AC 2010-1148: CODING PRACTICES FOR EMBEDDED SYSTEMS

Michael Pook, Boise State University

Sin Ming Loo, Boise State University

Arlen Planting, Boise State University

Josh Kiepert, Boise State University

Derek Klein, Boise State University

Coding Practices for Embedded Systems

Abstract

Far too often, students focus on creating a working project without any regard for the quality, readability, and maintainability of their code. Students are not usually made to realize how learning and applying good coding practices can improve the success of their projects and make them more valuable to future employers. By adhering to a set of simple coding practices, students can create projects that are easier to read, maintain, predict (no unexpected features or "ghost" effects), and reuse. Many students have developed a misconception that a copious amount of commenting can make up for the shortcomings of code that is poorly organized and/or difficult to understand. Although good commenting is important and can make the purpose of code blocks clearer, paragraphs of explanation do more to clutter a project than they do to improve its quality. Proper commenting should be used in conjunction with other important practices in order to create a project with the cleanest and clearest code possible. Distinctive naming of components (files, variables, constants, etc.) and a logical layering of code can help to make changes to code far simpler, less time-consuming, and produce more predictable results. This paper provides a guide to good coding practices. Some of the topics covered will include proper use of commenting, code modularization, layering, descriptive naming, cohesion, and avoiding race conditions.

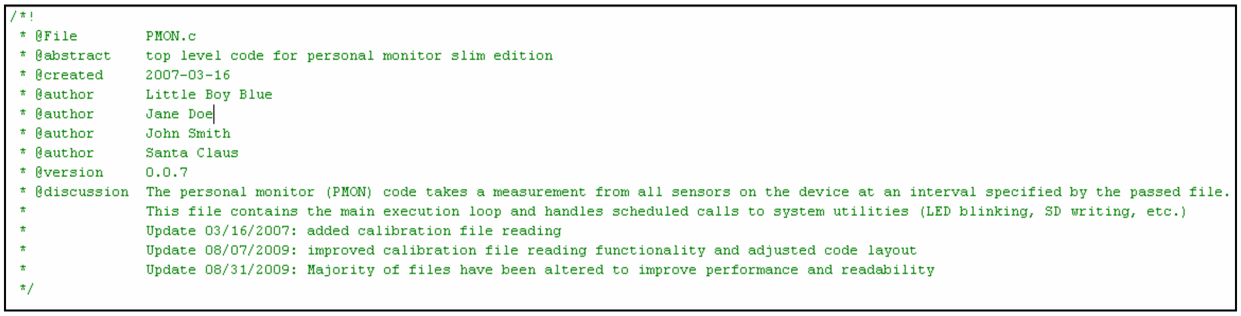
Introduction

The standard for many embedded systems courses is to stress the importance of functioning code. Although the primary goal of a design is to obtain a functioning product, students should be taught the importance of good coding practices as well. Many programs stress the need for good commenting but fail to cover the other guidelines. As a result, students fall under the false impression that a novel's worth of commenting can compensate for bad code. Coding should be taught in a similar fashion to writing an essay. The writing process is done in multiple iterations of review and refinement. Students should first write functioning code that meets specification. Then, they need to review and edit their code until they have achieved the best possible solution (i.e. most readable, reusable, maintainable, and predictable code). By following a simple set of rules, students can create code that is easy to read, maintain, predict, and reuse. Obtaining this valuable skill will make them more valuable to future employers and save them valuable design, debug, and test time. The following sections cover basic coding practices and the reasons for adhering to them.

Commenting

One of the most commonly overlooked and most important coding practices is the commenting of finished code. Comments are usually added as an afterthought with little or no effort made on the part of the programmer to make clear, concise statements about their design. One should always consider the consequences of poorly commenting code or leaving out comments entirely. From the perspective of a student, a project coded for a particular class one semester could contain valuable pieces of code that might be used for a project in another class. However, after leaving a project for a semester or more, said student may find the process of deciphering their poorly commented work to be tedious and more time consuming than it would be to start the project from scratch. From an industry standpoint, poorly commented/uncommented code can make the processes of fixing, maintaining, or adding functionality to a project very difficult. This situation is especially true when the programmer making the modifications or reusing the code was not the original designer. Thus, by not taking the proper steps to clearly comment code, a programmer can negatively affect his future productivity as well as the productivity of his colleagues.

