Embedded C Coding Standard

M J Bauer 2012

Contents

1 Introduction

This standard defines “firmware” as software written specifically for embedded micro-controllers, typically with a minimal “operating system” (real-time kernel), if any. This document is concerned with coding practices and conventions applied to source code written in the C programming language.

The words "***shall***", "***should***" and "***may***" have special meanings in this standard:

- "***Shall***" is a mandatory directive, which must be followed.
- "***Should***" is a strong recommendation. It is expected that it will be followed, but there may be circumstances where it is not preferable.
- "***May***" is a more flexible guideline.

2 References

…

…

…

3 Definitions

TERM	**DEFINITION**
MCU	Micro-Controller Unit (a micro-processor with on-chip memory and peripherals)
IDE	Integrated Development Environment (Software for computer workstation)
ISR	Interrupt Service Routine

4 Program Structure

The topic of Program Structure is intended to be covered in a separate document.

5 Organisation

5.1 Source File

The source file containing the function `main()` should be named `"main-<whatever>.c".`
The `<whatever>` part is optional, but helps identify the application.

A source file should not exceed 2000 lines, except files defining a large data object, where it would be awkward to split a file, e.g. a graphics image, font glyph table, etc.

Every C source file shall begin with a completed comment block containing at least the filename, project name, company name, copyright notice, author, date created and overview:

```
    /**
     * Module      : <filename>.c
     * Project     : ...
     * Originated  : 20yy.mm.dd  MJB -- Copyright XYZ-Co Pty Ltd
     *
     * Overview    : ...
     *               ...
     */
```

5.2 Functions

Every major function definition should be preceded by a completed comment block containing at least the function name, overview (brief description), entry arguments (if any) and return value:

```
    /*****************************************************************************
     * Function:     <name>
     *
     * Overview:     <brief description>
     *
     * Entry args:  <type and purpose of each argument>
     *
     * Return val:   <type and purpose of return value>
     */
```

A function should not exceed 100 lines of code (excluding title comment block).

Detailed function descriptions should be placed in a separate firmware architecture document.

5.3 Usage of Tabs

Tabs may be used for leading indents, but shall not be used anywhere else on a line.

5.4 Pre-processor Directives

Pre-processor directives should all begin at the left margin, i.e. not indented:

```
    #ifdef USE_TEST_CODE_A
    :
    :
    #endif // USE_TEST_CODE_A
```

Where there are many lines of code following `#if, #ifdef`, the matching `#endif` should have a trailing comment (as above).

Nesting of `#if` and `#ifdef` directives should be kept below 3 levels.

5.5 Usage of Braces

Matching braces {…} shall be on the same indent level, except where the statement(s) within the braces can fit on one line.

5.6 Code Formatting

Source lines shall not exceed 120 character places and should not exceed 100.

Leave one or two blank lines (no more than two) between functions.

Leave a blank line between logical bunches of code lines to improve readability.

Don't insert more than two blank lines anywhere.

5.7 Usage of Whitespace Characters

A space shall be inserted after keywords.

A space shall be inserted on either side of a binary operator:

```
    roo = wombat + tail;   // correct
    emu = emu+1;           // prohibited
    mask = mask &0xFF;     // prohibited
```

…except when using '+' or '-' in an expression for an array index:

```
    row = table[i+1];      // permitted
```

A space shall be inserted on either side of assignment operators (`=, +=, &=`, etc) and comparison operators (`==, >=, <=, !=, &&, ||`):

```
    if (signal != 0) result = TRUE;    // correct
    if (signal!=0) result=TRUE;        // prohibited
```

No space shall be inserted between a unary operator and its operand, e.g.

```
    mask = mask & ~(1 << 4);    // correct
    if (!isprint(c)) ...        // correct
    data = *pData;              // correct
    mask = mask & ~ BIT4;       // prohibited
    pointer = & packetData;     // prohibited
    dat = * pData;              // prohibited
```

Casts are not subject to the same rules as other unary operators, e.g.

```
        temp = (float) sensorReading;   // a space after a cast is optional
```

No space should be inserted between an array identifier and its index expression, e.g.

```
        value = lookup [idx];    // non-preferred
```

