Embedded Systems - Shape the World

Jonathan Valvano and Ramesh Yerraballi

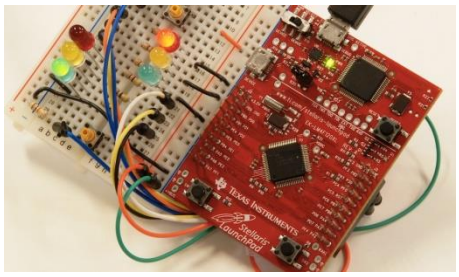


Table of Contents

1.0 Introduction

An embedded system combines mechanical, electrical, and chemical components along with a computer, hidden inside, to perform a single dedicated purpose. There are more computers on this planet than there are people, and most of these computers are single-chip microcontrollers that are the brains of an embedded system. Embedded systems are a ubiquitous component of our everyday lives. We interact with hundreds of tiny computers every day that are embedded into our houses, our cars, our bridges, our toys, and our work. As our world has become more complex, so have the capabilities of the microcontrollers embedded into our devices. Therefore the world needs a trained workforce to develop and manage products based on embedded microcontrollers.

1.0.1 Impact

The innovative aspect of this class is to effectively teach a course with a substantial lab component within the MOOC format. If MOOCs are truly going to transform the education, then they must be able to deliver laboratory classes. This offering will go a long way in unraveling the perceived complexities in delivering a laboratory experience to tens of thousands of students. If successful, the techniques developed in this class will significantly transform the MOOC environment. We believe effective education requires students to learn by doing. In the traditional academic setting this active learning is delivered in a lab format. A number of important factors have combined that allow a lab class like this to be taught at this time. First, we have significant support from industrial partners ARM Inc and Texas Instruments. Second, the massive growth of embedded microcontrollers has made the availability of lost-cost development platforms feasible. Third, your instructors have the passion, patience, and experience of delivering quality lab experiences to large classes. Fourth, on-line tools now exist that allow students to interact and support each other.

The overall educational objective of this class is to allow students to discover how the computer interacts with its environment. It will provide hands-on experiences of how an embedded system could be used to solve problems. The focus of this introductory course will be understanding and analysis rather than design. It takes an effective approach to learning new techniques by doing them. We feel we have solved the dilemma in learning a laboratory-based topic like embedded systems where there is a

tremendous volume of details that first must be learned before hardware and software systems can be designed.

1.0.2 Approach and Learning Philosophy

The approach taken in this course is to learn by doing in a bottom-up fashion. One of the advantages of a bottom-up approach to learning is that the student begins by mastering simple concepts. Once the student truly understands simple concepts, he or she can then embark on the creative process of design, which involves putting the pieces together to create a more complex system. True creativity is needed to solve complex problems using effective combinations of simple components. Embedded systems afford an effective platform to teach new engineers how to program for three reasons. First, there is no operating system. Thus, in a bottom-up fashion the student can see, write, and understand all software running on a system that actually does something. Second, embedded systems involve real input/output that is easy for the student to touch, hear, and see. Third, embedded systems are employed in many every-day products, motivating students to see firsthand, how engineering processes can be applied in the real world.

1.0.3 Audience and Background

This course is intended for beginning college students with some knowledge of electricity as would have been taught in an introductory college physics class. Secondly, it is expected students will have some basic knowledge of programming and logic design. No specific language will be assumed as prior knowledge but this class could be taken as their second programming class.

1.1 Structure and Objectives

The analog to digital converter (ADC) and digital to analog converter (DAC) are the chosen mechanism to bridge the computer and electrical worlds. Electrical engineering concepts include Ohms Law, LED voltage/current, resistance measurement, and motor control. Computer engineering concepts include I/O device drivers, debugging, stacks, FIFO queues, local variables and interrupts. The hardware construction is performed on a breadboard and debugged using a multimeter (students learn to measure voltage). Software is developed in C; all labs will be first simulated then run on the real microcontroller. Software debugging occurs during the simulation stage. Verification occurs in both stages.

The course has 11 labs and a final project. Each lab has a small and well-defined educational objective. Students begin by learning the fundamental concepts via lectures, interactive animations and readings. The second task is for students to observe an expert working through an example lab project (interactive tutorial where the students are required to follow along by building exactly what the instructor is building). Third, students are examined to make sure they understand the concepts by solving homework problems. Fourth, they are given a lab assignment where they must design hardware and software. Students connect circuits to their microcontroller board and write software to run on the board. The automatic grading system to verify specifications have been met. If the students are unsuccessful they will interact with their peers and be able to attempt the lab again.

1.1.1 Learning objectives:

Although the students are engaged with a fun and rewarding lab experience, the educational pedagogy is centered on fundamental learning objectives. After the successful conclusion of this class, students should be able to understand the basic components of a computer, write C language programs that perform I/O functions and implement simple data structures, manipulate numbers in multiple formats, and understand how software uses global memory to store permanent information and the stack to store temporary information. Our goal is for students to learn these concepts:

1. Understanding how the computer stores and manipulates data,
2. The understanding of embedded systems using modular design and abstraction,
3. C programming: considering both function and style,

4. The strategic use of memory,
5. Debugging and verification using a simulator and on the real microcontroller
6. How input/output using switches, LEDs, DACs, ADCs, motors, and serial ports,
7. The implementation of an I/O driver, multithreaded programming,
8. Understanding how local variables and parameters work,
9. Analog to digital conversion (ADC), periodic sampling,
10. Simple motors (e.g., open and closed-loop stepper motor control),
11. Digital to analog conversion (DAC), used to make simple sounds,
12. Design and implementation of elementary data structures.

1.2 Syllabus

The best way to understand what you will learn in this class is to list the labs you will complete and the example projects we will build. You will complete each lab first in simulation and then on the real board. For each module we will design a system and you will build and test a similar system as part of the lab for that module.

Following is the list of all modules, the corresponding examples we will build in each and the relevant lab you will complete. Some of the modules do not have examples or labs.

- **Module 1: Welcome and Introduction to course and staff**
  Your Lab 1. Install the Keil IDE and drivers for programming the labs

- **Module 2: Fundamental concepts: numbers, computers, and the ARM Cortex M processor**
  Our Example. Develop a system that toggles an LED on the LaunchPad
  Your Lab 2. Run existing project on LaunchPad with switch input and LED output

- **Module 3: Electronics: resistors, voltage, current and Ohm's Law**

- **Module 4: Digital Logic: transistors, flip flops and logic functions**
  Your Lab 4. Debug C software that inputs from two switches and outputs an LED output

- **Module 5: Introduction to C programming**
  Our Example. Develop a system that inputs and outputs on the serial port
  Your Lab 5. Write a C function and perform input/output on the serial port

- **Module 6: Microcontroller Input/Output**
  Our Example. Develop a system that inputs from a switch and toggles an LED output
  Your Lab 6. Write C software that inputs from a switch and toggles an LED output

- **Module 7: Design and Development Process**
  Our Example. Develop a system that outputs a pattern on an LED
  Your Lab 7. Write C functions that inputs from one switch and outputs to two LEDs

- **Module 8: Interfacing Switches and LEDs**
  Our Example. Develop a system with an external switch and LED
  Your Lab 8. Interface an external switch and LED and write input/output software.

- **Module 9: Arrays and Functional Debugging**
  Our Example. Develop a system that debugs by dumping data into an array
  Your Lab 9. Write C functions using array data structures that collect/debug your system.

- **Module 10: Finite State Machines**

Our Example 1. Develop a simple finite state machine
Our Example 2. Develop a vending machine using a finite state machine
Our Example 3. Develop a stepper motor robot using a finite state machine
Your Lab 10. Interface 3 switches and 6 LEDs and create a traffic light finite state machine

- **Module 11: UART - The Serial Interface, I/O Synchronization**
  Our Example . Develop a communication network using the serial port
  Your Lab 11. Write C functions that output decimal and fixed-point numbers to serial port

- **Module 12: Interrupts**
  Our Example 1. Develop a system that outputs a square wave using interrupts
  Our Example 2. Develop a system that inputs from a switch using interrupts
  Our Example 3. Develop a system that outputs to a DC motor that uses pulse width modulation
  Your Lab 12. Design and test a guitar tuner, producing a 440 Hz tone

- **Module 13: DAC and Sound**
  Our Example. Develop a system that outputs analog signal with a R-2R digital to analog converter
  Your Lab 13. Design and test a digital piano, with 4 inputs, digital to analog conversion, and sound

- **Module 14: ADC and Data Acquisition**
  Our Example 1. Develop a system that inputs an analog signal with an analog to digital converter
  Our Example 2. Develop an autonomous robot that uses two DC motors and two distance sensors
  Your Lab 14. Design and test a position measurement, with analog to digital conversion and calibrated output

- **Module 15: Systems Approach to Game Design**
  Your Lab 15. Design and test a hand-held video game, which integrates all components from previous labs.

1.2.1 Assessment

There will be five components of assessment. First, each module will have a quiz, which is a set of multiple choice/numerical questions that must be answered. If the student does not pass this quiz, then they can ask for help in the discussion forums and be allowed to retake the quiz. The second assessment involves solving the lab in simulation, and the third assessment is completing a physical lab. This means the student will wire up a simple circuit, write microcontroller code, and run the software on the real computer. Added to the student's software will be a grading engine that can assess the quantitative performance of the system. The labs result in a numerical score describing how many of the lab requirements the student successfully completed.

1. **10% quizzes**
2. **45% labs in simulation**
3. **45% labs running on the real board**

Chapter 2: Fundamental Concepts
Embedded Systems - Shape The World
Jonathan Valvano and Ramesh Yerraballi

This chapter covers the basic foundation concepts needed to build upon in this course. Specifically we will look at number representation, digital logic, embedded system components, and computer architecture: the Central Processing Unit (Arithmetic Logic Unit, Control Unit and Registers), the memory and the Instruction Set Architecture (ISA).

Learning Objectives:

- Understand the binary number system, hexadecimal representation and the representation of signed and unsigned integers.
- Learn what an Embedded System is.
- Know the terms: microcontroller (CPU, Busses, Memory, I/O ports)
- Learn the composition of a CPU.
- Know the terms: Arithmetic Logic Unit, Control Unit, Registers, Bus, Von Neumann, Harvard
- Learn the memory map
- See some of the Cortex M instruction set.

2.0. Introduction

Embedded systems are a ubiquitous component of our everyday lives. We interact with hundreds of tiny computers every day that are embedded into our houses, our cars, our toys, and our work. As our world has become more complex, so have the capabilities of the microcontrollers embedded into our devices. The ARM® Cortex™-M family represents a new class of microcontrollers much more powerful than the devices available ten years ago. The purpose of this class is to present the design methodology to train young engineers to understand the basic building blocks that comprise devices like a cell phone, an MP3 player, a pacemaker, antilock brakes, and an engine controller.

An embedded system is a system that performs a specific task and has a computer embedded inside. A system is comprised of components and interfaces connected together for a common purpose. This class is an introduction to embedded systems. Specific topics include microcontrollers, fixed-point numbers, the design of software in C, elementary data structures, programming input/output including interrupts, analog to digital conversion, digital to analog conversion.

In general, the area of embedded systems is an important and growing discipline within electrical and computer engineering. In the past, the educational market of embedded systems has been dominated by simple microcontrollers like the PIC, the 9S12, and the 8051. This is because of their market share, low cost, and historical dominance. However, as problems become more complex, so must the systems that solve them. A number of embedded system paradigms must shift in order to accommodate this growth in complexity. First, the number of calculations per second will increase from millions/sec to billions/sec. Similarly, the number of lines of software code will also increase from thousands to millions. Thirdly, systems will involve multiple microcontrollers supporting many simultaneous operations. Lastly, the need for system verification will continue to grow as these systems are deployed into safety critical applications. These changes are more than a simple growth in size and bandwidth. These systems must employ parallel programming, high-speed synchronization, real-time operating systems, fault tolerant design, priority interrupt handling, and networking. Consequently, it will be important to provide our students with these types of design experiences. The ARM platform is both low cost and provides the high-performance features required in future embedded systems. In addition, the ARM market share is large and will continue to grow. As of July 2013, ARM reports that over 35 billion ARM processors have been shipped from over 950 companies. Furthermore, students trained on the ARM will be equipped to design systems across the complete spectrum from simple to complex. The purpose of this course is to bring engineering education into the 21st century.

2.1. Binary number systems

To solve problems using a computer we need to understand numbers and what they mean. Each digit in a **decimal** number has a place and a value. The **place** is a power of 10 and the **value** is selected from the set {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. A decimal number is simply a combination of its digits multiplied by powers of 10. For example

$$1984 = 1 \bullet 10^3 + 9 \bullet 10^2 + 8 \bullet 10^1 + 4 \bullet 10^0$$

Fractional values can be represented by using the negative powers of 10. For example,

$$273.15 = 2 \bullet 10^2 + 7 \bullet 10^1 + 3 \bullet 10^0 + 1 \bullet 10^{-1} + 5 \bullet 10^{-2}$$

In a similar manner, each digit in a **binary** number has a place and a value. In binary numbers, the place is a power of 2, and the value is selected from the set {0, 1}. A binary number is simply a combination of its digits multiplied by powers of 2. To eliminate confusion between decimal numbers and binary numbers, we will put a subscript 2 after the number to mean binary. Because of the way the microcontroller operates, most of the binary numbers in this class will have 8, 16, or 32 bits. An 8-bit number is called a **byte**, and a 16-bit number is called a **halfword**. For example, the 8-bit binary number for 106 is

$$01101010_2 = 0 \bullet 2^7 + 1 \bullet 2^6 + 1 \bullet 2^5 + 0 \bullet 2^4 + 1 \bullet 2^3 + 0 \bullet 2^2 + 1 \bullet 2^1 + 0 \bullet 2^0 = 64+32+8+2 = 106$$

Checkpoint 2.1: What is the numerical value of the 8-bit binary number 11111111_2?

Binary is the natural language of computers but a big nuisance for us humans. To simplify working with binary numbers, humans use a related number system called **hexadecimal**, which uses base 16. Just like decimal and binary, each hexadecimal digit has a place and a value. In this case, the place is a power of 16 and the value is selected from the set {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}. As you can see, hexadecimal numbers have more possibilities for their digits than are available in the decimal format; so, we add the letters A through F, as shown in Table 2.1. A hexadecimal number is a combination of its digits multiplied by powers of 16. To eliminate confusion between various formats, we will put a 0x or a $ before the number to mean hexadecimal. Hexadecimal representation is a convenient mechanism for us humans to define binary information, because it is extremely simple for humans to convert back and forth between binary and hexadecimal. Hexadecimal number system is often abbreviated as “hex”. A **nibble** is defined as 4 binary bits, or one hexadecimal digit. Each value of the 4-bit nibble is mapped into a unique hex digit, as shown in Table 2.1.

Hex Digit	Decimal Value	Binary Value
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A or a	10	1010
B or b	11	1011
C or c	12	1100
D or d	13	1101
E or e	14	1110
F or f	15	1111

Table 2.1. Definition of hexadecimal representation.

For example, the hexadecimal number for the 16-bit binary 0001 0010 1010 1101 is

$$0x12AD = 1 \bullet 16^3 + 2 \bullet 16^2 + 10 \bullet 16^1 + 13 \bullet 16^0 = 4096+512+160+13 = 4781$$

Observation: In order to maintain consistency between assembly and C programs, we will use the 0x format when writing hexadecimal numbers in this class.

Checkpoint 2.2: What is the numerical value of the 8-bit hexadecimal number 0xFF?

As illustrated in Figure 2.1, to convert from binary to hexadecimal we can:

1) Divide the binary number into right justified nibbles,
2) Convert each nibble into its corresponding hexadecimal digit.

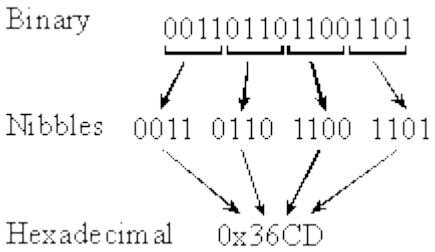


Figure 2.1. Example conversion from binary to hexadecimal.

Checkpoint 2.3: Convert the binary number 01000101_2 to hexadecimal.
Checkpoint 2.4: Convert the binary number 110010101011_2 to hexadecimal.

As illustrated in Figure 2.2, to convert from hexadecimal to binary we can:

1) Convert each hexadecimal digit into its corresponding 4-bit binary nibble,
2) Combine the nibbles into a single binary number.

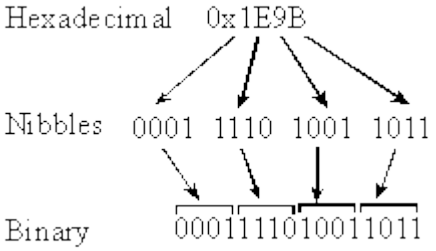


Figure 2.2. Example conversion from hexadecimal to binary.

Checkpoint 2.5: Convert the hex number 0x40 to binary.
Checkpoint 2.6: Convert the hex number 0x63F to binary.
Checkpoint 2.7: How many binary bits does it take to represent 0x123456?

Computer programming environments use a wide variety of symbolic notations to specify the numbers in hexadecimal. As an example, assume we wish to represent the binary number 01111010. Some assembly languages use $7A. Some assembly languages use 7AH. The C language uses 0x7A. Patt's LC-3 simulator uses x7A.

Precision is the number of distinct or different values. We express precision in alternatives, decimal digits, bytes, or binary bits. **Alternatives** are defined as the total number of possibilities. For example, an 8-bit number format can represent 256 different numbers. An 8-bit **digital to analog converter** (DAC) can generate 256 different analog outputs. An 8-bit **analog to digital converter** (ADC) can measure 256 different analog inputs. Table 2.2 illustrates the relationship between precision in binary bits and precision in alternatives. The operation $[[\mathbf{x}]]$ is defined as the greatest integer of $\mathbf{x}$. E.g., [[2.1]] [[2.9]] and [[3.0]] are all equal to 3. The **Bytes** column in Table 2.1 specifies

how many bytes of memory it would take to store a number with that precision assuming the data were not packed or compressed in any way.

Binary bits	Bytes	Alternatives
8	1	256
10	2	1024
12	2	4096
16	2	65536
20	3	1,048,576
24	3	16,777,216
30	4	1,073,741,824
32	4	4,294,967,296
n	[[n/8]]	2^n

Table 2.2. Relationship between bits, bytes and alternatives as units of precision.

Checkpoint 2.8: How many bytes of memory would it take to store a 50-bit number?

A **byte** contains 8 bits as shown in Figure 2.3, where each bit $b_7,...,b_0$ is binary and has the value 1 or 0. We specify b_7 as the **most significant bit** or MSB, and b_0 as the least significant bit or LSB.

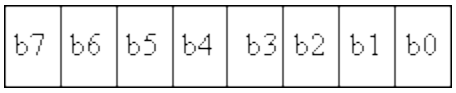


Figure 2.3. 8-bit binary format.

If a byte is used to represent an unsigned number, then the value of the number is

$$N = 128 \bullet b_7 + 64 \bullet b_6 + 32 \bullet b_5 + 16 \bullet b_4 + 8 \bullet b_3 + 4 \bullet b_2 + 2 \bullet b_1 + b_0$$

Notice that the significance of bit *n* is 2^n. There are 256 different unsigned 8-bit numbers. The smallest unsigned 8-bit number is 0 and the largest is 255. For example, 00001010_2 is 8+2 or 10. Other examples are shown in Table 2.3. The least significant bit can tell us if the number is even or odd.

binary	hex	Calculation	decimal
00000000_2	0x00		0
00100001_2	0x21	32+1	33
01000110_2	0x46	64+4+2	70
10001011_2	0x8B	128+8+2+1	139
11111111_2	0xFF	128+64+32+16+8+4+2+1	255

Table 2.3. Example conversions from unsigned 8-bit binary to hexadecimal and to decimal.

Checkpoint 2.9: Convert the binary number 01101001_2 to unsigned decimal.

Checkpoint 2.10: Convert the hex number 0x54 to unsigned decimal.

The **basis** of a number system is a subset from which linear combinations of the basis elements can be used to construct the entire set. The basis represents the "places" in a "place-value" system. For positive integers, the basis is the infinite set {1, 10, 100, …}, and the "values" can range from 0 to 9. Each positive integer has a unique set of values such that the dot-product of the value vector times the basis vector yields that number. For example, 2345 is (…, 2,3,4,5)•(…, 1000,100,10,1), which is 2*1000+3*100+4*10+5. For the unsigned 8-bit number system, the basis elements are

{1, 2, 4, 8, 16, 32, 64, 128}

The values of a binary number system can only be 0 or 1. Even so, each 8-bit unsigned integer has a unique set of values such that the dot-product of the values times the basis yields that number. For

example, 69 is (0,1,0,0,0,1,0,1)•(128,64,32,16,8,4,2,1), which equals 0*128+1*64+0*32+0*16+0*8+1*4+0*2+1*1. Conveniently, there is no other set of 0's and 1's, such that set of values multiplied by the basis is 69. In other words, each 8-bit unsigned binary representation of the values 0 to 255 is unique.

One way for us to convert a decimal number into binary is to use the basis elements. The overall approach is to start with the largest basis element and work towards the smallest. More precisely, we start with the most significant bit and work towards the least significant bit. One by one, we ask ourselves whether or not we need that basis element to create our number. If we do, then we set the corresponding bit in our binary result and subtract the basis element from our number. If we do not need it, then we clear the corresponding bit in our binary result. We will work through the algorithm with the example of converting 100 to 8-bit binary, see Table 2.4. We start with the largest basis element (in this case 128) and ask whether or not we need to include it to make 100? Since our number is less than 128, we do not need it, so bit 7 is zero. We go the next largest basis element, 64 and ask, "do we need it?" We do need 64 to generate our 100, so bit 6 is one and we subtract 100 minus 64 to get 36. Next, we go the next basis element, 32 and ask, "do we need it?" Again, we do need 32 to generate our 36, so bit 5 is one and we subtract 36 minus 32 to get 4. Continuing along, we do not need basis elements 16 or 8, but we do need basis element 4. Once we subtract the 4, our working result is zero, so basis elements 2 and 1 are not needed. Putting it together, we get 01100100_2 (which means 64+32+4).

Checkpoint 2.11: In this conversion algorithm, how can we tell if a basis element is needed?

Observation: If the least significant binary bit is zero, then the number is even.
Observation: If the right-most n bits (least sign.) are zero, then the number is divisible by 2^n.
Observation: Consider an 8-bit unsigned number system. If bit 7 is low, then the number is between 0 and 127, and if bit 7 is high then the number is between 128 and 255.

Number	Basis	Need it?	bit	Operation
100	128	no	bit 7=0	none
100	64	yes	bit 6=1	subtract 100-64
36	32	yes	bit 5=1	subtract 36-32
4	16	no	bit 4=0	none
4	8	no	bit 3=0	none
4	4	yes	bit 2=1	subtract 4-4
0	2	no	bit 1=0	none
0	1	no	bit 0=0	none

Table 2.4. Example conversion from decimal to unsigned 8-bit binary to hexadecimal.

Checkpoint 2.12: Give the representations of the decimal 45 in 8-bit binary and hexadecimal.
Checkpoint 2.13: Give the representations of the decimal 200 in 8-bit binary and hexadecimal.

One of the first schemes to represent signed numbers was called **one's complement**. It was called one's complement because to negate a number, we complement (logical not) each bit. For example, if 25 equals 00011001_2 in binary, then –25 is 11100110_2. An 8-bit one's complement number can vary from -127 to +127. The most significant bit is a sign bit, which is 1 if and only if the number is negative. The difficulty with this format is that there are two zeros +0 is 00000000_2, and –0 is 11111111_2. Another problem is that one's complement numbers do not have basis elements. These limitations led to the use of two's complement.

The **two's complement** number system is the most common approach used to define signed numbers. It is called two's complement because to negate a number, we complement each bit (like one's

complement), then add 1. For example, if 25 equals 00011001_2 in binary, then –25 is 11100111_2. If a byte is used to represent a signed two's complement number, then the value of the number is

$$N = -128 \bullet b_7 + 64 \bullet b_6 + 32 \bullet b_5 + 16 \bullet b_4 + 8 \bullet b_3 + 4 \bullet b_2 + 2 \bullet b_1 + b_0$$

Observation: One usually means two's complement when one refers to signed integers.
There are 256 different signed 8-bit numbers. The smallest signed 8-bit number is -128 and the largest is 127. For example, 10000010_2 equals -128+2 or -126. Other examples are shown in Table 2.5.

binary	Hex	Calculation	decimal
00000000_2	0x00		0
00010010_2	0x12	16+2	18
00100110_2	0x26	32+4+2	38
11000111_2	0xC7	-128+64+4+2+1	-57
11111111_2	0xFF	-128+64+32+16+8+4+2+1	-1

Table 2.5. Example conversions from signed 8-bit binary to hexadecimal and to decimal.

