2.5. Quality Design

Embedded system development is similar to other engineering tasks. We can choose to follow well-defined procedures during the development and evaluation phases, or we can meander in a haphazard way and produce code that is hard to test and harder to change. The ultimate goal of the system is to satisfy the stated objectives such as accuracy, stability, and input/output relationships. Nevertheless it is appropriate to separately evaluate the individual components of the system. Therefore in this section, we will evaluate the quality of our software. There are two categories of performance criteria with which we evaluate the "goodness" of our software. Quantitative criteria include dynamic efficiency (speed of execution), static efficiency (memory requirements), and accuracy of the results. Qualitative criteria center on ease of software maintenance. Another qualitative way to evaluate software is ease of understanding. If your software is easy to understand then it will be:

Easy to debug (fix mistakes)
Easy to verify (prove correctness)
Easy to maintain (add features)

Common Error: Programmers who sacrifice clarity in favor of execution speed often develop software that runs fast, but is error-prone and difficult to change.

Golden Rule of Software Development
Write software for others as you wish they would write for you.

2.5.1. Quantitative Performance Measurements

In order to evaluate our software quality, we need performance measures. The simplest approaches to this issue are quantitative measurements. **Dynamic efficiency** is a measure of how fast the program executes. It is measured in seconds or processor bus cycles. **Static efficiency** is the number of memory bytes required. Since most embedded computer systems have both RAM and ROM, we specify memory requirement in global variables, stack space, fixed constants and program. The global variables plus the stack must fit into the available RAM. Similarly, the fixed constants plus the program must fit into the available ROM. We can also judge our embedded system according to whether or not it satisfies given requirements and constraints, like accuracy, cost, power, size, reliability, and time-table.

2.5.2. Qualitative Performance Measurements

Qualitative performance measurements include those parameters to which we cannot assign a direct numerical value. Often in life the most important questions are the easiest to ask, but the hardest to answer. Such is the case with software quality. So therefore we ask the following qualitative questions. Can we prove our software works? Is our software easy to understand? Is our software easy to change? Since there is no single approach to writing the best software, we can only hope to present some techniques that you may wish to integrate into your own software style. In fact, this book devotes considerable effort to the important issue of developing quality software. In particular, we will study self-documented code, abstraction, modularity, and layered software. These issues indeed play a profound effect on the bottom-line financial success of our projects. Although quite real, because there is often not an immediate and direct relationship between a software's quality and profit, we may be mistakenly tempted to dismiss the importance of quality.

To get a benchmark on how good a programmer you are, take the following two challenges. In the first challenge, find a major piece of software that you have written over 12 months ago, and then see if you can still understand it enough to make minor changes in its behavior. The second challenge is to exchange with a peer a major piece of software that you have both recently written (but not written together), then in the same manner, see if you can make minor changes to each other's software.

Observation: You can tell if you are a good programmer if 1) you can understand your own code 12 months later, and 2) others can make changes to your code.

2.5.3. Attitude

Good engineers employ well-defined design processes when developing complex systems. When we work within a structured framework, it is easier to prove our system works (verification) and to modify our system in the future (maintenance.) As our software systems become more complex, it becomes increasingly important to employ well-defined software design processes. Throughout this book, a very detailed set of software development rules will be presented. This book focuses on real-time embedded systems written in assembly language and C, but most of the design processes should apply to other languages as well. At first, it may seem radical to force such a rigid structure to software. We might wonder if creativity will be sacrificed in the process. True creativity is more about good solutions to important problems and not about being sloppy and inconsistent. Because software maintenance is a critical task, the time spent organizing, documenting, and testing during the initial development stages will reap huge dividends throughout the life of the software project.

Observation: The easiest way to debug is to write software without any bugs.

We define **clients** as programmers who will use our software. A client develops software that will call our functions. We define **coworkers** as programmers who will debug and upgrade our software. A coworker, possibly ourselves, develops, tests, and modifies our software.