A programmer should place comments wherever a need for explanation is apparent and leave out comments in areas where they might obscure code rather than make it clearer. However, some sections of a project should always be commented regardless of the programmer's preference. One such section is at the beginning of every source file. The comment block at the beginning of a source file should include the name of the file, the date the file was created, a list of authors, and a discussion of the file's functionality. More information can be included as needed, but these four items should never be excluded (see Figure 1).



```
/*!
 * @File         PMON.c
 * @abstract     top level code for personal monitor slim edition
 * @created      2007-03-16
 * @author       Little Boy Blue
 * @author       Jane Doe
 * @author       John Smith
 * @author       Santa Claus
 * @version      0.0.7
 * @discussion   The personal monitor (PMON) code takes a measurement from all sensors on the device at an interval specified by the passed file.
 *               This file contains the main execution loop and handles scheduled calls to system utilities (LED blinking, SD writing, etc.)
 *               Update 03/16/2007: added calibration file reading
 *               Update 08/07/2009: improved calibration file reading functionality and adjusted code layout
 *               Update 08/31/2009: Majority of files have been altered to improve performance and readability
 */
```

Figure 1: Comment Block for the Top of a Source File

Additionally, comments should also always be included upon instantiation to explain the purpose of all global variables, typedefs, and structures/unions. Another important commenting practice is to place a comment before each function. Function comments should include a description of the function's purpose as well as a description of returned and passed parameters. Function comments can be written in the same format as the source file comment. A reader should be able to determine what the function does without having to read the code. Comments should only be placed inside a function if the function contains overly complex blocks that would hinder a developer if left unexplained. The final section where comments should be placed within a project is the header file. Header files should have a similar comment to source files at the top and include comments for any structures/unions, type definitions, or constants. Copying the function comments to their prototypes in the header file is also desirable to improve readability/maintainability.

Although thorough commenting is important, too many comments can make code look cluttered and difficult to read. A programmer should be able to comment his code without writing something for every line or including a novel at the start of each function. No one can give a general rule for the number of lines or words needed to properly comment a given block of code. A good method for determining the adequacy of a comment is to read it from the perspective of someone with little to no knowledge of the project in question. The programmer should try to imagine that he/she is viewing the project after having left it for five years. Given this frame of mind, the programmer should be able to determine if more commenting is needed to make the functionality of the project (code block, file, etc.) clearer.

Naming Conventions

The names given to the different objects which constitute a project (variables, constants, functions, source files, etc.) can contribute greatly to its readability. Names should be chosen to be as descriptive as possible without making the name as long as a paragraph. When determining the name for any given object, a programmer should base their decision on the purpose/function of said object.

Source Files

The names of source files and functions should be chosen such that they give a basic idea of the given object's purpose at a glance. Additionally, files/functions with interdependencies or related purposes should have names that indicate their relationship to one another and adhere to the same format. For example: given a project with three source files responsible for different aspects of wireless communication, the names "wireless1," "wireless2," and "wireless3" are completely inappropriate. The one responsible for creating/joining networks should have a name like "Network_Initialization." The file responsible for framing data to create transmittable packets should have a name like "Network_Packetization." Finally, the file responsible for transmitting packets and facilitating general communications should have a name like "Network_Communication." Although there exist many options for appropriate names for such files, the three given examples give the programmer reading the code a basic idea of what they will find in each file. In addition, each name is brief and the common word "Network" appears first in all three thus showing a relationship between the three files while maintaining a common format.

Variables and Constants

The most common naming problem with many coding projects is the use of meaningless variable/constant names such as single letters, the word "count," and the word "index." The most common cause of this problem (as well as most naming problems) is the "coding zone." For example: while in the "zone," programmers get ideas for loops and state machines that require some integer index in order to keep track of which task to accomplish next. They type as fast as they can without regard to proper coding practices in order to get the idea/functionality written on their screen before they forget. As a result, an index used to track the state of the "Airbag Deployment State Machine" gets named "a" or "i" instead of "Airbag_Deployment_State." Such a practice is perfectly fine to use when testing an idea for a new piece of code. Reasonable people do not expect a programmer to make perfect code on the first attempt. The idea of writing code in one iteration while following every naming guideline as well as every other good coding practice rule is utterly absurd. That would be the equivalent of trying to write a doctoral dissertation in one iteration without reading through and editing the completed rough draft. As with writing any paper, writing code is an iterative process involving careful review and refinement. The problem arises when a programmer fails to edit their own code. Once the functionality of new code has been verified, the programmer should choose appropriate names based on careful review of each object's purpose. One method for determining whether or not a variable's/constant's name is adequately descriptive is to determine the appropriate commenting for that portion of code. If the commenting includes a lengthy description (more than a brief mention in a single sentence) of the variable/constant, the name should be reconsidered.

