MotorWare Coding Standard

February 2013

Contents

Coding Standard "Top 10" List

1. Variable Names

- All variables are camel case starting with a lowercase letter. Global variables must begin with the letter 'g'.
    - e.g uint16_t loopCounter; uint16_t gDataBuffer.
- Type definitions must begin with their module name in all capital letters, followed by an underscore (i.e. '_').
    - e.g. MODULE_FruitTypes
- Defined constants must be in all caps and start with the module name.
    - e.g. #define MODULE_REGISTER_BIT (1 << 4)

2. Function Names

- All functions must begin with their module name in all capital letters, followed by an underscore (i.e. '_').
- Following the underscore, functions names must be camel case starting with a lowercase letter and should start with a verb describing the action of the function
    - e.g. UART_putChar(); PWM_setPeriod(); TIMER_getValue();

3. Comments

- All comments must use the single line comment delimiter "//" and must be indented to the same level of the code they document.

4. Function Prototypes

- All functions must be prototyped in a module's corresponding header file.
- All function declarations must contain extern keyword. All inline functions must be static.
- Prototypes must include Doxygen comments. See the example header file in the appendix for more details.

5. Indentation

- Each indent level is four (4) spaces and all white space only contains spaces (i.e. no tabs)
- Continued lines must be indented eight (8) spaces.

6. Bracketed statements must be in one of the following forms

```
if (expr)
 {
    stmt;
}
else
 {
    stmt;
}
```

```
do
 {
    stmt;
}
```

```
for(expr; expr; expr)
{
    stmt;
}
```

```
switch (expr)
 {
    case CONST_1:
        break;
    case CONST_2:
        break;
    default:
        break:
}
```

7. File Structure

- All software in a file must reside in one of the following sections. The sections must appear in the following order within a file - header, includes, defines, typedefs, globals, function prototypes. See the example header file in the appendix for more details.

8. Data Types

- bool_t for Boolean values
- C99 types for others: (u)int_leastX_t (portable), (u)int_fastX_t (architecture specific) and (u)intX_t (architecture specific), where X is 8, 16, 32 and 64

9. Macro Definitions

- Only single line macros are allowed. Inline functions must be used to optimize functions.

10. Not needed. ☺

1 Introduction

This file contains coding standards for Portable C code and Architectural Specific C code. It is meant to be a mechanism by which software can be created that is uniform in structure and functionality. While each programmer has their own coding style, the intent of these guidelines is to standardize the following

1. Files
2. Embedded documentation
3. Data types (portable, architecture specific)
4. Naming Conventions
5. Coding syntax

2 File Conventions

This section describes coding standards related to source and header files.

2.1 General Conventions

1. English is the language of choice for variable names, function names, comments, etc.
2. File content must be kept within 80 columns.
3. Special characters are not to be used.
4. Examples: Tab, page break, etc.

2.2 Layout

1. Loops must be left justified.
2. The basic indentation must be 4 spaces.
3. Single spaces and newlines are the only allowed white space characters.
4. Unix end-of-line character (‘\n’ or ASCII LF (0x0A)) will be used on all files.
5. All source files must contain a header denoting the file name and path. See the example header file in the appendix for more details.
6. All source files must contain a copyright statement attributing the code to Texas Instruments. The year in the copyright statement refers the year in which the code was created.
7. No random blank line pattern must exist in the code. If one (or two) blank lines are used to space the code, then this convention must be followed throughout the file.
8. All functions in the file must be alphabetized, except for inline functions that depend upon other inline functions in the same file. In this situation, the dependent inline functions should appear at the bottom of the section.

2.3 File Names

File names must follow the function naming conventions.

2.3.1 C File Names

1. The extension for the source file name must be *.c
2. The extension for the header file name must be *.h

2.3.2 Header files

1. All public functions in a source file must have function prototypes in a header file of the same file name.

3 Embedded Documentation

1. All variables, defines, type definitions and function prototypes must have Doxygen comments. See the appendix for a header file example that uses Doxygen comments.
2. Documentation must be generated with the latest stable version of Doxygen and Latex.

4 Data Types

This section describes the acceptable data types for software development.

4.1 Portable

The purpose of this section is to provide a mechanism for portable software development across different processor architectures. By using portable data types, the software can be ported across architectures without modification. However, this feature does not always provide the most cycle efficient implementation.

