standard rules

2 SPIRIT OF THE STANDARD

1. Keep the *spirit* of the standard. Where you have a coding decision to make and there is no direct standard, then you should always keep within the spirit of the standard.

2. All code must comply to ANSI C standard and must compile without warning under at least the compiler used in the project. Any warnings that cannot be eliminated should be commented in the code.

3. Keep it *simple*. Break down complexity into simpler chunks. Clearly comment necessary complexity.

4. Be *explicit*. Avoid implicit or obscure features of the language. Say what you mean.

5. Be *consistent*. Use the same rules as broadly as possible.

6. Minimize scope. This includes logical and visual scope. Be more explicit about items with wider scope.

7. Existing code will not be required to be rewritten to adhere to these guidelines, but it is suggested that whenever existing code is modified try to update it with the conventions outlined in this document. This will ensure that old code will be upgraded over time.

8. Divide header files to reflect the files they serve. For each subsystem, have: one or more interface files; one common file; lower level files as required.

3 NAMING CONVENTIONS

3.1 GENERAL

1. **{Req}** All the names (folders, files, code...) are in English. All the comments are also in English.
2. **{Req}** Names must be chosen so that the code is self-documenting.
   A programmer should be able to determine what a function does or a variable is for just by reading its name and without analyzing the code.
3. **{Req}** Macros, enumeration constants and global constant and typedef names should be stylistically distinguished from function, and variable names.

3.2 FILE NAMES

4. File names are composed of a base name and an optional suffix.
5. **{Req}** File names must represent the content or role of the file.
6. Only lowercase letters, digits, underscore and at most one suffix are allowed in a file name for portability reasons between DOS and Windows. (e.g. mpu9250.h, lis3dsh.c)
7. **{Req}** Header file names must have the extension ".h".
8. **{Req}** C source file names must have the extension ".c".
9. **{Req}** Assembler source file names must have the extension ".s".
10. **{Req}** Pre-compiled library must have the extension ".a" or ".lib", depending on the compiler used.

3.3 CONSTANTS

11. **{Req}** Constant names must be in uppercase, with multiple words separated by an underscore (e.g. `APP_ADV_INTERVAL` or `THOUSAND`).

3.4 VARIABLES

12. **{Req}** Variable names should be short, but meaningful (e.g. `NumPageToWrite`).
13. **{Req}** Avoid single letter variables since they are not meaningful (e.g. "`Index`" instead of "`i`"). Exception is allowed only for variables used as loop counter.
14. **{Req}** Local variables must be named with the content of the variable. Each word must be in lowercase and underscores are allowed (e.g. `deviceaddress`, `status_reg`).
15. **{Req}** Global variables must be named with the content of the variable. Each word must be capitalized, with no underscores allowed (e.g. `DataLength`).
16. **{Req}** Use the following prefixes to denote special types: p = pointer and a = array.

```
uint32_t *pBuffer;
uint8_t aTxBuffer[50];
```

3.5 FUNCTIONS

17. **{Req}** Function names must be self-explaining, and identify as far as possible the action performed or the information provided by the function.
18. Function names should be prefixed with the name of the module to which they belong (e.g. `HAL_ADC_Init`)
19. **{Req}** Static function must not have any prefix.

3.6 TYPES

20. **{Req}** Types names must be suffixed with _t

3.7 OTHER NAMES

21. **{Req}** Macros, with or without parameters, should also be in all upper case.
22. **{Req}** Enumeration constants and global typedef names should be in all uppercase with individual words separated by underscores, (e.g. `GPIO_MODE`)

4 C CODING RULES

4.1 PREPROCESSOR

23. A `#else` and a `#endif` should be followed on the same line by a comment based on the test used by the `#if` statement as following:

```
#if defined ENABLE_DEBUG_UART
...
#endif /* ENABLE_DEBUG_UART */
```

```
#if defined BSW_ENABLE_BLE
...
#else /* BSW BLE Disabled */
...
#endif /* BSW_ENABLE_BLE */
```

24. **{Req}** All use of the “pragma” directive must be documented and explained.

4.2 CONSTANTS

25. **{Req}** Variables that do not need to be modified must be declared as constants.
26. **{Req}** A constant macro defined with #define, its value must be casted with the appropriate type and encapsulated with parentheses, e.g.:

```
#define VDD_VALUE ((uint32_t)3300)
```

Exception is allowed only for macro used for conditional compilation, where the type cast and parentheses can be omitted.

27. **{Req}** Constants used in one file are defined within this file. A constant used in more than one file is defined in a header file.
28. Do not use a `#define` simply too shorten a name as the following example:
```
#define TNOP TotalNumberOfPackets
```
29. **{Req}** The use of 'magic numbers' (like 100 and 10) is not recommended as they are hard to change in a systematic way, instead use macros (see example).