Variables and Constants

A programmer's choices for how to use variables, constants, and other data types can have a profound effect on the code's maintainability and readability. Proper use of variable scope, function parameters, constants, structures, and unions deserve careful consideration.

Scope

One of the difficult decisions encountered when creating a variable is determining its scope. Many new programmers who have been taught to write programs meant to run on a computer will claim that the decision is quite simple. If there is any question as to what the scope should be, the initial reaction is just to make a global instantiation. This makes the coding process easier by allowing any function in any source file of the project to access the variable at any time. However, a global instantiation comes with its own set of problems that must be carefully weighed when writing for an embedded system. Some embedded system designers have been known to say that global variables should only be used if the programmer is willing to write a three page report explaining the need for each global instantiation. Although the three pages might seem a little extreme, the sentiment behind the statement is correct for two reasons. First and foremost, memory in an embedded system is usually a valuable and scarce resource. A global variable takes a permanent piece of that memory. Secondly, global variables detract from a project's readability. While passed parameters within a function are commented by said function's comment block, global variables get their own comments upon instantiation. Thus, in order to determine the purpose of a global variable, a programmer would have to search for the comment outside of the function they are trying to understand. Therefore, whenever the situation permits, variables used by multiple functions should be passed as parameters and returned values in order to decrease memory usage and increase code readability.

Function Parameters

Upon making the decision that passing variables between a set of functions will work better for a given situation, the programmer will have to determine how to pass said variables. Given a situation in which multiple functions must modify the same variables instantiated by one function, pointers are the best choice of parameter type. This removes the need to return parameters and allows the programmer to use the return value for error checking and debugging. However, if the function instantiating the variables has no use for them after passing them off to another function, pointers become superfluous and the variable itself can be passed. Another situation to consider is one in which data from a large array must be used. Rather than constantly passing a large array, a pointer should be used to minimize data transmission regardless of whether or not the original function still has use for the array.

Constants

Constants (the type; not the value) are a very useful tool for maintaining the readability of a project. Unfortunately, they are often ignored by programmers. Almost every constant value used in a project should take the form of a descriptively named word or phrase (see Naming Conventions section). For example: multiplying a sensor reading by 9.72 tells a person reading the code nothing about why that value is used. Assuming the original programmer is unavailable or can not remember the reason for this seemingly arbitrary value, a programmer trying to understand the code would have to search through datasheets to find the purpose of this constant value. However, if the original programmer took the time to type "constant float gravitational_constant = 9.72" or "#define gravitational_constant 9.72" and used this descriptive name instead of the value, they could save themselves and/or their colleagues valuable time.

Structures and Unions

Two more useful tools that are commonly ignored by less experienced programmers are structures and unions. In fact, many students just learning to write code for an embedded systems class seem to avoid these data types like the plague. This is very strange considering how simple they are to use and all of the benefits they present to programmers. Structures and unions allow a programmer to group variables used for similar purposes under one name. This allows said collection of variables to be passed between functions using a single name. For instance: consider a system that controls warning lights for a car. Figure 2 (a) shows a portion of code using multiple variables, and Figure 2 (b) depicts the same code using structures and unions. The most obvious difference between these two pieces of code is the amount of typing required for each. The code in Figure 2 (a) requires more typing and is very repetitive. In addition to requiring less typing for this portion of the project, the code in Figure 2 (b) organizes the code into layers and gives this section of the project a sense of hierarchy. Furthermore, the code in Figure 2 (b) contains more white space which results in a more readable project.

```
unsigned char right_rear_tire_pressure;
unsigned char right_front_tire_pressure;
unsigned char left_rear_tire_pressure;
unsigned char left_front_tire_pressure;
unsigned int engine_temperature;
unsigned int engine_oil;
unsigned int engine_service;
...
right_rear_tire_pressure= 0;
right_front_tire_pressure= 1;
left_rear_tire_pressure= 0;
left_front_tire_pressure= 0;
...
activate_system_warnings(right_rear_tire_pressure, right_front_tire_pressure,
                         left_rear_tire_pressure, left_front_tire_pressure,
                         engine_temperature, engine_oil, engine_service);
```