4.1.1 General

1. The endianness of the processor is assumed to be little endian. If the processor is big endian, the complex data type definitions below must be modified.

4.1.2 Portable Data Types

1. The following data types must be used for developing portable code. When defined, the data type from the C99 standard [1] is used. For signed fractional numbers, a data type style similar to the C99 integer style has been defined.

Type Name	Description
bool_t	A single Boolean value
char	A character
int_least8_t	A signed integer with a width of at least 8 bits
uint_least8_t	An unsigned integer with a width of at least 8 bits
int_least16_t	A signed integer with a width of at least 16 bits
uint_least16_t	An unsigned integer with a width of at least 16 bits
int_least32_t	A signed integer with a width of at least 32 bits
uint_least32_t	An unsigned integer with a width of at least 32 bits
int_least64_t	A signed integer with a width of at least 64 bits
uint_least64_t	An unsigned integer with a width of at least 64 bits
int_fast8_t	The fastest signed integer with a width of at least 8 bits
uint_fast8_t	The fastest unsigned integer with a width of at least 8 bits
int_fast16_t	The fastest signed integer with a width of at least 16 bits
uint_fast16_t	The fastest unsigned integer with a width of at least 16 bits
int_fast32_t	The fastest signed integer with a width of at least 32 bits
uint_fast32_t	The fastest unsigned integer with a width of at least 32 bits
int_fast64_t	The fastest signed integer with a width of at least 64 bits
uint_fast64_t	The fastest unsigned integer with a width of at least 64 bits
float16_t	A signed fractional number with at least 16 bits of precision
float32_t	A signed fractional number with at least 32 bits of precision
float64_t	A signed fractional number with at least 64 bits of precision
float128_t	A signed fractional number with at least 128 bits of precision

1. The following definition shall be used for Boolean values.

```
typedef  char  bool_t
```

2. The following definitions shall be used for a complex data values.

```
typedef struct _cplx_least16_t_
{
   int_least16_t  imag;
   int_least16_t  real;
} cplx_least16_t

typedef struct _cplx_least32_t_
{
  int_least32_t  imag;
  int_least32_t  real;
} cplx_least32_t
```

Note that a compiler is free to choose appropriate sizes (ie bit widths) for its target hardware, subject to the restriction that a short and an int are at least 16 bits, a long is at least 32 bits, the number of bits in a short is not larger than the number of bits in an int and the number of bits in an int is not larger than the number of bits in a long.

The purpose of using these data types instead of the inherent C data types is to allow for portable C code that can be migrated across multiple processor architectures and to maintain a valid memory architecture by controlling the mapping of the portable data types to the data types supported by a given target processor's compiler for the given processor architecture.

4.2 Architectural Specific

The purpose of this section is to provide a mechanism for exact bit width representations of variables and their corresponding memory locations. This feature is important so that registers can be correctly represented in C code and to ensure the correct used of C intrinsics when optimizing software for a given target processor.

4.2.1 General

1. The endianness of the processor is assumed to be little endian. If the processor is big endian, the complex data type definitions below must be modified.

4.2.2 Architectural Specific Data Types

1. The following data types must be used for developing architectural specific C code.

Type Name	Description
bool_t	A char width
char	A character
int8_t	A signed integer with a width of exactly 8 bits
uint8_t	An unsigned integer with a width of exactly 8 bits
int16_t	A signed integer with a width of exactly 16 bits
uint16_t	An unsigned integer with a width of exactly 16 bits
int32_t	A signed integer with a width of exactly 32 bits
uint32_t	An unsigned integer with a width of exactly 32 bits
int64_t	A signed integer with a width of exactly 64 bits
uint64_t	An unsigned integer with a width of exactly 64 bits

1. The following definition shall be used for Boolean values.

```
typedef  char  bool_t
```

2. The following definitions shall be used for a complex data values

```
typedef struct _cplx16_t_
{
  int16_t  imag;
  int16_t  real;
```

```
} cplx16_t

typedef struct _cplx32_t_
{
  int32_t  imag;
  int32_t  real;
} cplx32_t
```

5 Naming Conventions

This section defines the naming conventions for variables, functions, constants in C.