Writing quality software has a lot to do with attitude. We should be embarrassed to ask our coworkers to make changes to our poorly written software. Since so much software development effort involves maintenance, we should create software modules that are easy to change. In other words, we should expect each piece of our code will be read by another engineer in the future, whose job it will be to make changes to our code. We might be tempted to quit a software project once the system is running, but this short time we might save by not organizing, documenting, and testing will be lost many times over in the future when it is time to update the code.

As project managers, we must reward good behavior and punish bad behavior. A company, in an effort to improve the quality of their software products, implemented the following policies.

> The employees in the customer relations department receive a bonus for every software bug that they can identify. These bugs are reported to the software developers, who in turn receive a bonus for every bug they fix.

Checkpoint 2.7: Why did the above policy fail horribly?

We should demand of ourselves that we deliver bug-free software to our clients. Again, we should be embarrassed when our clients report bugs in our code. We should be mortified when other programmers find bugs in our code. There are a few steps we can take to facilitate this important aspect of software design.

Test it now. When we find a bug, fix it immediately. The longer we put off fixing a mistake the more complicated the system becomes, making it harder to find. Remember that bugs do not go away on their own, but we can make the system so complex that the bugs will manifest themselves in mysterious and obscure ways. For the same reason, we should completely test each module individually, before combining them into a larger system. We should not add new features before we are convinced the existing system is bug-free. In this way, we start with a working system, add features, and then debug this system until it is working again. This incremental approach makes it easier to track progress. It allows us to undo bad decisions, because we can always revert back to a previously working system. Adding new features before the old ones are debugged is very risky. With this sloppy approach, we could easily reach the project deadline with 100% of the features implemented, but have a system that doesn't run. In addition, once a bug is introduced, the longer we wait to remove it, the harder it will be to correct. This is particularly true when the bugs interact with each other. Conversely, with the incremental approach, when the project schedule slips, we can deliver a working system at the deadline that supports some of the features.

> **Maintenance Tip:** Go from working system to working system.

Plan for testing. How to test each module should be considered at the start of a project. In particular, testing should be included as part of the design of both hardware and software components. Our testing and the client's usage go hand in hand. In particular, how we test the module will help the client understand the context and limitations of how our component is to be used. On the other hand, a clear understanding of how the client wishes to use our hardware/software component is critical for both its design and its testing.

> **Maintenance Tip:** It is better to have some parts of the system that run with 100% reliability than to have the entire system with bugs.

Get help. Use whatever features are available for organization and debugging. Pay attention to warnings, because they often point to misunderstandings about data or functions. Misunderstanding of assumptions that can cause bugs when the software is upgraded, or reused in a different context than originally conceived. Remember that computer time is a lot cheaper than programmer time.

> **Maintenance Tip:** It is better to have a system that runs slowly than to have one that doesn't run at all.

Deal with the complexity. In the early days of microcomputer systems, software size could be measured in 100's of lines of source code using 1000's of bytes of memory. These early systems, due to their small size, were inherently simple. The explosion of hardware technology (both in speed and size) has led to a similar increase in the size of software systems. Some people forecast that by the next decade, automobiles will have 10 million lines of code in their embedded systems. The only hope for success in a large software system will be to break it into simple modules. In most cases, the complexity of the problem itself cannot be avoided. E.g., there is just no simple way to get to the moon. Nevertheless, a complex system can be created out of simple components. A real creative effort is required to orchestrate simple building blocks into larger modules, which themselves are grouped to create even larger systems. Use your creativity to break a complex problem into simple components, rather than developing complex solutions to simple problems.

> **Observation:** There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies and the other way is make it so complicated that there are no obvious deficiencies. C.A.R. Hoare, "The Emperor's Old Clothes," CACM Feb. 1981.

5.6. Writing Quality Software

5.6.1. Style Guidelines

The objective of this section is to present style rules when developing software. This set of rules is meant to guide not control. In other words, they serve as general guidelines rather than fundamental law. Choosing names for variables and functions involves creative thought, and it is intimately connected to how we feel about ourselves as programmers. Of the policies presented in this section, naming conventions may be the hardest habit for us to break. The difficulty is that there are many conventions that satisfy the "easy to understand" objective. Good names reduce the need for documentation. Poor names promote confusion, ambiguity, and mistakes. Poor names can occur because code has been copied from a different situation and inserted into our system without proper integration (i.e., changing the names to be consistent with the new situation.) They can also occur in the cluttered mind of a second-rate programmer, who hurries to deliver software before it is finished.

