Excerpts from
Introduction to Embedded Microcomputer Systems: Motorola 6811 and 6812 Simulation
Jonathan W. Valvano

1.2. Attitude

Good engineers employ well-defined design processes when developing complex systems. When we work within a structured frame work, 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, but most of the comments should apply to other situations 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 1.2:** 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 a mysterious and obscure fashion. 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, 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 previous 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 software design. Our testing and the client's usage go hand in hand. In particular, how we test the software module will help the client understand the context and limitations of how our software is to be used. On the other hand, a clear understanding of how the client wishes to use our software is critical for both the software 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 software system that runs slow than one that does run at all.*

In the early days of microcomputer systems, software size could be measured in 100's of lines of source code or 1000's of bytes of object code. These early systems, due to their small size, were inherently simple. The explosion of hardware technology (both in speed and size) has lead to a similar increase in the size of software 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 can not 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. Use our creativity to break a complex problem into simple components, rather than developing complex solutions to simple problems.

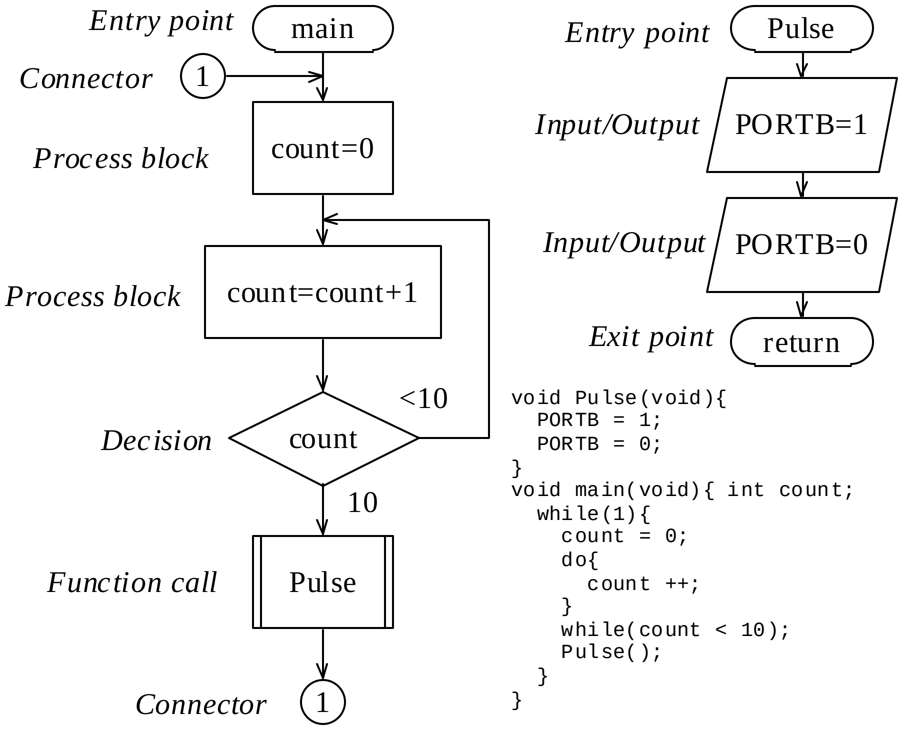
> ***Observation****: There are two ways of constructing a software design: one way is to make it so simple that there are* obviously *no deficiencies and the other way is make it so complicated that there are no* obvious *deficiencies. C.A.R. Hoare, "The Emperor's Old Clothes," CACM Feb. 1981.*

1.4. Flowcharts and structured programming

Next, we introduce the flowchart syntax that will be used throughout the book. Programs themselves are written in a linear or one-dimensional fashion. Writing programs this way is a natural process, because the computer itself usually executes the program in a top-to-bottom sequential fashion. This one-dimensional format is fine for simple programs, but conditional branching and function calls may create complex behaviors that are not easily observed in a linear fashion. Flowcharts are one way to describe software in a two-dimensional format, specifically providing convenient mechanisms to visualize conditional branching and function calls. You will not have to draw a flowchart for every program you write. However, flowcharts are very useful in the initial design stage of a software system to define complex algorithms. Flowcharts can also be used in the final documentation stage of a project, once the system is operational, in order to assist in its use or modification.

> ***Observation****:* ***TExaS*** *is one of the few software development systems that allow you to add flowcharts directly into your software as part of its documentation.*

Figure 1.2 illustrates the flowchart syntax, showing both the flowcharts and corresponding C program. The oval shapes define entry and exit points. The main *entry point* is the starting point of the software. Each function, or subroutine, also has an entry point. The *exit point* returns the flow of control back to the place from which the function was called. When the software runs continuously, as is typically the case in an embedded system, there will be no main exit point. We use rectangles to specify *process* blocks. In a high-level flowchart, a process block might involve many operations, but in a low-level flowchart, the exact operation is defined in the rectangle. The parallelogram will be used to define an *input/output* operation. Some flowchart artists use rectangles for both processes and input/output. Since input/output operations are an important part of embedded systems, we will use the parallelogram format, which will make it easier to identify input/output in our flowcharts. The diamond-shaped objects define a branch point or *decision* block. The rectangle with double lines on the side specify a call to a *predefined function*. In this book, functions, subroutines and procedures are terms that all refer to a well-defined section of code that performs a specific operation. Functions usually return a result parameter, while procedures usually do not. Functions and procedures are terms used when describing a high-level language, while subroutines often used when describing assembly language. When a function (or subroutine or procedure) is called, the software execution path jumps to the function, the specific operation is performed, and the execution path returns to the point immediately after the function call. Circles are used as *connectors*.



```
void Pulse(void){
  PORTB = 1;
  PORTB = 0;
}
void main(void){ int count;
  while(1){
    count = 0;
    do{
      count ++;
    }
    while(count < 10);
    Pulse();
  }
}
```

Figure 1.2. Example flowchart showing some common flowchart symbols.