Checkpoint 2.14: Convert the signed binary number 11011010_2 to signed decimal.
Checkpoint 2.15: Are the signed and unsigned decimal representations of the 8-bit hex number 0x95 the same or different?
For the signed 8-bit number system the basis elements are

{1, 2, 4, 8, 16, 32, 64, -128}

Observation: The most significant bit in a two's complement signed number will specify the sign.

Notice that the same binary pattern of 11111111_2 could represent either 255 or –1. It is very important for the software developer to keep track of the number format. The computer cannot determine whether the 8-bit number is signed or unsigned. You, as the programmer, will determine whether the number is signed or unsigned by the specific assembly instructions you select to operate on the number. Some operations like addition, subtraction, and shift left (multiply by 2) use the same hardware (instructions) for both unsigned and signed operations. On the other hand, divide, and shift right (divide by 2) require separate hardware (instruction) for unsigned and signed operations.

Similar to the unsigned algorithm, we can use the basis to convert a decimal number into signed binary. We will work through the algorithm with the example of converting –100 to 8-bit binary, as shown in Table 2.6. We start with the most significant bit (in this case –128) and decide do we need to include it to make –100? Yes (without –128, we would be unable to add the other basis elements together to get any negative result), so we set bit 7 and subtract the basis element from our value. Our new value equals –100 minus –128, which is 28. We go the next largest basis element, 64 and ask, "do we need it?" We do not need 64 to generate our 28, so bit 6 is zero. Next we go the next basis element, 32 and ask, "do we need it?" We do not need 32 to generate our 28, so bit 5 is zero. Now we need the basis element 16, so we set bit 4, and subtract 16 from our number 28 (28-16=12). Continuing along, we need basis elements 8 and 4 but not 2, 1. Putting it together we get 10011100_2 (which means -128+16+8+4).

Number	Basis	Need it	bit	Operation
-100	-128	yes	bit 7=1	subtract -100 - -28
28	64	no	bit 6=0	none
28	32	no	bit 5=0	none
28	16	yes	bit 4=1	subtract 28-16
12	8	yes	bit 3=1	subtract 12-8
4	4	yes	bit 2=1	subtract 4-4

0	2	no	bit 1=0	none
0	1	no	bit 0=0	none

Table 2.6. Example conversion from decimal to signed 8-bit binary.

Observation: To take the negative of a two's complement signed number we first complement (flip) all the bits, then add 1.

A second way to convert negative numbers into binary is to first convert them into unsigned binary, then do a two's complement negate. For example, we earlier found that +100 is 01100100_2. The two's complement negate is a two-step process. First we do a logic complement (flip all bits) to get 10011011_2. Then add one to the result to get 10011100_2.

A third way to convert negative numbers into binary uses the number wheel. Let n be the number of bits in the binary representation. We specify precision, M=2^n, as the number of distinct values that can be represented. To convert negative numbers into binary is to first add M to the number, then convert the unsigned result to binary using the unsigned method. This works because binary numbers with a finite n are like the minute-hand on a clock. If we add 60 minutes, the minute-hand is in the same position. Similarly if we add M to or subtract M from an n-bit number, we go around the number wheel and arrive at the same place. This is one of the beautiful properties of 2's complement: unsigned and signed addition/subtraction are same operation. In this example we have an 8-bit number so the precision is 256. So, first we add 256 to the number, then convert the unsigned result to binary using the unsigned method. For example, to find –100, we add 256 plus –100 to get 156. Then we convert 156 to binary resulting in 10011100_2. This method works because in 8-bit binary math adding 256 to number does not change the value. E.g., 256-100 has the same 8-bit binary value as –100.

Checkpoint 2.16: Give the representations of -54 in 8-bit binary and hexadecimal.
Checkpoint 2.17: Why can't you represent the number 150 using 8-bit signed binary?

When dealing with numbers on the computer, it will be convenient to memorize some **Powers of 2** as shown in Table 2.7.

exponent	decimal
2^0	1
2^1	2
2^2	4
2^3	8
2^4	16
2^5	32
2^6	64
2^7	128
2^8	256
2^9	512
2^{10}	1024 about a thousand
2^{11}	2048
2^{12}	4096
2^{13}	8192
2^{14}	16384
2^{15}	32768
2^{16}	65536
2^{20}	about a million
2^{30}	about a billion
2^{40}	about a trillion

Table 2.7. Some powers of two that will be useful to memorize.

Checkpoint 2.18: Use Table 2.7 to determine the approximate value of 2^{32}?

A **halfword** or **double byte** contains 16 bits, where each bit $b_{15},...,b_0$ is binary and has the value 1 or 0, as shown in Figure 2.4.

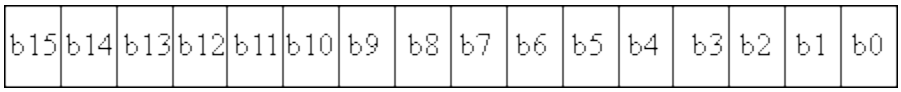


Figure 2.4. 16-bit binary format.

If a halfword is used to represent an unsigned number, then the value of the number is

$$N = 32768 \bullet b_{15} + 16384 \bullet b_{14} + 8192 \bullet b_{13} + 4096 \bullet b_{12} + 2048 \bullet b_{11} + 1024 \bullet b_{10} + 512 \bullet b_9 + 256 \bullet b_8 + 128 \bullet b_7 + 64 \bullet b_6 + 32 \bullet b_5 + 16 \bullet b_4 + 8 \bullet b_3 + 4 \bullet b_2 + 2 \bullet b_1 + b_0$$

There are 65536 different unsigned 16-bit numbers. The smallest unsigned 16-bit number is 0 and the largest is 65535. For example, 0010000110000100_2 or 0x2184 is 8192+256+128+4 or 8580. Other examples are shown in Table 2.8.

binary	hex	Calculation	decimal
0000000000000000_2	0x0000		0
0000010000000001_2	0x0401	1024+1	1025
0000110010100000_2	0x0CA0	2048+1024+128+32	3232
1000111000000010_2	0x8E02	32768+2048+1024+512+2	36354
1111111111111111_2	0xFFFF	32768+16384+8192+4096+2048+1024 +512+256+128+64+32+16+8+4+2+1	65535

Table 2.8. Example conversions from unsigned 16-bit binary to hexadecimal and to decimal.

Checkpoint 2.19: Convert the 16-bit binary number 0010000001101010_2 to unsigned decimal.
Checkpoint 2.20: Convert the 16-bit hex number 0x1234 to unsigned decimal.

For the unsigned 16-bit number system the basis elements are
{1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768}

Checkpoint 2.21: Convert the unsigned decimal number 1234 to 16-bit hexadecimal.
Checkpoint 2.22: Convert the unsigned decimal number 10000 to 16-bit binary.

There are also 65536 different signed 16-bit numbers. The smallest two's complement signed 16-bit number is –32768 and the largest is 32767. For example, 1101000000000100_2 or 0xD004 is –32768+16384+4096+4 or –12284. Other examples are shown in Table 2.9.

binary	hex	Calculation	decimal
0000000000000000_2	0x0000		0
0000010000000001_2	0x0401	1024+1	1025
0000110010100000_2	0x0CA0	2048+1024+128+32	3232
1000010000000010_2	0x8402	-32768+1024+2	-31742
1111111111111111_2	0xFFFF	- 32768+16384+8192+4096+2048+1024 +512+256+128+64+32+16+8+4+2+1	-1

Table 2.9. Example conversions from signed 16-bit binary to hexadecimal and to decimal.

If a halfword is used to represent a signed two's complement number, then the value of the number is

$$N = -32768 \bullet b_{15} + 16384 \bullet b_{14} + 8192 \bullet b_{13} + 4096 \bullet b_{12} + 2048 \bullet b_{11} + 1024 \bullet b_{10} + 512 \bullet b_9 + 256 \bullet b_8 + 128 \bullet b_7 + 64 \bullet b_6 + 32 \bullet b_5 + 16 \bullet b_4 + 8 \bullet b_3 + 4 \bullet b_2 + 2 \bullet b_1 + b_0$$

Checkpoint 2.23: Convert the 16-bit hex number 0x1234 to signed decimal.
Checkpoint 2.24: Convert the 16-bit hex number 0xABCD to signed decimal.

For the signed 16-bit number system the basis elements are
{1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, -32768}

Common Error: An error will occur if you use 16-bit operations on 8-bit numbers, or use 8-bit operations on 16-bit numbers.

Maintenance Tip: To improve the clarity of your software, always specify the precision of your data when defining or accessing the data.

Checkpoint 2.25: Convert the signed decimal number 1234 to 16-bit hexadecimal.
Checkpoint 2.26: Convert the signed decimal number –10000 to 16-bit binary.

A **word** on the ARM Cortex M will have 32 bits. Consider an unsigned number with 32 bits, where each bit $b_{31},...,b_0$ is binary and has the value 1 or 0. If a 32-bit number is used to represent an unsigned integer, then the value of the number is

$$N = 2^{31} \bullet b_{31} + 2^{30} \bullet b_{30} + ... + 2 \bullet b_1 + b_0 = \text{sum}(2^i \bullet b_i) \text{ for } i=0 \text{ to } 31$$

There are 2^{32} different unsigned 32-bit numbers. The smallest unsigned 32-bit number is 0 and the largest is 2^{32}-1. This range is 0 to about 4 billion. For the unsigned 32-bit number system, the basis elements are

$$\{1, 2, 4, ... , 2^{29}, 2^{30}, 2^{31}\}$$

If a 32-bit binary number is used to represent a signed two's complement number, then the value of the number is

$$N = -2^{31} \bullet b_{31} + 2^{30} \bullet b_{30} + ... + 2 \bullet b_1 + b_0 = -2^{31} \bullet b_{31} + \text{sum}(2^i \bullet b_i) \text{ for } i=0 \text{ to } 30$$

There are also 2^{32} different signed 32-bit numbers. The smallest signed 32-bit number is -2^{31} and the largest is 2^{31}-1. This range is about -2 billion to about +2 billion. For the signed 32-bit number system, the basis elements are

$$\{1, 2, 4, ... , 2^{29}, 2^{30}, -2^{31}\}$$

Maintenance Tip: When programming in C, we will use data types **`char short`** and **`long`** when we wish to explicitly specify the precision as 8-bit, 16-bit or 32-bit. Whereas, we will use the **`int`** data type only when we don't care about precision, and we wish the compiler to choose the most efficient way to perform the operation. For most compilers for the ARM processor, **`int`** will be 32 bits.

Observation: When programming in assembly, we will always explicitly specify the precision of our numbers and calculations.

We will use **fixed-point** numbers when we wish to express values in our computer that have non-integer values. A fixed-point number contains two parts. The first part is a variable integer, called *I*. The variable integer will be stored on the computer. The second part of a fixed-point number is a fixed constant, called the **resolution** Δ. The fixed constant will NOT be stored on the computer. The fixed constant is something we keep track of while designing the software operations. The value of the

number is the product of the variable integer times the fixed constant. The integer may be signed or unsigned. An unsigned fixed-point number is one that has an unsigned variable integer. A signed fixed-point number is one that has a signed variable integer. The **precision** of a number system is the total number of distinguishable values that can be represented. The precision of a fixed-point number is determined by the number of bits used to store the variable integer. On most microcontrollers, we can use 8, 16, or 32 bits for the integer. With **binary fixed point** the fixed constant is a power of 2. An example is shown in Figure 2.5.

Binary fixed-point value = $I \bullet 2^n$ for some constant integer n

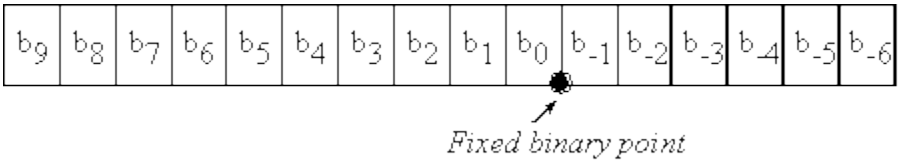


Figure 2.5. 16-bit binary fixed-point format with $\Delta=2^{-6}$.

2.2. Embedded Systems

To better understand the expression **embedded microcomputer system**, consider each word separately. In this context, the word "embedded" means hidden inside so one can't see it. The term "micro" means small, and a "computer" contains a processor, memory, and a means to exchange data with the external world. The word "system" means multiple components interfaced together for a common purpose. Systems have structure, behavior, and interconnectivity operating in a framework bound by rules and regulations. In an embedded system, we use ROM for storing the software and fixed constant data and RAM for storing temporary information. Many microcomputers employed in embedded systems use Flash EEPROM, which is an electrically-erasable programmable ROM, because the information can easily be erased and reprogrammed. The functionality of a digital watch is defined by the software programmed into its ROM. When you remove the batteries from a watch and insert new batteries, it still behaves like a watch because the ROM is nonvolatile storage. As shown in Figure 2.6, the term embedded microcomputer system refers to a device that contains one or more microcomputers inside. **Microcontrollers**, which are microcomputers incorporating the processor, RAM, ROM and I/O ports into a single package, are often employed in an embedded system because of their low cost, small size, and low power requirements. Microcontrollers like the Texas Instruments TM4C are available with a large number and wide variety of I/O devices, such as parallel ports, serial ports, timers, digital to analog converters (DAC), and analog to digital converters (ADC). The I/O devices are a crucial part of an embedded system, because they provide necessary functionality. The software together with the I/O ports and associated interface circuits give an embedded computer system its distinctive characteristics. The microcontrollers often must communicate with each other. How the system interacts with humans is often called the **human-computer interface** (HCI) or **man-machine interface** (MMI).

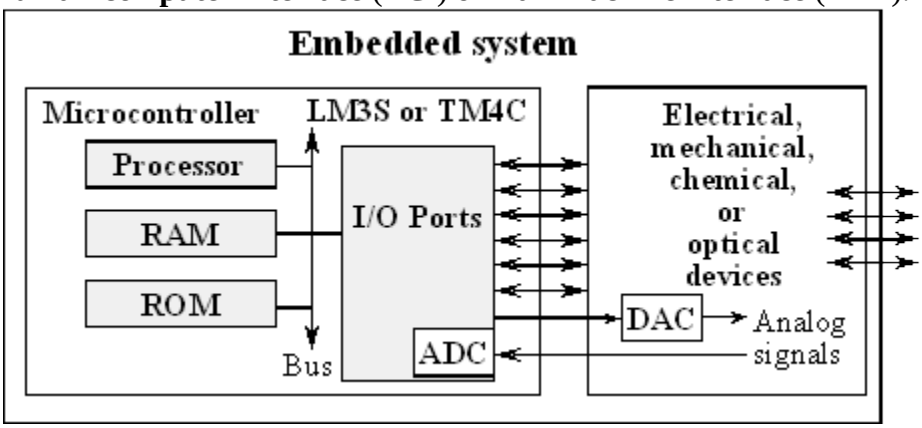


Figure 2.6. An embedded system includes a microcomputer interfaced to external devices.

Checkpoint 2.27: What is an embedded system?

A digital multimeter, as shown in Figure 2.7, is a typical embedded system. This embedded system has two inputs: the mode selection dial on the front and the red/black test probes. The output is a liquid crystal display (LCD) showing measured parameters. The large black chip inside the box is a microcontroller. The software that defines its very specific purpose is programmed into the ROM of the microcontroller. As you can see, there is not much else inside this box other than the microcontroller, a fuse, a rotary dial to select the mode, a few interfacing resistors, and a battery.

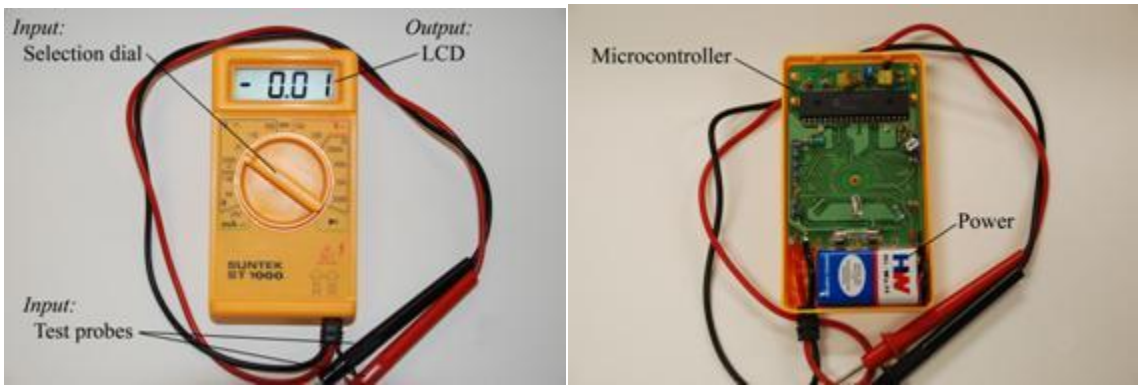


Figure 2.7. A digital multimeter contains a microcontroller programmed to measure voltage, current and resistance.

As defined previously, a microcomputer is a small computer. One typically restricts the term **embedded** to refer to systems that do not look and behave like a typical computer. Most embedded systems do not have a keyboard, a graphics display, or secondary storage (disk). There are two ways to develop embedded systems. The first technique uses a microcontroller, like the ARM Cortex M-series. In general, there is no operating system, so the entire software system is developed. These devices are suitable for low-cost, low-performance systems. On the other hand, one can develop a high-performance embedded system around a more powerful microcontroller such as the ARM Cortex A-series. These systems typically employ an operating system and are first designed on a development platform, and then the software and hardware are migrated to a stand-alone embedded platform.

Checkpoint 2.28: What is a microcomputer?

The external devices attached to the microcontroller allow the system to interact with its environment. An **interface** is defined as the hardware and software that combine to allow the computer to communicate with the external hardware. We must also learn how to interface a wide range of inputs and outputs that can exist in either digital or analog form. In this class we provide an introduction to microcomputer programming, hardware interfacing, and the design of embedded systems. In general, we can classify I/O interfaces into parallel, serial, analog or time. Because of low cost, low power, and high performance, there has been and will continue to be an advantage of using time-encoded inputs and outputs.

A **device driver** is a set of software functions that facilitate the use of an I/O port. One of the simplest I/O ports on the Stellaris® microcontrollers is a parallel port or General Purpose Input/Output (GPIO). One such parallel port is Port A. The software will refer to this port using the name **`GPIO_PORTA_DATA_R`**. Ports are a collection of pins, usually 8, which can be used for either input or output. If Port A is an input port, then when the software reads from **`GPIO_PORTA_DATA_R`**, it gets eight bits (each bit is 1 or 0), representing the digital levels (high or low) that exist at the time of the read. If Port A is an output port, then when the software writes to **`GPIO_PORTA_DATA_R`**, it sets the outputs on the eight pins high (1) or low (0), depending on the data value the software has written.

The other general concept involved in most embedded systems is they run in **real time**. In a real-time computer system, we can put an upper bound on the time required to perform the input-calculation-output sequence. A real-time system can guarantee a worst case upper bound on the response time between when the new input information becomes available and when that information is processed. This response time is called interface **latency**. Another real-time requirement that exists in many embedded systems is the execution of periodic tasks. A periodic task is one that must be performed at equal-time intervals. A real-time system can put a small and bounded limit on the time error between when a task should be run and when it is actually run. Because of the real-time nature of these systems, microcontrollers have a rich set of features to handle many aspects of time.

Checkpoint 2.29: An input device allows information to be entered into the computer. List some of the input devices available on a general purpose computer.

Checkpoint 2.30: An output device allows information to exit the computer. List some of the output devices available on a general purpose computer.

The embedded computer systems will contain a Texas Instruments TM4C123 microcontroller, which will be programmed to perform a specific dedicated application. Software for embedded systems typically solves only a limited range of problems. The microcomputer is embedded or hidden inside the device. In an embedded system, the software is usually programmed into ROM and therefore fixed. Even so, **software maintenance** (e.g., verification of proper operation, updates, fixing bugs, adding features, extending to new applications, end user configurations) is still extremely important. In fact, because microcomputers are employed in many safety-critical devices, injury or death may result if there are hardware and/or software faults. Consequently, testing must be considered in the original design, during development of intermediate components, and in the final product. The role of simulation is becoming increasingly important in today's market place as we race to build better and better machines with shorter and shorter design cycles. An effective approach to building embedded systems is to first design the system using a hardware/software simulator, then download and test the system on an actual microcontroller.

2.3. Introduction to Computers

A **computer** combines a processor, random access memory (RAM), read only memory (ROM), and input/output (I/O) ports. The common bus in Figure 2.8 defines the von Neumann architecture. Computers are not intelligent. Rather, you are the true genius. Computers are electronic idiots. They can store a lot of data, but they will only do exactly what we tell them to do. Fortunately, however, they can execute our programs quite quickly, and they don't get bored doing the same tasks over and over again. **Software** is an ordered sequence of very specific instructions that are stored in memory, defining exactly what and when certain tasks are to be performed. It is a set of instructions, stored in memory, that are executed in a complicated but well-defined manner. The **processor** executes the software by retrieving and interpreting these instructions one at a time. A **microprocessor** is a small processor, where small refers to size (i.e., it fits in your hand) and not computational ability. For example, Intel Xeon E7, AMD Fusion, and Sun SPARC are microprocessors. An ARM® Cortex™-M microcontroller includes a processor together with the bus and some peripherals.

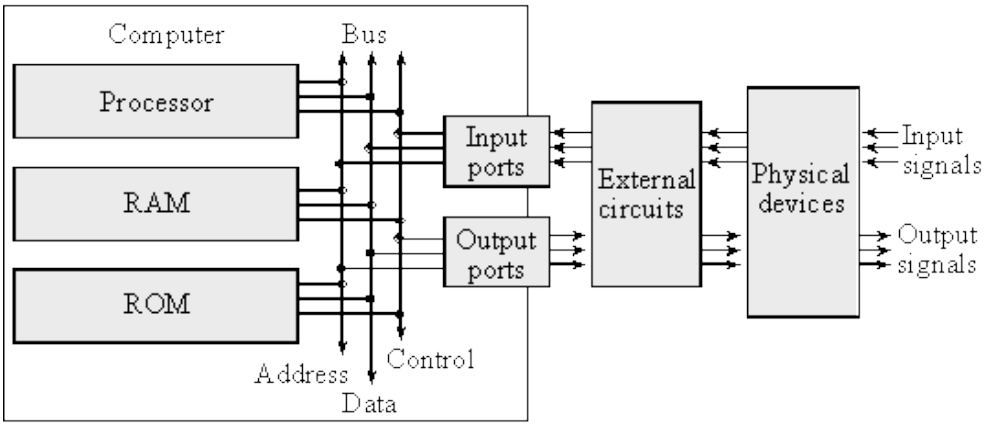


Figure 2.8. The basic components of a von Neumann computer include processor, memory and I/O.

A **microcomputer** is a small computer, where again small refers to size (i.e., you can carry it) and not computational ability. For example, a desktop PC is a microcomputer. Small in this context describes its size not its computing power. Consequently, there can be great confusion over the term microcomputer, because it can refer to a very wide range of devices from a PIC12C508, which is an 8-pin chip with 512 words of ROM and 25 bytes RAM, to the most powerful I7-based personal computer.

A **port** is a physical connection between the computer and its outside world. Ports allow information to enter and exit the system. Information enters via the input ports and exits via the output ports. Other names used to describe ports are I/O ports, I/O devices, interfaces, or sometimes just devices. A **bus** is a collection of wires used to pass information between modules.

A very small microcomputer, called a **microcontroller**, contains all the components of a computer (processor, memory, I/O) on a single chip. As shown in Figure 2.9, the Atmel ATtiny, the Texas Instruments MSP430, and the Texas Instruments TM4C123 are examples of microcontrollers. Because a microcomputer is a small computer, this term can be confusing because it is used to describe a wide range of systems from a 6-pin ATtiny4 running at 1 MHz with 512 bytes of program memory to a personal computer with state-of-the-art 64-bit multi-core processor running at multi-GHz speeds having terabytes of storage.

The computer can store information in **RAM** by writing to it, or it can retrieve previously stored data by reading from it. RAMs are **volatile**; meaning if power is interrupted and restored the information in the RAM is lost. Most microcontrollers have **static RAM** (SRAM) using six metal-oxide-semiconductor field-effect transistors (MOS or MOSFET) to create each memory bit. Four transistors are used to create two cross-coupled inverters that store the binary information, and the other two are used to read and write the bit.

