Embedded Systems Development Coding Standards

Revision History

Draft:	Oct 6, 2015
Released:	Oct 26, 2015

Table of Contents

Introduction

Adhering to a coding standard will ensure that your code is readable, and maintainable by others and will minimize bugs and code that is brittle i.e.: code that when modified suddenly has faults appear related or what may appear at first to be unrelated to modifications made.

Be sure to follow this standard when you write code.

This coding standard has been developed using the Software Engineering Technology (SET) coding standard as a starting point and incorporates the current thinking and best practices of the industry.

PS-1 Meaningful Identifiers

Create meaningful identifiers that can instantly be understood by any software developer who will ever look at your code.

Whenever possible, use the same words to form identifiers in your programs that you use in analysis and design.

Spell words out. Do not use arbitrary abbreviations, because they “mk the cd hrd to rd.”

It is acceptable to use single-letter names for the loop control variable in counted loops (for i = 0 . . .).

Do not hesitate to use multiple words in identifiers.

Only use acronyms that can be immediately understood by any software development professional. Spell out other acronyms.

Organization- or project-specific acronyms are acceptable in advanced courses, provided they can be immediately understood by anyone on the software development and maintenance teams

These standards encourage long identifiers, within reason. In general identifiers should be less than 30 characters.

Change names generated by tools to meaningful identifiers if the name is used in code.

Do not qualify more than 3 variables with a number suffix (department1, department2, department3). Use an array instead.

See Also: http://www.ganssle.com/articles/namingconventions.htm
http://www.ganssle.com/video/episode6-naming-conventions.html\

PS-2 Identifier Case Conventions

Constants - use Upper Case if #define is used
e.g:

```
#define IMAGEARRAYSIZE 1024
```

Use camelCase for variables that use the const keyword

e.g:

```
const int8_t  speakerPin = 2;
```

Function Names - Use camelCase:
e.g:

```
myFunction();
turnLedOn();
getSerialData();
```

Variables - Use camelCase.
e.g:

```
Char lcdLine1[] = "Hello World";
int8_t elapsedSeconds = 0;
```

Enumeration -use Upper Case
e.g.:

```
enum  DaysOfTheWeek
{
   SUN,
   MON,
    . . .
};
```

PS-3 Source Code Commenting

One of the most important things that a software developer can do is to comment their code. Source code commenting is done through Header Comments and Inline (or Body) Comments.

Header comments come in 2 different forms : File Header comments and Function Header comments. Use the following header comment formats when programming.

PS-3.1 File Header Comment

File and Function Header comments are very important in conveying information about your thought process to the reader of your code. File Header comments are found at the start of each source file you create (whether that be a C, C++, C#, etc. source file, a BASH shell script, an HTML web-page, …).

In C, a File Header Comment looks like :

```
/*
*  FILE          : mySourceFile.c
*  PROJECT       : PROG1345 - Assignment #1
*  PROGRAMMER    : Joe Student
*  FIRST VERSION : 2012-05-01
*  DESCRIPTION   :
*    The functions in this file are used to …
*/
```

In C, a Function Header Comment looks like :

```
//
// FUNCTION      : myFirstFunction
// DESCRIPTION   :
//   This function calculates tax on a
//   retail purchase in Ontario.
// PARAMETERS    :
//   double purchaseAmount : untaxed amount
//   double hstTaxRate     : HST tax rate
//   double pstTaxRate     : Provincial tax rate
// RETURNS       :
//   double : total tax amount (includes HST and
//            Provincial sales tax) or a value of
//            -1.00 on error
//
```

PS-3.2. Inline / Body Comment

Inline (or Body) Comments are those comments placed within a function or method. These comments are used to inform the reader about major code blocks, difficult to understand or obscure code, about workarounds that were implemented for unexpected problems, etc. Generally inline comments are used to document the flow of logic within your source code.

Inline comments can be found before a collection of statements or at the end of a single line of code. Typically, comments found at the end of a single line of code are meant to be short and to the point (a few words maximum) and use the "//" style of comment delimiter. If you have more to explain than can be stated in a few words, then use the "/* */" style of block commenting and place the block before the area of code you are describing.

