C Program Style Guidelines

For use in RBE3001 and RBE3002

Gary Pollice

ABSTRACT

This document provides the guidelines that students are expected to use for all programs written for the WPI RBE3001 and RBE3002 courses. This include programs for homework assignments, labs, and projects.

Table of Contents

1. Introduction

Well-written programs possess several traits. First, they work. Software that does not work is bad software. Second, well-written programs are designed to be flexible and robust, and the design is obvious in the structure of the program. Third, well-written programs are maintainable. Maintainable code is code that others can understand and modify. Code should be readable by any qualified reader—that is, a reader who is competent at writing code in the programming language. This manual provides information for students that will help them write maintainable code.

The manual is intended for WPI students in robotics engineering classes who write C and C++ programs. Developing good programming habits now, on fairly small projects, will prove valuable when you work on larger projects after graduation. Make good programming style a habit. The effort will be worth it.

1.1. General Philosophy

The main goal of using coding style guidelines is to make code readable. Strict adherence to guidelines, if such adherence causes the code to be less understandable, is discouraged. One must use common sense when formatting and commenting code. Code must communicate the programmer's intentions to the reader. You communicate your intentions by using well-chosen names for your functions, variables, and other programming entities, and commenting your code where appropriate.

Comments are necessary, but they must be good comments. A comment that states the obvious is a useless comment. In fact, it is worse than useless, it is a hindrance to understanding a program. Consider the following comment:

```
temperature = 0.0;        // set the temperature to 0
```

This comment adds nothing to the reader's understanding of the program. The line of code stands on its own. It's clear and unambiguous without the comment. The majority of comments in a program should be attached to files and modules, macro definitions (#defines), functions and methods, and variable declarations. Other comments should be used sparingly.

Comments should be written to convey two things to the reader:

1. The purpose of a method, variable, module, etc.
2. The rationale for the code. In other words, tell what forces caused you to write the code as you did, and to explain certain features of the code that are not obvious, such as machine-specific details or how algorithms work.

If there is one overriding principle to writing good code is that the code style must be consistent. Use consistent indentation, punctuation, symbol placement, and commenting style. When working on code that someone else has written, adapt to their style of code so that it looks like it was written by a single person. When working on a team programming project, the team should choose a coding style, make sure that every team member understands the style, and review code regularly to ensure adherence to the style.

This guide does not provide details about specific formatting styles. These can be found on several Web pages. Students are encouraged to find one that suits their dispositions and needs. There are some references to such guides in the bibliography.

The items that are specifically addressed by this guide are those that are most important to writing good code for the RBE course. We assume that some consistent C coding style has been adopted. The information here adds to these or specifies particular areas that you should pay attention to.

2. C/C++ Elements

When writing C and C++ code for embedded systems certain elements of style are most important to readability and understanding. This section addresses these and describes the preferred way of using these elements in your code.

2.1. Header files

Header files are used to describe a particular capability provided in the code—either code that is written by the programmer or in a library that the programmer is using. Header files should be cohesive. That is, they should contain definitions and prototypes that apply to a single, specific capability. An example would be a set of definitions that are specific to a particular microprocessor.[1] Several guidelines apply to the structure and content of header files.

2.1.1. Separate header files for separate devices

When building robotic systems you will deal with various types of peripheral devices. When developing code for these devices, *place information about each specific device and device type in its own header file*.

2.1.2. Include header files once

Large C and C++ systems typically have many header files that are included explicitly or implicitly when a file is compiled. You can guarantee that a file is included just one time by checking to see if a macro value is defined at the

[1] For example, see how there are individual header files for each of the microprocessor or microprocessor families in the Win AVR tool chain.

.

beginning of a header file. If the value is not defined, then the header file is included. When the file is included, it defines the macro value. The following example shows this for a file named `sensor_defs.h`.

```
#ifndef SENSOR_DEFS_H
#define SENSOR_DEFS_H
     ...
     /* content of the header file */
     ...
#endif SENSOR_DEFS_H
```

2.2. Naming conventions

One of the most important things a programmer can do to make code readable and understandable is to use meaningful names. While it may take a little longer to type a meaningful name, the benefits far outweigh the extra effort.

2.2.1. Function and variable names should convey their purpose

Every function is written for a purpose. Every variable has a purpose. Their names should convey their purpose. Consider the following code. What does it do? It seems to be evaluating something, but it's very hard to decide exactly what. What is the argument, `pv`? What is an `SV`?

```
SV * my_eval_pv(char *pv) {
     SV* result;
     result = eval_pv(pv, TRUE) ;
     return result ;
}
```

This code came from a project that has something to do with embedded Perl. There were no comments describing the function and one comment in the code saying “eval_pv(..., TRUE) means die if Perl traps an error.” This doesn't actually help much. Certainly, knowing the context of the code and the project domain might help, but it will require some effort to understand this code.

Now consider this following code example.