> **Common error.** *In general, it is bad programming style to develop software that requires a lot of connectors when drawing its flowchart.*

> **Checkpoint 1.7:** *Using a flowchart describe the control algorithm that a toaster must use to cook toast. Assume the inputs are toast temperature in F, and desired temperature in F. The output is heat (on/off).*

There are an almost infinite number of operations one can perform on a computer, and the key to developing great products is to select the correct ones. Just like hiking through the woods, we need to develop guidelines (like maps and trails) to keep us from getting lost. One of the fundamental issues when developing software, especially in assembly language, is to maintain a consistent structure. One such framework is called **structured programming**. Most high-level languages (with the exception of `goto`) force the programmer to write structured programs. Structured programs are built from three basic building blocks: the **sequence**, the **conditional**, and the **while-loop**. At the lowest level, the process block contains simple and well-defined commands, like the process blocks shown in Figure 1.2. I/O functions are also low-level building blocks. Structured programming involves combining existing blocks into more complex structures, as shown in Figure 1.3.

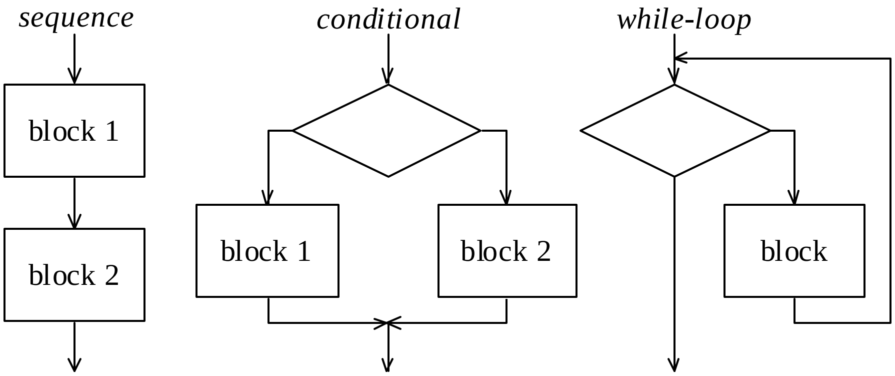


Figure 1.3. Flowchart showing the basic building blocks of structured programming.

1.5. Product development cycle

In this section, we will introduce the product development process in general. The basic approach is introduced here, and the details of these concepts will be presented throughout the remaining chapters of the book. As we learn software/hardware development tools and techniques, we can place them into the framework presented in this section. As illustrated in Figure 1.4, the development of a product follows an analysis -design-implementation-testing cycle. For complex systems with long life-spans, we tranverse multiple times around the development cycle. For simple systems, a one-time pass may suffice.

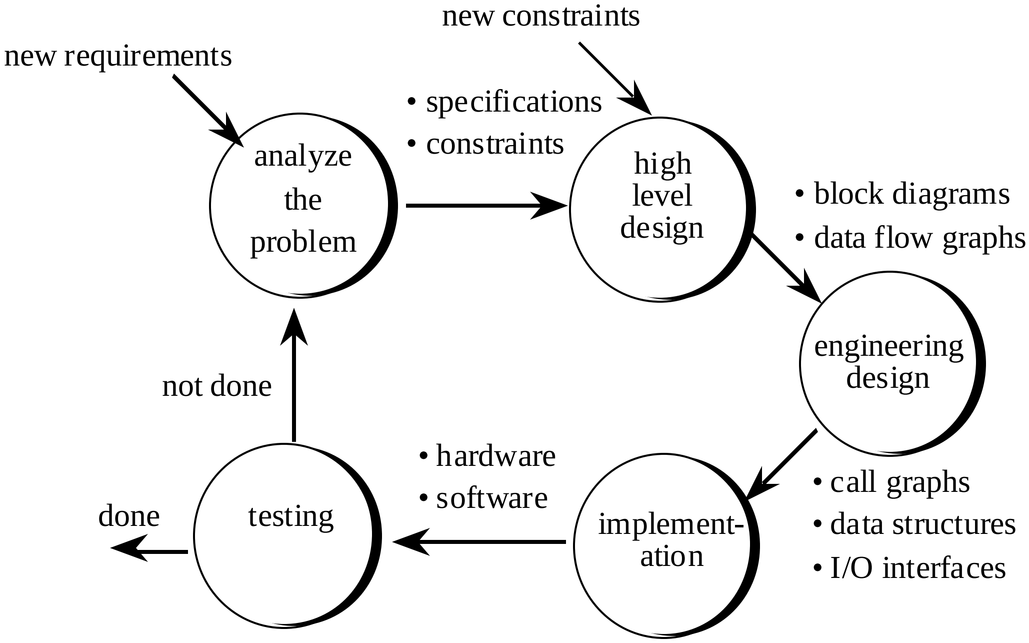


Figure 1.4. Software development cycle.

During the **analysis phase**, we discover the requirements and constraints for our proposed system. We can hire consultants and interview potential customers in order to gather this critical information. A *requirement* is a specific parameter that the system must satisfy. We begin by rewriting the system requirements, which are usually written in general form, into a list of detailed *specifications*. In general, specifications are detailed parameters describing how the system should work. For example, a requirement may state that the system should fit into a pocket, whereas a specification would give the exact size and weight of the device. For example, suppose we wish to build a thermometer. During the analysis phase, we would determine obvious specifications such as range, resolution, accuracy, and speed. There may be less obvious requirements to satisfy, such as weight, size, battery life, product life, ease of calibration, display readability, and reliability. A *constraint* is a limitation, within which the system must operate. The system may be constrained to such factors as compatibility with other products, use of specific electronic and mechanical parts as other devices, interfaces with other instruments and test equipment, and development schedule.