(a)

```
struct
{
        union
        {
                unsigned int all_four;
                struct
                {
                        unsigned char right_rear;
                        unsigned char right_front;
                        unsigned char left_rear;
                        unsigned char left_front;
                } individual;
        } tire_pressure;
        struct
        {
                unsigned int temperature;
                unsigned int oil;
                unsigned int service;
        } engine;
} system_warnings;|
...
system_warnings.tire_pressure.all_four= 0;
system_warnings.tire_pressure.individual.right_front= 1;
...
activate_system_warnings(&system_warnings);
```

(b)

Figure 2: (a) C Code for Passing Multiple Variables: (b) C Code for Passing Multiple Variables Using Structures and Unions

An additional benefit to using structures and unions is the ability to create shadow variables to keep track of register states in peripheral devices. Using these data types, a programmer can create an exact copy (or shadow variable) of the current status of each register in the external device. Then, when the system decides to write changes to the peripheral's registers, it can simply make its changes to the shadow variables and call a routine to write the shadow variables to the device registers. This process allows programmers working on the project at a later date to understand the operation of the peripheral device without needing datasheets. Thus, the readability/maintainability of the project is improved.

Project Organization

Project organization plays a key role in creating a project that is easy to read and maintain. Programmers should be aware of how cohesion, coupling, layering, and the proper use of header files can affect their projects.

Cohesion and Coupling

In general, programmers writing code for an embedded system should try to make a project with the highest possible cohesion and the lowest possible coupling. In basic terminology, this translates into the desire to have the source files of a project split into independent modules with well defined tasks. Each module may consist of one or more source files as long as all source files in a module share a common goal and naming convention (to indicate the relationship).

The way in which functions will be grouped into a module is dependent upon a given project. The programmer should first look at all of the functions required for system operation and group functions with related goals into a single module. Once all functions have been placed in a module, the programmer should check for interdependencies between modules. This process should be repeated until the configuration having modules with the most well defined tasks and the fewest amount of interdependencies is obtained. Modules with well defined tasks will allow someone searching for specific functions to find them in the least amount of time. Also, modules with little or no dependencies on other modules are easier to reuse in future projects. For example: the goal of a given project may be to write data received on the network to memory. This would indicate that the network communications and memory writing functions share a common goal in the project. Thus, a logical grouping might be to create one module for memory interface functions (initialization, reading, etc.) and one module for network communications including the memory writing function. However, due to the memory initialization routines, closer inspection shows that this makes the network communications module dependent upon the memory module. Additionally, if the function for writing data to memory is stored in the network communications module of a given project, a programmer is likely to waste time searching the memory interface module before checking other modules. Thus, the network communications module is not reusable without the memory module, and the code is difficult to read and maintain. However, if the memory writing function is moved to the memory interface module, the dependency is removed and the memory writing function is located in the most logical module.

Layering

With any code written for an embedded system, the creation of a project hierarchy is a necessity in order to make the project easy to maintain, reuse, and read. The code for an embedded system can be split into two main layers. The first is the low-level code responsible for any necessary bit banging and interfacing to actual system hardware. Examples of this type of code include functions responsible for such tasks as implementing communications protocols, mapping microcontroller pin functions, and editing peripheral device registers. The second is the high-level code which uses the low-level code to complete the main tasks of a given system. Examples of this type of code include functions responsible for such tasks as sending data to be written to memory, scheduling sensor readings, and using communications protocols to write messages to some external device.

It should be noted that more layering can (and in most cases should) occur inside each of the two main layers. The number of subdivisions within each layer is dependent on the project and should be determined at the discretion of the programmer. Properly layered code should keep the higher-level functions completely independent from the lower-level functions. This allows a system designer to change a system's hardware and keep high-level functionality unchanged. The only requirement for changing hardware should be a new set of low-level functions that can communicate with the new hardware. Thus, higher-level code is made to be hardware independent and can be easily ported to run on any platform. For example: consider a system that is responsible for taking measurements from a set of sensors and control emergency systems based on the collected data. Such a system would have low-level functions to handle communication with each sensor and high-level functions to call the necessary low-level code in order to ask for a sensor measurement. Given that one of the current sensors becomes obsolete and is no longer manufactured, the system designer must change to a new sensor. If the code is properly separated, the designer should only need to change the one file that is responsible for low-level communication with this specific sensor. Thus, proper layering has made maintenance of this system simpler and has allowed for the reuse of high-level code on the new hardware.

Header Files

Header files have two important jobs. First, they act as links between source files and make the process of organizing and layering of code far simpler. They contain all of the information needed for another source file to communicate with their corresponding source file. Second, they provide the programmer with a quick reference for their source file. The comment block located at the top of a source file should also appear at the top of its header file.