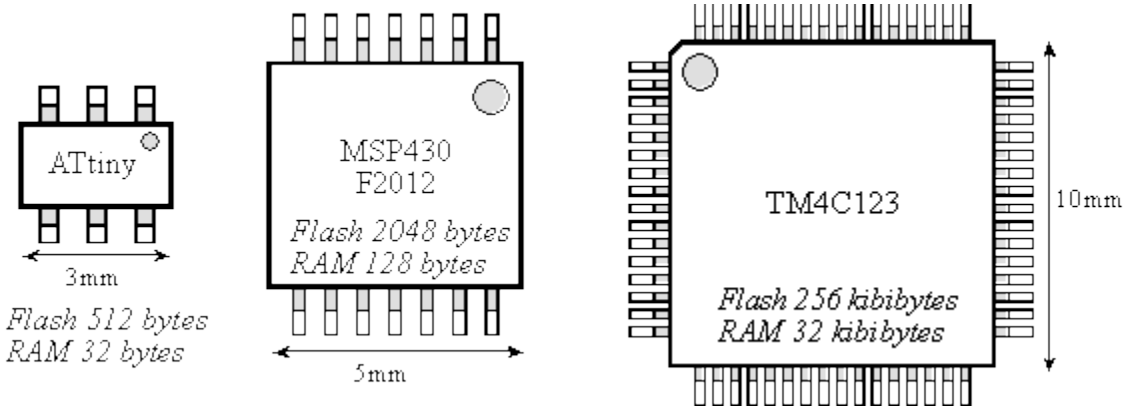


Figure 2.9. A microcontroller is a complete computer on a single chip.

Information is programmed into **ROM** using techniques more complicated than writing to RAM. From a programming viewpoint, retrieving data from a ROM is identical to retrieving data from RAM. ROMs are nonvolatile; meaning if power is interrupted and restored the information in the ROM is retained. Some ROMs are programmed at the factory and can never be changed. A Programmable ROM (PROM) can be erased and reprogrammed by the user, but the erase/program sequence is typically 10000 times slower than the time to write data into a RAM. Some PROMs are erased with ultraviolet light and programmed with voltages, while electrically erasable PROM (EEPROM) are both erased and programmed with voltages. We cannot program ones into the ROM. We first erase the ROM, which puts ones into the entire memory, and then we program the zeros as needed. **Flash ROM** is a popular type of EEPROM. Each flash bit requires only two MOSFET transistors. The input (gate) of one transistor is electrically isolated, so if we trap charge on this input, it will remain there for years. The other transistor is used to read the bit by sensing whether or not the other transistor has trapped charge. In regular EEPROM, you can erase and program individual bytes. Flash ROM must be erased in large blocks. On many of Stellaris family of microcontrollers, we can erase the entire ROM or just a 1024-byte block. Because flash is smaller than regular EEPROM, most microcontrollers have a large flash into which we store the software. For all the systems in this class, we will store instructions and constants in flash ROM and place variables and temporary data in static RAM.

Checkpoint 2.31: What are the differences between a microcomputer, a microprocessor, and a microcontroller?

Checkpoint 2.32: Which has a higher information density on the chip in bits per mm^2: static RAM or flash ROM? Assume all MOSFETs are approximately the same size in mm^2.

Figure 2.10 shows a simplified block diagram of a microcontroller based on the ARM® Cortex™-M processor. It is a **Harvard architecture** because it has separate data and instruction buses. The Cortex-M instruction set combines the high performance typical of a 32-bit processor with high code density typical of 8-bit and 16-bit microcontrollers. Instructions are fetched from flash ROM using the ICode bus. Data are exchanged with memory and I/O via the system bus interface. On the Cortex-M4 there is a second I/O bus for high-speed devices like USB. There are many sophisticated debugging features utilizing the DCode bus. The nested vectored interrupt controller (NVIC) manages **interrupts**, which are hardware-triggered software functions. Some internal peripherals, like the NVIC communicate directly with the processor via the private peripheral bus (PPB). The tight integration of the processor and interrupt controller provides fast execution of interrupt service routines (ISRs), dramatically reducing the interrupt latency.

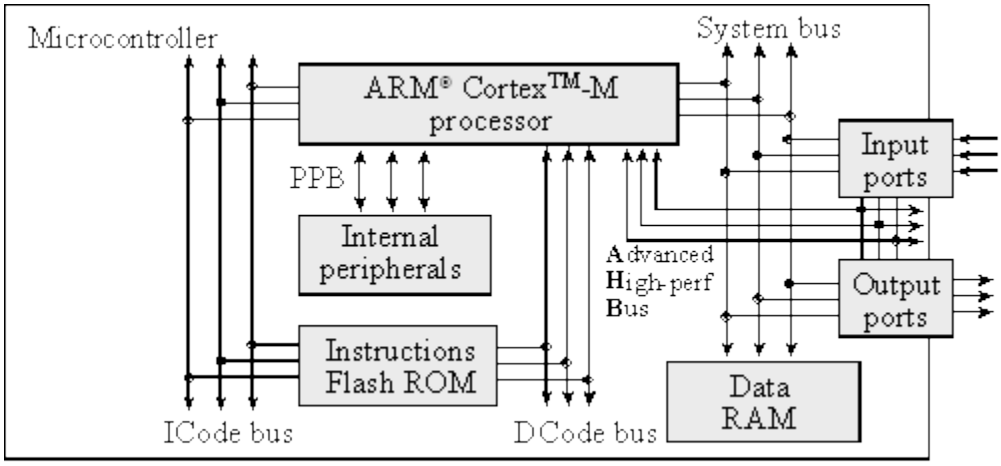


Figure 2.10. Harvard architecture of an ARM® Cortex-M-based microcontroller.

Even though data and instructions are fetched 32-bits at a time, each 8-bit byte has a unique address. This means memory and I/O ports are byte addressable. The processor can read or write 8-bit, 16-bit, or 32-bit data. Exactly how many bits are affected depends on the instruction, which we will see later in this chapter.

2.4. I/O Ports

The external devices attached to the microcontroller provide functionality for the system. An **input port** is hardware on the microcontroller that allows information about the external world to be entered into the computer. The microcontroller also has hardware called an **output port** to send information out to the external world. Most of the pins shown in Figure 2.11 are input/output ports.

An **interface** is defined as the collection of the I/O port, external electronics, physical devices, and the software, which combine to allow the computer to communicate with the external world. An example of an input interface is a switch, where the operator toggles the switch, and the software can recognize the switch position. An example of an output interface is a light-emitting diode (LED), where the software can turn the light on and off, and the operator can see whether or not the light is shining. There is a wide range of possible inputs and outputs, which can exist in either digital or analog form. In general, we can classify I/O interfaces into four categories

Parallel - binary data are available simultaneously on a group of lines
Serial - binary data are available one bit at a time on a single line
Analog - data are encoded as an electrical voltage, current, or power
Time - data are encoded as a period, frequency, pulse width, or phase shift

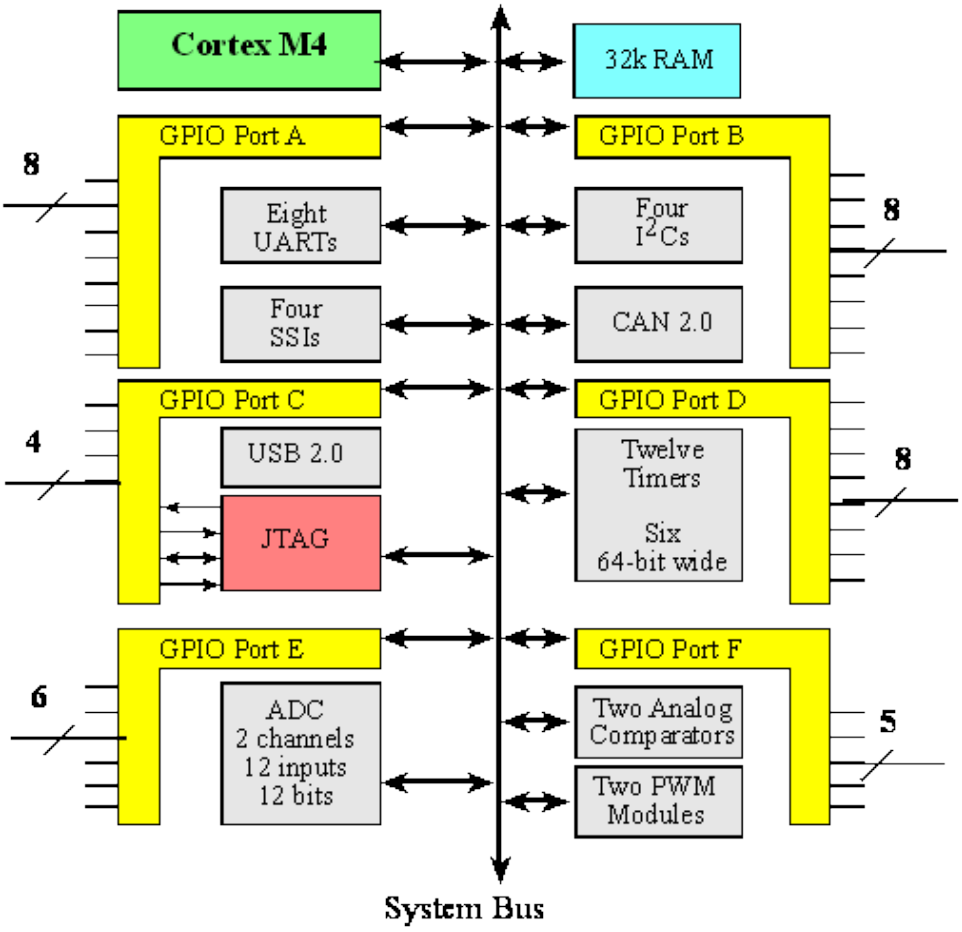


Figure 2.11. Architecture of TM4C123 microcontroller.

Reading Assignment:
The PDF document for the Launchpad Microcontroller Cortex M4

2.5. CPU Registers

Registers are high-speed storage inside the processor. The registers are depicted in Figure 2.12. R0 to R12 are general purpose registers and contain either data or addresses. Register R13 (also called the stack pointer, SP) points to the top element of the stack. Register R14 (also called the link register, LR) is used to store the return location for functions. The LR is also used in a special way during exceptions, such as interrupts. Interrupts are covered in Chapter 12. Register R15 (also called the program counter, PC) points to the next instruction to be fetched from memory. The processor fetches an instruction using the PC and then increments the PC.

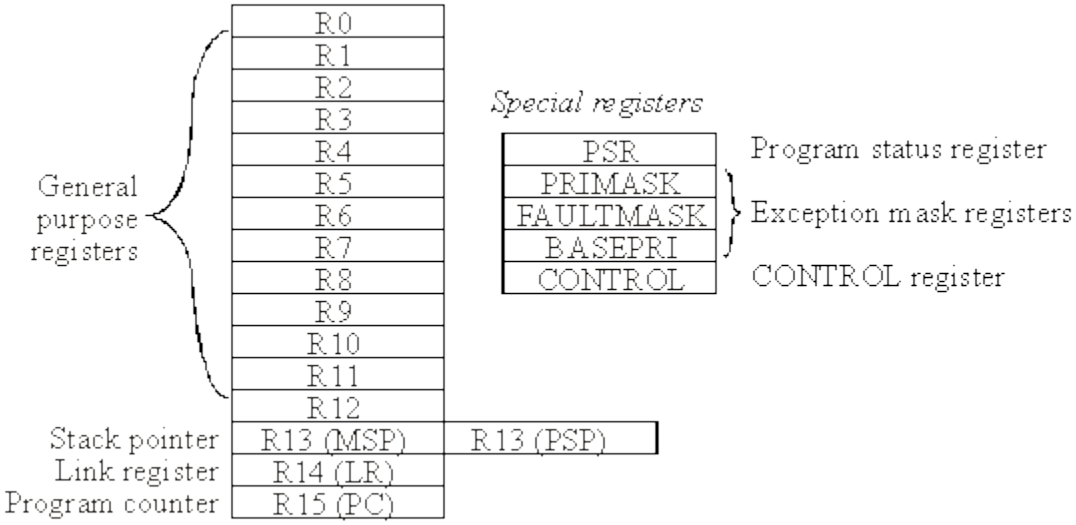


Figure 2.12. Registers on the ARM® Cortex-M processor.

The ARM Architecture **Procedure Call Standard**, AAPCS, part of the ARM **Application Binary Interface** (ABI), uses registers R0, R1, R2, and R3 to pass input parameters into a C function. Also according to AAPCS we place the return parameter in Register R0. In this class, the SP will always be the main stack pointer (MSP), not the Process Stack Pointer (PSP).

There are three status registers named Application Program Status Register (APSR), the Interrupt Program Status Register (IPSR), and the Execution Program Status Register (EPSR) as shown in Figure 2.13. These registers can be accessed individually or in combination as the **Program Status Register** (PSR). The N, Z, V, C, and Q bits give information about the result of a previous ALU operation. In general, the **N bit** is set after an arithmetical or logical operation signifying whether or not the result is negative. Similarly, the **Z bit** is set if the result is zero. The **C bit** means carry and is set on an unsigned overflow, and the **V bit** signifies signed overflow. The **Q bit** indicates that "saturation" has occurred – while you might want to look it up, saturated arithmetic is beyond the scope of this class.

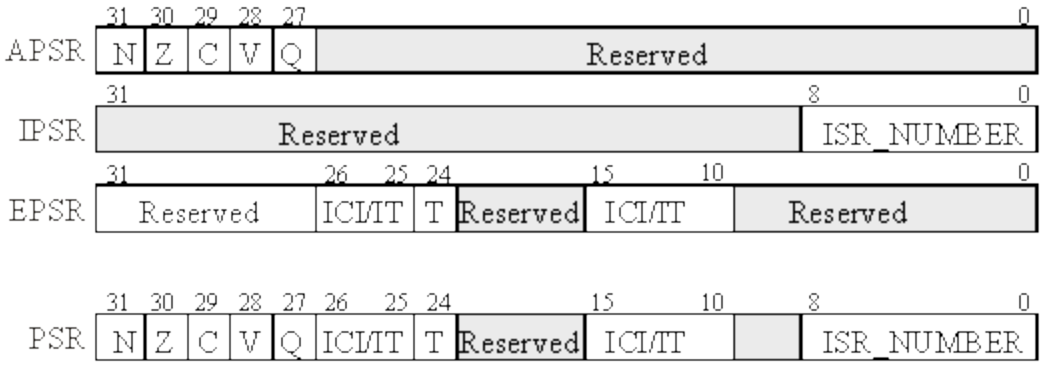


Figure 2.13. The program status register of the ARM® Cortex-M processor.

The **T bit** will always be 1, indicating the ARM® Cortex™-M processor is executing Thumb® instructions. The ISR_NUMBER indicates which interrupt if any the processor is handling. Bit 0 of the special register **PRIMASK** is the interrupt mask bit. If this bit is 1, most interrupts and exceptions are not allowed. If the bit is 0, then interrupts are allowed. Bit 0 of the special register **FAULTMASK** is the fault mask bit. If this bit is 1, all interrupts and faults are not allowed. If the bit is 0, then interrupts and faults are allowed. The nonmaskable interrupt (NMI) is not affected by these mask bits. The **BASEPRI** register defines the priority of the executing software. It prevents interrupts with lower or equal priority but allows higher priority interrupts. For example if **BASEPRI** equals 3, then requests with level 0, 1, and 2 can interrupt, while requests at levels 3 and higher will be postponed. A lower number means a higher priority interrupt. The details of interrupt processing will be presented in subsequent chapters.

2.6. Assembly Language

For this online class, Section 2.6 can be skipped. It is included for those readers who seek some basic understanding of low-level assembly programming. This section focuses on the ARM® Cortex™-M assembly language. There are many ARM® processors, and this class focuses on Cortex-M microcontrollers, which executes Thumb® instructions extended with Thumb-2 technology. This class will not describe in detail all the Thumb instructions. Rather, we focus on only a subset of the Thumb® instructions. This subset will be functionally complete without regard to minimizing code size or optimizing for execution speed. Furthermore, we will show general forms of instructions, but in many cases there are specific restrictions on which registers can be used and the sizes of the constants. For further details, please refer to the ARM® Cortex™-M Technical Reference Manual.

2.6.1. Syntax

Assembly language instructions have four fields separated by spaces or tabs. The **label field** is optional and starts in the first column and is used to identify the position in memory of the current instruction. You must choose a unique name for each label. The **opcode field** specifies the processor command to execute. The **operand field** specifies where to find the data to execute the instruction. Thumb instructions have 0, 1, 2, 3, or 4 operands, separated by commas. The **comment field** is also optional and is ignored by the assembler, but it allows you to describe the software making it easier to understand. You can add optional spaces between operands in the operand field. However, a semicolon must separate the operand and comment fields. Good programmers add comments to explain the software.

```
Label    Opcode  Operands        Comment
Func     MOV     R0, #100      ; this sets R0 to 100
                 BX  LR        ; this is a function return
```

Observation: A good comment explains why an operation is being performed, how it is used, how it can be changed, or how it was debugged. A bad comment explains what the operation does. The comments in the above two assembly lines are examples of bad comments.

When describing assembly instructions we will use the following list of symbols

Ra Rd Rm Rn Rt and **Rt2** represent registers
{Rd,} represents an optional destination register
#imm12 represents a 12-bit constant, 0 to 4095
#imm16 represents a 16-bit constant, 0 to 65535
operand2 represents the flexible second operand as described in Section 3.4.2
{cond} represents an optional logical condition as listed in Table 2.10
{type} encloses an optional data type
{S} is an optional specification that this instruction sets the condition code bits
Rm {, shift} specifies an optional shift on **Rm**
Rn {, #offset} specifies an optional offset to **Rn**

For example, the general description of the addition instruction

```
ADD{cond} {Rd,} Rn, #imm12
```

could refer to either of the following examples.

```
ADD   R0,#1          ; R0=R0+1
ADD   R0,R1,#10      ; R0=R1+10
```

Table 2.10 shows the conditions **{cond}** that we will use for conditional branching.

Suffix	*Flags*	*Meaning*
EQ	Z = 1	Equal
NE	Z = 0	Not equal
CS or **HS**	C = 1	Higher or same, unsigned ≥
CC or **LO**	C = 0	Lower, unsigned <
MI	N = 1	Negative
PL	N = 0	Positive or zero
VS	V = 1	Overflow
VC	V = 0	No overflow
HI	C = 1 and Z = 0	Higher, unsigned >
LS	C = 0 or Z = 1	Lower or same, unsigned ≤
GE	N = V	Greater than or equal, signed ≥
LT	N ≠ V	Less than, signed <
GT	Z = 0 and N = V	Greater than, signed >
LE	Z = 1 and N ≠ V	Less than or equal, signed ≤
AL	Can have any value	Always. This is the default when no suffix specified

Table 2.10. Condition code suffixes used to optionally execution instruction.

It is much better to add comments to explain how or even better why we do the action. Good comments also describe how the code was tested and identify limitations. But for now we are learning what the instruction is doing, so in this chapter comments will describe what the instruction does. The assembly **source code** is a text file (with Windows file extension **.s**) containing a list of instructions. If register R0 is an input parameter, the following is a function that will return in register R0 the value (100*input+10).

```
Func MOV   R1,#100      ; R1=100
     MUL   R0,R0,R1     ; R0=100*input
     ADD   R0,#10       ; R0=100*input+10
     BX    LR           ; return 100*input+10
```

The **assembler** translates assembly source code into **object code**, which are the machine instructions executed by the processor. All object code is halfword-aligned. This means instructions can be 16 or 32 bits wide, and the program counter bit 0 will always be 0. The **listing** is a text file containing a mixture of the object code generated by the assembler together with our original source code.

```
Address     Object code  Label Opcode Operand        comment
0x000005E2  F04F0164     Func MOV    R1,#0x64      ; R1=100
0x000005E6  FB00F001          MUL    R0,R0,R1      ; R0=100*input
0x000005EA  F100000A          ADD    R0,R0,#0x0A   ; R0=100*input+10
0x000005EE  4770              BX     LR            ; return 100*input+10
```

When we **build** a project all files are assembled or compiled then linked together. The address values shown in the listing are relative to the particular file being assembled. When the entire project is built, the files are linked together, and the **linker** decides exactly where in memory everything will be. After building the project, it can be downloaded, which programs the object code into flash ROM. You are allowed to load and execute software out of RAM. But for an embedded system, we typically place executable instructions into nonvolatile flash ROM. The listing you see in the debugger will specify the absolute address showing you exactly where in memory your variables and instructions exist.

2.6.2. Addressing Modes and Operands

A fundamental issue in program development is the differentiation between data and address. When we put the number 1000 into Register R0, whether this is data or address depends on how the 1000 is used. To run efficiently, we try to keep frequently accessed information in registers. However, we need to access memory to fetch parameters or save results. The **addressing mode** is the format the instruction uses to specify the memory location to read or write data. The addressing mode is associated more specifically with the operands, and a single instruction could exercise multiple addressing modes for each of the operands. When the import is obvious though, we will use the expression "the addressing mode of the instruction", rather than "the addressing mode of an operand in an instruction". All instructions begin by fetching the machine instruction (op code and operand) pointed to by the PC. When extended with Thumb-2 technology, some machine instructions are 16 bits wide, while others are 32 bits. Some instructions operate completely within the processor and require no memory data fetches. For example, the **`ADD R1,R2`** instruction performs R1+R2 and stores the sum back into R1. If the data is found in the instruction itself, like **`MOV R0,#1`**, the instruction uses **immediate addressing** mode. A register that contains the address or the location of the data is called a **pointer** or **index** register. **Indexed addressing** mode uses a register pointer to access memory. The addressing mode that uses the PC as the pointer is called **PC-relative addressing** mode. It is used for branching, for calling functions, and accessing constant data stored in ROM. The addressing mode is called PC relative because the machine code contains the address difference between where the program is now and the address to which the program will access. The **`MOV`** instruction will move data within the processor without accessing memory. The **`LDR`** instruction will read a 32-bit word from memory and place the data in a register. With PC-relative addressing, the assembler automatically calculates the correct PC offset.

Register. Most instructions operate on the registers. In general, data flows towards the op code (right to left). In other words, the register closest to the op code gets the result of the operation. In each of these instructions, the result goes into R2.

```
MOV   R2,#100      ; R2=100, immediate addressing
LDR   R2,[R1]      ; R2= value pointed to by R1
ADD   R2,R0        ; R2= R2+R0
ADD   R2,R0,R1     ; R2= R0+R1
```

Register list. The stack push and stack pop instructions can operate on one register or on a list of registers. SP is the same as R13, LR is the same as R14, and PC is the same as R15.

```
PUSH  {LR}          ; save LR on stack
POP   {LR}          ; remove from stack and place in LR
PUSH  {R1,R2,LR}    ; save registers and return address
POP   {R1,R2,PC}    ; restore registers and return
```

Immediate addressing. With immediate addressing mode, the data itself is contained in the instruction. Once the instruction is fetched no additional memory access cycles are required to get the data. Notice the number 100 (0x64) is embedded in the machine code of the instruction shown in Figure 2.14. Immediate addressing is only used to get, load, or read data. It will never be used with an instruction that stores to memory.

```
MOV   R0,#100       ; R0=100, immediate addressing
```

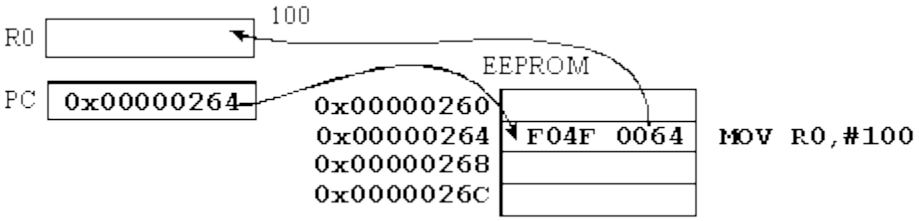


Figure 2.14. An example of immediate addressing mode, data is in the instruction.

Indexed addressing. With indexed addressing mode, the data is in memory and a register will contain a pointer to the data. Once the instruction is fetched, one or more additional memory access cycles are required to read or write the data. In these examples, R1 points to RAM. In this class, we will focus on just the first two forms of indexed addressing.

```
LDR   R0,[R1]       ; R0= value pointed to by R1
LDR   R0,[R1,#4]    ; R0= word pointed to by R1+4
LDR   R0,[R1,#4]!   ; first R1=R1+4, then R0= word pointed to by R1
LDR   R0,[R1],#4    ; R0= word pointed to by R1, then R1=R1+4
LDR   R0,[R1,R2]    ; R0= word pointed to by R1+R2
LDR   R0,[R1,R2, LSL #2] ; R0= word pointed to by R1+4*R2
```

In Figure 2.15, R1 points to RAM, the instruction **`LDR R0,[R1]`** will read the 32-bit value pointed to by R1 and place it in R0. R1 could be pointing to any valid object in the memory map (i.e., RAM, ROM, or I/O), and R1 is not modified by this instruction.