Names should have meaning. If we observe a name away from the place where it is defined, the meaning of the object should be obvious. The object **`TxFifo`** is clearly the transmit first in first out circular queue. The function **`LCD_OutString`** will output a string to the LCD.

Avoid ambiguities. Don't use variable names in our system that are vague or have more than one meaning. For example, it is vague to use **`temp`**, because there are many possibilities for temporary data, in fact, it might even mean temperature. Don't use two names that look similar, but have different meanings.

Give hints about the type. We can further clarify the meaning of a variable by including phrases in the variable name that specify its type. For example, **`dataPt timePt putPt`** are pointers. Similarly, **`voltageBuf timeBuf pressureBuf`** are data buffers. Other good phrases include **`Flag Mode U L Index Cnt`**, which refer to Boolean flag, system state, unsigned 16-bit, signed 32-bit, index into an array, and a counter respectively.

Use the same name to refer to the same type of object. For example, everywhere we need a local variable to store an ASCII character we could use the name **`letter`**. Another common example is to use the names **`i j k`** for indices into arrays. The names **`V1 R1`** might refer to a voltage and a resistance. The exact correspondence is not part of the policies presented in this section, just the fact that a correspondence should exist. Once another programmer learns which names we use for which types of object, understanding our code becomes easier.

Use a prefix to identify public objects. A public variable is shared between two modules. A public function is a function in one module that can be called from another module. An underline character will separate the module name from the function name. Public objects have the underline and private objects do not. As an exception to this rule, we can use the underline to delimit words in all upper-case name (e.g., **`MIN_PRESSURE equ 10`**). Functions that can be accessed outside the scope of a module (i.e., public) will begin with a prefix specifying the module to which it belongs. It is poor style to create public variables, but if they need to exist, they too would begin with the module prefix. The prefix matches the module name containing the object. For example, if we see a function call, **`BL LCD_OutString`** we know the public function belongs to the LCD module. Notice the similarity between this syntax (e.g., **`LCD_Init`**) and the corresponding syntax we would use if programming the module in C++ (e.g., **`LCD.Init()`**). Using this convention, we can distinguish public and private objects.

Use upper and lower case to specify the allocation of an object. We will define I/O port addresses and other constants using no lower-case letters, like typing with caps-lock on. In other words, names without lower-case letters refer to objects with fixed values. **`TRUE FALSE`** and **`NULL`** are good examples of fixed-valued objects. As mentioned earlier, constant names formed from multiple words

will use an underline character to delimit the individual words. E.g., **MAX_VOLTAGE UPPER_BOUND FIFO_SIZE**. Permanently allocated variables are global, with a name beginning with a capital letter, but including some lower-case letters. Temporarily allocated variables are called local, and the name will begin with a lower-case letter, and may or may not include upper case letters. Since all functions are permanently allocated, we can start function names with either an upper-case or lower-case letter. Using this convention, we can distinguish constants, globals and locals.

> An object's properties (public/private, local/global, constant/variable) are always perfectly clear at the place where the object is defined. The importance of the naming policy is to extend that clarity also to the places where the object is used.

Use capitalization to delimit words. Names that contain multiple words should be defined using a capital letter to signify the first letter of the word. Recall that the case of the first letter specifies whether it is a local or global variable. Some programmers use the underline as a word-delimiter, but except for constants, we will reserve underline to separate the module name from the name of a public object. Table 5.3 overviews the naming convention presented in this section.

Object type	Examples of names that identify type
Constants	**CR SAFE_TO_RUN PORTA STACK_SIZE START_OF_RAM**
Local variables	**maxTemperature lastCharTyped errorCnt**
Private global variable	**MaxTemperature LastCharTyped ErrorCnt RxFifoPt**
Public global variable	**DAC_MaxTemperature Key_LastCharTyped Network_ErrCnt**
Private function	**ClearTime wrapPointer InChar**
Public function	**Timer_ClearTime RxFifo_Put Key_InChar**

Table 5.3. Examples of names.