5.1 Variable Naming Conventions

1. Local variable names must be mixed case starting with lower case.

   Examples: fruitType, vegetableType

2. Global variables must be preceded by the letter g.

   Example: gFruitType, gVegetableType

3. Variables must never have dual meaning within the scope of the variable

   Example: i should not be used as a loop variable in one portion of code and as a current value in another portion of code within the same scope

4. Type names must start with a module name, in all capital letters, followed by an underscore. (e.g. MODULE_). The remaining type name must be mixed case starting with upper case.

   Examples: MODULE_FruitTypes, MODULE_VegetableTypes

5. Constants must start with a module name, in all capital letters, and separated by an underscore. Constants must be all upper case using an underscore

   Examples: MODULE_APPLE_COLOR, MODULE_CUCUMBER_COLOR

6. Object names must start with the object name (e.g. OBJECT), in all capital letters, and end with _Obj.

   Example: OBJECT_Obj

7. Handle names must start with the object name (e.g. OBJECT), in all capital letters, and end with _Handle. Note, a handle is just an object pointer.

   Example: OBJECT_Handle

8. Class names must be mixed case starting with upper case.

   Examples: FruitClass,VegetableClass

9. Acronyms must not be in upper case when used as a name.

   Example: genDspCode() instead of genDSPCode()

10. Quantity variables must contain a consistent prefix.

    Examples: numApples, nApples

11. Entity variables must contain a consistent suffix.

    Examples: apple_i, appleNum

12. Loop variables must have meaningful names.

    Examples: appleCnt, cucumberCnt (instead of i,j)

13. Abbreviations in names must be avoided.

    Examples: computeAverage() instead of compAvg()

14. Negated Boolean variables must be avoided.

    Examples: isError instead of isNoError

15. Generic variables must have the same name as their type.

    Examples: void displayFruitType (Fruit fruit)

16. Pointer variable names must start with the letter p.

    Examples: int *pApples

17. Pointers and references must have their symbol next to the variable name and not next to the data type.

Example: int *pApples instead of int * pApples.

18. Const must be used for all constant function arguments.

    Examples void displayFruitType(const Fruit fruit)

19. Enumerated constants must be prefixed with a common name and the first element must be initialized.

    Example:
    typedef enum
    {
    FRUIT_Apple = 0,
    FRUIT_Orange
    } Fruits_e;

20. Enumeration type definitions must be followed by _e.
21. Type definitions must be followed by _t.
22. Any variable representing a fractional value must be defined using an _iqX data type except for values in GLOBAL_Q format.

5.2 Function Naming Conventions

1. Function names must start with a module name, in all capital letters. Following the underscore, function names must start with a verb and must be mixed case starting with lower case.

   Examples: MODULE_getFruitType, MODULE_computeAverageAppleSize()

2. The terms get/put is used when a function accesses an entity directly.
3. The term compute is used when a function computes an entity.
4. The term find is used when a function searches for an entity.
5. The term is is used for Boolean functions.
6. The term init is used when a function initializes an entity and must return a handle to an entity.
7. The term setup is used when a function configures an entity for operation.
8. The term run is used when a function runs an entity.
9. The term gen is used when a function generates an entity.
10. The term enable/disable must be used when a function turns on/off an entity.
11. The term set/clear must be used when a function sets or clears a state from an entity.

5.3 Constants

1. The following values must be used for true(1) and false(0).
2. The following values must be used for pass(1) and fail(0).

5.4 Statements

1. Comparisons must not be made to known numerical values. Instead, assign the value to a variable and make comparisons to this variable.

   Example: if(fruit == FRUIT_Apple) instead of if(fruit == 0)

2. Loop test variables must be initialized immediately before the loop.

5.5 Coding Syntax

This section described the coding syntax for the C code itself.

1. No spaces shall exist between unary or primary operators (->, ., [], (), sizeof, etc.)

   Examples:
   i++
   !e
   (char *)p
   -17
   sizeof(uint16_t)
   a[i]
   s.member
   p->member
   f(x,y)

2. One space shall exist between other operators and their operands

   Examples:
   x = f(y) * (z + 2);
   p != NULL && (p->member == -1)

3. Parentheses are to be used to improve clarity of expressions with several precedence levels
4. Nested operations will always use parentheses to define operation order

   Example:
   (x < y) & mask
   Expressions with side effects are not to be used in function calls

Example:
```
t = min(x++, y); //WRONG
t = min(x, y);   //RIGHT
x++;
```

5. When breaking long expressions into more than one line, the beginning of each new line must start with an operator and be indented eight (8) spaces.