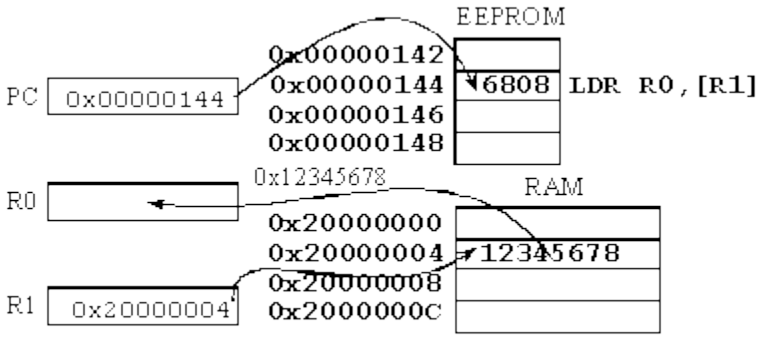


Figure 2.15. An example of indexed addressing mode, data is in memory.

In Figure 2.16, R1 points to RAM, the instruction **`LDR R0,[R1,#4]`** will read the 32-bit value pointed to by R1+4 and place it in R0. Even though the memory address is calculated as R1+4, the Register R1 itself is not modified by this instruction.

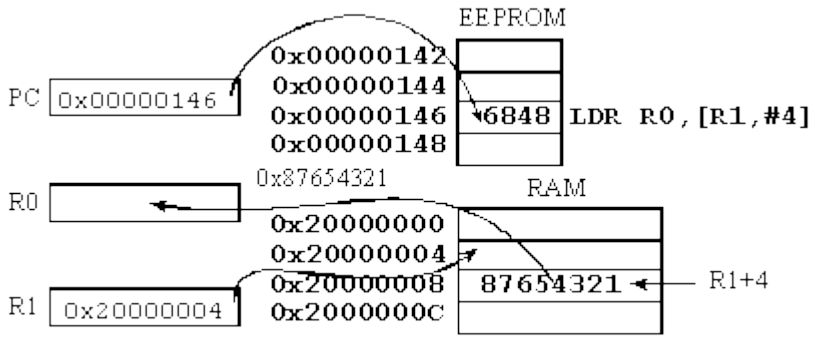


Figure 2.16. An example of indexed addressing mode with offset, data is in memory.

PC-relative addressing. PC-relative addressing is indexed addressing mode using the PC as the pointer. The PC always points to the instruction that will be fetched next, so changing the PC will cause the program to branch. A simple example of PC-relative addressing is the unconditional branch. In assembly language, we simply specify the label to which we wish to jump, and the assembler encodes the instruction with the appropriate PC-relative offset.

```
B      Location      ; jump to Location, using PC-relative addressing
```

The same addressing mode is used for a function call. Upon executing the **`BL`** instruction, the return address is saved in the link register (LR). In assembly language, we simply specify the label defining the start of the function, and the assembler creates the appropriate PC-relative offset.

```
BL     Subroutine    ; call Subroutine, using PC-relative addressing
```

Typically, it takes two instructions to access data in RAM or I/O. The first instruction uses PC-relative addressing to create a pointer to the object, and the second instruction accesses the memory using the pointer. We can use the **`=Something`** operand for any symbol defined by our program. In this case **`Count`** is the label defining a 32-bit variable in RAM.

```
LDR    R1,=Count     ; R1 points to variable Count, using PC-relative
LDR    R0,[R1]       ; R0= value pointed to by R1
```

The operation caused by the above two **`LDR`** instructions is illustrated in Figure 2.17. Assume a 32-bit variable **`Count`** is located in the data space at RAM address 0x2000.0000. First, **`LDR R1,=Count`** makes R1 equal to 0x2000.0000. I.e., R1 points to **`Count`**. The assembler places a constant 0x2000.0000 in code space and translates the **`=Count`** into the correct PC-relative access to the constant (e.g., **`LDR R1,[PC,#28]`**). In this case, the constant 0x2000.0000, the address of **`Count`**, will be located at PC+28. Second, the **`LDR R0,[R1]`** instruction will dereference this pointer, bringing the 32-bit contents at location 0x2000.0000 into R0. Since **`Count`** is located at 0x2000.0000, these two instructions will read the value of the variable into R0.

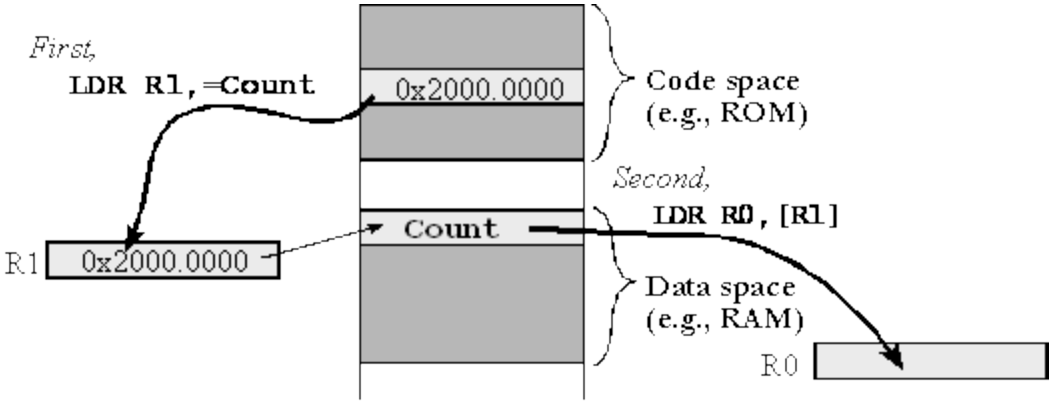


Figure 2.17. Indexed addressing using R1 as a register pointer to access memory. Data is moved into R0. Code space is where we place programs and data space is where we place variables.

Flexible second operand <op2>. Many instructions have a flexible second operand, shown as **<op2>** in the descriptions of the instruction. **<op2>** can be a constant or a register with optional shift. The flexible second operand can be a constant in the form **#constant**

```
ADD Rd, Rn, #constant   ;Rd = Rn+constant
```

where **constant** is calculated as one of these four, **X** and **Y** are hexadecimal digits:

- Constant produced by shifting an unsigned 8-bit value left by any number of bits
- Constant of the form **0x00XY00XY**
- Constant of the form **0xXY00XY00**
- Constant of the form **0xXYXYXYXY**

We can also specify a flexible second operand in the form **Rm {,shift}**. If **Rd** is missing, **Rn** is also the destination. For example:

```
ADD Rd, Rn, Rm {,shift}   ;Rd = Rn+Rm
ADD Rn, Rm {,shift}       ;Rn = Rn+Rm
```

Where, **Rm** is the register holding the data for the second operand, and **shift** is an optional shift to be applied to **Rm**. The optional **shift** can be one of these five formats:

ASR #n Arithmetic (signed) shift right **n** bits, $1 \le n \le 32$.
LSL #n Logical (unsigned) shift left **n** bits, $1 \le n \le 31$.
LSR #n Logical (unsigned) shift right **n** bits, $1 \le n \le 32$.
ROR #n Rotate right **n** bits, $1 \le n \le 31$.
RRX Rotate right one bit, with extend.

If we omit the shift, or specify **LSL #0**, the value of the flexible second operand is **Rm**. If we specify a shift, the shift is applied to the value in **Rm**, and the resulting 32-bit value is used by the instruction. However, the contents in the register **Rm** remain unchanged. For example,

```
ADD R0,R1,LSL #4       ; R0 = R0 + R1*16 (R1 unchanged)
ADD R0,R1,R2,ASR #4    ; signed R0 = R1 + R2/16 (R2 unchanged)
```

An **aligned access** is an operation where a word-aligned address is used for a word, dual word, or multiple word access, or where a halfword-aligned address is used for a halfword access. Byte accesses are always aligned. The address of an aligned word access will have its bottom two bits equal to zero. An **unaligned** word access means we are accessing a 32-bit object (4 bytes) but the address is not evenly

divisible by 4. The address of an aligned halfword access will have its bottom bit equal to zero. An unaligned halfword access means we are accessing a 16-bit object (2 bytes) but the address is not evenly divisible by 2. The Cortex-M processor supports unaligned access only for the following instructions:

LDR	Load 32-bit word
LDRH	Load 16-bit unsigned halfword
LDRSH	Load 16-bit signed halfword (sign extend bit 15 to bits 31-16)
STR	Store 32-bit word
STRH	Store 16-bit halfword

Transfers of one byte are allowed for the following instructions:

LDRB	Load 8-bit unsigned byte
LDRSB	Load 8-bit signed byte (sign extend bit 7 to bits 31-8)
STRB	Store 8-bit byte

When loading a 32-bit register with an 8- or 16-bit value, it is important to use the proper load, depending on whether the number being loaded is signed or unsigned. This determines what is loaded into the most significant bits of the register to ensure that the number keeps the same value when it is promoted to 32 bits. When loading an 8-bit unsigned number, the top 24 bits of the register will become zero. When loading an 8-bit signed number, the top 24 bits of the register will match bit 7 of the memory data (signed extend). Note that there is no such thing as a signed or unsigned store. For example, there is no **STRSH**; there is only **STRH**. This is because 8, 16, or all 32 bits of the register are stored to an 8-, 16-, or 32-bit location, respectively. No promotion occurs. This means that the value stored to memory can be different from the value located in the register if there is overflow. When using **STRB** to store an 8-bit number, be sure that the number in the register is 8 bits or less.

All other read and write memory operations generate a usage fault exception if they perform an unaligned access, and therefore their accesses must be address aligned. Also, unaligned accesses are usually slower than aligned accesses, and some areas of memory do not support unaligned accesses. But unaligned accesses may allow programs to use memory more efficiently at the cost of performance. The tradeoff between speed and size is a common motif.

Observation: We add a dot in the middle of 32-bit hexadecimal numbers (e.g., 0x2000.0000). This dot helps the reader visualize the number. However, this dot should not be used when writing actual software.

Common Error: Since not every instruction supports every addressing mode, it would be a mistake to use an addressing mode not available for that instruction.

Checkpoint 2.33: What is the addressing mode used for?
Checkpoint 2.34: Assume R3 equals 0x2000.0000 at the time **`LDR R2,[R3,#8]`** is executed. What address will be accessed? If R3 is changed, to what value will R3 become?
Checkpoint 2.35: Assume R3 equals 0x2000.0000 at the time **`LDR R2,[R3],#8`** is executed. What address will be accessed? If R3 is changed, to what value will R3 become?

2.7. Address Space

Microcontrollers within the same family differ by the amount of memory and by the types of I/O modules. All LM3S and TM4C microcontrollers have a Cortex-M processor. There are hundreds of members in this family; some of them are listed in Table 2.11.

Part number	*RAM*	*Flash*	*I/O*	*I/O modules*
LM3S811	8	64	32	PWM
LM3S1968	64	256	52	PWM
LM3S2965	64	256	56	PWM, CAN
LM3S3748	64	128	61	PWM, DMA, USB
LM3S6965	64	256	42	PWM, Ethernet
LM3S8962	64	256	42	PWM, CAN, Ethernet, IEEE1588
LM4F110B2QR	12	32	43	floating point, CAN, DMA
LM4F120H5QR	32	256	43	floating point, CAN, DMA, USB
TM4C123GH6PM	32	256	43	floating point, CAN, DMA, USB, PWM
	KiB	KiB	pins	

Table 2.11. Memory and I/O modules (all have SysTick, RTC, timers, UART, I^2C, SSI, and ADC).

The memory map of TM4C123 is illustrated in Figure 2.18. Although specific for the TM4C123, all ARM® Cortex™-M microcontrollers have similar memory maps. In general, Flash ROM begins at address 0x0000.0000, RAM begins at 0x2000.0000, the peripheral I/O space is from 0x4000.0000 to 0x5FFF.FFFF, and I/O modules on the private peripheral bus exist from 0xE000.0000 to 0xE00F.FFFF. In particular, the only differences in the memory map for the various 180 members of the LM3S/TM4C family are the ending addresses of the flash and RAM. Having multiple buses means the processor can perform multiple tasks in parallel. The following is some of the tasks that can occur in parallel

ICode bus Fetch opcode from ROM
DCode bus Read constant data from ROM
System bus Read/write data from RAM or I/O, fetch opcode from RAM
PPB Read/write data from internal peripherals like the NVIC
AHB Read/write data from high-speed I/O and parallel ports (M4 only)

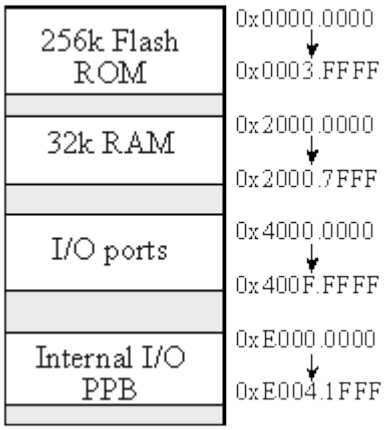


Figure 2.18. Memory map of the TM4C123.

When we store 16-bit data into memory it requires two bytes. Since the memory systems on most computers are byte addressable (a unique address for each byte), there are two possible ways to store in memory the two bytes that constitute the 16-bit data. Freescale microcomputers implement the **big endian** approach that stores the most significant byte at the lower address. Intel microcomputers implement the **little endian** approach that stores the least significant byte at the lower address. The Texas Instruments TM4C microcontrollers use the little endian format. Many ARM® processors are **biendian**, because they can be configured to efficiently handle both big and little endian data. Instruction fetches on the ARM are always little endian. Figure 2.19 shows two ways to store the 16-bit number 1000 (0x03E8) at locations 0x2000.0850 and 0x2000.0851. We also can use either the big or little endian approach when storing 32-bit numbers into memory that is byte (8-bit) addressable. Figure 2.20 shows the big and little endian formats that could be used to store the 32-bit number 0x12345678 at locations 0x2000.0850 through 0x2000.0853.

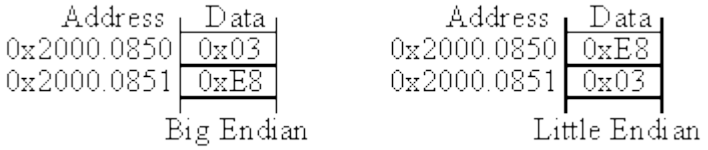


Figure 2.19. Example of big and little endian formats of a 16-bit number.

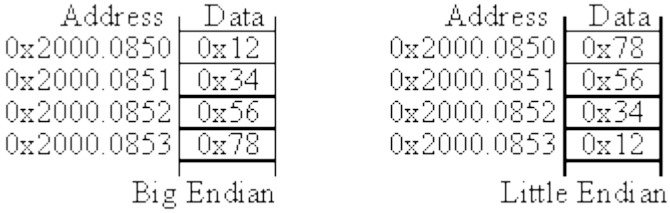


Figure 2.20. Example of big and little endian formats of a 32-bit number.

In the previous two examples, we normally would not pick out individual bytes (e.g., the 0x12), but rather capture the entire multiple byte data as one nondivisable piece of information. On the other hand, if each byte in a multiple byte data structure is individually addressable, then both the big and little endian schemes store the data in first to last sequence. For example, if we wish to store the four ASCII characters 'LM3S', which is 0x4C4D3353 at locations 0x2000.0850 through 0x2000.0853, then the ASCII 'L'=0x4C comes first in both big and little endian schemes, as illustrated in Figure 2.21.

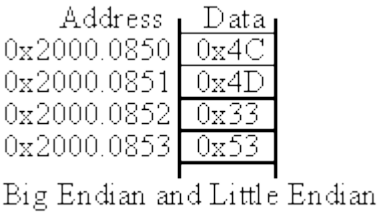


Figure 2.21. Character strings are stored in the same for both big and little endian formats.

The terms "big and little endian" come from Jonathan Swift's satire Gulliver's Travels. In Swift's book, a Big Endian refers to a person who cracks their egg on the big end. The Lilliputians were Little Endians because they insisted that the only proper way is to break an egg on the little end. The Lilliputians considered the Big Endians as inferiors. The Big and Little Endians fought a long and senseless war over the best way to crack an egg.

Common Error: An error will occur when data is stored in Big Endian by one computer and read in Little Endian format on another.

2.8. Software Development Process

In this class we will begin with assembly language, and then introduce C. However, the process described in this section applies to both assembly and C. Either the ARM Keil™ **uVision®** or the Texas Instruments **Code Composer Studio™ (CCStudio)** integrated development environment (IDE) can be used to develop software for the Texas Instruments microcontrollers. Both include an editor, assembler, compiler, and simulator. Furthermore, both can be used to download and debug software on a real microcontroller. Either way, the entire development process is contained in one application, as shown in Figure 2.22. In this course, we will use ARM Keil™ **uVision.**

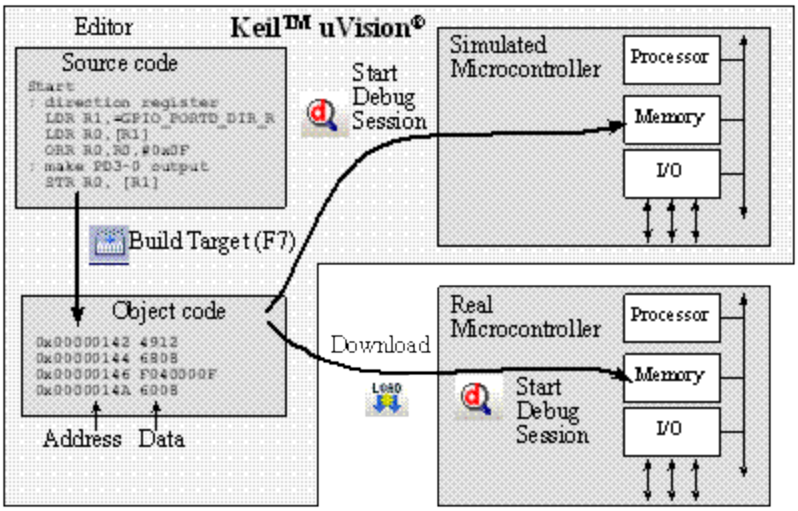


Figure 2.22. Assembly language or C development process.

To develop software, we first use an **editor** to create our **source code**. Source code contains specific set of sequential commands in human-readable-form. Next, we use an **assembler** or **compiler** to translate our source code into object code. On ARM Keil™ uVision® we compile/assemble by executing the command **Project->Build Target** (short cut F7). Object code or machine instructions contains these same commands in machine-readable-form. Most assembly source code is one-to-one with the object code that is executed by the computer. For example, when programming in a high level language like C or Java, one line of a program can translate into several machine instructions. In contrast, one line of assembly code usually translates to exactly one machine instruction. The assembler/compiler may also produce a **listing file**, which is a human-readable output showing the addresses and object code that correspond to each line of the source program. The **target** specifies the platform on which we will be running the object code. When testing software with the simulator, we choose the **Simulator** as the target. When simulating, there is no need to download, we simply launch the simulator by executing the **Debug->Start Debug Session** command. The simulator is an easy and inexpensive way to get started on a project. However, its usefulness will diminish as the I/O becomes more complex.

In a real system, we choose the real microcontroller via its JTAG debugger as the target. In this way the object code is downloaded into the EEPROM of the microcontroller. Most microcontrollers contain built-in features that assist in programming their EEPROM. In particular, we will use the JTAG debugger connected via a USB cable to download and debug programs. The JTAG is both a **loader** and a **debugger**. We program the EEPROM by executing the **Flash->Download** command. After downloading we can start the system by hitting the reset button on the board or we can debug it by executing **Debug->Start Debug Session** command in the uVision® IDE.

In contrast, the loader on a general purpose computer typically reads the object code from a file on a hard drive or CD and stores the code in RAM. When the program is run, instructions are fetched from RAM. Since RAM is volatile, the programs on a general purpose computer must be loaded each time the system is powered up.

For embedded systems, we typically perform initial testing on a simulator. The process for developing applications on real hardware is identical except the target is switched from a simulated microcontroller to the real microcontroller. It is best to have a programming reference manual handy when writing assembly language. These three reference manuals for the Cortex M4 processor are available as pdf files and are posted on the book web site.

A description of each instruction can also be found by searching the Contents page of the help engine included with the ARM Keil™ uVision® or TI CCStudio applications. There are a lot of settings

required to create a software project from scratch. I strongly suggest those new to the process first run lots of existing projects. Next, pick an existing project most like your intended solution, and then make a copy of that project. Finally, make modifications to the copy a little bit at a time as you morph the existing project into your solution. After each modification verify that it still runs. If you take a project that runs, make hundreds of changes to it, and then notice that it no longer runs, you will not know which of the many changes caused the failure.

2.9. Chapter 2 Quiz

2.1 Make this a matching definition with the word

a) Precision ---------- number of different values

b) Hexadecimal ----------base sixteen

c) Fixed point ---------- number system that can be used to define non-integer values

d) Energy ---------- defines the amount of work that can be done

e) Resistance ---------- potential divided by flow

2.2 How many bits is each?

a) Binary bit ---------- 1

b) Nibble ---------- 4

c) Byte ---------- 8

d) Halfword ----------16

e) Word ---------- 32

Chapter 3: Introduction to Electronics
Embedded Systems - Shape The World
Jonathan Valvano and Ramesh Yerraballi

This chapter introduces electronics. We will learn Ohm's Law. This chapter is foundational, laying the ground work for the remainder of the book.

Learning Objectives:

- Understand current, voltage, power, energy.
- Learn Ohm's Law.
- Understand Kirchoffs Voltage and Current Laws for circuit analysis
- Learn to recognize common circuit configurations like "voltage dividers" and "current dividers"

3.0. Introduction

Most students reading this will have had some prior training in electronics. However, this brief section will provide an overview of the electronics needed to understand electric circuits in this class. **Current**, *I*, is defined as the movement of electrons. In particular, 1 ampere (A) of current is 6.241×10^{18} electrons per second, or one coulomb per second. Current is measured at one point as the number of electrons travelling per second. Current has an amplitude and a direction. Because electrons are negatively charged, if the electrons are moving to the left, we define current as flowing to the right. **Voltage**, *V*, is an electrical term representing the potential difference between two points. The units of voltage are volts (V), and it is always measured as a difference. Voltage is the electromotive force or potential to produce current. We will see two types of conducting media: a **wire** and a **resistor**. Wires, made from copper, will allow current to freely flow, but forcing current to flow through a resistor will require energy. The electrical property of a resistor is resistance in ohms (**Ω**). Ideally, a wire is simply a resistor with a resistance of 0 **Ω**. The basic relation between voltage, current, and resistance for a resistor is known as **Ohm's Law**, which can be written three ways:

$V = I * R$	$Voltage = Current * Resistance$
$I = V / R$	$Current = Voltage / Resistance$
$R = V / I$	$Resistance = Voltage / Current$

The left side of Figure 3.1 shows a circuit element representation of a resistor, of resistance *R*. Whenever we define voltage, we must clearly specify the two points across which the potential is defined. Typically we label voltages with + and –, defining the voltage *V* as the potential to produce current from the + down to the –. When defining current we draw an arrow signifying the direction of the current. If the voltage *V* is positive, then the current *I* will be positive meaning the current is down in this figure. However, because electrons have negative charge, the electrons are actually flowing up. According to the passive sign convention, we define positive current as the direction of the flow of positive charge (or the opposite direction of the flow of negative charge). The middle of Figure 3.1 shows a circuit with a 1 k**Ω** resistor placed across a 3.7V battery. 1 k**Ω** exactly the same as 1000 **Ω**, just like 1 km is the same as 1000 m. According to Ohm's Law, 3.7 mA of current will flow down across the resistor. 1 mA exactly the same as 0.001 A, just like 1 mm is the same as 0.001 m. In this circuit, current flows clockwise from the + terminal of the battery, down across the resistor, and then back to the – terminal of the battery.

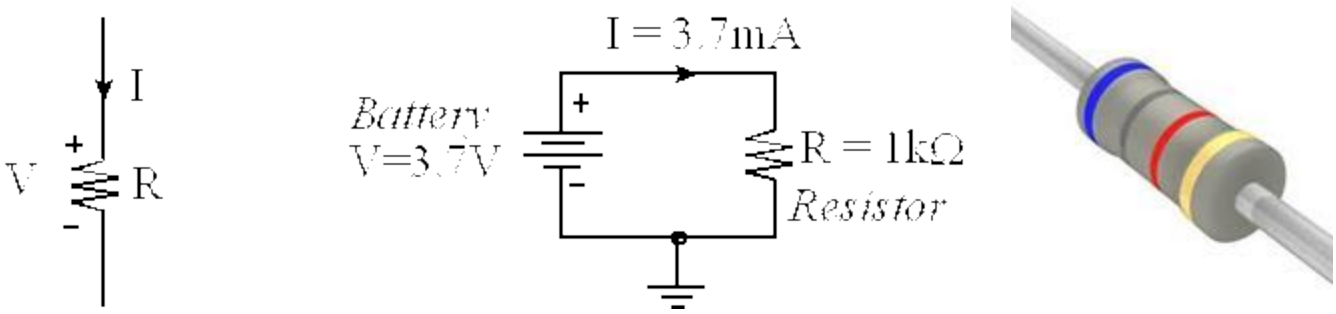


Figure 3.1. The voltage and current definitions; a circuit with a battery; and a drawing of a resistor.