Checkpoint 5.12: Just by looking at its name, how can you tell if a function is private or public?

Checkpoint 5.13: Just by looking at its name, how can you tell if a variable is local or global?

The Single Entry Point is at the Top. In assembly language, we place a single entry point of a subroutine at the first line of the code. By default, C functions have a single entry point. Placing the entry point at the top provides a visual marker for the beginning of the subroutine.

The Single Exit Point is at the Bottom. Most programmers prefer to use a single exit point as the last line of the subroutine. Some programmers employ multiple exit points for efficiency reasons. In general, we must guarantee the registers, stack, and return parameters are at a similar and consistent state for each exit point. In particular, we must deallocate local variables properly. If you do employ multiple exit points, then you should develop a means to visually delineate where one subroutine ends and the next one starts. You could use one line of comments to signify the start a subroutine and a different line of comments to show the end of it. Program 5.22 employs distinct visual markers to see the beginning and end of the subroutine.

Label Names. Some of the assembly examples used weak label names like "**loop**", "**done**", "**end**", and "**out**". These names technically work, and they might make sense for very short examples. But they can cause problems when functions are copied from one file to another, resulting in multiple "**loop**" labels. One or both needs to be changed before it even assembles. Labels inside functions can be preceded with the function name, such as "**AbsOK**" or with an underscore "**Abs_OK**".

```
;------------Abs-----------
; Take the absolute value of a number.
; Input: R0 is 32-bit signed number
; Output: R0 is 31-bit absolute value
Abs   CMP R0, #0  ; is number (R0) >= 0?
      BPL AbsOK   ; if so, already positive
```

```
//************Abs************
// Input: signed 32-bit
// Output: absolute value
uint32_t Abs(int32_t n){
  if(n<0){
    n = -n;
```

```
      EOR R0, R0, #0xFFFFFFFF ; invert bits
      ADD R0, R0, #1          ; add one
AbsOK BX  LR                  ; return
;------------end of Abs------------
```

```
  }
  return (uint32_t) n;
}
```

Program 5.22. Examples that use comments to delineate its beginning and end.

Observation: Having the first and last lines of a subroutine be the entry and exit points makes it easier to debug, because it will be easy to place debugging instruments (like breakpoints).

Common error: If you place a debugging breakpoint on the last BL LR of a subroutine with multiple exit points, then sometimes the subroutine will return without generating the break.

Write Structured Programs. A structured program is one that adheres to a strict list of program structures, previously defined in Section 1.8 and further elaborated in Section 5.2. When we program in C (with the exception of **`goto`**, which by the way you should only use with extreme care) we are forced to write structured programs due to the syntax of the language. One technique for writing structured assembly language is to adhere to the program structures shown in Figure 1.29. In other words, restrict the assembly language branching to configurations that mimic the software behavior of **`if`**, **`if-else`**, **`do-while`**, **`while`**, **`for`** and **`switch`**. Structured programs are much easier to debug, because execution proceeds only through a limited number of well-defined pathways. When we use well-understood assembly branching structures, then our debugging can focus more on the overall function and less on how the details are implemented.

The Registers Must Be Saved. When working on a software team it is important to establish a rule whether or not subroutines will save/restore registers. Establishing this convention is especially important when a mixture of assembly and high-level language is being used, or if the software project remains active for long periods of time. It is safest to save and restore registers that are modified (most programmers do not save/restore the PSR) and output parameter(s) returned in a register. Exceptions to this rule can be made for those portions of the code where speed is most critical. According to AAPCS, we will preserve R4 through R11, but not preserve R0–R3, R12, or the PSR. Remember to always save the link register if your function calls another function. Stack operations function more efficiently if the SP remains aligned on an 8-byte boundry. We maintain the stack on an 8-byte alignment by pushing and popping an even number of registers. Subsequent function calls over-write the return location for your function, making yours unable to return. It is also important to ensure that the SP is the same on exit from your function as it was at entry.