No space should be inserted between a function identifier and its argument list, e.g.

```
        putch (c);               // non-preferred
```

If there is only a single short statement following "if (...)", it may be on the same line, e.g.

```
        if (flag) c = '0';       // permitted
```

5.8 Loops

Loop constructs shall employ braces, even if the loop contains only one statement, or none:

```
while (signal == 0)   // preferred
{
    // WAIT FOR SIGNAL
}

while (signal == 0) { /* WAIT FOR SIGNAL */ }   // acceptable

for (count = 0; count < 20000; count++);  // prohibited
```

6 Commenting Style

Code should be written such that it is largely “self commenting”.

Use meaningful identifier names which help to annotate the code and structure the code so that it is readable.

Comments shall be included to explain what is not obvious from reading the code.

Comment lines shall be at the same indent level as the next code line following.

Exception:

Where `//` is used to comment out a code line, the `//` shall be at the left margin, but the code (text) must be correctly indented (as if it were not commented out):

```
//    while (signal == 0)   // preferred
//    {
//          count = 0;
//    }
```

Use `“/** .. */” or “//== .. =”` for multi-line function banners, module title blocks, etc. Use, sparingly, if at all, to separate functions or logical groups of functions as overuse unnecessarily bloats the file size.

```
/****************************************************************************/
//============================================================================
```

Use `//` (line-style comments) everywhere else.

7 Naming Conventions

Use Pascal Case (aka "Title Case") for Function Names, e.g.:

```
    InitializeSystem();
```

Use Camel Case for other identifiers, e.g.:

```
    ledTurnOnDelayTime;
```

NOTE: ***Do not use Hungarian Notation.*** *This is no longer fashionable! Examples…*

```
    chInitial      // char, "Initial"
    fpPrice        // floating point, "Price"
```

Use prefixes to indicate the scope of variables, e.g.

```
    extern  type  g_globalVariable;              // Global
    static  type  m_privateToCodeModule;         // Within the source file
    static  volatile type  v_volatileVariable;   // Shared with ISR
```

Use suffix "`_t`" (and/or Pascal Case) to indicate `typedefs`, e.g.:

```
    typedef  struct  ComplexNumber
    {
        float alpha;
        float sigma;
    }  ComplexNumber_t;
```

Variables and function arguments representing signals and measurable quantities (voltage, time, distance, speed, mass, force, etc) may have a suffix appended to their names indicating their units. Examples:

```
        mainPhotoDetectorReading_mV
        preSampleDelayTime_ticks
        motorPosition_usteps
```

Symbolic constants (including enumerated values) shall be all uppercase using underscore to separate words, e.g.

```
        MAX_GROUP_NUMBER, CONNECTION_TIMEOUT
```

Other macro definitions should also be uppercase, with exceptions, e.g. common typedefs:

```
    uint16
    int32
    bool, etc.
```

Variables with a broad scope should have meaningful names; variables with a small scope may have short names, e.g. `i, j, k` for loop variables & array indexes, `c` for char.

Underscores may be used to separate logical parts of long identifier names, e.g.:

```
    uint32  rotorStepperMotor_HomePosition;

    #define USE_HARDWARE_CONFIGURATION_XYZ  (TRUE)
```

Where there is a set of lengthy identifiers, including function names sharing many common word combinations, their names should be constructed with the "most significant" words first, followed by "less significant" words. In the following example, the words "Opto", "Detector" and "Reading" are common to a number of function names:

```
    int  OptoDetectorMainReadingFetch(...);      // preferred
    int  FetchMainReadingFromOptoDetector(...);  // discouraged
```

In this way, identifiers are logically grouped and more easily categorized, e.g.

```
    OptoDetectorMainReadingFetch();
    OptoDetectorMainReadingStore();
    OptoDetectorAuxReadingFetch();
    OptoDetectorAuxReadingStore();
```

8 Declarations

Header files (*.h) shall include a construct that prevents multiple inclusion, e.g.

```
    #ifndef SYSTEM_H    //first statement in header file "system.h"
    #define SYSTEM_H
     :
     :
    #endif // SYSTEM_H    //last line in header file
```

Header files shall not include source code which generates executable object code, or which generates or allocates data storage in an object module. Otherwise, there is no practical distinction between a header file and a code file.

