Embedded Programming Style Guide

Roger M. Kieckhafer

Rev: 08/28/19

Abstract:
We do not want to go overboard with overly-stringent, straight-jacket standards on programming style. However, painful experience has shown that having a few common conventions really helps with debugging, diagnosis, & grading. Therefore, we do hereby decree that the standards & practices defined herein are mandatory for all software source code and/or Verilog modules written for this course.

This style guide was developed in collaboration with several instructors in both the CS & ECE departments. Thus, most of these standards reflect common contemporary practices that should be familiar to you. Others may be more reflective of this instructor's preferences, but in either case, it is most important that programming standards exist and that they be enforced.

Failure to comply with this style guide will result in deductions from graded work.

- For minor violations of this style guide, points will be deducted from the grade for your code.
- For major violations of this style guide, neither the TA nor the instructor will even attempt to read or grade your code. It will receive a grade of Zero, as if it was not even turned in.
1 Naming Conventions

All names shall be *semantically* meaningful, i.e. descriptive of their meaning or purpose in the context of the *application* or *algorithm*. For example:

- Meaningless names such as $a, b, c$, etc are usually not acceptable.
- Names such as $x, y, z$ are acceptable only if they refer to Cartesian coordinates. And don't even think about using names like $xx, yy, zz$.
- For loop indices, $i, j, k$ are usually just fine, because we are all conditioned to look for them as loop indices. However, we have seen cases where even the author forgets what ordering was intended, especially for nested loops. If confusion is possible, use meaningful names e.g.:
  - $iRow, jCol$ : reveals both semantics & nesting order for a matrix
  - $iMsg, jWord$ : does same for words within a set of messages

1.1 Constant Names

Constant names shall be in *upper* case with words separated by underscores.

- Examples: `MAX_LOOP_COUNT, MIN_TIME_DELAY`.

1.2 Variable names

Variable Names shall be in *Lower Camel* case with no underscores. Specifically:

- The first character of the name shall be in *lower* case.
- After that, the first character of each word in the name shall be in upper case.

- Examples: `loopCounter, prevTimeStamp, aReallyLongVarName`.

1.3 Function and Module Names

Names for functions, procedures, & interrupt service routines (ISRs), and Verilog modules shall be in *Upper Camel* case with no underscores. Specifically:

- The first character of the name shall be in *upper* case.
- After that, the first character of each word in the name shall be in upper case.
- Additionally, function names should represent a *verb* or short *action phrase*.

- Examples: `LogNewMsg(), CalcTimeDelay(), TimeoutIsrResponse()`.

---

1 Our vendors do not follow our conventions (e.g. TI constants have no underscores – very annoying). You're stuck with names & aliases pre-defined by them. However, all names *created or defined by you* must comply with this style guide.
1.4 User Defined Type Names

Type names for user-defined data types (typedef types) shall be formatted as follows:

- The main body of the name shall be in Lower Camel case with no underscores
- The type name shall be terminated by appending “_t”
- Examples: msgFrameBuff_t, usbMsgFormat_t.

1.5 Vendor Defined Type Names

Never use the “standard” C data types (int, long, char, etc). They are not uniformly defined from one compiler to the next. Therefore, vendors provide type aliases which define the word length explicitly in the type definition. Always use these data types instead of standard C types.

1.5.1 For EE-3173 (Altera Nios)

```
#include <alt_types.h> & always use the Altera-defined integer types:
```

- Signed: alt_8, alt_16, alt_32, alt_64
- Unsigned: alt_u8, alt_u16, alt_u32, alt_u64

1.5.2 For EE-4735 (TI MSP430)

```
#include “stdint.h” & always use the TI-defined integer types:
```

- Signed: int8_t, int16_t, int32_t, int64_t,
- Unsigned: uint8_t, uint16_t, uint32_t, uint64_t.

1.6 Union Field Names

C Unions are particularly useful in embedded programming, especially for moving data between objects of different sizes (e.g. streaming from a 32-bit register to an 8-bit I/O port).