Common Error: If the calling routine expects a subroutine to save/restore registers, and it doesn't, then information will be lost.

Observation: If the calling routine does not expect a subroutine to save/restore registers, and it does, then the system executes a little slower and the object code is a little bigger than it could be.

Common Error: When a mixture of C and assembly language programs are integrated, then an error may occur if the AAPCS rules are changed because there may be a change in if registers are saved/restored, or how parameters are passed.

Use High-Level Languages Whenever Possible. It may seem odd to have a rule about high-level languages in a section about assembly language programming. It is even odder to make this statement in a book devoted to assembly language programming. In general, we should use high-level languages when memory space and execution speed are less important than portability and maintenance. When execution speed is important, you could write the first version in a high-level language, run a profiler (that will tell you which parts of your program are executed the most), then optimize the sections of code using up the most execution time by writing them in assembly language. If a C language implementation just doesn't run fast enough, you could consider a more powerful compiler or a faster microcomputer.

Observation: High-level language programmers who are well acquainted with the underlying assembly language of the machine have a better understanding of how their machine and software work.

Minimize Conditional Branching. Every time software makes a conditional branch, there are two possible outcomes that must be tested (branch or not branch.) In the example shown in Program 5.23, assume we wish to set a 32-bit Flag if Port A bit 7 is true. A flag will be true if it is any nonzero value, and false if it is zero. A conditional branch could be avoided by solving the problem in another way. We will define a PA7 label to be the bit-specific address for this pin (0x40004200).

Observation: Software can be made easier to understand by reworking the approach in order to reduce the number of conditional branches.

Checkpoint 5.14: If a system has 20 conditional branches, how many potential execution paths might there be through the software?

```
PA7  EQU 0x40004200
; set flag with conditional branch
; Input: none
; Output: none
; Modifies: R1, R2
SetFlagConditional
       LDR R2, =PA7  ; R2 = 0x40004200
       LDR R1, [R2]  ; R1 = PA7
       LDR R2, =Flag ; R2 = &Flag
       CMP R1, #0x80 ; is PA7 == 0x80?
       BNE SFCClr
SFCSet MOV R1, #-1   ; PA7 is high
       STR R1, [R2]  ; Flag = -1
       B   SFCEnd
SFCClr MOV R1, #0    ; PA7 is low
       STR R1, [R2]  ; Flag = 0
SFCEnd BX  LR
; set flag with no conditional branch
; Input: none
; Output: none
; Modifies: R1, R2
SetFlagNoConditional
    LDR R2, =PA7  ; R2 = 0x40004200
    LDR R1, [R2]  ; R1 = PA7
    LDR R2, =Flag ; R2 = &Flag
    STR R1, [R2]  ; Flag = 0x00 or 0x80
    BX  LR
```

```
// uses conditional branch
void SetFlag(void){
  if(PA7){
    Flag = -1;   // PA7 is 0x80
  } else{
    Flag = 0;    // PA7 is 0
  }
}
// no conditional branch
void SetFlag(void){
  Flag = PA7; // 0 or 0x80
}
```

Program 5.23. Sometimes we can remove a conditional branch and simplify the program.

5.6.2. Comments

Discussion about comments was left for last, because they are the least important aspect involved in writing quality software. It is much better to write well-organized software with simple interfaces having operations so easy to understand that comments are not necessary.

The beginning of every file should include the file name, purpose, hardware connections, programmer, date, and copyright. E.g.,

```
; filename  adtest.s
; Test of TM4C123 12-bit ADC
; 1 Hz sampling and output to the serial port
; Last modified 7/1/15 by Jonathan W. Valvano
; Copyright 2015 by Jonathan W. Valvano, valvano@mail.utexas.edu
;    You may use, edit, run or distribute this file
;    as long as the above copyright notice remains
```