   Example:
   ```
   if(MODULE_variableName == (MODULE_DEFINE_1 || MODULE_DEFINE_2
         || MODULE_DEFINE_3)
          {
             //Do something
           }
   ```

6. All subordinate statements are enclosed in braces and indented four (4) spaces
7. One statement will exist per line except in the vae of the for() statement, which can have multiple statements with the for() statement.
8. Left and right braces which follow a statement that opens a subordinate set of statements will be fully left justified with both braces on their open line.

   Example
   ```
   if(variable == CONST)
   {

   //Do Something
   }
   ```

9. Left and Right braces which open and close a function will be fully left justified with both braces on their own line.

   Example
   ```
   void FOO_getBar(void)
   {
   //Do Something

   return;
   }
   ```

10. All functions will have an explicit return statement. No spaces will exist between the return and “()”.

Example
return(value);

11. The use of switch statements should be minimal. Use if-else statements instead.
12. All public or externally callable functions (API functions) shall be extern'ed and prototyped in a header file corresponding to the function's module name.

Example
extern void FOO_getBar(void);

Example
static inline void FOO_getBar(void)
{
// Do Something

return;
}

13. Function declarations shall follow the single space after comma rule.
14. There shall be a single blank line between declarations in a function body and the first statement in the function. If there are no declarations a single blank line will follow the opening brace.

Example
void FOO_averageBar(uint8_t bar)
{
static uint8_t barAccum;

barAccum += bar; //Note blank line above this line

return; //Note blank line above this line
}

15. Macro definition must follow one of the following forms:

a. Single Token
#define QUE_enqueue QUE_put
b. Function or Macro call
#define C62_enableGie(mask) HWI_restore(mask)

c. Enclosed in Parentheses

```
#define C62_mask(devid) ((uint16_t)(1 << (devid) – 1))
```

d. Macro function definitions are not allowed, use inline functions instead.

Example: Not Allowed

```
#define QUE_remove(elem) { \
((QUE_Elem *)(elem))->prev->next = ((QUE_Elem *)(elem))->next; \
((QUE_Elem *)(elem))->next->prev = ((QUE_Elem *)(elem))->prev; \
}
```

Example: Allowed

```
static inline void QUE_remove(const uint32_t elem)
{

((QUE_Elem *)(elem))->prev->next = ((QUE_Elem *)(elem))->next;
((QUE_Elem *)(elem))->next->prev = ((QUE_Elem *)(elem))->prev;

return;
}
```

16. In all cases, the use of a macro parameter must always be enclosed in parentheses.
17. Braces must be used for one or more lines of conditionally blocked code.
18. Left and right braces must be left justified and on their own line.
19. The indentation inside of braces shall be 4 spaces.
20. The first two lines before the file header and the last line of a header file must be a compiler directive.

Example

```
#ifndef _FILENAME_H_
#define _FILENAME_H_
:
#endif  // end of _FILENAME_H_ definition
```

where FILENAME is replaced by the actual file name in all capital letters.

21. Include statements must be located at the top of a file only.
22. Include statements must be sorted and grouped. A common grouping scheme is hierarchical in terms of system defined, user defined, etc.

23. All source files must contain a header denoting the file name and path.
24. All source files must contain a copyright statement attributing the code to Texas Instruments. The year in the copyright statement refers to the year in which the code was created.
25. Comments will use the C99 comment delimiter (i.e. //).
26. Comments will be indented to the same level as the code.
27. #ifdef and #endif statements will not be indented. Their use must be constrained to the following cases:

    a. Header Files
    b. Debug
    c. Conditionally compiled code (Flash/RAM build configs)

28. The last line of a source file must be the following statement // end of file followed by a newline character .
29. Bit-fields are not allowed.
30. The break statement can only be used in switch statements.