Checkpoint 1.8: *What's the difference between a requirement and a specification?*

During the **high-level design** phase, we build a conceptual model of the hardware/software system. It is in this model that we exploit as much abstraction as appropriate. The project is broken in modules or subcomponents. Modular design will be presented in Chapter 9. During this phase, we estimate the cost, schedule, and expected performance of the system. At this point we can decide if the project has a high enough potential for profit. A *data flow graph* is a block diagram of the system, showing the flow of information. Arrows point from source to destination. The rectangles represent hardware components and the ovals are software modules. We use data flow graphs in the high-level design, because they describe the overall operation of the system while hinding the details of how it works. A data flow graph for a simple thermometer is shown in Figure 1.5. The sensor converts temperature in an electrical resistance. The amplifier converts resistance into the 0 to +5V voltage range required by the ADC. The ADC converts analog voltage into a digital sample. The `ADC routines`, using the ADC and timer hardware, collect samples and calculate voltages. The `calculation` software uses a table data structure to convert voltage to temperature. Voltage and temperature data are represented as fixed-point numbers within the computer. The temperature data is passed to the `LCD routines` creating ASCII strings, which will be sent to the *liquid crystal display* (LCD) module. The user will be able to select the Fahrenheit or Centigrade scale using a switch.

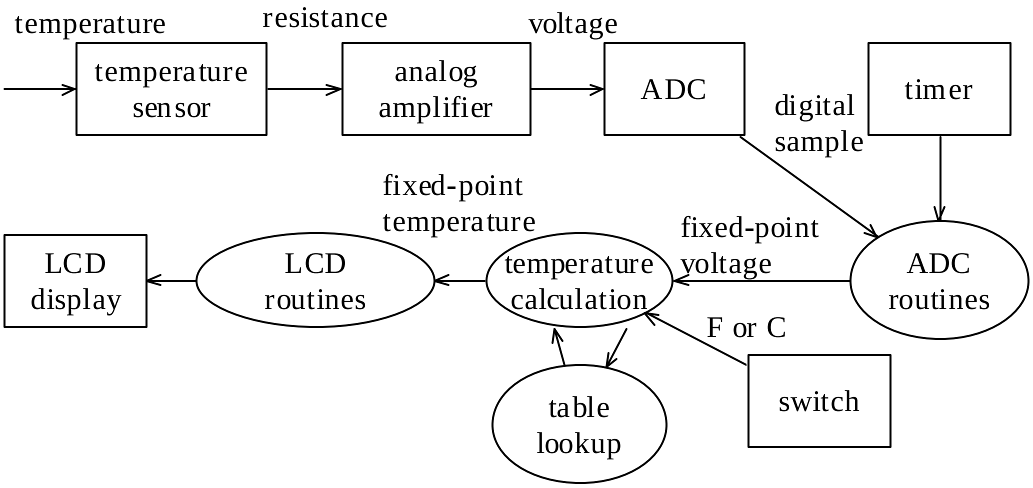


Figure 1.5. A data flow graph showing how the temperature signal passes through a simple thermometer.

The next phase is **engineering design**. We begin by constructing a preliminary design. This system includes the overall top down hierarchical structure, the basic I/O signals, shared data structures and overall software scheme. At this stage there should be a simple and direct correlation between the hardware/software systems and the conceptual model developed in the high-level design. Next, we finish the top down hierarchical structure, and built mock-ups of the mechanical parts (connectors, chassis, cables etc.) and user software interface. Sophisticated 3-D CAD systems can create realistic images of our system. Detailed hardware designs must include mechanical drawings. It is a good idea to have a second source, which is an alternative supplier that can sell our parts if the first source can't deliver on time. *Call-graphs* are a graphical way to define how the software/hardware modules interconnect. *Data structures*, which will be presented in Chapter 10, include both the organization of information and mechanisms to access the data. A call-graph for a simple thermometer is shown in Figure 1.6. Again, rectangles represent hardware components and ovals show software modules. An arrow points from the calling routine to the module it calls. The I/O ports are organized into groups and placed at the bottom of the graph. A high-level call-graph, like the one shown in Figure 1.6, shows only the high-level hardware/software modules. A detailed call-graph would include each software function and I/O port. Normally, hardware is passive and the software initiates hardware/software communication, but as we will learn in Chapter 11, it is possible for the hardware to interrupt the software and cause certain software modules to be run. In this system, the timer hardware will cause the `ADC` software to collect a sample. The `main program` gets the next sample from the `ADC` software, converts it to temperature, and displays the result by calling the `LCD` interface software.

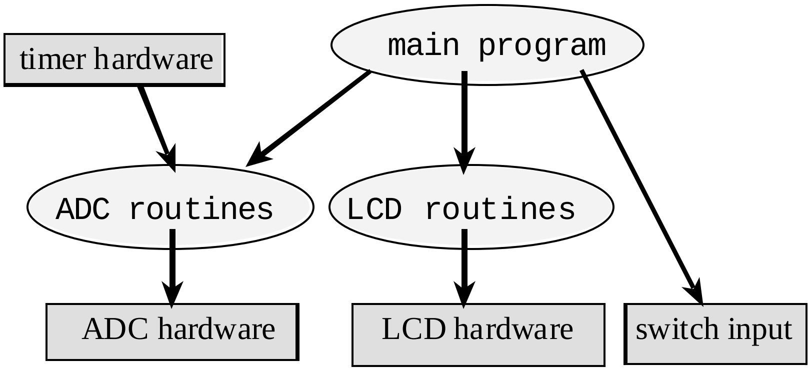


Figure 1.6. A call flow graph for a simple thermometer.