- For union declarations, the name declaration should explicitly reveal its length.
- Examples: Altera Nios: TI MSP430:

```
Union
{ alt_u32  u32;
  alt_u8   u8[4];
} msgHeader;
```

```
Union
{ uint32_t u32;
  uint8_t  u8[4];
} msgHeader;
```

- Field lengths are then clearly obvious to the most casual reader, e.g.:
  - msgHeader.u32 vs. msgHeader.u8[i].
2 General Appearance

There is always disagreement over the more cosmetic aspects of programming. In such cases, it is often less important exactly which standard is enforced, and more important that some standard is enforced for uniformity and readability. Thus, the following standards are mandatory for this course.

2.1 Indentation

Each nesting level shall be indented 2 to 4 spaces from the previous level, no more & no less. Whichever value you choose must be consistent throughout the program (pick one & stick to it).

Never use Tabs to indent any code, as various editors and printer drivers treat tabs in non-uniform manners, thereby ruining the intent of the indentation.

2.2 Line Spacing

Excessive white space between lines only serves to spread the code over more pages, making it harder to read as a whole. Therefore:

- All lines that fall within a logical block of code shall be placed with no blank lines between successive lines of code (think of a “logical block” as being like a paragraph of text).
- Blank lines shall be used between logical blocks of code to help distinguish the blocks (as would be done between paragraphs of text).

2.3 Line Composition

Although there are no hard and fast rules for doing so, make a reasonable attempt to enhance the readability of your code. It will pay off when tracking down a bug or seeking help from the TA.

In particular, parentheses and white space do much to improve readability and clarify the intent of a statement, even when the compiler does not require them. Examples:

- \[a=b+c*d+e*f;\]  // clear to compiler, tedious to read
- \[a = b + c*d + e*f;\]  // easier to read due to spaces
- \[a = b + (c*d) + (e*f);\]  // easier to read due to parens

2.4 Line Wrapping

Long lines shall not be allowed to randomly wrap around to the next line. Rather:

- Line breaks shall be forced at less than 80 characters from the extreme left margin.
- Continuation lines shall not violate the indentation level of the first line (i.e. they may not start to the left of the first line).
- Place your forced line breaks so as to maximize readability of the entire statement, e.g.:

\[
\text{result} = (\text{longVariableName} \ast \text{anotherLongName}) \\
+ (\text{thirdLongName} \ast \text{fourthLongName}) \\
+ (\text{somethingElse} \ast \text{oneLastVariable});
\]
2.5 Braces
For a block of code that requires braces:

- **Every** brace (both opening & closing) shall appear as the *first character on a new line*.
- All braces shall be indented to the same column as the parent (conditional) statement.
- You *may* optionally place the first statement of the enclosed block on the same line as the *opening* brace, properly indented of course.
- You *may* omit the braces around a block of code that comprises only a single-statement.
- **Special Case**: in embedded code, it is often convenient to write an “empty” loop to do nothing but kill time. In this case, it is a good idea to place an empty pair of braces at the end of the line, just before the semicolon, as a visual flag that this really *is* an empty loop. Examples:

  ```c
  while (irqFlag == 0) {};       // wait for external irq signal
  for (i = 0; i < I_MAX; i++) {}; // stall for I_MAX loop passes
  ```

3 Formatting of Comments
Use copious comments, since you will need to re-use some code several weeks after you wrote it. Comments must be meaningful in a *semantic* or *algorithmic* context, not a syntactic context, e.g.:

- **Useless**: \( r = \sqrt{x \times x + y \times y} \); // \( r = \) square root of \( x^2 + y^2 \)
- **Useful**: \( r = \sqrt{x \times x + y \times y} \); // \( r = \) range fm source to target

3.1 General Comment Style
- Use // instead of /* ... */ for all C comments (our C compilers do allow it).
- **Standalone full-line** comments may be used at the start of a block of code to describe the block as a whole. However, they are *not* the preferred method to comment a single line of code.
- **Trailing in-line** comments are preferred for single lines of code. If a comment must continue on a new line, start the next comment line in the same column as the first comment line.