Checkpoint 3.1: There is 1 V across a resistor, and 5 mA is flowing. What is the resistance?
Checkpoint 3.2: There is 2 V across a 100Ω resistor. How much current is flowing?
Checkpoint 3.3: What happens if you place a wire directly from + terminal to the –terminal of a battery?

There are two analogous physical scenarios that might help you understand the concept of voltage, current, and resistance. The first analogy is flowing water through a pipe. We place a large reservoir of water in a tower, connect the water through a pipe, and attach a faucet at the bottom of the pipe, see Figure 3.2. In this case pressure is analogous to voltage, water flow is analogous to current, and fluid resistance of the faucet is analogous to electrical resistance. Notice that water pressure is defined as the potential to cause water to flow, and it is measured between two places. Pressure has a polarity and water flow has a direction. If the faucet is turned all the way off, its resistance is infinite, and no water flows. If the faucet is turned all the way on, its resistance is not zero, but some finite amount. As we turn the faucet we are varying the fluid resistance. The fluid resistance will determine the amount of flow:

Flow = Pressure/Resistance

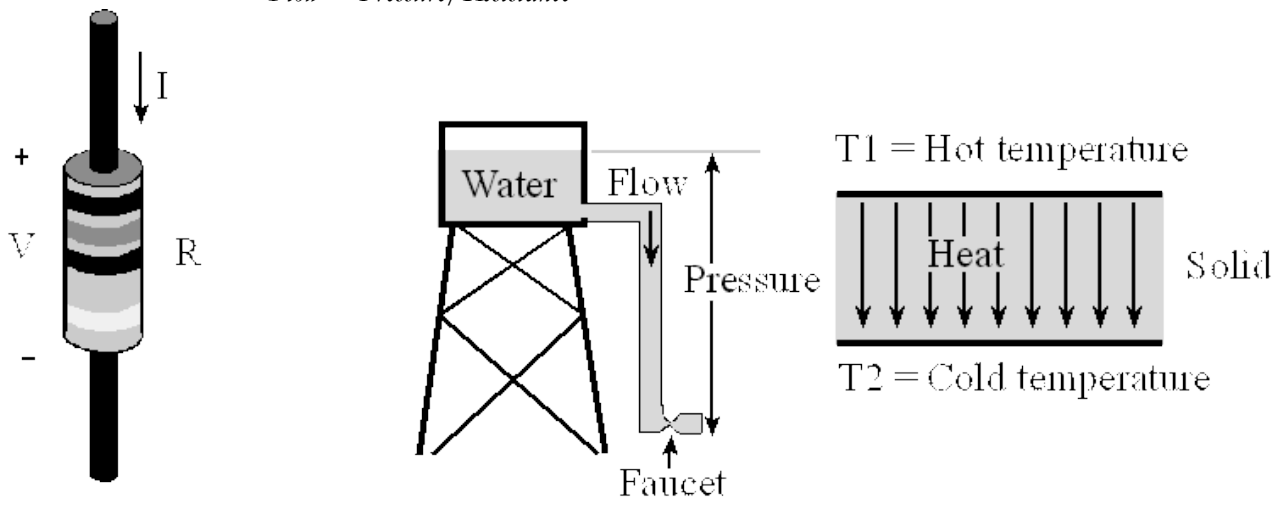


Figure 3.2. Three analogous physical systems demonstrating Ohm's Law.

Checkpoint 3.4: If pressure is measured in Newtons/m^2 (Pascal) and flow measured in m^3/sec, what are the units of fluid resistance?

A second analogy is heat flow across a solid. If we generate a temperature gradient across a solid, heat will flow from the hot side to the cold side (right side of Figure 3.2). This solid could be a glass window on a house or the wall of your coffee cup. In this case temperature gradient is analogous to voltage, heat flow is analogous to current, and thermal resistance of the solid is analogous to electrical resistance. Notice that potential is defined as the temperature difference between two places. Heat flow also has a direction. If the coffee cup is made from metal, its thermal resistance is low, lots of heat will flow, and the coffee cools off quickly. If the coffee cup is made of Styrofoam, its resistance is high, little heat will

flow, and the coffee remains hot for a long time. The temperature difference divided by the thermal resistance will determine the amount of flow:

$$Flow = (T1-T2)/Resistance$$

Checkpoint 3.5: If heat flow is measured in watts (Joules/sec) and temperature measured in ºC, what are the units of thermal resistance?

The R-value of insulation put in the walls and ceiling of a house is usually given in units per square area, e.g., $m^2 \cdot ºC/W$. The amount of heat flow across a wall is:

$$Flow = Area * (T1-T2)/R\text{-}value$$

Another important parameter occurring when current flows through a resistor is **power**. The power (*P* in watts) dissipated in a resistor can be calculated from voltage (*V* in volts), current (*I* in amps), and resistance (*R* in ohms). Interestingly, although voltage has a polarity (+ and –) and current has a direction, power has neither a polarity nor a direction.

$P = V * I$ $\quad Power = Voltage * Current$

$P = V^2 / R$ $\quad Power = Voltage^2 / Resistance$

$P = I^2 * R$ $\quad Power = Current^2 * Resistance$

Checkpoint 3.6: There is 1 V across a resistor, and 5 mA is flowing. How much power is being dissipated?
Checkpoint 3.7: There is 2 V across a 100Ω resistor. How much power is being dissipated?

The **energy** (*E* in joules) stored in a battery can be calculated from voltage (*V* in volts), current (*I* in amps), and time (*t* in seconds). In a manner similar to power, energy has neither a polarity nor a direction.

$E = V * I * t$ $\quad Energy = Voltage * Current * time$

$E = P * t$ $\quad Energy = Power * time$

3.1. Electric Circuits

A switch is an element used to modify the behavior of the circuit (Figure 3.3). If the switch is pressed, its resistance is 0, and current can flow across the switch. If the switch is not pressed, its resistance is infinite, and no current will flow. In reality, the ON-resistance of a switch is less than 0.1Ω, but this is so close to zero, we can assume the ideal value of 0 in most cases. Similarly, the OFF-resistance is actually greater than 100MΩ, but this is so close to infinity that we can again assume the ideal value of infinity. The classic electrical circuit involves a battery, a light bulb (modeled in this circuit as a 100Ω resistor), and a switch.

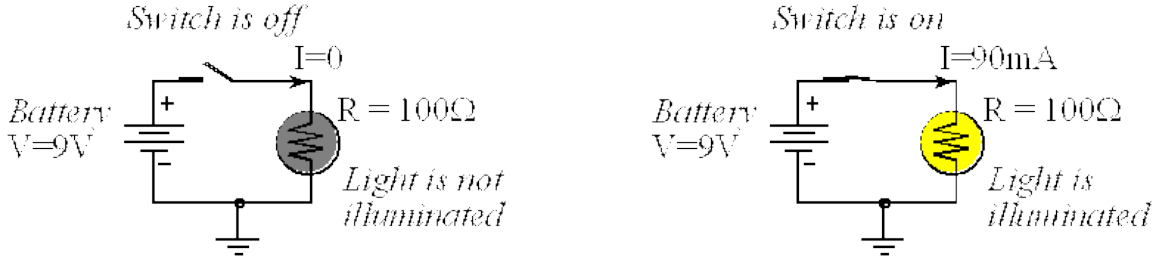


Figure 3.3. When the switch is open, no current can flow, and the bulb does not emit light. When the switch is closed, 90 mA of current will flow, and the bulb emits light.

Checkpoint 3.8: If the switch is on, how much power is being dissipated in the bulb?
Checkpoint 3.9: If the battery has 1000 joules stored in it, how long will the light be illuminated, assuming the voltage is constant?

There are a few basic rules that allow us to solve for voltages and currents within a circuit comprised with batteries, switches, and resistors.

Current always flows in a loop. In Figure 3.3 when the switch is pressed, current flows out of the + side of battery, across the switch, through the light and back to the – side of the battery. When there is no loop, no current can flow. In Figure 3.3 when the switch is off, the loop is broken, and no current will flow.

Kirchhoff's Voltage Law (KVL). The sum of the voltages around the loop is zero. For a battery, we label the + and – sides exactly the way the battery is labeled. For a resistor, we label the current arrow and the voltage + – like the left side of Figure 3.1. The important step is the direction of the current arrow must match the polarity of the corresponding voltage. It is common practice to draw arrows in the direction the currents actually flow, so the voltages will be positive. However, sometimes we don't know which way the current will flow, so we can just guess. If we happen to guess wrong, both the current and voltage will calculate to be negative and the correct behavior will still be obtained. We can think of the switch as a resistor of either 0 or infinity resistance, so it too can be labeled with a current arrow and a voltage polarity. Figure 3.4 shows the light circuit redrawn to show voltages and currents. As we are going around a circle and pass from + to –, we add that voltage. However, if we pass across an element from – to + ve subtract that voltage.

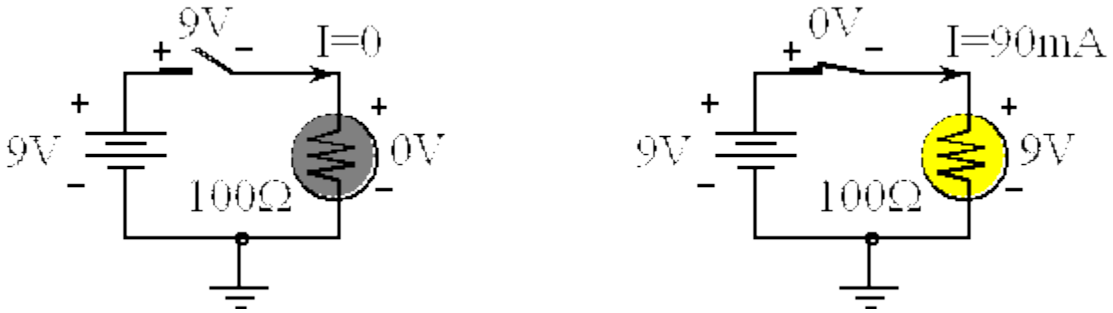


Figure 3.4. The voltages around a loop will sum to zero (KVL).

Kirchhoff's Current Law (KCL). The sum of the currents into a node equal the sum of the currents leaving a node as shown in Figure 3.5. To solve circuits using KCL and KVL, the current arrow across a resistor goes from the + voltage to the – voltage. Conversely, the current arrow across a battery goes from the – voltage to the + voltage. This is the same thing as saying current comes out of the battery's + terminal and into the battery's – terminal. At Node A, there is one incoming current and one outgoing current. This is a simple but important fact that *I1 = I2*. At Node B, there is one incoming current and two outgoing currents. Therefore, *I3 = I4+I5*. There are two currents into NodeC and two currents out of NodeC; thus, *I6+I7 = I8+I9*.

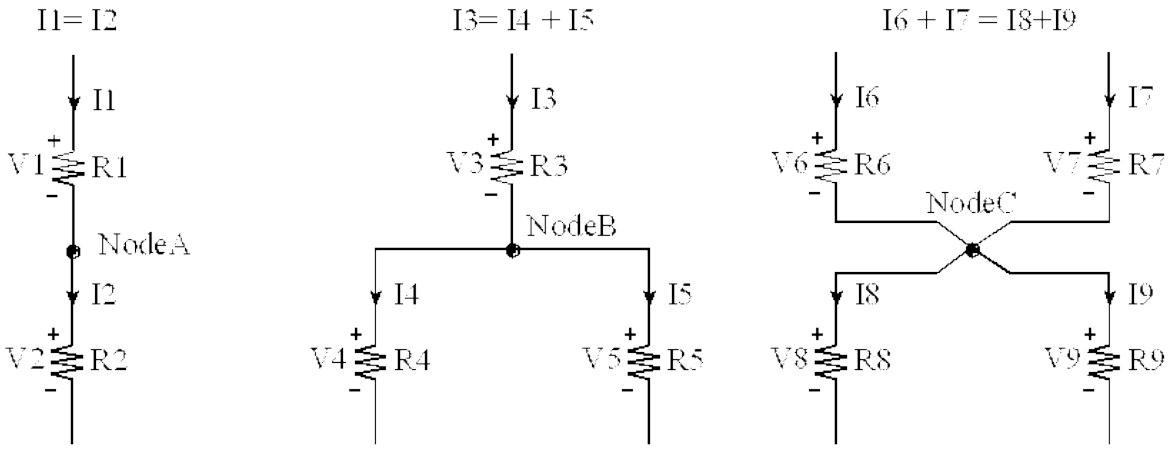


Figure 3.5. The sum of the currents into a node will equal the sum of the currents leaving (KCL).

Observation: If at all possible, draw the circuit so current flows down across the resistors and switches. As a secondary rule have currents go left to right across resistors and switches.

Series resistance. If resistor *R1* is in series with resistor *R2*, this combination behaves like one resistor with a value equal to *R1+R2*. See Figure 3.6. This means if replace the two series resistors in a circuit with one resistor at *R= R1+R2*, the behavior will be the same. The *V* equals *V1+V2*. By KCL, the currents through the two resistors are the same. These two facts can be used to derive the **voltage divider rule**

$$V2 = I*R2 = (V/R)*R2 = V*R2/(R1+R2)$$

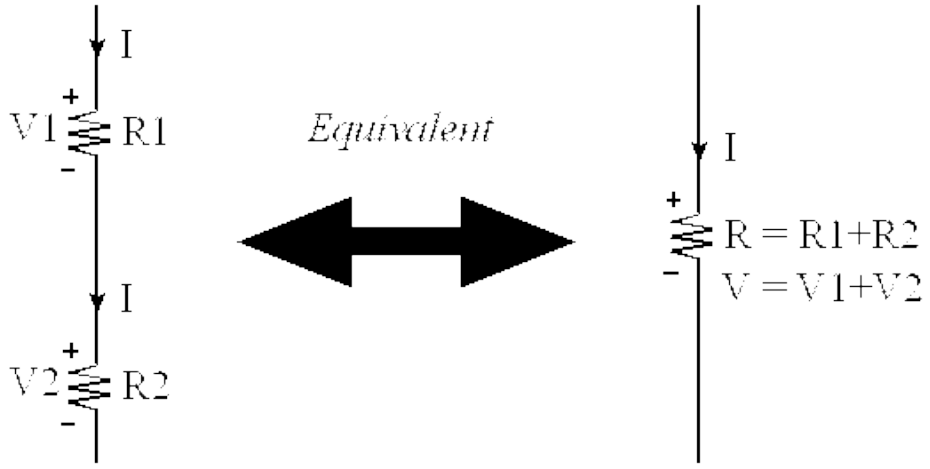


Figure 3.6. The series combination of two resistors, R1 R2, is equivalent to one resistor at R1+R2.

Checkpoint 3.10: Using Figure 3.6, assume I is 1mA, R1 is 2k**Ω** and R2 is 3k**Ω**, what is V?
Checkpoint 3.11: Using Figure 3.6, assume V is 10V, R1 is 2k**Ω** and R2 is 3k**Ω**, what is V2?

Parallel resistance. If resistor *R1* is in parallel with resistor *R2*, this combination behaves like one resistor with a value equal to

$$R = \frac{R1 * R2}{R1 + R2} = \frac{1}{\frac{1}{R1} + \frac{1}{R2}}$$

See Figure 3.7. This means we can replace the two parallel resistors in a circuit with one resistor at *R= R1*R2/(R1+R2)*. The voltages across *R1* and *R2* will be the same because of KVL. Due to KCL, *I=I1+I2*. These facts can be used to derive the **current divider rule**

$$I1 = V/R1 = (I*R)/R1 = I*(R1*R2/(R1+R2))/R1 = I*R2/(R1+R2)$$
$$I2 = V/R2 = (I*R)/R2 = I*(R1*R2/(R1+R2))/R2 = I*R1/(R1+R2)$$
$$I = I1+I2$$

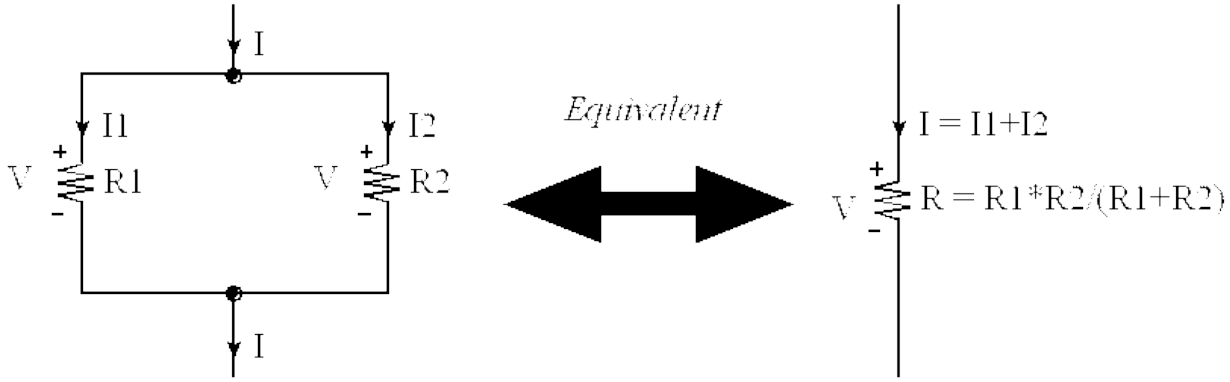


*Figure 3.7. The parallel combination of two resistors, R1 R2, is equivalent to one resistor at R1*R2/(R1+R2).*

Checkpoint 3.12: Using Figure 3.7, assume I is 1mA, R1 is 2k**Ω** and R2 is 4k**Ω**, what is V?
Checkpoint 3.13: Using Figure 3.7, assume V is 10V, R1 is 2k**Ω** and R2 is 4k**Ω**, what is I2?

3.2. Chapter 3 Quiz

3.1 Make this a matching definition with the word
a) Voltage ---------- an electrical potential.
b) Current ---------- the flow of charge (electrons)
c) Power ---------- the rate of energy change.
d) Energy ---------- defines the amount of work that can be done
e) Resistance ---------- potential divided by flow

3.2 Make this a matching definition with the word
a) Ohm's Law .---------- Voltage equals current times resistance
b) Kirchhoff's Current Law (KCL).---------- The sum of the currents into a node equal the sum of the currents leaving a node
c) Kirchhoff's Voltage Law (KVL). ---------- The sum of the voltages around the loop is zero.

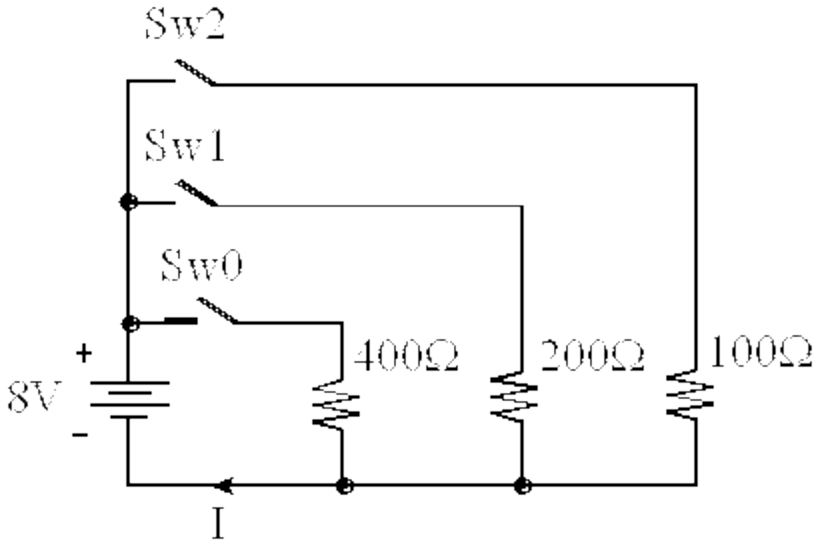
3.3 Know the formula (multiple choice for each)
a) Voltage current power .---------- P = V*I
b) Energy voltage current time.---------- E = V*I*time
c) Power voltage resistance. ---------- $P = V^2/R$.

3.4 Fill in this table with the equivalent resistance (all values are in ohms)

R1	R2	R1 in series with R2	R1 in parallel with R2
1000	4000		
1000	9000		
1000		5000	
1000			750
		4000	1000

3.5 2 V is applied across the parallel combination of a 1000Ω and a 4000Ω resistor. What is the voltage across the 4000Ω resistor? What is the current through the 4000Ω resistor?

3.6 Consider this 3-bit digital to analog converter. We define the logic state of each switch as 0 or 1, where 0 means not pushed and 1 means pushed. Define a 3-bit number *n* (0 to 7) which specifies the three switch positions. *n* = 0 means none are pushed. *n* = 1 means Sw0 is pushed. *n* = 2 means Sw1 is pushed. *n* = 3 means Sw1 and Sw0 are pushed. *n* = 4 means Sw2 is pushed. *n* = 5 means Sw2 and Sw0 are pushed. *n* = 6 means Sw2 and Sw1 are pushed. *n* = 7 means all are pushed. Derive a relationship between the current *I* and the number *n*. Multiple choice
a) $I = 0$
b) $I = n$*20mA (answer)
c) $I = n^2$+10mA
d) I = 140 mA
e) none of the above


Sw2
Sw1
Sw0
+
8V
-
400Ω
200Ω
100Ω
I

Chapter 4. Digital Logic
Embedded Systems - Shape The World
Jonathan Valvano and Ramesh Yerraballi

This chapter introduces digital logic. We will first define what it means to be digital, and then introduce logic, voltages, gates, flip flops, registers, adders and memory. This chapter is foundational, laying the ground work for the remainder of the class.

Learning Objectives:

- Understand N-channel and P-channel MOS transistors.
- Learn digital logic as implemented on a computer.
- Know how to build simple logic from transistors.
- Learn how to construct the basic components of a computer from the logic gates.
- Know the terms: flip flop, register, binary adder and memory.

4.1. Binary Information Implemented with MOS transistors

Information is stored on the computer in binary form. A binary **bit** can exist in one of two possible states. In **positive logic**, the presence of a voltage is called the '1', true, asserted, or high state. The absence of a voltage is called the '0', false, not asserted, or low state. Figure 4.1 shows the output of a typical complementary metal oxide semiconductor (CMOS) circuit. The left side shows the condition with a true bit, and the right side shows a false. The output of each digital circuit consists of a p-type transistor "on top of" an n-type transistor. In digital circuits, each transistor is essentially on or off. If the transistor is **on**, it is equivalent to a short circuit between its two output pins. Conversely, if the transistor is **off**, it is equivalent to an open circuit between its outputs pins.

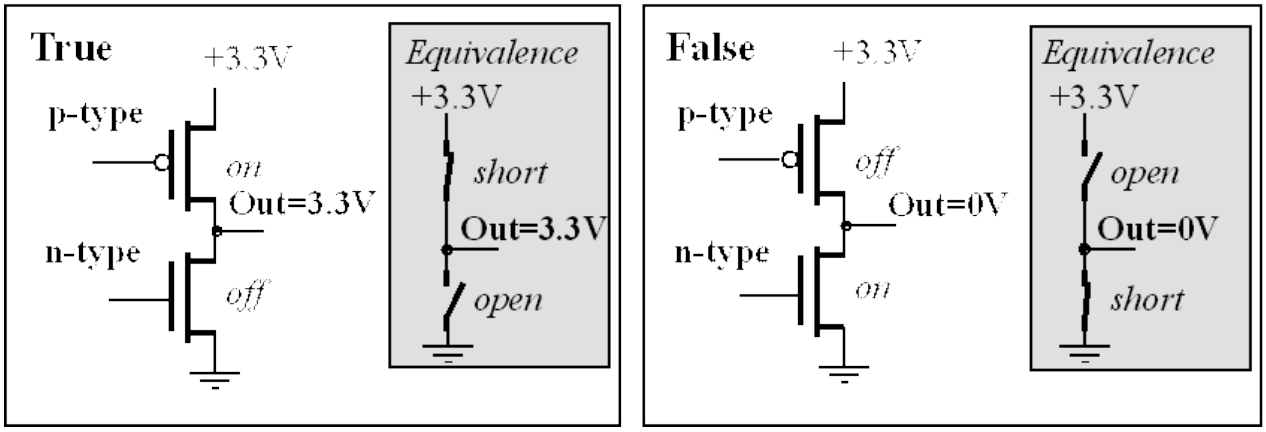


Figure 4.1. A binary bit is true if a voltage is present and false if the voltage is 0.

Every family of digital logic is a little different, but on a Stellaris® microcontroller powered with 3.3 V supply, a voltage between 2 and 5 V is considered high, and a voltage between 0 and 1.3 V is considered low, as drawn in Figure 4.2. Separating the two regions by 0.7 V allows digital logic to operate reliably at very high speeds. The design of transistor-level digital circuits is beyond the scope of this class. However, it is important to know that digital data exist as binary bits and encoded as high and low voltages.