> ***Observation**: If module A calls module B, and B returns data, then a data flow graph will show an arrow from B to A, but a call-graph will show an arrow from A to B.*

The next phase is **implementation**. An advantage of a top-down design is that implementation of subcomponents can occur concurrently. During the initial iterations of the development cycle, it is quite efficient to implement the hardware/software using simulation. One major advantage of simulation is that it is usually quicker to implement an initial product on a simulator versus constructing a physical device out of actual components. Rapid prototyping is important in the early stages of product development. This allows for more loops around the analysis-design-implementation-testing cycle, which in turn leads to a more sophisticated product.

Recent software and hardware technological developments have made significant impacts on the software development for embedded microcomputers. The simplest approach is to use a cross-assembler or cross-compiler to convert source code into the machine code for the target system. The machine code can then be loaded into the target machine. Debugging embedded systems with this simple approach is very difficult for two reasons. First, the embedded system lacks the usual keyboard and display that assist us when we debug regular software. Second, the nature of embedded systems involves the complex and real-time interaction between the hardware and software. These real-time interactions make it impossible to test software with the usual single-stepping and print statements.

The next technological advancement that has greatly affected the manner in which embedded systems are developed is simulation. Because of the high cost and long times required to create hardware prototypes, many preliminary feasibility designs are now performed using hardware/software simulations. A simulator is a software application that models the behavior of the hardware/software system. If both the external hardware and software program are simulated together, even although the simulated time is slower than the clock on the wall, the real-time hardware/software interactions can be studied.

During the **testing** phase, we evalutate the performance of our system. First, we debug the system and validate basic functions. Next, we use careful measurements to optimize performance such as static efficiency (memory requirements), dynamic efficiency (execution speed), accuracy (difference between truth and measured), and stability (consistent operation.)

Maintenance is the process of correcting mistakes, adding new features, optimizing for execution speed or program size, porting to new computers or operating systems, and reconfiguring the system to solve a similar problem. No system is static. Customers may change or add requirements or constraints. To be profitable, we probably will wish to tailor each system to the individual needs of each customer. Maintenance is not really a separate phase, but rather involves additional loops around the development cycle.

1.6. Quality Programming

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 around 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 does work and can't be changed.*

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

1.6.1. 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 CPU 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 object code. The global variables plus maximum stack size must be less than the available RAM. Similarly, the fixed constants plus program size must be less than the ROM size. We can judge our software system according to whether or not it satisfies given constraints, like software development costs, memory available, and time-table.

1.6.2. Qualitative Performance Measurements

Qualitative performance measurements include those parameters to which we can not 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 a immediate and direct relationship between a software's quality and profit, we may be mistakenly tempted to dismiss their importance.

To get a benchmark on how good a programmer you are, take the following two challenges. In the first test, find a major piece of software that you have written over 12 months ago, then see if you can still understand it enough to make minor changes in its behavior. The second test 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.*

9.3. Naming convention

Choosing names for variables and functions involves creative thought, and it 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 out of the context of the program in which it exists, 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 display.

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. An underline character will separate the module name from the function name. As an exception to this rule, we can use the underline to delimit words in all upper-case name (e.g., `#define MIN_PRESSURE 10`). Functions that can be accessed outside the scope of a module 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 file name containing the object. For example, if we see a function call, `LCD_OutString("Hello world");` we know the public function belongs to the LCD module, where the policies are defined in `LCD.h` and the implementation in `LCD.c`. Notice the similarity between this syntax (e.g., `LCD_init()`) and the corresponding syntax we would use if programming the module as a class in C++ (e.g., `LCD.init()`). Using this convention, we can easily distinguish public and private objects. If the variable is public, because the name has an underline, then the first letter of the name after the underline should be capitalized (e.g., `my_Count` is a public variable belonging the the module "`my`" and defined in the header file `my.h`.)

Use upper and lower case to specify the scope of an object. We will define I/O ports and 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`. Global objects will begin with a capital letter, but include some lower-case letters. Local variables will begin with a lower-case letter, and may or may not include upper case letters. Since all functions are global, 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 is the local or global. Some programmers use the underline as a word-delimiter, but except for constants, we will reserve underline to separate the module name from the variable name. Table 9.1 overviews the naming convention presented in this section.

Checkpoint 9.5: *How can tell if a function is private or pubic?*

Checkpoint 9.6: *How can tell if a variable is local or global?*

type	examples
constants	`CR SAFE_TO_RUN PORTA STACK_SIZE START_OF_RAM`
local variables	`maxTemperature lastCharTyped errorCnt`
private global variable	`MaxTemperature LastCharTyped ErrorCnt`
public global variable	`DAC_MaxTemperature Key_LastCharTyped Network_ErrorCnt file_OpenFlag`
private function	`ClearTime wrapPointer InChar`
public function	`Timer_ClearTime RxFifo_Put Key_InChar`

Table 9.1. Examples of names.

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

short my_Data1; // in global ram, public
static short Data2; // in global ram, private
const short DATA3=5; // in EEPROM, if in C file, public if in h file

9.5. C language Style Guidelines

9.5.1. Code File Structure, the *.c file

One of the recurring themes of this software style section is consistency. Maintaining a consistent style will help us locate and understand the different components of our software, as well as prevent us from forgetting to include a component or worse including it twice. The following regions should occur in this order in every code file (e.g., `file.c`).