.

```
void handleChangeInSwitches()
{
      int switchNum;
      uint8_t newSwitchSettings = SWITCH_PINS;
      uint8_t LEDsToToggle = 0x00;

      for (switchNum = 0; switchNum < NUMBER_OF_SWITCHES;
              switchNum ++) {
             uint8_t oldBit, newBit;
             oldBit =
               (lastSwitchSettings >> switchNum ) & 0x01;
             newBit =
               (newSwitchSettings >> switchNum ) & 0x01;
             if (oldBit == 1 && newBit == 0) {
                    LEDsToToggle |= (1 << switchNum );
             }
      }

      if (LEDsToToggle & 0xFF) {
             toggleLEDs(LEDsToToggle);
      }
      lastSwitchSettings = newSwitchSettings;
}
```

While you still need to know what the code's context is—which is aided by comments on the function that are not shown—you should be able to understand the code much more easily than the previous example.

2.2.2. Format multi-word names consistently

There are two common ways of naming code elements such as functions and variables. The first is called "CamelCase." (Wikipedia CamelCase) This style strings multiple words together, using a capital letter for the first letter of each word, except possibly for the first word in the multi-word name. The code in the previous code snippet uses CamelCase.

The second method of formatting multi-word names is by separating each word with an underscore. The choice is one of preference. However, you should adopt a style and stick with it. In the above code snippet, this style is used for the type, `uint8_t`. You will often have a case where you are using bodies of code, such as those supplied with the WinAVR tools, that use different formatting conventions. This is unavoidable, but you should be consistent in the code you write.

.

2.3. Symbolic constants and macros

C and C++ programs can be quite unreadable. There are often constant values that must be used in order to correctly implement an algorithm or satisfy the requirements for a function. Good C and C++ programs make liberal use of symbolic constants. If the constants are well-named, they significantly improve the readability of the program.

2.3.1. Replace literal constants with symbolic constants

One sign of fragile programs is the use of "magic constants" in the code. These are numbers or other constants in the code that represent specific values that have particular meanings for the application or application domain. Using the literal value for the constant may make sense to the programmer, but may not be obvious to the reader. Sometimes the same literal value is used for multiple purposes and it is impossible to tell this by reading the code.

Whenever you use a literal value in your code, consider whether it has a specific meaning. If it does, consider defining a symbolic constant for that value.

Consider the following code snippet. The value 65535 is used twice. Is it just a number, or is there some special meaning to it?

```
if (result < 65535) {
     // do something here
}
...
if (stackSize < 65535) {
     // do something else
}
```

What if the code were written as follows?

```
if (result < MAX_UNSIGNED_INT) {
     // do something here
}
...
if (stackSize < MAX_STACK_SIZE) {
     // do something else
}
```

.

Here we can easily see that the values have different meanings and purposes. Further, if we need to port the code where there are different sizes for unsigned integers and stacks, we will simply have to change the value of the defined symbolic constant.

2.3.2. Symbolic constant names

Notice that the symbolic constant names in the last section are written quite differently than other names. This is the usual way that C and C++ programmers write symbolic constants. Even if you use CamelCase style for naming your entities, you should name your symbolic constants according to this style.

Symbolic constants are written with all uppercase letters and words in the name are separated by underscores. The following code declares the constants used in the last code snippet.

```
#define MAX_UNSIGNED_INT 65536
#define MAX_STACK_SIZE   65536
```

2.3.3. Do not put comments in macros

When you create a macro, including a symbolic constant, using the `#define` preprocessor operator, you may be tempted to include comments that illuminate the macro's meaning. This can lead to problems when you use the macro in your code. Depending upon the language preprocessor, the comment may become part of the body of the macro's replacement string. This may produce errors or unwanted results.

Look at the following definition and its use .

```
#define BOIL_F 212  // Fahrenheit boiling point
...
    bdiff1 = BOIL_F - degrees;
```

When the compiler actually gets to see the line where the symbolic constant is used, it might see this:

```
bdiff1 = 212  // Fahrenheit boiling point -
```

Clearly, this is not valid code. Be careful with macro definitions. One of the most complex parts of many C programs is figuring out the macros that the programmers have defined. Simplicity is one of the mot important characteristics of understandable macros and the programs that use them.

.

3. Commenting Your Code

No matter how well you name your program elements, format your code, and use a consistent style, there is one more thing you must do to help others understand your code. You must write meaningful comments that provides contextual information and design rationale to the reader.

There are several tools available that process source code files, extract comments that are formatted specifically for the tool, and produce excellent documentation for your code. For the RBE courses we will use the Doxygen tool. (Doxygen) Doxygen can process code written in several programming languages and output the documentation in various formats.

This section describes the recommended set of comments that you should provide for any of your programs. Commands shown in the examples can be found in the Doxygen manual for Doxygen version 1.5.8.[2]

3.1. Doxygen command format

You have the choice of introducing Doxygen commands with either the ′\′ character or the ′@′ character. Choose one and use it consistently. Do not mix command introduction characters. Code in our examples will usually be documented using the ′\′ character to introduce the commands.