Checkpoint 4.1. What does binary mean?

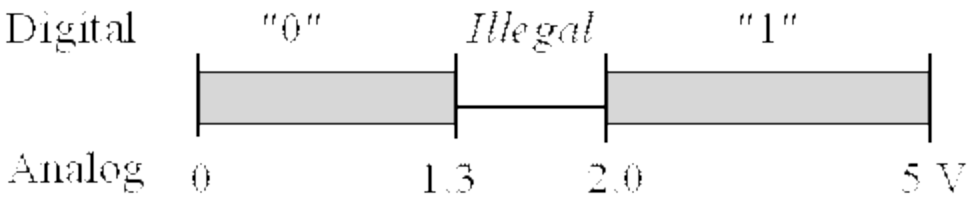


Figure 4.2. Mapping between analog voltage and the corresponding digital meaning on the TM4C123.

The maximum allowable voltage that the input will consider as low is called V_{IL}. For the TM4C123, V_{IL} is 1.3V. The minimum allowable voltage that the input will consider as high is called V_{IH}. For the TM4C123, V_{IH} is 2.0V.

The output pin also has a range of normal operating voltages. When the output is low, the maximum possible voltage that an output can be is called V_{OL}. For the TM4C123, V_{OL} is 0.4V. This means the output of the TM4C123 will be between 0 and 0.4V when the microcontroller is sending a low. When the output is high, the minimum possible voltage that an output can be is called V_{OH}. For the TM4C123, V_{OH} is 2.4V. This means the output of the TM4C123 will be between 2.4 and 3.3V when the microcontroller is sending a high. This information can be found in Section 24.2 of **TM4C123 data sheet**.

If the information we wish to store exists in more than two states, we use multiple bits. A collection of 2 bits has 4 possible states (00, 01, 10, and 11). A collection of 3 bits has 8 possible states (000, 001, 010, 011, 100, 101, 110, and 111). In general, a collection of *n* bits has 2^n states. For example, a **byte** contains eight bits, and is built by grouping eight binary bits into one object, as shown in Figure 4.3. Another name for a collection of eight bits is **octet** (*octo* is Latin and Greek meaning 8.) Information can take many forms, e.g., numbers, logical states, text, instructions, sounds, or images. What the bits mean depends on how the information is organized and more importantly how it is used. This figure shows one byte in the state representing the binary number 01100111. Again, the output voltage 3.3V means true or 1, and the output voltage of 0V means false or 0.

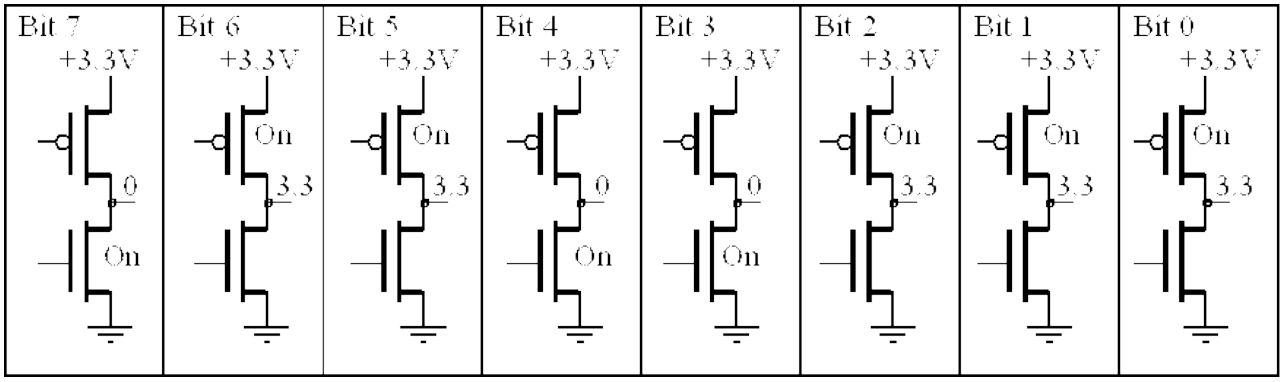


Figure 4.3. A byte is comprised of 8 bits, in this case representing the binary number 01100111.

What these 8 bits mean depends on how the computer software chooses to interpret them. Possibilities include but are not limited to an unsigned integer, a signed integer, a part of a machine code, and a character.

Checkpoint 4.2. Assume the circuit in Figure 4.3 contains an unsigned integer. What is the smallest unsigned integer that can be represented? What is the largest unsigned integer that can be represented?
Checkpoint 4.3. Assume the circuit in Figure 4.3 contains a signed 2's complement integer. What is the smallest unsigned integer that can be represented? What is the largest unsigned integer that can be represented?

Checkpoint 4.4. If the data stored in Figure 4.3 represent characters, how many characters could it represent?

4.2. Digital Logic

In this section, we will give just a little taste of how the computer digital logic in the computer works. Transistors made with metal oxide semiconductors are called MOS. In the digital world MOS transistors can be thought of as voltage controlled switches. Circuits made with both p-type and n-type MOS transistors are called complementary metal oxide semiconductors or CMOS. The 74HC04 is a high-speed CMOS NOT gate, as shown in Figure 4.4.

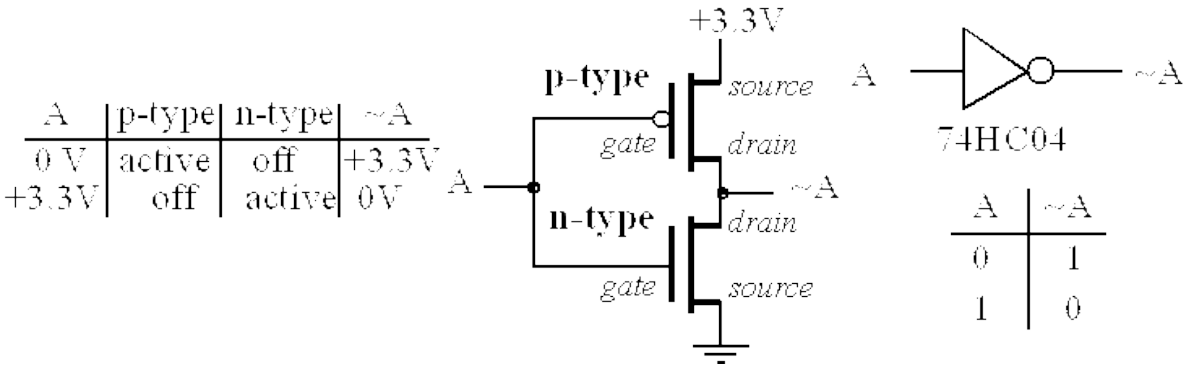


Figure 4.4. CMOS implementation of a NOT gate.

There are just a few rules one needs to know for understanding how CMOS transistor-level circuits work. Each transistor acts like a switch between its source and drain pins. In general, current can flow from source to drain across an active p-type transistor, and no current will flow if the switch is open. From a first approximation, we can assume no current flows into or out of the gate. For a p-type transistor, the switch will be closed (transistor active) if its gate is low. A p-type transistor will be off (its switch is open) if its gate is high.

The gate on the n-type works in a complementary fashion, hence the name complementary metal oxide semiconductor. For an n-type transistor, the switch will be closed (transistor active) if its gate is high. An n-type transistor will be off (its switch is open) if its gate is low. Therefore, consider the two possibilities for the circuit in Figure 4.4. If the input *A* is high (+3.3V), then the p-type transistor is off and the n-type transistor is active. The closed switch across the source-drain of the n-type transistor will make the output low (0V). Conversely, if *A* is low (0V), then p-type transistor is active and the n-type transistor is off. The closed switch across the source-drain of the p-type transistor will make the output high (+3.3V).

The AND, OR, EOR digital logic takes two inputs and produces one output; see Figure 4.5 and Table 4.1. We can understand the operation of the AND gate by observing the behavior of its six transistors. If both inputs *A* and *B* are high, both T3 and T4 will be active. Furthermore, if *A* and *B* are both high, T1 and T2 will be off. In this case, the signal labeled **~(A&B)** will be low because the T3–T4 switch combination will short this signal to ground. If *A* is low, T1 will be active and T3 off. Similarly, if *B* is low, T2 will be active and T4 off. Therefore if either *A* is low or if *B* is low, the signal labeled **~(A&B)** will be high because one or both of the T1, T2 switches will short this signal to +3.3V. Transistors T5 and T6 create a logical complement, converting the signal **~(A&B)** into the desired result of *A*&*B*. We can use the **and** operation to extract, or **mask**, individual bits from a value.

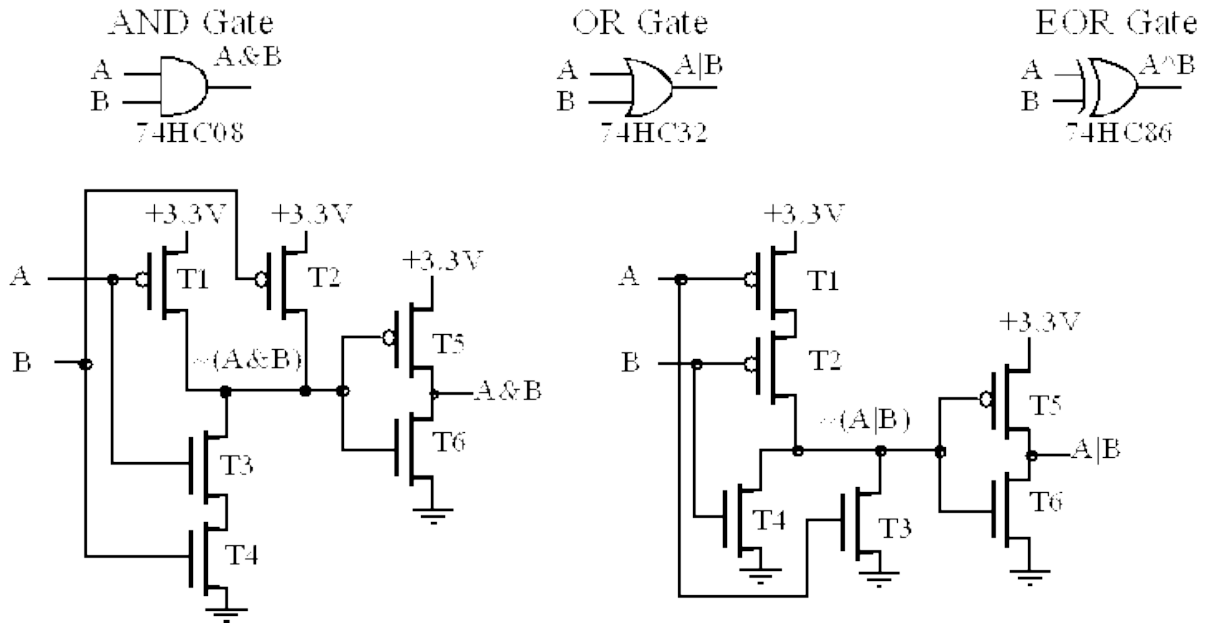


Figure 4.5. Logical operations can be implemented with discrete transistors or digital gates.

A	B	AND	NAND	OR	NOR	EOR	Ex NOR
0	0	0	1	0	1	0	1
0	1	0	1	1	0	1	0
1	0	0	1	1	0	1	0
1	1	1	0	1	0	0	1
Symbol		A&B	~(A&B)	A|B	~(A|B)	A^B	~(A^B)

Table 4.1. Two-input one-output logical operations.

We can understand the operation of the OR gate by observing the behavior of its six transistors. If both inputs *A* and *B* are low, both T1 and T2 will be active. Furthermore, if *A* and *B* are both low, T3 and T4 will be off. In this case, the signal labeled **~(A|B)** will be high because the T1–T2 switch combination will short this signal to +3.3V. If *A* is high, T3 will be active and T1 off. Similarly, if *B* is high, T4 will be active and T2 off. Therefore if either *A* is high or if *B* is high, the signal labeled **~(A|B)** will be low because one or both of the T3, T4 switches will short this signal to ground. Transistors T5 and T6 create a logical complement, converting the signal *~(A|B)* into the desired result of *A|B*. We use the OR operation to set individual bits.

When writing software we will have two kinds of logic operations. When operating on numbers (collection of bits) we will perform **logic** operations bit by bit. In other words, the operation is applied independently on each bit. In C, the logic operator for AND is **&**. For example, if number *A* is 01100111 and number *B* is 11110000 then

A = 01100111
B = 11110000
A&B 01100000

The other type of logic operation occurs when operating on **Boolean** values. In C, the condition false is represented by the value 0, and true is any nonzero value. In this case, if the Boolean *A* is 01100111 and *B* is 11110000 then both *A* and *B* are true. The standard value for true is the value 1. In C, the Boolean operator for AND is **&&**. Performing Boolean operation yields

A = 01100111
B = 11110000
A&&B 1

In C, the logic operator for OR is **|**. The logic operation is applied independently on each bit **.** E.g.,

```
   A =   01100111
   B =   11110000
A | B    11110111
```

In C, the Boolean operator for OR is **||**. Performing Boolean operation of **true** OR **true** yields **true**. Although 1 is the standard value for a true, any nonzero value is considered as true.

```
   A =   01100111
   B =   11110000
A || B          1
```

Other convenient logical operators are shown as digital gates in Figure 4.6. The **NAND** operation is defined by an AND followed by a NOT. If you compare the transistor-level circuits in Figures 4.5 and 4.6, it would be more precise to say AND is defined as a NAND followed by a NOT. Similarly, the OR operation is a **NOR** followed by a NOT. The **exclusive NOR** operation implements the bit-wise equals operation.

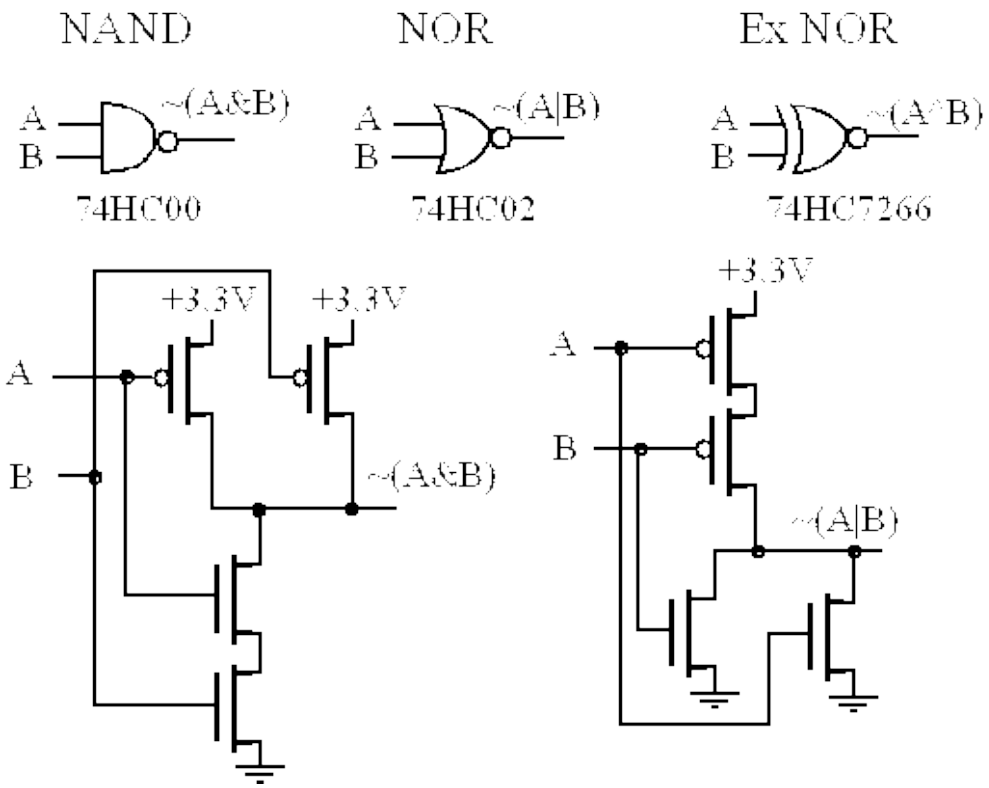


Figure 4.6. Other logical operations can also be implemented with MOS transistors.

Boolean Algebra is the mathematical framework for digital logic. Some fundamental laws of Boolean Algebra are listed in Table 4.2. With these laws, we consider A, B, C either as Booleans or as individual bits of a logic operation.

A & B = B & A	Commutative	Law
A | B = B | A	Commutative	Law
(A & B) & C = A & (B & C)	Associative	Law
(A | B) | C = A | (B | C)	Associative	Law
(A | B) & C = (A & C) | (B & C)	Distributive	Law
(A & B) | C = (A | C) & (B | C)	Distributive	Law

A & 0 = 0	Identity of 0
A | 0 = A	Identity of 0
A & 1 = A	Identity of 1
A | 1 = 1	Identity of 1
A | A = A	Property of OR
A | (~A) = 1	Property of OR
A & A = A	Property of AND
A & (~A) = 0	Property of AND
~(~A) = A	Inverse
~(A | B) = (~A) & (~B)	De Morgan's Theorem
~(A & B) = (~A) | (~B)	De Morgan's Theorem

Table 4.2. Fundamental laws of Boolean Algebra.

Checkpoint 4.5. Let A bit an 8-bit number, and consider the operation B=A&0x20, where A&0x20 is performed bit by bit. Now, if we consider B as a Boolean value, what is the relationship between A and B?
Checkpoint 4.6. Let C be an 8-bit number and consider the operation C=C&0xDF. How does this operation affect C?
Checkpoint 4.7. Let D bit an 8-bit number, and consider the operation D=D|0x20. How does this operation affect D?

When multiple operations occur in a single expression, **precedence** is used to determine the order of operation. Usually NOT is evaluated first, then AND, and then OR. This order can be altered using parentheses.

There are multiple ways to symbolically represent the digital logic functions. For example, ~*A* *A*' $\overline{A}$!*A* and ¬*A* are five ways to represent NOT(*A*). One can use the pipe symbol (|) or the plus sign to represent logical OR: *A*|*B* *A*+*B*. In this class we will not use the plus sign to represent OR to avoid confusion with arithmetic addition. One can use the ampersand symbol (&) or a multiplication sign (* • ×) to represent logical AND: *A*&*B* *A*•*B*. In this class we will not use the multiplication sign to represent AND to avoid confusion with arithmetic multiplication. Another symbolic rule is adding a special character (* n \) to a name to signify the signal is negative logic (0 means true and 1 means false). These symbols do not signify an operation, but rather are part of the name used to clarify its meaning. E.g., Enable* is a signal than means enable when the signal is zero.

Checkpoint 4.8. Let C bit an 8-bit number. Are these two operations the same or different? C=C&0xDF C=C&(~0x20)

4.3. Flip-flops are used for storage

While we're introducing digital circuits, we need digital storage devices, which are essential components used to make registers and memory. The simplest storage device is the **set-reset latch**. One way to build a set-reset latch is shown on the left side of Figure 4.7. If the inputs are *S**=0 and *R**=1, then the *Q* output will be 1. Conversely, if the inputs are *S**=1 and *R**=0, then the *Q* output will be 0. Normally, we leave both the *S** and *R** inputs high. We make the signal *S** go low, then back high to set the latch, making *Q*=1. Conversely, we make the signal *R** go low, then back high to reset the latch, making *Q*=0. If both *S** and *R** are 1, the value on *Q* will be remembered or stored. This latch enters an unpredictable mode when *S** and *R** are simultaneously low.

The **gated D latch** is also shown in Figure 4.7. The front-end circuits take a data input, *D*, and a control signal, *W*, and produce the *S** and *R** commands for the set-reset latch. For example, if *W*=0, then the latch is in its quiescent state, remembering the value on Q that was previously written. However, if *W*=1, then the data input is stored into the latch. In particular, if *D*=1 and *W*=1, then *S**=0 and *R**=1, making *Q*=1. Furthermore, if *D*=0 and *W*=1, then *S**=1 and *R**=0, making *Q*=0. So, to use the gated latch, we

first put the data on the *D* input, next we make *W* go high, and then we make *W* go low. This causes the data value to be stored at *Q*. After *W* goes low, the data does not need to exist at the *D* input anymore. If the *D* input changes while *W* is high, then the *Q* output will change correspondingly. However, the last value on the *D* input is remembered or latched when the *W* falls, as shown in Table 4.3.

The **D flip-flop**, shown on the right of Figure 4.7, can also be used to store information. D flip-flops are the basic building block of RAM and registers on the computer. To save information, we first place the digital value we wish to remember on the *D* input, and then give a rising edge to the **clock** input. After the rising edge of the **clock**, the value is available at the *Q* output, and the *D* input is free to change. The operation of the clocked D flip-flop is defined on the right side of Table 4.3. The 74HC374 is an 8-bit D flip-flop, such that all 8 bits are stored on the rising edge of a single clock. The 74HC374 is similar in structure and operation to a register, which is high-speed memory inside the processor. If the gate (*G*) input on the 74HC374 is high, its outputs will be HiZ (floating), and if the gate is low, the outputs will be high or low depending on the stored values on the flip-flop. The D flip-flops are edge-triggered, meaning that changes in the output occur at the rising edge of the input clock.

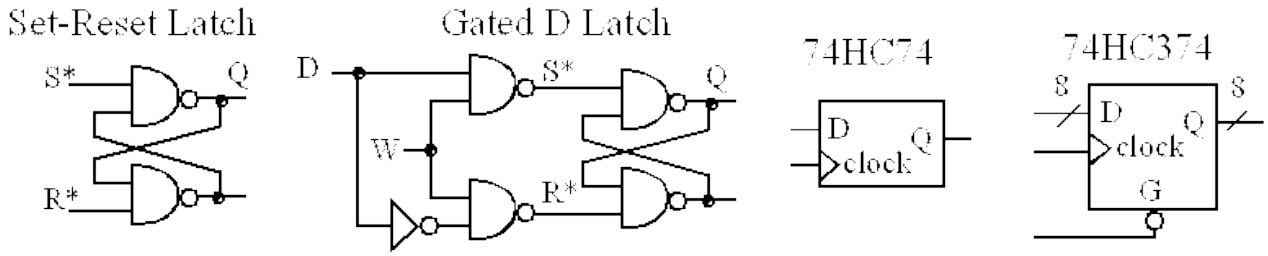


Figure 4.7. Digital storage elements.

D	W	Q
0	0	Q_{old}
1	0	Q_{old}
0	1	0
1	1	1
0	↓	0
1	↓	1

D	clock	Q
0	0	Q_{old}
0	1	Q_{old}
1	0	Q_{old}
1	1	Q_{old}
0	↑	0
1	↑	1

Table 4.3. D flip-flop operation. Q_{old} is the value of the D input at the time of fall of W or rise of clock.

The **tristate driver**, shown in Figure 4.8, can be used dynamically control signals within the computer. It is called tristate because there are three possible outputs: high, low, and HiZ. The tristate driver is an essential component from which computers are built. To activate the driver, we make its gate (*G**) low. When the driver is active, its output (*Y*) equals its input (*A*). To deactivate the driver, we make its *G** high. When the driver is not active, its output *Y* floats independent of *A*. We will also see this floating state with the open collector logic, and it is also called HiZ or high impedance. The HiZ output means the output is neither driven high nor low. The operation of a tristate driver is defined in Table 4.4. The 74HC244 is an 8-bit tristate driver, such that all 8 bits are active or not active controlled by a single gate. The 74HC374 8-bit D flip-flop includes tristate drivers on its outputs. Normally, we can't connect two digital outputs together. The tristate driver provides a way to connect multiple outputs to the same signal, as long as at most one of the gates is active at a time.