Opening comments. The first line of every file should contain the file name. This is because some printers do not automatically print the name of the file. Remember that these opening comments will be duplicated in the corresponding header file (e.g., `file.h`) and are intended to be read by the client, the one who will use these programs. If major portions of this software are copied from copyrighted sources, then we must satisfy the copyright requirements of those sources. The rest of the opening comments should include

- the overall purpose of the software module,
- the names of the programmers,
- the creation (optional) and last update dates,
- the hardware/software configuration required to use the module, and
- any copyright information.

Including .h files. Next, we will place the `#include` statements that add the necessary header files. Adding other code files, if necessary, will occur at the end of the file, but here at the top of the file we include just the header files. Normally the order doesn't matter, so we will list the include files in a hierarchical fashion starting with the lowest level and ending at the highest high. If the order of these statements is important, then write a comment describing both what the proper order is and why the order is important. Putting them together at the top will help us draw a call-graph, which will show us how our modules are connected. In particular, if we consider each code file to be a separate module, then the list of `#include` statements specifies which other modules can be called from this module. Of course one header file is allowed to include other header files. In general, we should avoid having one header file include other header files. This restriction makes the organizational structure of the software system easier to observe. Be careful to include only those files that are absolutely necessary. Adding unnecessary include statements will make our system seem more complex than it actually is.

`extern` references. After including the header files, we can declare any external variables or functions. External references will be resolved by the linker, when various modules are linked together to

create a single executable application. Placing them together at the top of the file will help us see how this software system fits together (i.e., is linked to) other software systems.

#define statements. After external references, we should place the #define macros and #define constants. Since these definitions are located in the code file (e.g., file.c), they will be private. This means they are available within this file only. If the client does not need to use or change the macro or constant, then it should be made private by placing it here in the code file. Conversely, if we wish to create a public constant or macro, then we place it in the header file for this module.

struct union enum statements. After the define statements, we should create the necessary data structures using struct union and enum. Again, since these definitions are located in the code file (e.g., file.c), they will be private.

Global variables and constants. After the structure definitions, we should include the global variables and constants. If we specify the global as static then it will be private, and can only be accessed by programs in this file. If we do not specify the global as static then it will be public, and can only be accessed any program (that program defines it as extern and the linker will resolve the reference). We put all the globals together before any function definitions to symbolize the fact that any function in this file has access to these globals. If we have a permanent variable that is only access by one function, then it should be defined as a static local. The **scope** of a variable includes all the software in the system that can access it. In general, we wish to minimize the scope of our data.

```
short publicGlobal; // accessible by any function via extern declaration
static short privateGlobal; // accessible in this file only
void function(void){
static short veryPrivateGlobal; // accessible by this function only
}
```

Maintain order in our system by restricting direct access to our data.

Prototypes of private functions. After the globals, we should add any necessary prototypes. Just like global variables, we can restrict access to private functions by defining them as static. Prototypes for the public functions will be included in the corresponding header file. In general, we will arrange the code implementations in a top-down fashion. Although not necessary, we will include the parameter names with the prototypes. Descriptive parameter names will help document the usage of the function. For example, which of the following prototypes is easier to understand?

```
static void plot(short, short);
static void plot(short time, short pressure);
```

Implementations of the functions. The heart of the implementation file will be, of course, the implementations. Again, private functions should be defined as static. The functions should be sequenced in a logical manner. The most typical sequence is top-down, meaning we begin with the highest level and finish with the lowest level. Another appropriate sequence mirrors the manner in which the functions will be used. For example, start with the initialization functions, followed by the operations, and end with the shutdown functions. For example:

open
input
output
close

Including .c files. At the end of the file, we will place the #include statements that add the necessary code files. If our compiler supports projects, then it is a good idea to take advantage of this feature. The project simplifies the management of large software systems by providing organizational structure to the software system. If we use projects, then including code files will be unnecessary, and hence should be avoided. If our compiler does not support projects, or if we are writing software for multiple compilers, then including code files allows a large software project to be constructed simply by compiling the file with the main() program in it. Including header files at the top of the file allows this module to accessing public variables and functions of the other module. On the other hand, including code files at the end of the file prevents this module from accessing private variables and functions of the other module.

If our compiler supports **assert()** functions, use them liberally. In particular, place them at the beginning of functions to test the validity of the input parameters. Place them after calculations to test the validity of the results. Place them inside loops to verify indices and pointers are valid. There is a secondary benefit to using **assert()**. The **assert()** statements themselves provide documentation of the assumptions made by the programmer.

9.5.2. Header File Structure, the *.h file

Once again, maintaining a consistent style facilitates understanding and helps to avoid errors of omission. Definitions made in the header file will be public, i.e., accessible by all modules. As stated earlier, it is better to make global variables private rather than placing them in the header file . Similarly, we should avoid placing actual code in a header file.

There are two types of header files. The first type of header file has no corresponding code file. In other words, there is a `file.h`, but no `file.c`. In this type of header, we can list global constants and helper macros. Examples of global constants are I/O port addresses (e.g., `HC12.h`) and calibration coefficients. Debugging macros could be grouped together and placed in a `debug.h` file. We will not consider software in these types of header files as belonging to a particular module.

The second type of header file does have a corresponding code file. The two files, e.g., `file.h`, and `file.c`, form a software module. In this type of header, we define the prototypes for the public functions of the module. The `file.h` contains the policies (behavior or what it does) and the `file.c` file contains the mechanisms (functions or how it works.) The following regions should occur in this order in every header file (e.g., `file.h`).

Opening comments. The first line of every file should contain the file name. This is because some printers do not automatically print the name of the file. Remember that these opening comments should be duplicated in the corresponding header file (e.g., `file.c`) and are intended to be read by the client, the one who will use these programs. We should repeat copyright information as appropriate. The rest of the opening comments should include

- the overall purpose of the software module,
- the names of the programmers,
- the creation (optional) and last update dates,
- the hardware/software configuration required to use the module, and
- any copyright information.

