EE 382 Style Guide

March 2, 2018

This is a short document describing the coding style for this class. All code written in this class is assumed to follow this coding style.

1 Indentation

Indentations should be 4 characters, using spaces rather than 'hard' tabs (the tab character). The rationale behind this is that the indentation you use should remain constant across different computers and editors. This can be changed by a setting in your text editor or development environment.

Do not put multiple assignments on a single line. Avoid tricky expressions. Both of these are easier in assembly, especially as you cannot have multiple assignments on a single line. Keep this in mind.

Do not leave whitespace at the end of lines. It is tacky and takes up extra space.

2 Breaking Lines/Strings

The purpose of a coding style is to encourage readability and maintainability using a given toolset. As such, the greatest consideration is to maintain legibility. For legacy compatibility, you *may* limit your lines to 80 characters wide, although this is not required.

3 Variable Naming

C and Assembly are spartan languages, and their variable naming should respect that. As such, do not use overly verbose names ('ThisVariableIsATemporaryCounter') where a short name ('tmp') will do. **However**, this is not intended to instruct you to use the shortest name possible. Recall that the point of the coding style is to improve readability — using only single letter variable names defeats the purpose.

Mixed-case names are frowned upon — names with underscores should be used instead. For example, if you have a function that writes data to a LCD display, it should be called something along the lines of `lcd_write_data`, not `lcddtwrt` or some other gibberish. Similarly, all functions, but *especially* global functions, should have descriptive names. See previous.

Local variable names should be short and to the point. If you have some random integer loop counter, it should probably be called `i`. Calling it `loop_counter` is non-productive, if there is no chance of it being misunderstood. Similarly, `tmp` can be just about any type of variable that is used to hold a temporary value.

4 Functions/Subroutines

Functions should be short and sweet, and do just one thing. They should fit on one or two screenfuls of text (the ISO/ANSI screen size is 80x24, as we all know), and do one thing and do that well.

The maximum length of a function is inversely proportional to the complexity and indentation level of that function. So, if you have a conceptually simple function that is just one long (but simple) case-statement, where you have to do lots of small things for a lot of different cases, it's OK to have a longer function.

However, if you have a complex function, and you suspect that a less-than-gifted first-year high-school student might not even understand what the function is all about, you should adhere to the maximum limits all the more closely. Use helper functions with descriptive names (you can ask the compiler to in-line them if you think it's performance-critical, and it will probably do a better job of it than you would have done).

Another measure of the function is the number of local variables. They shouldn't exceed 5-10, or you're doing something wrong. Re-think the function, and split it into smaller pieces. A human brain can generally easily keep track of about 7 different things, anything more and it gets confused. You know you're brilliant, but maybe you'd like to understand what you did 2 weeks from now.

In function prototypes, include parameter names with their data types. Although this is not required by the C language, it is preferred because it is a simple way to add valuable information for the reader.

5 Commenting

Comments are a wonderful invention that allows you to tell what your code does to a reader. There is, however, a danger of over-commenting. **Do not** explain *how* your code functions — it is better to write it in a way that the working is obvious.

As such, you generally want your comments to tell WHAT your code does, not HOW. For functions, you can make small comments to note or warn about something particularly clever (or ugly), but try to avoid excess. Instead, put the comments at the head of the function, telling people what it does, and possibly WHY it does it.

6 Assembly Specific

6.1 Alignment/Indentation

For assembly files, the preferred layout is for each line to be of the form `[label:] [mnemonic] [operands] [; comment]`, a full line comment, or blank. For example:

```
.equ LEDPORT, PORTA
```

```
start:
        out   DDRA, 0x10
        ldi   r16, 0xFA

; Loop until we reach zero
loop:   dec   r16
        breq  done
        in    r26, LEDPORT
        com   r26
        out   LEDPORT, r26
        jmp   loop          ; Return to the beginning of the loop

done:
        nop
        jmp   done          ; Infinite loop!
```

Note the vertical alignment of the mnemonics, operands and end-of-line comments. This may take multiple indents.

6.2 Capitalization

When using assembly directives for labels for various registers, values, or other objects, the label tends to be capitalized. This can allow you to see, at a glance, where you are using global names or (more) local names. However, you may write assembly in all capitol letters, in lowercase letters, or in some dark mix of the two. This class prefers for assembly to be written in lowercase, *but* it is not required due to code examples in the textbook using capitol letters. Remember that the point of a coding style is to increase readability.