6 References

1. Programming languages – C, International Standard ISO/IEC 9899:1999(E), Second Edition, December 1, 1999.

7 Appendix A

A. Example Header File

This section provides a standard header file, whose structure should be followed by each module. Portions of this example should be modified to represent the specific module. From example, replace MODULE and module with the appropriate module names.

```
#ifndef _MODULE_H_

#define _MODULE_H_

//#############################################################################
//

//! \file   file_path_from_root_of_sw_package/module.h

//!

//! \brief  Contains examples of the MCU coding standard implemented in a

//!         header file

//

//  Group:            C2000, MSP430, Stellaris, etc.

//  Target Device:  TMS320F28069

//

//  (C) Copyright 2013, Texas Instruments, Inc.

//#############################################################################

// $TI Release: PACKAGE NAME $

// $Release Date: PACKAGE RELEASE DATE $

//#############################################################################
```

```
//*****************************************************************************
// the includes
//*****************************************************************************

//!
//! \defgroup MODULE

//!
//! \ingroup MODULE
//@{

#ifdef __cplusplus
extern "C" {
#endif

//*****************************************************************************
// #defines
//*****************************************************************************

//! \brief Description of this define
//!
#define MODULE_REGISTERNAME_BITFIELDNAME (1 << 4)

//! \brief Description of this define
```

```
//!
#define MODULE_clearState STANDARD_clear

//! \brief Description of this define
//!
#define MODULE_putData(data) UART_putData(data)

//! \brief Description of this define
//!
#define MODULE_maskData(data) ((data >> 8) & 0x00FF)

//*****************************************************************************
// typedefs
//*****************************************************************************

//! \brief Description of this enumeration
//!
typedef enum
{
    MODULE_enumeration_name1_1 = 0,    //!< Denotes enumeration state 1
    MODULE_enumeration_name1_2,        //!< Denotes enumeration state 2
    MODULE_enumeration_name1_3         //!< Denotes enumeration state 3
} MODULE_enumeration_name1_e;


//! \brief Description of this enumeration
```

```
//!
typedef enum
{
  MODULE_enumeration_name2_1 = (1 << 0),     //!< Denotes interrupt number 1
  MODULE_enumeration_name2_2 = (1 << 1),     //!< Denotes interrupt number 2
  MODULE_enumeration_name2_3 = (1 << 2),     //!< Denotes interrupt number 3
  MODULE_enumeration_name2_4 = (1 << 3),     //!< Denotes interrupt number 4
  MODULE_enumeration_name2_5 = (1 << 4),     //!< Denotes interrupt number 5
  MODULE_enumeration_name2_6 = (1 << 5),     //!< Denotes interrupt number 6
  MODULE_enumeration_name2_7 = (1 << 6),     //!< Denotes interrupt number 7
  MODULE_enumeration_name2_8 = (1 << 7),     //!< Denotes interrupt number 8
} MODULE_enumeration_name2_e;


//! \brief Defines the module (MODULE) object
//!
typedef struct _MODULE_Obj_
{
  uint8_t   tmp;   //!< a filler value for the object
} MODULE_Obj;


//! \brief Defines the module (MODULE) handle
//!
typedef struct MODULE_Obj *MODULE_Handle;
```

```
//*****************************************************************************
// globals
//*****************************************************************************

//! \brief Defines the MODULE object
//!
extern MODULE_Obj MODULE;

//*****************************************************************************
// the function prototypes
//*****************************************************************************

//! \brief        Does something which is described here
//! \param[in] handle  The module (MODULE) handle
void MODULE_doSomething(MODULE_Handle handle);

//! \brief     Gets data from an object
//! \param[in] handle  The module (MODULE) handle
//! \return       Object data
inline uint8_t MODULE_getData(MODULE_Handle handle)
{
    MODULE_Obj *module = (MODULE_Obj *)handle;

    return (module->tmp);
} // end MODULE_getData()
```

```
//! \brief        Initializes the MODULE object handle

//! \param[in] pMemory    A pointer to the memory for the module (MODULE) object

//! \param[in] numBytes   The number of bytes allocated for the module (MODULE)
object, bytes

//! \return       The module (MODULE) handle

MODULE_Handle MODULE_init(void *pMemory,const size_t numBytes);



#ifdef __cplusplus

}

#endif // extern "C"


//@}  // ingroup



#endif  // end of _MODULE_H_ definition
```