3.2 Opening Comment Blocks
Each main program, function, or procedure, *must* start with an opening comment block. Each opening block must contain *at least*:

- A brief description its major function (at the level of application semantics),
- A list of the arguments and definitions of their *semantic* meanings,
- If there is a return variable, a definition of its *semantic* meaning.

---

\(^2\)Neither the TA nor the instructor will even attempt to read, grade, or help you debug un-commented code.
• For example, something similar to the following will do very nicely for a C function.

```c
alt_32 CopyLst16 (alt_16* srcPtr, alt_16* dstPtr, alt_u8 lstLen)
// Func: Copy a list of 16-bit words from one memory buffer
// to a second memory buffer
// Args: srcPtr = ptr to 1st word of source buffer
//       dstPtr = ptr to 1st word of destination buffer
//       lstLen = length of the list (in words)
// Retn: value of the final word in the list (sign extended)
// -------------------------------------------------------------
{
    body of function goes here...
    return (finalWord);
}
```

4 Nios Assembly Code Conventions (EE-3173 Only)

The following additional requirements apply to Nios assembly language programs.

• Use the Nios “***io” versions of Load/Store instructions when going from/to IO devices. This convention not only makes your code portable to the /f core, but it also significantly enhances readability of the code.

• Follow standard Nios conventions for register usage, including arguments, returns, who pushes and pops which registers (caller or callee), etc. In particular:
  ◦ Do not push registers not used in your function – it’s a waste of time & space.
  ◦ If you push any registers, then properly push & adjust the sp register (in case your function is interrupted).
  ◦ If your function does not call other functions, then you need not push the ra register.

• Opening blocks for assembly programs and functions must contain at least:
  ◦ A brief description its major function (at the level of application semantics),
  ◦ Semantic definitions for all registers used, e.g.:

```asm
CopyLst32:
    # In asm, function name = label name
    # Func: Copy a string of 32-bit words from one memory buffer
    # to a second memory buffer
    # r6 = Source address of current word
    # r7 = Destination address of current word
    # r8 = Number of words remaining to be copied
    # r9 = Temp storage of data word between Load & Store instrs
    # Retn: After return, r9 holds value of final word copied
    # -------------------------------------------------------------
    body of function goes here...
    ret
```

© 2019, R. M. Kieckhafer
5 Verilog-Specific Conventions (EE-3173 Only)

In addition to the generic standards specified above, the following apply to Verilog designs:

- Keywords `begin` & `end` shall follow the same usage & placement rules as braces in C,
- Keyword `endmodule` shall start in the same column as keyword `module`.
- Module & instance names shall be in **Upper Camel** case (like function names),
- In hardware design, I/O pin names are often beyond your control, as they are pre-defined by previous convention (e.g. a, b, z, or D, Q, Ck). However, all user defined names must follow the naming rules defined in Section 1.
- As with C functions, Verilog modules opening comment blocks that must contain at least:
  - A brief description its major function (at the level of application semantics),
  - A list of all input, output & inout variables in the port list and their semantic meanings.

6 Hardcopies

All hardcopies of code handed in shall adhere to the following:

- The font shall be at least **10-points** in size.
- The font shall be **fixed-width** (not proportional width). As a few examples:
  - Acceptable fonts include: courier, courier new, consoLas, & monospace,
  - Unacceptable fonts include: arial, palatino linotype, & times new roman.
- All indentation shall be in compliance with Section 2.1, above.
- All code shall be **single**-spaced, in compliance with Section 2.2, above.
- Uncontrolled line wrapping shall be prevented, in compliance Section 2.4, above.
- Font colors and toner density must be high density, high contrast and easy to read.
  Do not expect us to read a light colored font printed with a weak toner cartridge.\(^3\)
- If you submit a monochrome printout of a color file, use pure black font only (no graytones).

\(^3\) Neither the TA nor the instructor will even attempt to read, grade, or help you debug code on a faint, mis-formatted, or unreadable printout.