Including .h files. Nested includes in the header file should be avoided. As stated earlier, nested includes obscure the manner in which the modules are interconnected. On the other, an implementation file can include other header and implementation files.

`#define` *statements*. Public constants and macros are next. Special care is required to determine if a definition should be made private or public. One approach to this question is to begin with everything defined as private, and then shift definitions into the public category only when deemed necessary for the client to access in order to use the module. If the parameter relates to what the module does or how to use the module, then it should probably be public. On the other hand, if it relates to how it works or how it is implemented, it should probably be private.

`struct union enum` *statements*. The definitions of public structures allow the client software to create data structures specific for this module.

Global variables and constants. If at all possible, public global variables should be avoided. Public constants follow the same rules as public definitions. If the client must have access to a constant to use the module, then it could be placed in the header file.

Prototypes of public functions. The prototypes for the public functions are last. Just like the implementation file, we will arrange the code implementations in a top-down fashion. Comments should be directed to the client, and these comments should clarify what the function does and how the function can be used.

9.5.3. Formatting

The rules set out in this subsection are not necessary for the program to compile or to run. Rather the intent of the rules are to make the software easier to understand, easier to debug, and easier to change.

Just like beginning an exercise program, these rules may be hard to follow at first, but the discipline will pay dividends in the future.

Make the software easy to read. I strongly object to hardcopy printouts of computer programs during the development phase of a project. At this time, there are frequent updates made by multiple members of the software development team. Because a hardcopy printout will be quickly obsolete, we should develop and debug software by observing it on the computer screen. In order to eliminate horizontal scrolling, no line of code should be more than 80 characters wide. If we do make hard copy printouts of the software at the end of a project, this rule will result in a printout that is easy to read.

Indentation should be set at 2 spaces. When transporting code from one computer to another, the TAB settings may be different. So, what looks good on one computer may look ugly on another. For this reason, we should avoid TABs and use just spaces. Local variable definitions can go on the same line as the function definition, or in the first column on the next line.

Be consistent about where we put spaces. Similar to English punctuation, there should be no space before a comma or a semicolon, but there should be at least one space or a carriage return after a comma or a semicolon. There should be no space before or after open or close parentheses. Assignment and comparison operations should have a single space before and after the operation. One exception to the single space rule is if there are multiple assignment statements, we can line up the operators and values. For example

```
  data     = 1;
  pressure = 100;
  voltage  = 5;
```

Be consistent about where we put braces {}. Misplaced braces cause both syntax and semantic errors, so it is critical to maintain a consistent style. Place the opening brace at the end of the line that opens the scope of the multi-step statement. The only code that can go on the same line after an opening brace is a simple local variable declaration or a comment. Placing the open brace near the end of the line provides a visual clue that a new code block has started. Place the closing brace on a separate line to give a vertical separation showing the end of the multi-step statement. The horizontal placement of the close brace gives a visual clue that the following code is in a different block. For example

```
void main(void){ int i, j, k;
  j = 1;
  if(sub0(j)){
    for(i = 0; i < 6; i++){
      sub1(i);
    }
    k = sub2(i, j);
  }
  else{
    k = sub3();
  }
}
```

Use braces after all if else for do while case and switch commands, even if the block is a single command. This forces we to consider the scope of the block making it easier to read and easier to change. For example, assume we start the following code.

```
  if(flag)
    n = 0;
```

Now, we add a second statement that we want to execute also if the flag is true. The following error might occur if we just add the new statement.

```
  if(flag)
    n = 0;
    c = 0;
```

If all of our blocks are enclosed with braces, we would have started with the following.

```
  if(flag){
    n = 0;
  }
```

Now, when we add a second statement, we get the correct software.

```
  if(flag){
    n = 0;
    c = 0;
  }
```

9.5.4. Code Structure

Make the presentation easy to read. We define presentation as the look and feel of our software as displayed on the screen. If at all possible, the size of our functions should be small enough so the majority of the code fits on a single computer screen. We must consider the presentation as a two-dimensional object. Consequently, we can reduce the 2-D *area* of our functions by encapsulating components and defining them as private functions, or by combining multiple statements on a single line. In the horizontal dimension, we are allowed to group multiple statements on a single line only if the collection makes sense. We should list multiple statements on a single line, if we can draw a circle around the statements and assign a simple collective explanation to the code.

> ***Observation:*** *Most professional programmers do not create hard copy printouts of the software. Rather, software is viewed on the computer screen.*

Another consideration related to listing multiple statements on the same line is debugging. The compiler often places debugging information on each line of code. For example, the ICC11 and ICC12 compilers place a label specifying the starting address of the assembly code that implements the line. Breakpoints in some systems can only be placed at the beginning of a line.

Consider the following three presentations. Since the compiler generates exactly the same code in each case, the computer execution will be identical. Therefore, we will focus on the differences in style. The first example has a horrific style.

```
void testFilter(short start, short stop, short step){ short x,y;
  initFilter(); SCI_OutString("x(n) y(n)"); SCI_OutChar(CR);
  for(x=start;x<=stop; x=x+step){ y=filter(x); SCI_OutUDec(x);
  SCI_OutChar(SP); SCI_OutUDec(y); SCI_OutChar(CR);} }
```

The second example places each statement on a separate line. Although written in an adequate style, it is unnecessarily vertical.

```
void testFilter(short start, short stop, short step){
short x;
short y;
  initFilter();
  SCI_OutString("x(n) y(n)");
  SCI_OutChar(CR);
  for(x = start; x <= stop; x = x+step){
    y = filter(x);
    SCI_OutUDec(x);
    SCI_OutChar(SP);
    SCI_OutUDec(y);
    SCI_OutChar(CR);
  }
}
```