The example on the next page shows the body of a function in C and the types of inline / body comments that might exist.

```
*  FUNCTION      : myFirstFunction
*
*  DESCRIPTION   : This function calculates tax on a retail purchase in Ontario.
*
*  PARAMETERS    : double purchaseAmount : untaxed amount
*                  double hstTaxRate     : HST tax rate
*                  double pstTaxRate     : Provincial tax rate
*
*  RETURNS       : double : total tax amount (includes HST and Provincial sales tax)
*                           or a value of -1.00 on error
*/
double myFirstFunction(double purchaseAmount, double hstTaxRate, double provTaxRate)
{
    double totalTaxAmount = 0.00;    // the running total of tax amount
    double workingTaxRate = 0.00;    // used to hold the effective tax rate being charged
    int    isSpecialPSTItem = 0;     // used to indicate the item is PST-only


    /* On July 1, 2010 the HST took over the "tax-rate du jour" in the
     * Province of Ontario.  The HST replaced the PST and GST that were being
     * added to most retail items.  However, after this date, there are still some
     * items that only have PST charged on them.
     * This function needs to be updated to determine which those PST-only items
     * are and calculate the totalTaxAmount correctly.                              */
    if( isSpecialPSTItem == 1 )
    {
        // function still needs to do this ...
    }
    else
    {
        // make sure we were given valid parameter values
        if(( purchaseAmount > 0.00 ) && ( hstTaxRate > 0.00 ))
        {
           /* Ontario has a 13% HST tax rate - the person calling this function
            * may choose to pass this value is as 0.13 or 13.00 - we need to detect
            * and handle this                                                       */
           workingTaxRate = hstTaxRate;
           if( hstTaxRate > 1.00 )   // person passed in 13.00 style amount
           {
               workingTaxRate = hstTaxRate / 100.00;
           }
           totalTaxAmount = purchaseAmount * workingTaxRate;
        }
        else
        {
           // we received some invalid input - signify an error
           totalTaxAmount = -1.00;
        }
    }
    return( totalTaxAmount );
}
```

PS-4 Variables and Magic Numbers

There should only be one variable declaration per line of code.

Do this:

```
Int32_t   elapsedSeconds =    0;
Int32_t   elapsedHours =      0;
Int32_t   elapsedDays =       0;
```

Not this:

```
Int32_t elapsedSeconds, elapsedHours, elapsedDays = 0;
```

As only elapsedDays will be initialized to 0.

This is also considered acceptable:

```
Int32_t elapsedSeconds=0, elapsedHours=0, elapsedDays = 0;
```

Variables need to be initialized at the time of declaration.

Use constants instead of literals (magic numbers) within your code. There are exceptions to this rule depending on the context.

Do this:

```
const float scalingFactor = 5.0/1024.0; //0-5V scaling factor
```

not this:

```
#define O25VSF 0.004882813
```

Avoid global variables whenever possible. They can make your code tough to

PS-5 Source Code Style and Formatting

There are many facets to how you should and shouldn't format your source code … what style you need or should use … The following guidelines will help you ensure that any source code you produce will be readable and maintainable.

PS-5.1 Indentation

Consistently indent (and brace) the body of classes and interfaces, functions / methods, and other control flow code

PS-5.2 Line Length and Wrapping

Q: How long is too long?

A: If you find that the line of code makes you scroll to the right (within your editor or IDE) – or when you print your source code – the line wraps onto the next printed line then your line of code is too long!

Q: What can you do about it?

A: If you have a single line of code that is very long, then you need to split the line so that it doesn't cause scrolling to the right.

Stack lines of code so that they do not scroll off the screen or wrap when printed, and indent the second and subsequent lines.

e.g.: consider the following line of code :

```
if ((((controlTemp & tempSetting) || ((airPressure > airPressureLimit) && (lightVariation > LI
```

The line of code is so long, it doesn't even fit in the document! So imagine how hard it would be for someone to read, follow and understand your code!

A better way of writing this line of code would be :

```
if ((((controlTemp & tempSetting) ||
       (airPressure > airPressureLimit) &&
       (lightVariation > LIGHT_LIMIT))) &&
       (humidityLevel > DRY))
```

Or maybe this way to show the levels of brackets and their associations :

```
if (
       (
         ((controlTemp & tempSetting) ||
          ((airPressure > airPressureLimit) &&
(lightVariation > LIGHT_LIMIT)
         )
       ) && (humidityLevel > DRY)
    )
```

The point is, you want to make sure that you break longer lines of code up, so as to not have to scroll to the right and left while reading the code

If your long line of code is as a result of adding a comment to the end of a line of code, then change your same-line comment to a comment before the line of code :

In this case the comment makes the line of code too long. We need to comment and code as shown here :

```
// check to see if the current temperature has reached the control setting value
if (controlTemp & tempSetting)
```

We often run into these long lines of code when we are declaring and defining functions. In this case, we simply split the function declaration (or definition) over multiple lines by have one parameter per line

```
double mySecondFunction( int controlTemp,
                         int tempSetting,
                         double airPressure,
                         double airPressureLimit,
                         int lightVariation,
                         int humidityLevel);
```

PS-5.3 Blank Lines and Code Crowding

Remember that the name of the game is to make your code readable and maintainable:

Leave at least one blank line after code modules and control flow constructs

Use blank lines to separate logical blocks of code

Always code the control flow constructs over multiple lines rather than crowding them onto a single line

Ensure that your code blocks are enclosed in curly braces spread over multiple lines.

Do this:

```
for(i = 0; i < LIMIT; i++)
{
    someValue++;
}
```

Not this:

```
for(i = 0; i < LIMIT; i++) someValue++;
```

PS-5.4 Operator and Delimiter Spacing

When it comes to writing each line of code, place a blank space on each side of an operator. This makes the code more readable.