Exception: Functions declared as "inline" may be defined in a header file.

`#include` directives shall be located at the top of a file only.

Header files (*.h) shall not `#include` C source files (*.c).

C source files should not `#include` other C source files.

Individual source files should compile cleanly in isolation from other source files. In other words, successful compilation of a source file should not be dependent on prior compilation of others.

In general, scope of variables should be kept to the minimum necessary.

Module private (`static`) data may be made accessible to external modules by means of function calls. This is preferable to using global variables.

Local (auto) variable declarations shall be placed at the top of the function body, before any executable statements, with the following exception.

In a `switch` statement, local variables may be declared within individual cases, provided that the case body is enclosed within braces and the variable definitions are placed at the top of the case body (as in a function definition).

```
    case SET_CAL_FACTOR:
    {
        uint8  *pktData = (uint8 *) m_packetBuffer + MSG_HEADER_SIZE;
        float   paramVal;

        memcpy((uint8 *) &paramVal, pktData, sizeof(float));

        if (paramVal >= 0.5 && paramVal <= 2.0)
        {
            ParamBuffer.calFactor = paramVal;
            g_storePersistentData = TRUE;    // trigger B/G task
        }
        break;
    }
```

A blank line shall be left between local variable declarations and the first line of the body of code in which they are referenced, as in the above example.

Variables and function arguments representing signals and measurable quantities (voltage, time, distance, speed, mass, force, etc) shall have a comment in their declaration indicating the units of the variable (mV, us, MHz, mm, steps/s, kg, etc), except where a variable name (suffix) explicitly indicates the unit of measurement.

9 Expressions and Statements

Parentheses should be used to guarantee correct evaluation of expressions. This also helps in code reviews to make the intent of the expression more obvious.

Switch statements shall not be nested.

There should be no more than one `return` statement in a function.

Assignments and operand arithmetic should not be put in conditional expressions, e.g.

```
    if ((c = getch()) == ASCII_ESC) ... ;   // not recommended

    if (++counter >= TOP_COUNT) ... ;       // not recommended
```

Convoluted pointer expressions shall be avoided.

Pointers should not exceed 2 levels of indirection.

Pointer arithmetic, other than these examples, should be avoided:

```
    pointer++;
    pointer--;
    pointer += CONSTANT;    // where CONSTANT is of type integer
    pointer -= CONSTANT;
    pointer += expression;  // where expression evaluates to integer
    pointer -= expression;
```

If an expression following “`if (...)`” or “`else`” contains a macro reference, the expression should be contained within braces to guarantee the intended program flow, e.g.:

```
    if (temp > 50.0) { TURN_LED_ON(); }
    else  { TURN_LED_OFF(); }
```

10 Portability

Never use the expression:

```
    if (whatever == TRUE) ...
```

because TRUE may be any non-zero value! If a source file (or function) containing this expression is ported to another application, where TRUE is defined differently, the conditional expression could evaluate incorrectly. The following alternative expression will always evaluate as intended:

```
    if (whatever) ...    // correct test for “whatever is TRUE”
```

Conversely, there is nothing wrong with using the expression:

```
    if (whatever == FALSE) ...
```

Although, the preferred equivalent expression is simply:

```
    if (!whatever) ...   // preferred test for “whatever is FALSE”
```

NOTE: Beware of word-length and "endian" dependencies, particularly where defining data structures for use with inter-platform communications (e.g. packet structures).

Define platform-independent data types in a common header file, e.g.:

```
    typedef signed char     int8;
    typedef unsigned char   uint8;
    typedef signed short    int16;
    typedef unsigned short  uint16;
    typedef signed long     int32;
    typedef unsigned long   uint32;

    #ifndef bool
    typedef unsigned char   bool;
    #endif
```

Low-level hardware-dependent code (peripheral I/O access) should be confined to dedicated "I/O driver" modules, as far as practicable. I/O device driver modules should provide functions for interfacing with higher-level "application code" modules.