    In general; macros must be used for values dependent from HW/SW specification (e.g. LCD resolution), that impact the code features (e.g. number of supported UART interfaces, in serial bridge application), configuration (e.g. size of communication buffers) and evolution (e.g. address of supported I2C EEPROM).

    Example: should avoid using magic number like 76800
```
for(index = 76800; index != 0; index--)
{}
```
    Use macros instead, which help to make the code easier to understand and to maintain:
```
#define LCD_PIXEL_WIDTH ((uint32_t)320)
#define LCD_PIXEL_HEIGHT ((uint32_t)240)
for(index = (LCD_PIXEL_WIDTH * LCD_PIXEL_HEIGHT); index != 0; index--)
{}
```

4.3 TYPES

30. **{Req}** Uses ANSI standard data types defined in the ANSI C header file <stdint.h>
```
/* exact-width signed integer types */
typedef signed char int8_t;
typedef signed short int int16_t;
typedef signed int int32_t;
typedef signed __int64 int64_t;
/* exact-width unsigned integer types */
typedef unsigned char uint8_t;
typedef unsigned short int uint16_t;
typedef unsigned int uint32_t;
typedef unsigned __int64 uint64_t;
```
Note: A deviation is allowed for counter loop; "`int`" should be used as type of the counter variable, to ensure portability across different architectures

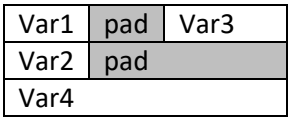
31. IO type qualifiers are used to specify the access to volatile variables (e.g. registers):
```
#define __I volatile const
#define __IO volatile
```

32. As type sizes are used on the target processor, do not make any assumption about the size of the types, instead use **`sizeof()`** operator.

4.4 VARIABLES

33. Global variables should be avoided unless they are really necessary, and good reasons should be given for their use.
34. **{Req}** Variables whose value can change independently of the execution of the program (e.g. input register value, variable shared between interruption and main processes) must be declared as `volatile`.
35. **{Req}** Do not declare a variable as `extern` in a source file. Instead include the header file where the variable is declared as `extern`.
36. **{Req}** Group variables of the same type together. This is the best way to ensure that as little padding data as possible is added by the compiler. For example,

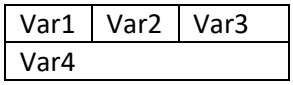
```
uint8_t  Var1;
uint16_t Var3;
uint8_t  Var2;
uint32_t Var4;
```



occupies 12 bytes, with 4 bytes of padding.

While,

```
uint8_t  Var1;
uint8_t  Var2;
uint16_t Var3;
uint32_t Var4;
```



occupies 8 bytes, without any padding.

37. **{Req}** Local variables must be declared at the top of the function in which they are used.
38. **{Req}** Local variables must be assigned a value before being used.
39. **{Req}** For local variables, use word-sized variables rather than half word and byte: to avoid additional shifts to ensure that variables only occupy specified space within 32-bit register.
40. **{Req}** When comparing a variable and a constant in a conditional statement, the variable should be on the left hand size of the operator as the following example:

```
if (Variable == CONSTANT)
if (Variable <= CONSTANT)
```

41. If a variable needs to keep its contents over multiple calls to a function, then that variable should be declared as a local static and not a global static variable.
42. **{Req}** The ‘‘pointer’’ qualifier, ‘*’, must be with the variable name rather than with the type.

    `int8_t *Var1, *Var2, *Var3;`

    instead of:

    `int8_t* Var1, Var2, Var3;`

    which is wrong, since ‘var2’ and ‘var3’ do not get declared as pointers.

4.5 FUNCTIONS

43. **{Req}** Functions must have prototype declarations and the prototype shall be visible at both the function definition and call.
44. **{Req}** For each function parameter the type given in the declaration and definition must be identical, and the return types must also be identical.
45. Try to ensure that functions take four or fewer arguments. These will not use the stack for argument passing.
46. If a function needs more than four arguments, put the arguments in a structure, and pass a pointer to the structure to function. This will reduce the number of parameters and optimize memory usage.
47. Avoid functions with a variable number of parameters. Varargs functions effectively pass all their arguments on the stack.
48. Pass pointers to structures instead of passing the structure itself.
49. **{Req}** Declaration of a function with `extern` is forbidden. Include the header file where the prototype of the function is declared.
50. **{Req}** Always declare function type, at least `void`.
51. **{Req}** If a function has an empty argument list this should be clearly indicated using `(void)`.
52. **{Req}** Good idea to make functions static if they do not get called from other modules.
53. Group function calls. When function calls are separated by other operations, each call forces the compiler to store any register-allocated variable to memory. When calls are consecutive, these values only need to be saved once.