The last example groups the two variable definitions together because the collection can be considered as a single object. The variables are related to each other. Obviously, x and y are the same type (16-bit signed), but in a physical sense, they would have the same units. For example, if x represents a signal in mV, then y is also a signal in mV. Similarly, the SCI output sequences cause simple well-defined operations.

```
void testFilter(short start, short stop, short step){ short x, y;
  initFilter();
  SCI_OutString("x(n) y(n)"); SCI_OutChar(CR);
  for(x = start; x <= stop; x = x+step){
    y = filter(x);
    SCI_OutUDec(x); SCI_OutChar(SP); SCI_OutUDec(y); SCI_OutChar(CR);
  }
}
```

The "make the presentation easy to read" guideline sometimes comes in conflict with the "be consistent where we place braces" guideline. For example, the following example is obviously easy to read, but violates the placement of brace rule.

```
  for(i = 0; i < 6; i++) dataBuf[i] = 0;
```

When in doubt, we will always be consistent where we place the braces. The correct style is also easy to read.

```
  for(i = 0; i < 6; i++){
    dataBuf[i] = 0;
  }
```

Employ modular programming techniques. Complex functions should be broken into simple components, so that the details of the lower-level operations are hidden from the overall algorithms at the higher levels. An interesting question arises:

"Should a subfunction be defined if it will only be called from a single place?"

The answer to this question, in fact the answer to all questions about software quality, is yes if it makes the software easier to understand, easier to debug, and easier to change.

Minimize scope. In general, we hide the implementation of our software from its usage. The scope of a variable should be consistent with how the variable is used. In a military sense, we ask the question, *"Which software has the need to know?"* Global variables should be used only when the lifetime of the data is permanent, or when data needs to be passed from one thread to another. Otherwise, we should use local variables. When one module calls another, we should pass data using the normal parameter-passing mechanisms. As mentioned earlier, we consider I/O ports in a manner similar to global variables. There is no syntactic mechanism to prevent a module from accessing an I/O port, since the ports are at fixed and known absolute addresses. The Intel Pentium does have a complex hardware system to prevent unauthorized software from accessing I/O ports, but the details are beyond the scope of this book. So for our embedded system, we must rely on the **does-access** rather than the **can-access** method. In other words, we must have the discipline to restrict I/O port access only in the module that is designed to access it. For similar reasons, we should consider each interrupt vector address separately, grouping it with the corresponding I/O module. In particular, rather than having one long list of interrupt vectors for the entire system, each interrupt vector should be separately defined along with the software that supports the other I/O hardware of the module. For example, the serial port interrupt vector should be specified in the same file as the serial port interrupt handler.

Use types. Using a `typedef` will clarify the format of a variable. It is another example of the separation of mechanism and policy. New data types and structures will begin with an upper case letter. The `typedef` allows use to hide the representation of the object and use an abstract concept instead. For example

```
typedef short Temperature;
void main(void){ Temperature lowT, highT;
}
```

This allows us to change the representation of temperature without having to find all the temperature variables in our software. Not every data type requires a `typedef`. We will use types for those objects of fundamental importance to our software, and for those objects for which a change in implementation is anticipated. As always, the goal is to clarify. If it doesn't make it easier to understand, easier to debug, or easier to change, don't do it.

Prototype all functions. Public functions obviously require a prototype in the header file. In the implementation file, we will organize the software in a top-down hierarchical fashion. Since the highest level functions go first, prototypes for the lower-level private functions will be required. Grouping the low-level prototypes at the top provides a summary overview of the software in this module. Include both the type and name of the input parameters. Specify the function as `void` even if it has no parameters. These prototypes are easy to understand:

```
void start(unsigned short seriod, void(*functionPt)(void));
short divide(short dividend, short divisor);
void SCI_Init(void);
```

These prototypes are harder to understand:

```
start(unsigned short, (*)());
short divide(short, short);
SCI_Init();
```

Declare function return types explicitly. In general, we can remove ambiguities by clarifying exactly what we want. Unless the number of parameters is large, we will place the return type, the function name, and the input parameters on a single line. If there is still room within the 80-character line limit, we can add some local variable declarations to this line. The following are good examples of the first line of several functions.

```
void main(void){ int i;
void SCI_OutUDec(unsigned short number){
unsigned short SCI_InUHex(void){
int RxFifo_Put(char data){
```

Declare data and parameters as `const` *whenever possible.* Declaring an object as const has two advantages. The compiler can produce more efficient code when dealing with parameters that don't change. The second advantage is to catch software bugs, i.e., situations where the program incorrectly attempts to modify data that it should not modify.

goto statements are not allowed. Debugging is hard enough without adding the complexity generated when using `goto`. A corollary to this rule is when developing assembly language software, we should restrict the branching operations to the simple structures allowed in C.