The beginning of every function should include a line delimiting the start of the function, purpose, input parameters, output parameters, and special conditions that apply. The comments at the beginning of the function explain the policies (e.g., how to use the function.) These comments, which are similar to the comments for the prototypes in the header file, are intended to be read by the client. E.g.,

```
;----------------------UART_InUDec-------------------------------
; Accepts ASCII input from the UART in unsigned decimal format
; and converts to a 32-bit unsigned number with a maximum of 65535
; If a number is above 2^32, it truncates without reporting error
; Backspace will remove last digit typed
; Inputs: none
; Outputs: Register R0 is the unsigned 32-bit value
```

Comments can be added to a variable or constant definition to clarify the usage. In particular, comments can specify the units of the variable or constant. For complicated situations, we can use additional lines and include examples. E.g.,

```
V1        SPACE  2  ; voltage at node 1 in mV, range -5000 mV to +5000 mV
Fs        SPACE  2  ; sampling rate in Hz
FoundFlag SPACE  1  ; 0 if keyword not yet found, 1 if found
RunMode   SPACE  1  ; 0, 1, 2, or 3 specifies system mode
; 0 means idle
; 1 means startup
; 2 means active run
; 3 means stopped
```

Comments can be used to describe complex algorithms. These types of comments are intended to be read by our coworkers. The purpose of these comments is to assist in changing the code in the future, or applying this code into a similar but slightly different application. Comments that restate the function provide no additional information, and actually make the code harder to read. Examples of bad comments include:

```
ADD R0,#1   ; add one to R0
MOV R1,#0   ; set R1 to 0
```

Good comments explain why the operation is performed, and what it means:

```
ADD R0,#1   ; maintain elapsed time in msec
MOV R1,# 0  ; switch to idle mode because no more data is available
```

We can add spaces so the comment fields line up. We should avoid tabs because they often do not translate well from one computer to another. In this way, the software is on the left and the comments can be read on the right.

I taught a large programming class one semester, and being an arrogant and lazy fellow, I thought I could write a grading program that accepts the students' programming assignments and automatically generates and records their grades. The second step will be to design a Massive Open Online Class, MOOC, on edX, and then I could teach the masses without ever having to show up for work. My grading program worked OK for the functional aspects of the students' software. My program generated inputs, called the students' program and compared the results with expected behavior. Where I utterly failed was in my attempts to automatically grade their software on style. I used the following three part "quality" statistic. First, I measured execution speed the student's software, s_i. Smaller times represent improved dynamic efficiency. Next, I measured the number of bytes in the object code, b_i. Again, a smaller number represents better static efficiency. Third, I used the number of ASCII characters in the source code, c_i, as a quantitative measure of documentation. For this parameter, bigger is better. In a typical statistical fashion, I used the average and standard deviation to calculate

$$\text{quality} = \frac{\bar{s} - s_i}{\sigma_s} + \frac{\bar{b} - b_i}{\sigma_b} + \frac{c_i - \bar{c}}{\sigma_c}$$

Half way through the semester, I happened to look at some assignments and was horrified to find the all-time worst software ever written from both a style and content basis. To improve speed and reduce size, the students cut so many corners that their code didn't really work anymore, it just appeared to work to my grading program. Then they took the ugly mess and filled it with nonsense comments, giving it the appearance of extensive documentation. To my students in that class that semester, I sincerely apologize. We should write comments for coworkers who must change our software, or clients who will use our software.

5.6.3. Inappropriate I/O and Portability

One of the biggest mistakes beginning programmers make is the inappropriate usage of I/O calls (e.g., screen output and keyboard input). An explanation for their foolish behavior is that they haven't had the experience yet of trying to reuse software they have written for one project in another project. Software portability is diminished when it is littered with user input/output. To reuse software with user I/O in another situation, you will almost certainly have to remove the input/output statements. In general, we avoid interactive I/O at the lowest levels of the hierarchy, rather return data and flags and let the higher level program do the interactive I/O. Often we add keyboard input and screen output calls when testing our software. It is important to remove the I/O that not directly necessary as part of the module function. This allows you to reuse these functions in situations where screen output is not available or appropriate. Obviously screen output is allowed if that is the purpose of the routine.

Common Error: Performing unnecessary I/O in a subroutine makes it harder to reuse at a later time.