3.2. File comments

All files should have a comment block at the top of the file. The comment should have at least the following items:

- Brief description.
- File name.
- Detailed description describing the purpose of the file and its contents.
- Authors names. *Whenever you modify a file you must make sure that you add your name to the list of authors if it is not already there.*
- Date of last modification.
- Version identifier. Select some numbering scheme and be consistent.

The following example shows an example of a file block comment that adheres to these guidelines.

[2] This manual can be downloaded from ftp://ftp.stack.nl/pub/users/dimitri/doxygen_manual-1.5.8.pdf.zip. Or accessed online. Doxygen User Manual, 2 Jan 2009, 17 Jan 2009 <http://www.stack.nl/~dimitri/doxygen/manual.html>.

.

```
/** \brief Toggle LEDs based upon button presses for the switches.
 *
 * \file ToggleInterrupt.c
 *
 * This program is similar to the Toggle1 program but we use
 * interrupts to detect the button presses rather than polling
 * for them in the loop.
 *
 * \author gpollice@cs.wpi.edu
 * \date 16-Jan-2009
 * \version 1
 */
```

3.3. Macro and symbolic constant comments

If you have taken the time to define a macro or symbolic constant, you should take the tome to let the reader know why you've done so. There are several ways you can add Doxygen comments to your program to document these elements. Doxygen allows you to add comments below the element (not recommended) or above the element immediately or in groups.

We recommend that you place comments above each element when you have defined the macro or symbolic constant in a header file that contains just macros and symbolic constants. This is shown in the following example.

```
/**
 * \def PORT_OUTPUT
 *    Value to store in a DDR to set the port to be output.
 /
#define PORT_OUTPUT           0xFF

 /**
 * \def PORT_INPUT
 *    Value to store in a DDR to set the port to be input.
 */
#define PORT_INPUT            0x00
```

When the symbolic constants and macros are part of a C or C++ implementation file, you may choose to document them in such a way that the documentation for a group of elements appears before the group is defined. Documenting the previous example in this manner looks like this.

.

```
/**
 * \def PORT_OUTPUT
 *    Value to store in a DDR to set the port to be output.
 *
 * \def PORT_INPUT
 *    Value to store in a DDR to set the port to be input.
 */
#define PORT_OUTPUT           0xFF
#define PORT_INPUT            0x00
```

The generated documentation is identical for both examples, but the second example makes it easier to find the definitions in the code.

Symbolic constants should always have a comment associated with them. Other macros need documentation when they are not obvious from their name and description. If there is any chance that the value can be misunderstood or misinterpreted, you should provide comments. If the value is dependent upon the context of the application or particular hardware and might vary in subsequent usage, you definitely should provide comments.

3.4. Commenting functions

Every function must be commented. The comments should state the intended purpose of the function and clarify for the reader complex or subtle points in the code. The comments should also identify any external sources that were used in developing the code. *Using publicly available code in your program is acceptable as long as you provide a citation to such code.*

The minimum contents of a function comment are:

- Function signature using the *fn* Doxygen command.
- Purpose of each parameter using *param*.
- Any return value using *return*. If there are some values that indicate special error conditions, you should identify them as such.
- A brief description using *brief*.
- Detailed description if the brief description is insufficient for complete understanding.

A sample function comment follows.

```
/** \fn void toggleLEDs(uint8_t toggleBits)
 * \brief Toggle the LEDs specified by the bits in the argument
 * \param [in] toggleBits bits signifying the LEDs to change
 * \return void
 */
```

.

3.5. Commenting other elements

The following sections describe the typical elements you will need to comment in your programs. When you use other program elements, such as unions or structs, you should provide comments for them as well. The above guidelines should give you an idea of the level of documentation you should provide.

Doxygen provides commands that let you customize your documentation. Develop your own style. Start simple and add as you need to. Remember to be as consistent as possible.

.

Bibliography

Déchelle, Maurizio De Cecco and François. "http://www.literateprogramming.com/ftsguide.pdf." 1995. literateprogramming.com. 17 Jan 2009 <http://www.literateprogramming.com/>.

Doxygen . 2 Jan 2009. 17 Jan 1009 <http://www.doxygen.org>.

Doxygen User Manual. 2 Jan 2009. 17 Jan 2009 <http://www.stack.nl/~dimitri/doxygen/manual.html>.

Sutton, Paul. Apache Developers' C Language Style Guide. 2008. 17 Jan 2009 <http://httpd.apache.org/dev/styleguide.html>.

Welch, Jenny and Walter, Jennifer. C Programming Style Guide. 17 Jan 2009 <http://faculty.cs.tamu.edu/welch/teaching/cstyle.html>.

Wikipedia CamelCase. 17 Jan 2009 <http://en.wikipedia.org/wiki/CamelCase>.