Do this:

```
totalPurchaseAmount = purchaseAmount * (1.00 + (hstTaxRate / 100.00));
```

Not this:

```
totalPurchaseAmount=purchaseAmount*(1.00+(hstTaxRate/100.00));
```

Also leave one blank space after a delimiter such as a comma or semi-colon (e.g. in a parameter list or for statement)

Do this:

```
double myFirstFunction(double purchaseAmount, double hstTaxRate, double provTaxRate);
```

and

```
for(i = 0; i < LIMIT; i++)
```

Not this:

```
double myFirstFunction(double purchaseAmount,double hstTaxRate,double provTaxRate);
```

and

```
for(i=0;i<LIMIT;i++)
```

PS-5.5 Curly Braces

The recommended curly brace formatting is as follows :

Align an opening curly brace under the first letter of the code above

Align closing braces with their matching opening counterparts

The opening and closing curly brace are alone on the line

This is the recommended formatting as Visual Studio supports this by default

```
if( someVariable > SOME_LIMIT)
{
    // do something here ...
    x = x + 5;
}
```

Another curly brace formatting style is known as the "K&R Style" (for Kernighan and Ritchie)

In this style, the opening brace is located on the line of code that the block opens with

The closing brace is aligned with the first let of the line of code

```
if( someVariable > SOME_LIMIT) {
    // do something here ...
    x = x + 5;
}
```

In the final style, the curly braces are aligned with the code in the block

```
if( someVariable > SOME_LIMIT)
    {
    // do something here ...
    x = x + 5;
    }
```

Regardless of the style of curly braces you choose – it is important that the same bracing style be used throughout your source code (see PS-5.1)

Always include curly braces in control flow statements – even if only one line of code is enclosed!

This is done for consistency, readability and maintainability … after all at some point in the source code's life there may need to be more than one line of code in that block. In coding this way initially, no confusion can result.

Do this:

```
If(alarmCondition)
{
     soundAlarm();
}
```

Not this:

```
If(alarmCondition)soundAlarm();
```

PS-5.6 Source Code Functions

Functions should only do the task their names suggest. Like this:

```
getSerialData();
displayRPM();
```

Functions ideally should fit on one screen. They rarely have to exceed 50 lines. If your functions require you to scroll down to view them in their entirety they are probably too long.

Functions should only have one return statement i.e.: a single point of exit.

Remove any disabled code (code enclosed in comments) before submitting your work for marking. As well, remove any misleading or incorrect comments

PS-5.7 Unit Testing

There is no explicit requirement in the ESD Coding Standard that says you must unit test your solution before submission, but the solution should be unit-testable:

Design and implement in a modular manner with well documented functions.

Remember that in a good solution, there is very little code that actually ends up in the main() function. The main() function acts as the puppet-master – calling upon other functions to perform the work.

PS-5.8 Known Issues / Bugs List

You are required to submit a Known Bugs and Issues text file with every coding solution. Use this document to tell the person marking the solution about :

Known bugs and issues in the solution

Assignment requirements that haven't been fully implemented

This document (titled bugs.txt) will contain your name, the date, which assignment the text file covers as well as the known bugs and issues and missed requirements in the following format.

```
Assignment  : PROG1345 C-Programming Assignment #4
Name        : Joe Student
Date        : October 1, 2012

BUGS/ISSUES
===========
 - the first time you open a file for reading the program works, but
   attempts to open another file fail
 - calculateTax() function has an issue that sometimes makes the
   calculation wrong in the decimal places

MISSED REQUIREMENTS
===================
 - did not implement menu choices 5, 6 and 7
```

PS-5.9 Functions – Declaration vs calling

When declaring a function always place a space after the function name and before the opening bracket.

e.g.:

```
Int8_t myFunction (int8_t val1, int8_t val2)
{
      //function body
      .
      .
      .
}
```

When calling the function place the bracket up against the function name without a space.

```
Void main (void)
{
      myFunction(2,5);
}
```

If you follow this convention it makes it easy to find function declarations with in your code. Simply search for “myFunction (“ -- searching for “myFunction(“ will find all the function calls to myFunction.

PS6 Conditional Expressions

Don't group < and > into an expression. While it makes sense and is entirely correct from a mathematics viewpoint, it is not from a programming view.

e.g.: while 20<x<40 is clear mathematically and states that x should be greater than 20 and less than 40, that is not how the coding will work

Consider:

```
int8_t x = 10;
if(20 <x<40)
{
      printf("x = %d and %d  is greater than 20 and less than 40",x,x);
}
```

The above code will print "x =10 and 10 is greater than 20 and less than 40" which is clearly not the case.

Either the C-compiler will detect during compilation that the first part of the expression 20<x will evaluate to either 0 or 1. And that means the second part of the expression will evaluate to:

0 < 40 or 1 < 40

Which in both cases is true and the code will always execute.

PS7 Global Variables

Minimize the use of global variables, use them only when absolutely necessary. Global variables make your code difficult to troubleshoot.