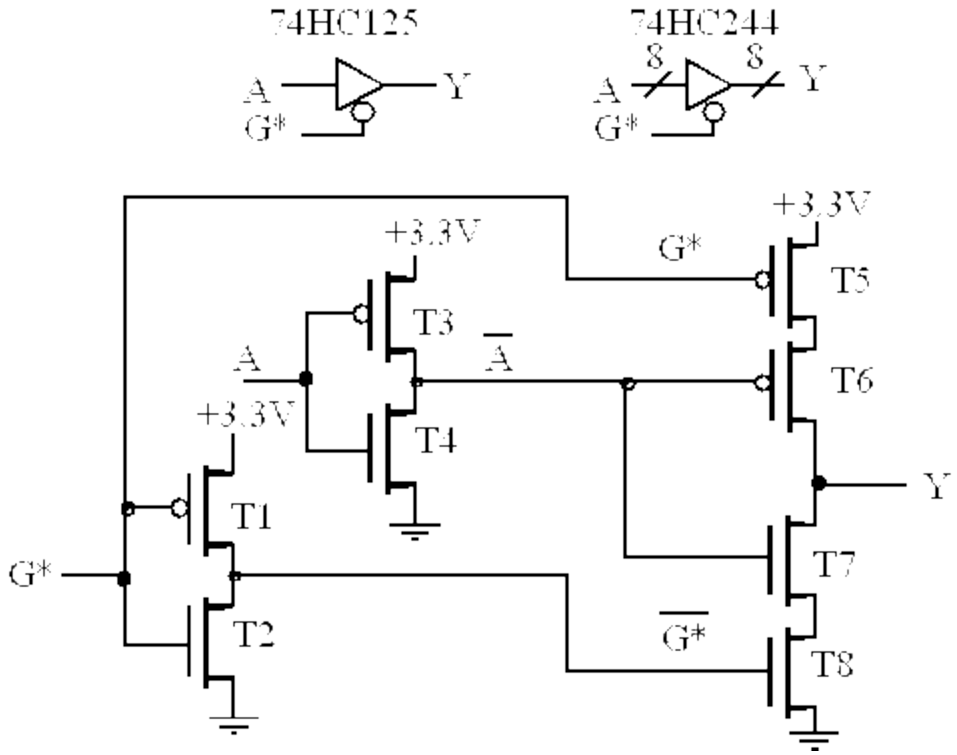


Figure 4.8. A 1-bit tristate driver and an 8-bit tristate driver (if G is low, then Y equals A, if G* is high, then Y is HiZ). The signal G* is negative logic.*

Table 4.4 describes how the tristate driver in Figure 4.8 works. Transistors T1 and T2 create the logical complement of *G**. Similarly, transistors T3 and T4 create the complement of *A*. An input of *G**=0 causes the driver to be active. In this case, both T5 and T8 will be on. With T5 and T8 on, the circuit behaves like a cascade of two NOT gates, so the output *Y* equals the input *A*. However, if the input *G**=1, both T5 and T8 will be off. Since T5 is in series with the +3.3V, and T8 in series with the ground, the output *Y* will be neither high nor low. I.e., it will float.

A	G*	T1	T2	T3	T4	T5	T6	T7	T8	Y
0	0	on	off	on	off	on	off	on	on	0
1	0	on	off	off	on	on	on	off	on	1
0	1	off	on	on	off	off	off	on	off	HiZ
1	1	off	on	off	on	off	on	off	off	HiZ

Table 4.4. Tristate driver operation. HiZ is the floating state, such that the output is not high or low.

The output of an open collector gate, drawn with the '×', has two states low (0V) and HiZ (floating) as shown in Figure 4.9. Consider the operation of the transistor-level circuit for the 74HC05. If *A* is high (+3.3V), the transistor is active, and the output is low (0V). If *A* is low (0V), the transistor is off, and the output is neither high nor low. In general, we can use an **open collector NOT** gate to switch current on and off to a device, such as a relay, an LED, a solenoid, or a small motor. The 74HC05, the 74LS05, the 7405, and the 7406 are all open collector NOT gates. 74HC04 is high-speed CMOS and can only sink up to 4 mA when its output is low. Since the 7405 and 7406 are transistor-transistor-logic (TTL) they can sink more current. In particular, the 7405 has a maximum output low current (I_{OL}) of 16 mA, whereas the 7406 has a maximum I_{OL} of 40 mA.

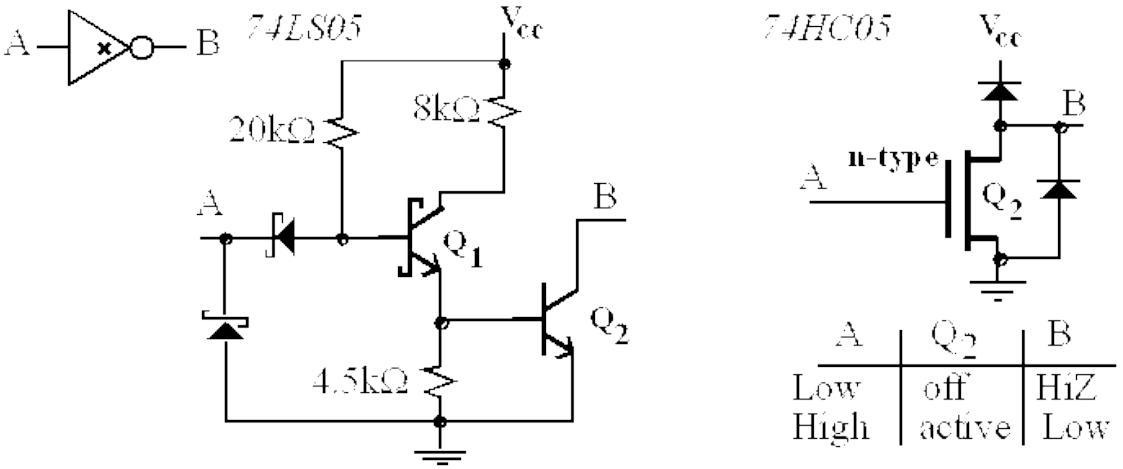


Figure 4.9. Two transistor implementations of an open collector NOT gate.

4.4. Binary Adder

The computer performs many arithmetic and logic operations. We will show one of them to illustrate some of the computation possible in the computer. We begin the design of an adder circuit with a simple subcircuit called a binary full adder, as shown in Figure 4.10. There are two binary data inputs *A*, *B*, and a carry input, C_{in}. There is one data output, S_{out}, and one carry output, C_{out}. As shown in Table 4.5, C_{in}, *A*, and *B* are three independent binary inputs each of which could be 0 or 1. These three inputs are added together (the sum could be 0, 1, 2, or 3), and the result is encoded in the two-bit binary result with C_{out} as the most significant bit and S_{out} as the least significant bit. C_{out} is true if the sum is 2 or 3, and S_{out} is true if the sum is 1 or 3.

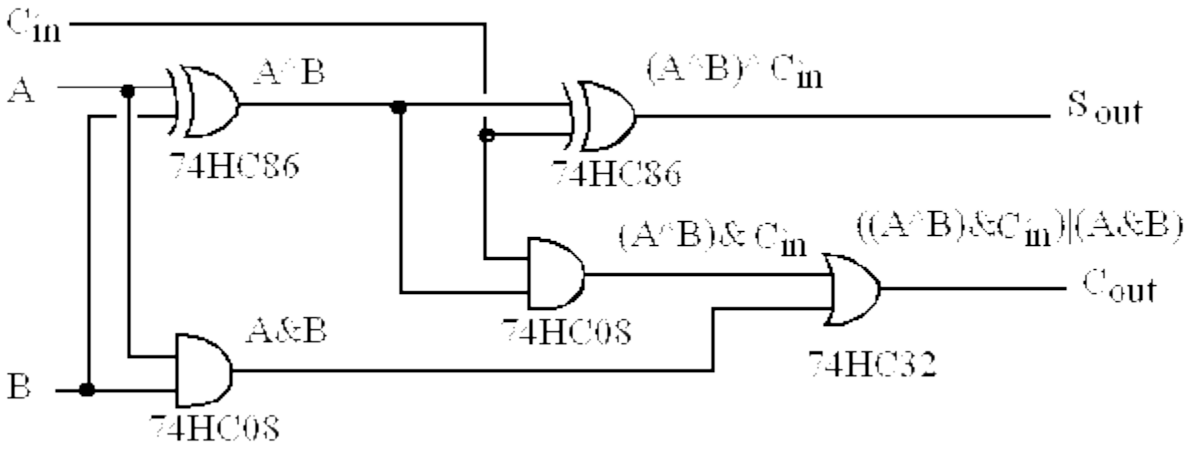


Figure 4.10. A binary full adder.

A	*B*	C_{in}	*A*+*B*+C_{in}	C_{out}	S_{out}
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	1	0	1
0	1	1	2	1	0
1	0	0	1	0	1
1	0	1	2	1	0
1	1	0	2	1	0
1	1	1	3	1	1

Table 4.5. Input/output response of a binary full adder.

Figure 4.11 shows an 8-bit adder formed by cascading eight binary full adders. Similarly, we build a 32-bit adder by cascading 32 binary full adders together. The carry into the 32-bit adder is zero, and the carry out will be saved in the carry bit.

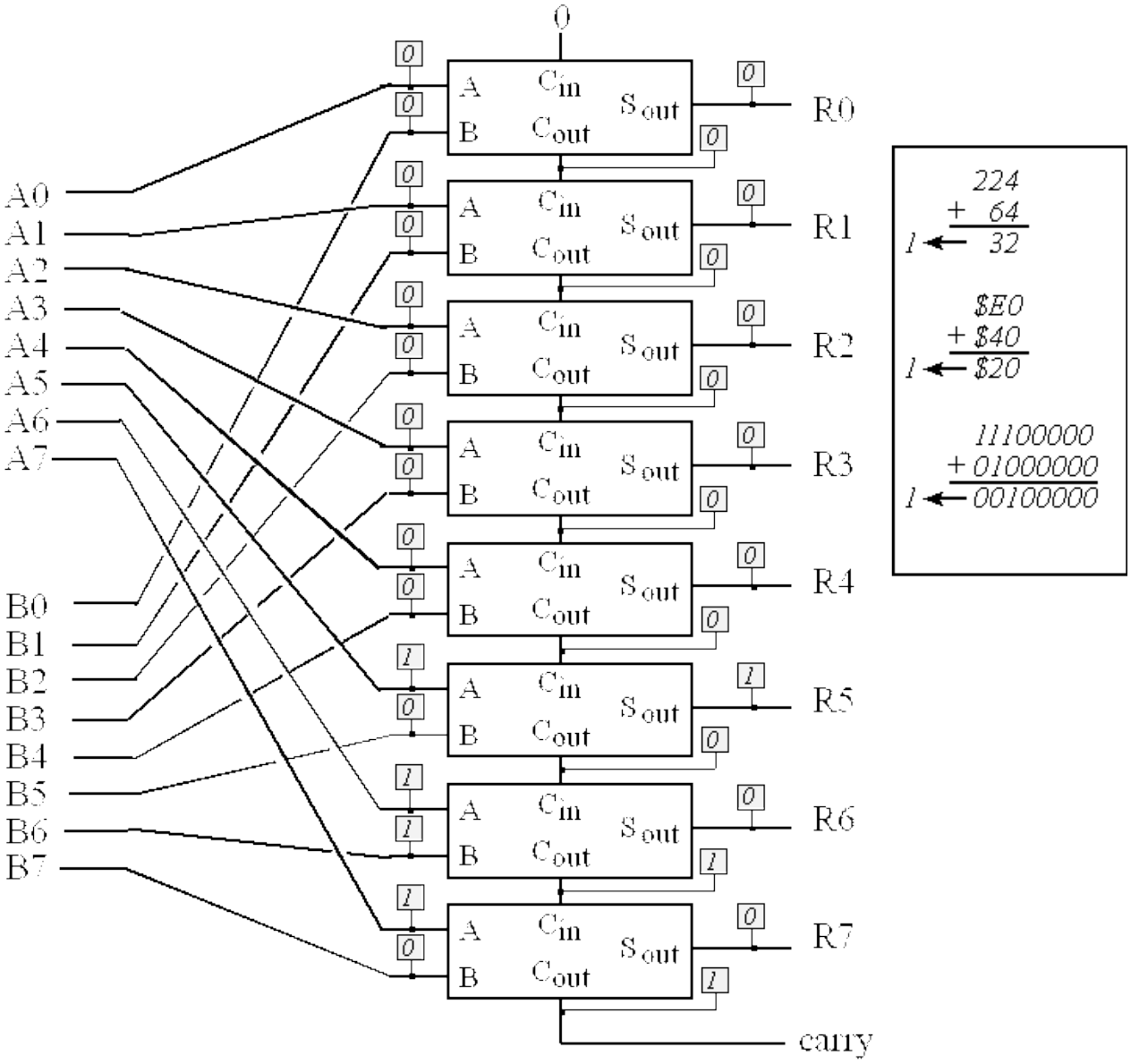


Figure 4.11. We make an 8-bit adder cascading eight binary full adders.

For an 8-bit unsigned number, there are only 256 possible values, which are 0 to 255. When we add two 8-bit numbers the sum can be any number from 0 to 510, which is a 9-bit number. The 9-bit result in Figure 4.11 exists as the 8 bits R7–R0 plus carry.

We can think of 8-bit unsigned numbers as positions along a circle, like a clock. There is a discontinuity in the clock at the 0|255 interface; everywhere else adjacent numbers differ by ±1. If we add two unsigned numbers, we start at the position of the first number a move in a clockwise direction the number of steps equal to the second number. If 96+64 is performed in 8-bit unsigned precision, the correct result of 160 is obtained. In this case, the carry bit will be 0 signifying the answer is correct. On the other hand, if 224+64 is performed in 8-bit unsigned precision, the incorrect result of 32 is obtained. In this case, the carry bit will be 1, signifying the answer is wrong.

Checkpoint 4.9. If A has the value 100 (0x64) and B has the value 50 (0x32), what will be the value of the output (R7-R0) of the circuit in Figure 4.11? Also what will the carry signal be?

Checkpoint 4.10. If A has the value 255 (0xFF) and B has the value 2 (0x02), what will be the value of the output (R7-R0) of the circuit in Figure 4.11? Also what will the carry signal be?

4.5. Digital Information stored in Memory

Memory is a collection of hardware elements in a computer into which we store information, as shown in Figure 4.12. For most computers in today's market, each memory cell contains one byte of information, and each byte has a unique and sequential address. The memory is called **byte-addressable** because each byte has a separate address. The address of a memory cell specifies its physical location, and its content is the data. When we **write** to memory, we specify an address and 8, 16, or 32 bits of data, causing that information to be stored into the memory. Typically data flows from processor into memory during a write cycle. When we **read** from memory we specify an address, causing 8, 16, or 32 bits of data to be retrieved from the memory. Typically data flows from memory into the processor during a read cycle. **Read Only Memory**, or ROM, is a type of memory where the information is programmed or burned into the device, and during normal operation it only allows read accesses. **Random Access Memory** (RAM) is used to store temporary information, and during normal operation we can read from or write data into RAM. The information in the ROM is **nonvolatile**, meaning the contents are not lost when power is removed. In contrast, the information in the RAM is **volatile**, meaning the contents are lost when power is removed. The system can quickly and conveniently read data from a ROM. It takes a comparatively long time to program or burn data into a ROM. Writing to Flash ROM is a two-step process. First, the ROM is erased, causing all the bits to become 1. Second, the system writes zeroes into the ROM as needed. Each of these two steps requires around 1 ms to complete. In contrast, it is fast and easy to both read data from and write data into a RAM. Writing to RAM is about 100,000 times faster (on the order of 10 ns). ROM on the other hand is much denser than RAM. This means we can pack more ROM bits into a chip than we can pack RAM bits. Most microcontrollers have much more ROM than RAM.

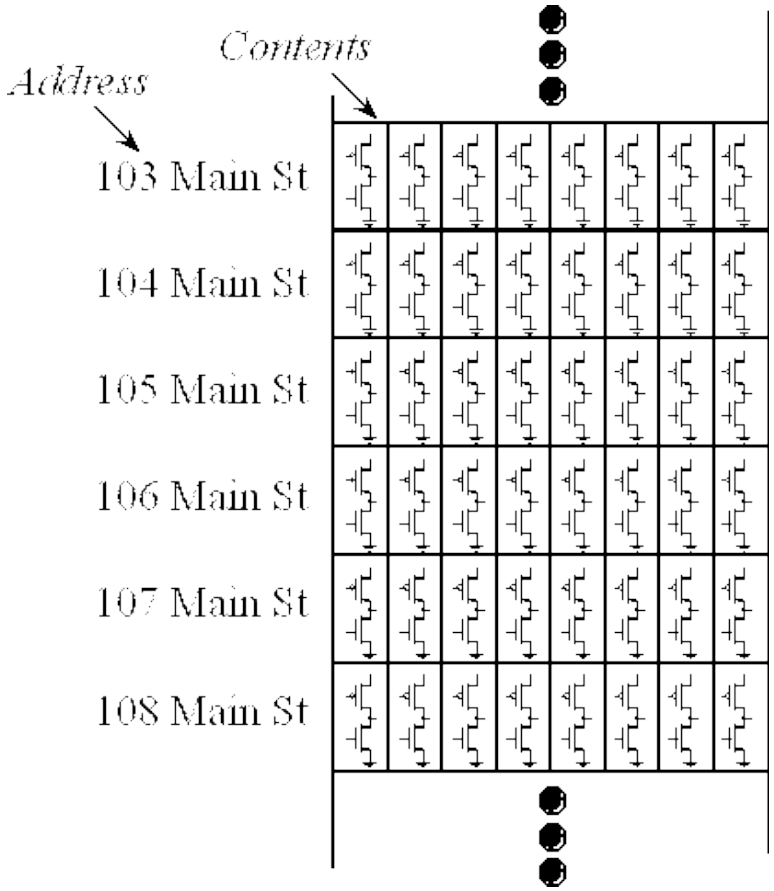


Figure 4.12. Memory is a sequential collection of data storage elements.

In the computer, we can build an 8-bit storage element, shown logically as Figure 4.12, by combining 8 flip-flops. This basic storage element is called a **register**, as shown in Figure 4.13. A **bus** is a collection of wires used to pass data from one place to another. In this circuit, the signals D7–D0 represent the data bus. Registers on the Stellaris® microcontrollers are 32-bits wide, but in this example we show an 8-bit register. We call it storage because as long the circuit remains powered, the digital information represented by the eight voltages Q7–Q0 will be remembered. There are two operations one performs on a register: write and read. To perform a write, one first puts the desired information on the 8 data bus wires (D7–D0). As you can see from Figure 4.13, these data bus signals are present on the D inputs of the 8 flip-flops. Next, the system pulses the **Write** signal high then low. This **Write** pulse will latch or store the desired data into the 8 flip-flops. The read operation will place a copy of the register information onto the data bus. Notice the gate signals of the tristate drivers are negative logic. This means if the **Read*** signal is high, the tristate drivers are off, and this register does not affect signals on the bus. However, the read operation occurs by setting the **Read*** signal low, which will place the register data onto the bus.

Checkpoint 4.11. What does negative logic mean?

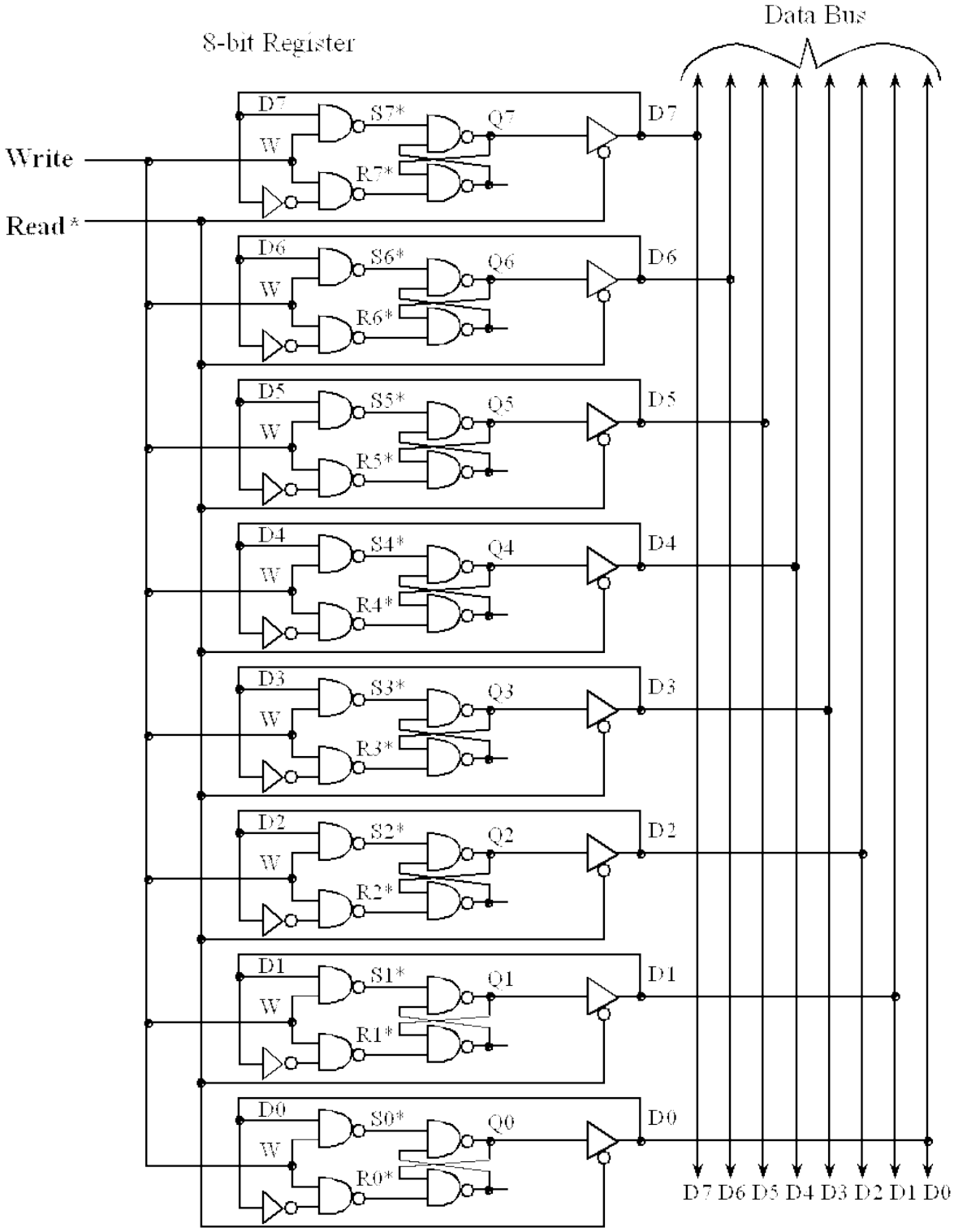


Figure 4.13. Digital logic implementation of a register.

A great deal of confusion exists over the abbreviations we use for large numbers. In 1998 the International Electrotechnical Commission (IEC) defined a new set of abbreviations for the powers of 2, as shown in Table 4.6. These new terms are endorsed by the Institute of Electrical and Electronics Engineers (IEEE) and International Committee for Weights and Measures (CIPM) in situations where the use of a binary prefix is appropriate. The confusion arises over the fact that the mainstream computer industry, such as Microsoft, Apple, and Dell, continues to use the old terminology. According to the companies that market to consumers, a 1 GHz is 1,000,000,000 Hz but 1 Gbyte of memory is 1,073,741,824 bytes. The correct terminology is to use the SI-decimal abbreviations to represent powers

of 10, and the IEC-binary abbreviations to represent powers of 2. The scientific meaning of 2 kilovolts is 2000 volts, but 2 kibibytes is the proper way to specify 2048 bytes. The term **kibibyte** is a contraction of kilo binary byte and is a unit of information or computer storage, abbreviated KiB.

1 KiB = 2^{10} bytes = 1024 bytes
1 MiB = 2^{20} bytes = 1,048,576 bytes
1 GiB = 2^{30} bytes = 1,073,741,824 bytes

These abbreviations can also be used to specify the number of binary bits. The term **kibibit** is a contraction of kilo binary bit, and is a unit of information or computer storage, abbreviated Kibit.

A **mebibyte** (1 MiB is 1,048,576 bytes) is approximately equal to a megabyte (1 MB is 1,000,000 bytes), but mistaking the two has nonetheless led to confusion and even legal disputes. In the engineering community, it is appropriate to use terms that have a clear and unambiguous meaning.

Value	*SI*	*Decimal*	*SI*	*Decimal*
1000^1	k		kilo-	
1000^2	M		mega-	
1000^3	G		giga-	
1000^4	T		tera-	
1000^5	P		peta-	
1000^6	E		exa-	
1000^7	Z		zetta-	
1000^8	Y		yotta-	

Value	*IEC*	*Binary*	*IEC*	*Binary*
1024^1	Ki		kibi-	
1024^2	Mi		mebi-	
1024^3	Gi		gibi-	
1024^4	Ti		tebi-	
1024^5	Pi		pebi-	
1024^6	Ei		exbi-	
1024^7	Zi		zebi-	
1024^8	Yi		yobi-	

Table 4.6. Common abbreviations for large numbers.