Each header file in a project should be organized into sections with logical groupings. Although there is no set order in which the sections should appear, it is important that their order is uniform throughout a project. Every header file should contain one section for prototypes of every function in the source file. Even if a function is not used externally, a prototype can be created in the header file for the purpose of uniform documentation. Although creating a private prototype in a header file is not the standard industry practice, doing so will improve documentation by providing future programmers working on the project with insight into the internal operations of the code and the way in which the project layered. Each prototype should be commented with the appropriate comment blocks as discussed in the Commenting section. Any prototypes not meant for external use should be commented and named to indicate as much to the reader. Another section should include any necessary typedefs and "#defines" (with comments for each) if any are needed for the source file.

Optimization

Although the primary goal of any piece of code is to function properly, optimized code should be near the top of the list of secondary goals. Optimized code not only improves maintainability, readability, and reusability, but also makes the behavior of code far more predictable.

The Main File

Ideally, the main file in a project should consist of one main function that calls functions from the modules to facilitate the desired functionality. There should be no global variables (see Variables and Constants section) and no general purpose functions in the main file. This would net the least confusing and most easy to understand flow of logic. That being said, an ideal situation is rarely obtainable especially in complex systems. However, a programmer should try to get as close to the ideal situation as possible. For instance, separating the general purpose functions from the main file should be a simple process. These functions should be moved to a "utility" or "system_general" (or another descriptive name) file to be called from the main. The objective is to improve readability by keeping the main file to a minimum length.

Race Conditions

A major concern for many programmers is the possibility of encountering race conditions in their projects. These occur when multiple functions use the same variable, and the performance of the code is dependent upon the timing when each function accesses said variable. This problem is more prevalent in embedded systems with interruptible code. For example: variable "race" in a given system is incremented somewhere inside function "determine_winner." At a predetermined point, the main file calls "determine_winner," but the function is interrupted just after reading the current value of "race" and before storing the new incremented value. Now, interrupt service routine (ISR) "ruin_the_race" increments "race" and releases control of the system. Next, "determine_winner" is allowed to write the value it calculated back to "race." Notice that "race" should have been incremented twice but, instead, was only incremented once. The stored value of "race" is invalid and could result in system failure. This is a very simplified example, but it provides an idea of the problems presented by race conditions. To avoid such problems, shared variables between functions and ISR's should be avoided when possible. When such a situation is unavoidable, the easiest method for preventing race conditions is disabling interrupts while sensitive sections of code are running. However, disabling interrupts can result in missing important system events. Thus, this option should only be used as a last resort. Following these rules will produce code with far more predictable results.

General Optimizations

Three general guidelines that should be used when optimizing code for an embedded system are minimize the use of delays, minimize redundant code, and minimize the amount of code in ISR's. First, delays used while polling for an event or to adjust timings are a waste of processing power. Forcing a processor to idle when it has other important functions to perform should be minimized to increase system performance. For example: consider a system with multiple high-level tasks running sequentially but completely independent from each other. If one task has to wait for a low-level function to finish polling for an event, all other tasks must also wait even though they have all necessary data to perform their functions. A better solution is to use interrupts instead of polling at the low-level to avoid stalling the entire system. Second, writing multiple instances of the same piece of code in different locations makes little sense. If a piece of code is reused, it should be turned into its own function. This reduces project size, increases readability, and saves time. Third, ISR's interrupt the normal flow of a system and prevent a system from continuing normal function until they release control of the processor. As such, code inside an ISR should be minimized to allow a system to resume normal function sooner. This will improve system performance and predictability.

Conclusion

Good coding practices are an important but commonly overlooked topic in embedded systems curriculum. Creating code that is easy to read, maintain, predict, and reuse is an important and valuable skill that will serve students well. Many students labor under the misconception that copious amounts of commenting can make up for the shortcomings of code that is poorly organized and/or difficult to understand. Although good commenting is important, writing lines of comments that outnumber the code in a project at a ratio of two to one will clutter a project rather than improve its quality. Sifting through comments can become tedious when attempting to make a quick change to old code. Distinctive naming of project components (files, variables, constants, etc.) and a logical layering of code can help to make such a process far simpler, less time consuming, and produce more predictable results. This indicates that proper commenting should be used in conjunction with other important practices in order to create a project with the cleanest and clearest code possible.

Bibliography

No reference material was used for the writing of this document. All statements made in this document are based on the collective experiences of the authors. These experiences include hands on research/programming (both for individual and collaborative projects), programming for industry projects, and taking/teaching classes at Boise State University.