++ and -- should not appear in complex statements. These operations should only appear as commands by themselves. Again, the compiler will generate the same code, so the issue is readability. The statement

```
  *(--pt) = buffer[n++];
```

should have been written as

```
  --pt;
  *(pt) = buffer[n];
  n++;
```

If it makes sense to group, then put them on the same line. The following code is allowed

```
  buffer[n] = 0; n++;
```

Be a parenthesis zealot. When mixing arithmetic, logical, and conditional operations, explicitly specify the order of operations. Do not rely on the order of precedence. As always, the major issue is clarity. Even if the following code were actually to perform the intended operation (which in fact it does not),

```
  if( x + 1 & 0x0F == y | 0x04)
```

the programmer assigned to modify it in the future will have a better chance if we had written

```
  if( ((x + 1) & 0x0F) == (y | 0x04))
```

Use `enum` instead of `#define` or `const`. The use of `enum` allows for consistency checking during compilation, and provides for easy to read software. A good optimizing compiler will create the exact object code for the following four examples. So once again, we focus on style. In the first example, we needed comments to explain the operations.

```
int Mode;  // 0 means error
void function1(void){
  Mode = 1; // no error
}

void function2(void){
  if(Mode == 0){ // error?
    SCI_OutString("error");
  }
}
```

In the second example, no comments are needed.

```
#define NOERROR 1
#define ERROR 0
int Mode;
void function1(void){
  Mode = NOERROR;
}
void function2(void){
  if(Mode == ERROR){
    SCI_OutString("error");
  }
}
```

In the third example, the compiler performs a type-match, making sure `mode`, `NOERROR`, and `ERROR` are the same type.

```
const int NOERROR = 1;
const int ERROR = 0;
int Mode;
void function1(void){
  Mode = NOERROR;
}
void function2(void){
  if(Mode == ERROR){
    SCI_OutString("error");
  }
}
```

Enumeration provides a check of both type and value. We can explicitly set the values of the enumerated types if needed.

```
enum Mode_state{ ERROR, NOERROR};
enum Mode_state mode;
void function1(void){
  mode = NOERROR;
}
void function2(void){
  if(mode == ERROR){
    SCI_outString("error");
  }
}
```

Don't use bit-shift for arithmetic operations. Microcomputer architectures and compilers used to be so limited that it made sense to perform multiply/divide by 2 using a shift operation. For example, when multiplying a number by 4, we might be tempted to write `data<<2`. This is wrong; if the operation is multiply, we should write `data*4`. Compiler optimization has developed to the point where the compiler can choose to implement `data*4` as either a shift or multiply depending on the instruction set of the computer. When we use `data*4`, we have code that is easier to understand than `data<<2`.

Reference:

Mike Dahlin, ed., "LESS Software Engineering", coding standard for the LESS group.

9.7. 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.c
// Test of 6812 8-bit ADC
// 1 Hz sampling and output to the serial port
// Last modified 1/7/05 by Jonathan W. Valvano
// Copyright 2005 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.,

```
//---------------------SCI_InUDec-----------------------------
// InUDec accepts ASCII input in unsigned decimal format
//     and converts to a 16 bit unsigned number
//     valid range is 0 to 65535
// Input: none
// Output: 16-bit unsigned number
// If you enter a number above 65535, it will truncate without an error
// Backspace will remove last digit typed
```

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 addition lines and include examples. E.g.,

```
short V1;            // voltage at node 1 in mV, range -5000 mV to +5000 mV
unsigned short Fs;   // sampling rate in Hz
int FoundFlag;       // 0 if keyword not yet found, 1 if found
unsigned short Mode; // determines system action, as one of the following three cases
#define IDLE 0
#define COLLECT 1
#define TRANSMIT 2
```

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:

```
  time++;    // add one to time
  mode = 0;  // set mode to zero
```

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

```
  time++;    // maintain elapsed time in msec
  mode = 0;  // switch to idle mode because no more data is available
```

We can add spaces so the comment fields line up. As stated earlier, we avoid tabs because they often do not translate from one system 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 would have been to write a self-study book, 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

$$\text{quality} = \frac{\bar{s}-s_i}{s_s} + \frac{\bar{b}-b_i}{s_b} + \frac{c_i-\bar{c}}{s_c}$$

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.

Appendix 9. Solutions Manual

A9.1. Checkpoint Solutions

Checkpoint 1.2: It failed because employees were rewarded for poor behavior. It is much better to punish poor behavior and reward good behavior.

Checkpoint 1.7:

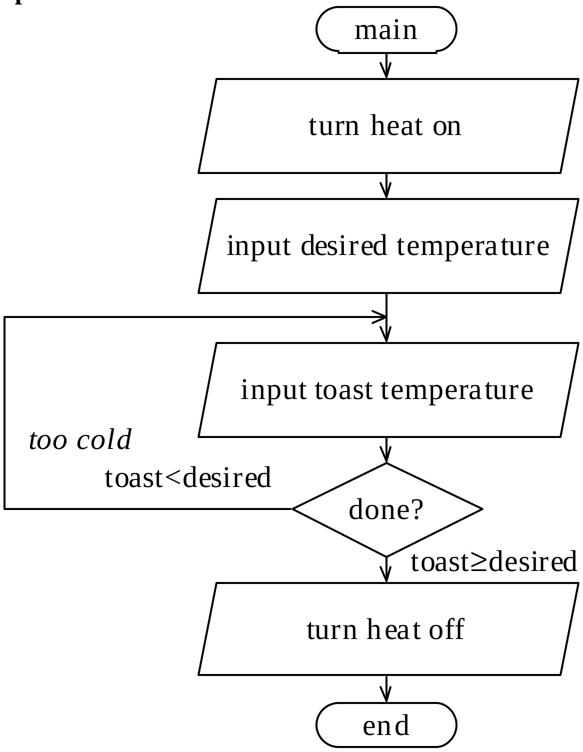


Figure A9.1. Flowchart showing a toaster algorithm.

Checkpoint 1.8: Both terms refer to parameters of a system, but the differences lie in the level of detail used to describe the parameter. A requirement is usually defined in general terms, whereas a specification entails detailed engineering rigor. A requirement often refers to an objective of the system, while a specification describes how well the actual device works.

Checkpoint 9.5. Public functions have an underline. E.g., `SCI_OutString`.

Checkpoint 9.6. Local variables begin with a lower case letter E.g., `myKey`. Global variables begin with an upper case letter E.g., `TheKey`.