7 C Specific

7.1 Indentation

The preferred way to place multiple indentations in a switch is to align the `switch` statement and its subordinate `case` labels in the same column. Then single indent the `case` bodies.

```
switch (var) {
case 'G':
case 'g':
    mem <<= 10;
    break;
case 'M':
case 'm':
    mem <<= 20;
    /* fall through */
default:
    break;
}
```

Do not put multiple statements on a single line unless it simplifies readability. Err on the side of splitting the line. **Do not do this:**

```
if (condition)  do_this;
    do_this_everytime;
```

7.2 Placing Brackets

An issue that often comes up in projects involving C code is the placement of braces. Unlike other issues, there are few technical reasons to choose one style over the other. Thus, the arbitrarily preferred way is to place the opening brace last on the line and place the closing brace on its own line.

```
if (condition) {
   do_this;
}
```

The one special case is for functions: The opening brace is placed at the beginning of the next line. This is indeed inconsistent, but functions are special compared to other braced environments in that they cannot be nested.

```
int main(void)
{
    /* Insert program here */
}
```

Note that the closing brace is on a line by itself, **except** in the cases where it is a continuation of the same statement, i.e. the `while` in a `do-while` statement or the `else` in an `if` statement.

```
do {
    /* ... something ... */
} while (condition);
```

Do not use braces where a single statement will do, but if any branch of a conditional has more than a single statement — then use braces for all branches.

7.3 Spaces

The general guideline is to use a space after most keywords. Thus, use a space after these keywords:

```
if, switch, case, for, do, while, goto
```

but not after:

```
sizeof, typeof, alignof, __attribute__
```

as these are often used with parentheses like `s = sizeof(struct file)`. With regards to parentheses, do not add spaces inside of the parentheses.

When declaring pointer data or functions that reference pointers, place the '*' adjacent to the data or function name and not adjacent to the type name. Example:

```
char *string;
void *memcpy(void *dest, const void *src, size_t n);
void free(void *ptr);
```

Use one space around most binary and ternary operators, such as:

```
=  +  -  <  >  *  /  %  |  &  ^  <=  >=  ==  !=  ?  :
```

but no space after unary operators, or on the inside of prefix or posfix operators. These are:

```
&  *  +  -  ~  !  sizeof  typeof  alignof  __attribute__  defined ++ -- ++ --
```

Also add no space around the `.` and `->` structure member operators.

Do not leave trailing whitespace.

7.4 Macros

Names of macros defining constants and labels in enums are capitalized.

```
#define CONSTANT 0x12345
```

Enums are preferred when defining several related constants.

CAPITALIZED macro names are appreciated but macros resembling functions may be named in lower case.

Generally, inline functions are preferable to macros resembling functions.

Macros with multiple statements should be enclosed in a do - while block:

```
#define macrofun(a, b, c)                       \
    do {                                        \
        if (a == 5)                             \
            do_this(b, c);                      \
    } while (0)
```

Things to avoid when using macros:

1) macros that affect control flow:

```
#define FOO(x)                                  \
    do {                                        \
        if (blah(x) < 0)                        \
            return -EBUGGERED;                  \
    } while (0)
```

is a very bad idea. It looks like a function call but exits the calling function; don't break the internal parsers of those who will read the code.

2) macros that depend on having a local variable with a magic name:

```
#define FOO(val) bar(index, val)
```

might look like a good thing, but it's confusing as hell when one reads the code and it's prone to breakage from seemingly innocent changes.

3) macros with arguments that are used as l-values: `FOO(x) = y;` will bite you if somebody e.g. turns FOO into an inline function.

4) forgetting about precedence: macros defining constants using expressions must enclose the expression in parentheses. Beware of similar issues with macros using parameters.

```
#define CONSTANT 0x4000
#define CONSTEXP (CONSTANT | 3)
```

5) namespace collisions when defining local variables in macros resembling functions:

```
#define FOO(x)                          \
({                                      \
    typeof(x) ret;                      \
    ret = calc_ret(x);                  \
    (ret);                              \
})
```

`ret` is a common name for a local variable - `__foo_ret` is less likely to collide with an existing variable.