54. **{Req}** In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses:

```
#define ABS (x) (((x) >= 0) ? (x) : -(x))
```

and not:

```
#define ABS (x) ((x >= 0) ? x : -x)
```

55. Library function that may need to be redefined by user, has to be declared as `__weak` (refer to the right keyword for your compiler). However, the function prototype must not have `__weak` keyword.
56. **{Req}** Do-while-zero construct must be used to wrap a series of one or more statements in macros:

```
#define __ETH_CLK_ENABLE() do {                          \
                             __ETHMAC_CLK_ENABLE();   \
                             __ETHMACTX_CLK_ENABLE();\
                             __ETHMACRX_CLK_ENABLE();\
                           } while(0)
```

4.6 LOOP STATEMENTS AND CONTROL FLOW

57. The loop termination condition can cause significant overhead if written without caution. Where possible:
    a. use simple termination conditions;
    b. write count-down-to-zero loops;
    c. use counters of type unsigned int;
    d. test for equality against zero.
58. **{Req}** Always provide a default for `switch` statements.
59. **{Req}** Always use braces `{}` to limit the block in **`if`**, **`while`** and **`do`**, even if there is a single statement.
60. Tests of a value against zero should be made explicit,

```
if (Counter != 0) /* Correct way of testing */
if (Counter) /* Not compliant */
```

61. 62. In a while loop, the increment or decrement should be at the top or bottom of the block. Do not increment or decrement loop counter variables in the middle of a block. This makes it harder to find the loop counter.
62. In `if..else`, the test in the `if` statement should be the one which is most of the time correct to avoid branching to the `else` statement.

63. In interrupt routine, avoid using loops that waits on hardware flags. If this is necessary, clearly document the reasons in the source code.
64. **{Req}** Loop polling on hardware flag must be conditioned with timeout value, to avoid endless loop. Exception is allowed for some cases where a blocking loop may be the intended behavior of the application, in this case need to clearly document the reasons.

4.7 MISCELLANEOUS

65. Code should be optimized. If the optimized code is non-intuitive then it must be well commented.
66. Do not assume that a NULL pointer is represented by binary zeros, the compiler will supply the required bit-pattern when it sees a constant '0' in a pointer context.
67. Unions shall not be used. However, it is recognized that sometimes such storage sharing may be required for reasons of efficiency. Where this is the case it is essential that measures are taken to ensure that the wrong type of data can never be accessed, that data is always properly initialized and that it is not possible to access parts of other data (e.g. due to alignment differences).
68. **{Req}** All code used for debug purpose must be written under conditional compilation and the condition is such as the debug code is not embedded if the condition is not present. (e.g. `#if defined CODE_DEBUG`)
69. Use special comments to indicate the presence of known bugs, enhancements and legacy code. (e.g. `//ToDo`)
70. Avoid the use of **`goto`** and **`continue`** statements. If their use is necessary, clearly document the reasons.
71. **{Req}** Header files should be used as a means of interface specification for a source module. For example, for a module wifi.c, the external interface containing data declarations, function prototypes and all information necessary to use the module should be declared in wifi.h.
72. **{Req}** To avoid nested includes of header files, use `#define` wrappers as follows:

```
#ifndef __WIFI_H
#define __WIFI_H
...
#endif /* __WIFI_H */
```

73. Macro used to configure the source code and/or special feature of the device, should be declared with a value, and the source code need to check on the value of the macro instead if it's declared or not. e.g. in header file, BLE_ADVERTISING_ENABLE is defined as below:

```
#define BLE_ADVERTISING_ENABLE      1
```

in the source file; a check is done on BLE_ADVERTISING_ENABLE value, if it's different from zero (i.e. user needs to activate this feature) then an action is done:

```
#if (BLE_ADVERTISING_ENABLE != 0)
___NRF52_BLE_ADVERTISING_OPERATION_ENABLE();
#endif /* BLE_ADVERTISING_ENABLE */
```

5 PRESENTATION AND STYLE

5.1 SOURCE AND HEADER FILES

74. **{Req}** Source and header files must use Doxygen style and based on the reference templates.
75. The file's header must contain a description of the purpose of the module and any other pertinent information about the module.

5.2 C SOURCE CODE

76. **{Req}** Line length is limited to 120 characters, when needed use "\" to split instruction on several lines.
77. All comments in source code files must be up-to-date at all times during that code's lifetime.
78. **{Req}** Code must be indented to highlight the block structure.
79. **{Req}** The size for indents is 2 spaces. Tabulations must not be used to indent code, although some editors will expand tabulations to the required number of spaces.
80. Use vertical and horizontal white space generously. Indentation and spacing should reflect the block structure of the code. There should be at least two blank lines between the end of one block and the comment of the next one.
81. Empty lines should be used to structure the source code clearly and should surround statements that logically belong together.
82. When wrapping lines, indent to past logical start of wrapped item.
83. **{Req}** Source code must only use ANSI C `/* ... */` style comment

```
/* Single line comment */
if (Condition) /* Another single line comment */
```

84. **{Req}** C99 and C++ comment style // must not be used
85. **{Req}** Do not keep portion of code commented out.
86. **{Req}** Comments should be indented to the same amount of the code it refers to.
87. **{Req}** Use blanks around all binary operators except " . " and "->", this will make C expressions more readable. The following are examples of readable code:

```
/* Space separation between identifiers and operators */
a = (b + c) * d;
/* Example to show function call assignment */
Value = ReadString(StringPtr);
/* Example to show identifiers separation: space after a comma or
semi-colon */
Result = SendData(ParamOne, ParamTwo, ParamThree);
```

Examples of unreadable code:

88. code, and code between compiling conditions are also indented. e.g.:

```
void Func(void)
{
 statement1;
#ifdef TEST1
 statement2;
#endif /* TEST1 */
 if (condition)
 {
  statement3;
#ifdef TEST2
  statement4;
#endif /* TEST2 */
 }
a=(b+c)*d;
Value=ReadString(StringPtr);
Result=Translate(ParamOne,ParamTwo,ParamThree);
/* Overuse of spaces */
a = ( b + c ) * d ;
Value = ReadString ( StringPtr ) ;
Result = SendData ( ParamOne , ParamTwo , ParamThree ) ;
```

89. **{Req}** Preprocessor statements are indented to the same level as the surrounding code, and code between compiling conditions are also indented. e.g.:

```
void Func(void)
{
 statement1;
#ifdef TEST1
 statement2;
#endif /* TEST1 */
 if (condition)
 {
  statement3;
#ifdef TEST2
  statement4;
#endif /* TEST2 */
 }
}
```

90. **{Req}** Long macro's must be split onto multiple lines using '\' separator. e.g.:

```
#define IS_USART_ALL_PERIPH(PERIPH) (((PERIPH) == USART1) || \
                                     ((PERIPH) == USART2) || \
                                     ((PERIPH) == USART3) || \
                                     ((PERIPH) == UART4)  || \
                                     ((PERIPH) == UART5)  || \
                                     ((PERIPH) == USART6))
```

91. **{Req}** Functions indentation is as following:

```
void RCC_PLLI2SConfig(uint32_t PLLI2SN, uint32_t PLLI2SR)
{
 ..statement;
}
```

92. **{Req}** `for` statement indentation is the following:

```
for (Item = ListHead; Item != 0; Item--){
 ..statement;
}
```

If the statement is too long, then the following indentation is used:

```
for (Item = ListHead;
.....Item != 0;
.....Item --)
{
 ..statement;
}
```

Curly brackets start and stop the loop body and are not indented.

93. **{Req}** `while` statement indentation is the following:

```
while (condition)
```

```
{
 ..statement;
}
```

If there are many stop conditions, then they can be split on several lines.

Curly brackets start and stop the while body and are not indented.

94. **{Req}** `do while` statement indentation is the following:

```
do
{
 ..statement;
} while (condition)
```

Curly brackets start and stop the while body and are not indented.

95. **{Req}** `switch` statement indentation is the following:

```
switch (Variable)
{
case CaseA:
..statement;
..break;
case CaseB:
..statement;
..break;
...

default:
..statement;
..break;
}
```

Curly brackets start and stop the while body and are not indented.

96. **{Req}** `if` statement indentation is the following:

```
if (first condition)
{
..statement;
}
else if (first condition)
{
..statement;
}
else
{
..statement;
}
```

Curly brackets start and stop the while body and are not indented.

97. **{Req}** A long string of conditional operator should be split onto separate line e.g.:

```
if ((List->Next == NULL) && (TotalCount < Needed) && (Needed <=
kMaxAlloc))
```

The following structure could be also used:

```
if ((List->Next == NULL) && \
    (TotalCount < Needed) && \
```

```
(Needed <= kMaxAlloc))
```

6 REFERENCES

- C/C++ Compiler Hints & Tips, ARM, 2002
- Declaring C Global Data (ARM DAI 0036B), ARM, 1998
- Writing Efficient C for ARM (RM DAI 0034A), ARM, 1998
- Working With Your Compiler, Hawn Prestridge, IAR Systems, 2011
- C Style: Standards and Guidelines, David Straker, Prentice Hall, 1992
- C Coding Standards, Alan Bridger, Atacama Large Millimeter Array, 2001
- MCD Application C Standard, STMicroelectronics, 2014