4.6. Chapter 4 Quiz

4.1 Which of the following is true. Positive logic is defined as
a) The configuration where the "true" state has a higher voltage than the "false" state.
b) The state where the signal is "high".
c) The state where the transistor is "on".
d) The state where the transistor is "off".
e) None of the above

4.2 If a voltage on an input of the TM4C123 is between 0 and 1.3 V, how is that input considered?
a) Low or logic "0"
b) Unknown or illegal
c) High or logic "1"

4.3 If a voltage on an input of the TM4C123 is between 1.3 and 2 V, how is that input considered?
a) Low or logic "0"
b) Unknown or illegal
c) High or logic "1"

4.4 If a voltage on an input of the TM4C123 is between 2 and 5 V, how is that input considered?
a) Low or logic "0"
b) Unknown or illegal
c) High or logic "1"

4.5 Calculate the logic expression for each set of inputs A, B, C

A	B	C	A & (B | C)
$0100_2 = 4$	$0101_2 = 5$	$0110_2 = 6$	
$0111_2 = 7$	$1010_2 = 10$	$0001_2 = 1$	
$1110_2 = 14$	$1001_2 = 9$	$0111_2 = 7$	

4.6 Calculate the Boolean expression for each set of inputs A, B, C

A	B	C	A&& (B || C)
True	False	False	
True	False	True	
False	True	False	
True	True	False	

Chapter 5: Introduction to C Programming
Embedded Systems - Shape the World
Jonathan Valvano and Ramesh Yerraballi

This chapter covers the C Programming language starting with the structure, constants and variable declarations, the main subroutine, simple input/output, arithmetic expressions, Boolean expressions, the assignment statement, the while loop and lastly simple functions with at most one input and one output.

Learning Objectives:

- Know the elements of a C program: What goes where, what are the syntax rules
- Know declarations: simple data types, **`char, short, long`** (**`unsigned`** and **`signed`**)
- Know the form of the mandatory main subroutine for a program to be executable
- Know the basic assignment statement: **`variable = expression;`**
- Know how to use **`printf`** and **`scanf`** for I/O.
- Know the basic form of the **`if`**-statement and the **`while(1)`** statement and how they use the conditional Boolean expression.
- Know the importance of functions and understand the use with inputs and outputs.

5.0 Introduction

This course presents the art and science of designing embedded systems. In this module we will introduce C programming. If you need to write a paper, you decide on a theme, and then begin with an outline. In the same manner, if you design an embedded system, you define its specification (what it does) and begin with an organizational plan. In this chapter, we will present three graphical tools to describe the organization of an embedded system: flowcharts, data flow graphs, and call graphs. You should draw all three for every system you design. In this section, we introduce the flowchart syntax that will be used throughout the class. Programs themselves are written in a linear or one-dimensional fashion. In other words, we type one line of software after another in a sequential 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. Flowcharts are very useful in the initial design stage of a software system to define complex algorithms. Furthermore, flowcharts can be used in the final documentation stage of a project, once the system is operational, in order to assist in its use or modification.

Where does one begin when learning a new skill? To me, I begin software development with the question, "What is it the program is supposed to do?" Next, I think of how I will test it. Testing is essentially a mechanism to see if the program does exactly what it is supposed to do, no more no less. Next, I list what are the inputs, and what are the outputs. Inside the system we have data. The data defines what do I know, so I define the data and give some examples. The software algorithm connects the inputs to the data, and software must connect the data to the outputs. Lastly, I test it. So you see I begin with testing and end with testing.

We will use flowcharts to illustrate what the software does (Figure 5.1). 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 **conditional** block. Inside the diamond we can define what is being tested. Each arrow out of a condition block must be labeled with the condition causing flow to go in that direction. There must be at least two arrows out of a condition block, but there could be more than two. However, the condition for each arrow must be mutually exclusive (you can't say "if I'm happy go left and if I'm tall go right" because it is unclear what you want the software to do if I'm happy and tall). Furthermore, the complete set of conditions must define all possibilities (you can't say "if temperature is less than 20 go right and if the temperature is above 40 go left" because you have not defined what to do if the temperature is between 20 and 40). The rectangle with double lines on the side specifies 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 are 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**. A connector with an arrow pointing out of the circle defines a label or a spot in the algorithm. There should be one label connector for each number. Connectors with an arrow pointing into the circle are jumps or goto commands. When the flow reaches a goto connector, the execution path jumps to the position specified by the corresponding label connector. It is bad style to use a lot of connectors.

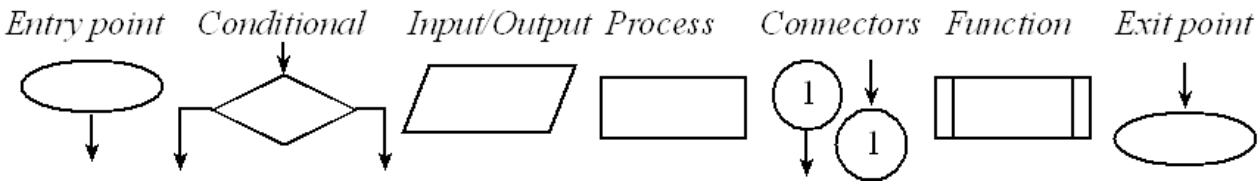


Figure 5.1. Flowchart symbols.

There are a seemingly unlimited number of tasks 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 fundamentals when developing software, regardless whether it is a microcontroller with 1000 lines of assembly code or a large computer system with billions of lines of code, is to maintain a consistent structure. One such framework is called structured programming. C is a structured language, which means we begin with a small number of simple templates, as shown in Figure 5.2. A good high-level language will force the programmer to write structured programs. Structured programs in C are built from three basic templates: the **sequence**, the **conditional**, and the **while-loop**. At the lowest level, the "block" contains simple and well-defined commands, like `Area = Height*Width;` I/O functions are also low-level building blocks. To program in C, we combine existing structures into more complex structures. Each of the "blocks" in Figure 5.2 is either a simple well-defined command or another structure.

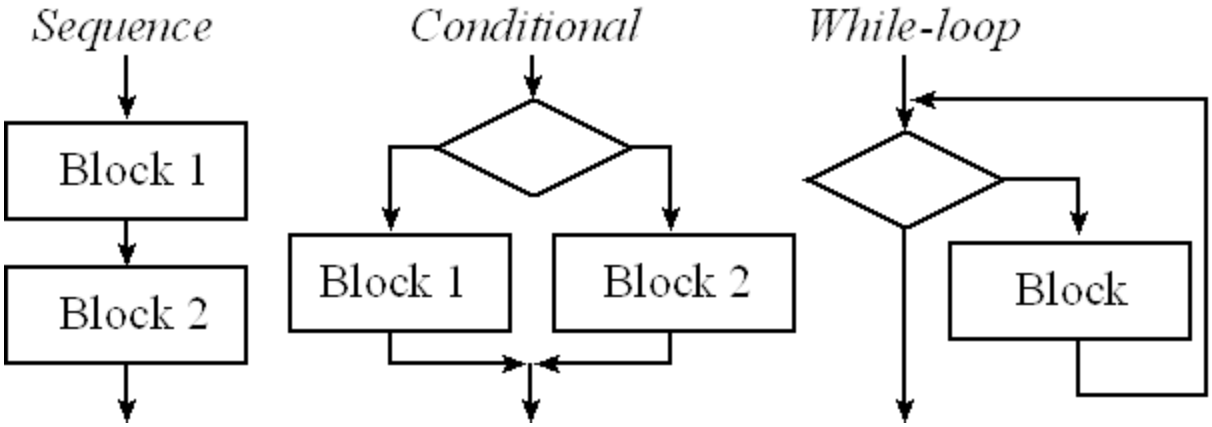


Figure 5.2. Flowchart showing the basic building blocks of structured programming.

Example 5.1: Using a flowchart describe the control algorithm that a toaster might use to cook toast. There will be a start button the user pushes to activate the machine. There is other input that measures toast temperature. The desired temperature is preprogrammed into the machine. The output is a heater, which can be on or off. The toast is automatically lowered into the oven when heat is applied and is ejected when the heat is turned off.

Solution: This example illustrates a common trait of an embedded system, that is, they perform the same set of tasks over and over forever. The program starts at **main** when power is applied, and the system behaves like a toaster until it is unplugged. Figure 5.3 shows a flowchart for one possible toaster algorithm. The system initially waits for the operator to push the start button. If the switch is not pressed, the system loops back reading and checking the switch over and over. After the start button is pressed, heat is turned on. When the toast temperature reaches the desired value, heat is turned off, and the process is repeated.

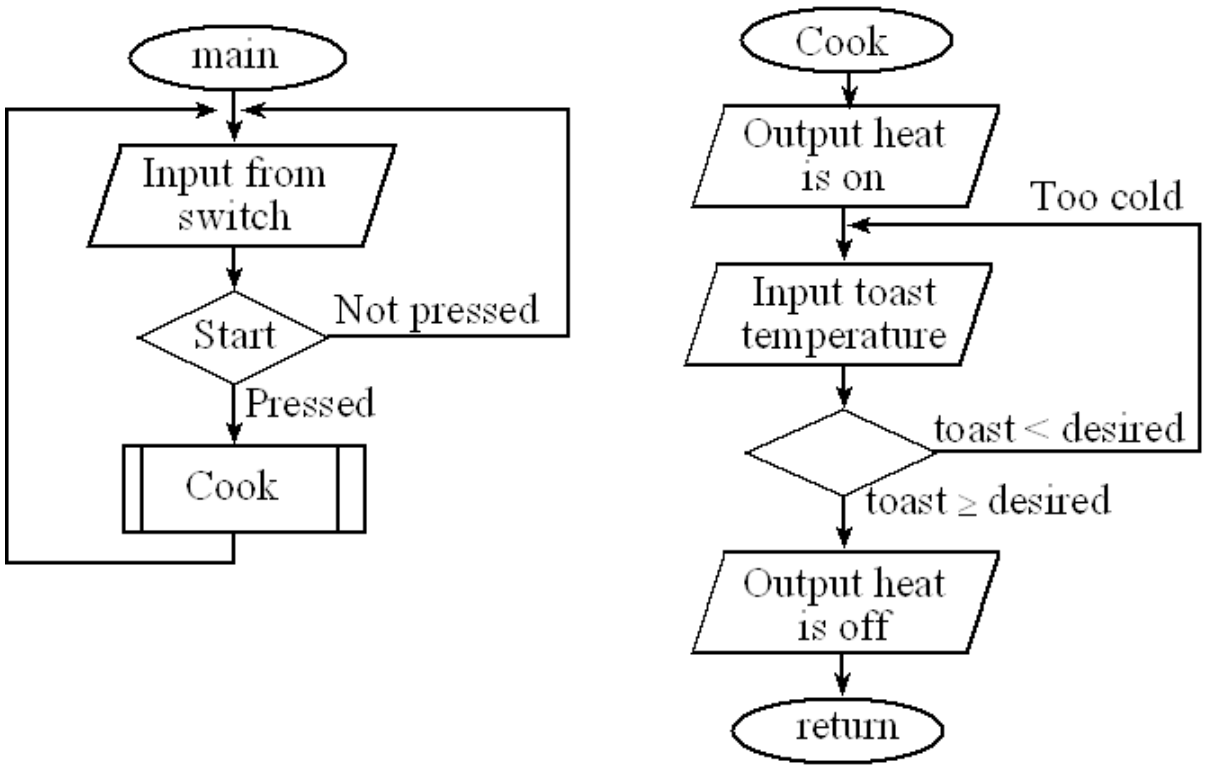


Figure 5.3. Flowchart illustrating the process of making toast.

Safety tip: When dealing with the potential for fire, you may want to add some safety features such as a time out or an independent check for temperature overflow.

Observation: The predefined functions in this chapter do not communicate any data between the calling routine and function. Data passed into a function are called input parameters, and data passed from the function back to the calling routine are called output parameters.
Observation: Notice in Figure 5.3 we defined a function Cook even though it was called from only one place. You might be tempted to think it would have been better to paste the code for the function into the one place it was called. There are many reasons it would be better to define the function as a separate software object: it will be easier to debug because there is a clear beginning and end of the function, it will make the overall system simpler to understand, and in the future we may wish to reuse this function for another purpose.

Example 5.2. The system has one input and one output. An event should be recognized when the input goes from 0 to 1 and back to 0 again. The output is initially 0, but should go 1 after four events are detected. After this point, the output should remain 1. Design a flowchart to solve this problem.

Solution: This example also illustrates the concept of a subroutine. We break a complex system into smaller components so that the system is easier to understand and easier to test. In particular, once we know how to detect an event, we will encapsulate that process into a subroutine, called **Event**. In this example, the **main** program first sets the output to zero, calls the function **Event** four times, then it sets the output to one. To detect the 0 to 1 to 0 edges in the input, it first waits for 1, and then it waits for 0 again. The letters **A** through **H** in Figure 5.4 specify the software activities in this simple example. In this example, execution is sequential and predictable.

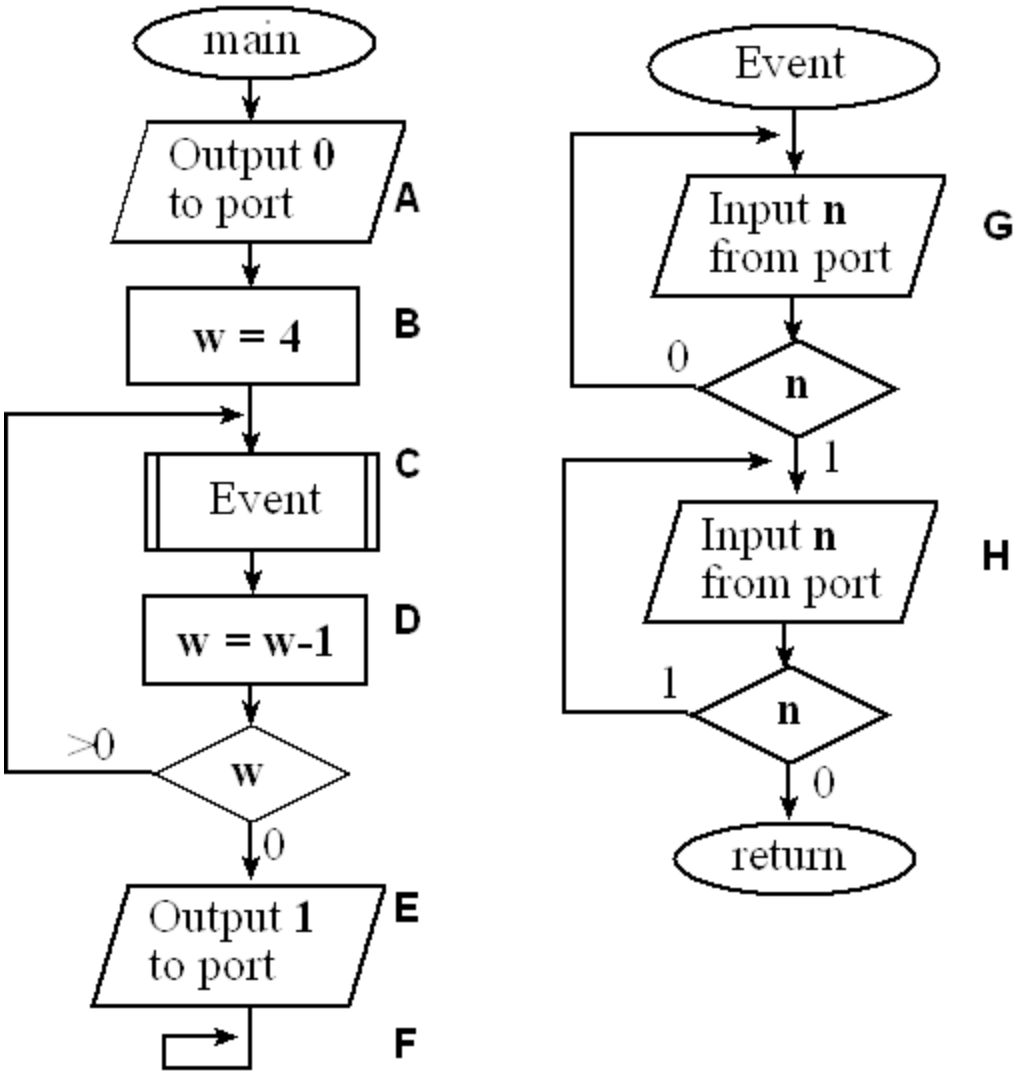
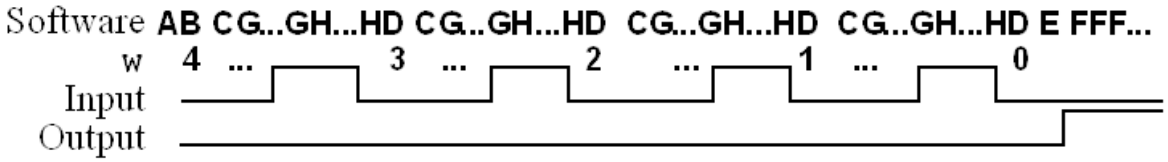


Figure 5.4. Flowchart illustrating the process waiting for four events.



Before we write software, we need to develop a plan. Software development is an iterative process. Even though we list steps the development process in a 1,2,3,4 order, in reality we cycle through these steps over and over. I like to begin with step 4), deciding how I will test it even before I decide what it does.

1) We begin with a list of the inputs and outputs. This usually defines what the overall system will do. We specify the range of values and their significance.
2) Next, we make a list of the required data. We must decide how the data is structured, what does it mean, how it is collected, and how it can be changed.
3) Next we develop the software algorithm, which is a sequence of operations we wish to execute. There are many approaches to describing the plan. Experienced programmers can develop the algorithm directly in C language. On the other hand, most of us need an abstractive method to document the desired sequence of actions. Flowcharts and pseudo code are two common descriptive formats. There are no formal rules regarding pseudo code, rather it is a shorthand for describing what to do and when to

do it. We can place our pseudo code as documentation into the comment fields of our program. Next we write software to implement the algorithm as define in the flowchart and pseudo code.
4) The last stage is debugging. Learning debugging skills will greatly improve the quality of your software and the efficiency at which you can develop code.

5.1 Background

We will use C in this class for two reasons. First, over the last ten years, it has ranked one or two out of all high-level languages. Second, C is by far the most common language for writing software for embedded systems.

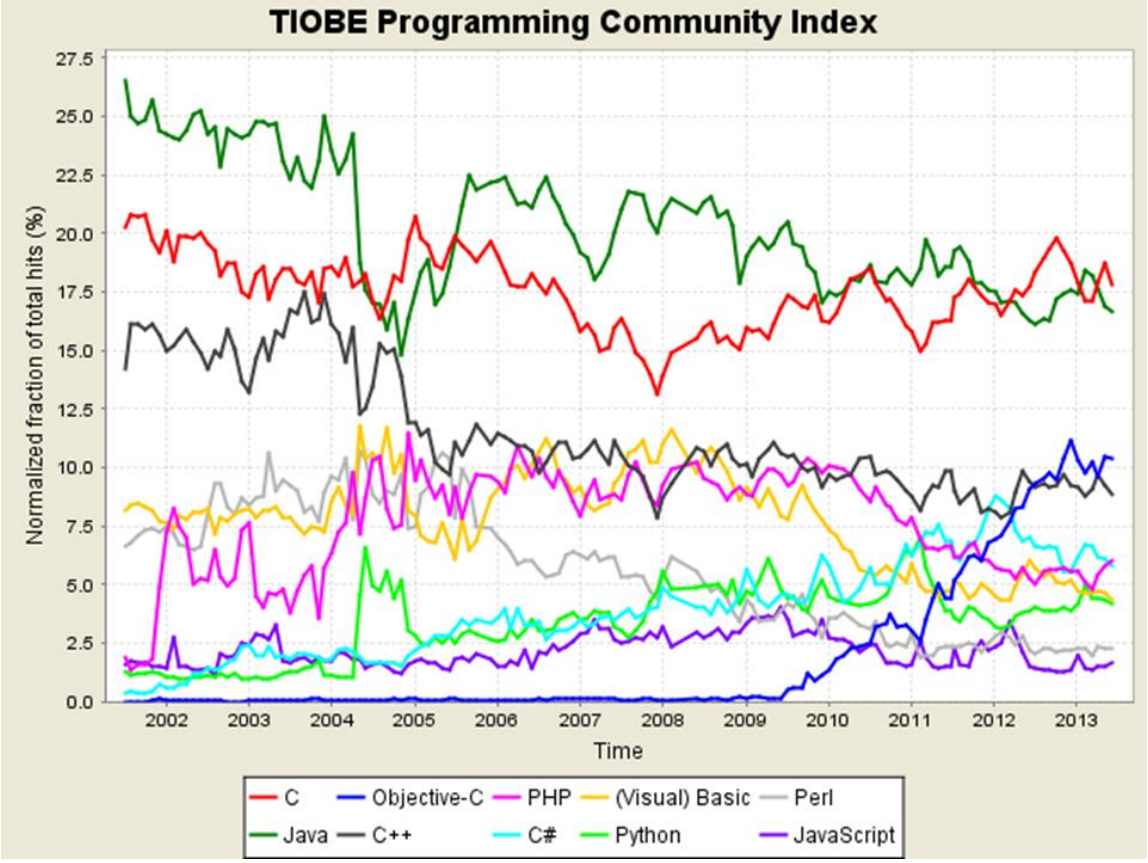


Figure 5.5. Graph of popular programming languages over time.

Position Jun 2013	Position Jun 2012	Programming Language	Ratings Jun 2013	Delta Jun 2012
1	1	C	17.809%	+0.08%
2	2	Java	16.656%	+0.39%
3	4	Objective-C	10.356%	+1.26%
4	3	C++	8.819%	-0.54%
5	7	PHP	5.987%	+0.70%
6	5	C#	5.783%	-1.24%
7	6	(Visual) Basic	4.348%	-1.70%
8	8	Python	4.183%	+0.33%

9	9	Perl	2.273%	+0.05%
10	11	JavaScript	1.654%	+0.18%
11	10	Ruby	1.479%	-0.20%
12	12	Visual Basic .NET	1.067%	-0.15%
13	17	Transact-SQL	0.913%	+0.21%
14	14	Lisp	0.879%	-0.11%
15	16	Pascal	0.779%	-0.07%
16	21	Bash	0.711%	+0.09%
17	19	PL/SQL	0.657%	+0.02%
18	13	Delphi/Object Pascal	0.602%	-0.55%
19	18	Ada	0.575%	-0.11%
20	22	MATLAB	0.563%	0.00%

Table 5.1. Top 20 popular programming languages.

C is a general-purpose programming language initially developed by Dennis Ritchie between 1969 and 1973 while at AT&T Bell Labs, see Figure 5.6. At the time, there were programming languages called A and another named B, so Ritchie decided to name his language C. Dennis Ritchie and Brian Kernighan wrote the first book on C, The C Programming Language. Ritchie was also one of the developers of the Unix operating system.

Figure 5.6. Dennis MacAlistair Ritchie

As C became more popular, many derivative languages were introduced. C++ was developed by Bjarne Stroustrup 1979-1983 also at Bell Labs. C++ is a language originally called “C plus classes”. In 1999, a professional standard version of C, called C99, was defined. When you download Tivaware (http://www.ti.com/tool/sw-tm4c) from Texas Instruments, you will notice TI’s example code for the TM4C123 has been written in C99. In this class however, we will use the more simple C language.

A **compiler** is system software that converts a high-level language program (human readable format) into object code (machine readable format). It produces software that is fast but to change the software we need to edit the source code and recompile.
C code (`z = x+y;`) → Assembly code (`ADD R2,R1,R0`) → Machine code (`0xEB010200`)

An **assembler** is system software that converts an assembly language program (human readable format) into object code (machine readable format).
Assembly code (`ADD R2,R1,R0`) → Machine code (`0xEB010200`)

An **interpreter** executes directly the high level language. It is interactive but runs slower than compiled code. Many languages can be compiled or interpreted. The original BASIC (Beginner's All-purpose Symbolic Instruction Code) was interpreted. This means the user typed software to the computer, and the interpreter executed the commands as they were typed. In this class, an example of the interpreter will be the command window while running the debugger. For more information on this interpreter, run Keil uVision and execute **Help->uVisionHelp**. Next, you need to click the **Contents** tab, open the **uVisionIDEusersGuide**, and then click **DebugCommands**. It will show you a list of debugger commands you can type into the command window.

A **linker** builds software system by connecting (linking) software components. In Keil uVision, the build command (**Project->BuildTarget**) performs both a compilation and a linking. The example code in this module has three software components that are linked together. These components are

```
startup.s
uart.c
main.c
```

A **loader** will place the object code in memory. In an embedded system, the loader will program object code into flash ROM. In Keil uVision, the download command (**Flash->Download**) performs a load operation.

A **debugger** is a set of hardware and software tools we use to verify system is operating correctly. The two important aspects of a good debugger are control and observability.

5.2 Structure and Organization of C

In assembly language, symbols placed at the beginning of each line have special meaning. On the contrary, C is a **free field language**. Except for preprocessor lines that begin with #, spaces, tabs and line breaks have the same meaning. This means we can place more than one statement on a single line, or place a single statement across multiple lines. We could write a function without any line breaks. Since we rarely make hardcopy printouts of our software, it is not necessary to minimize the number of line breaks. Furthermore, we could have added extra line breaks. I prefer the style of the program on the right because each line contains one complete thought or action. As you get more experienced, you will develop a programming style that is easy to understand. Although spaces, tabs, and line breaks are syntactically equivalent, their proper usage will have a profound impact on the readability of your software. The following three functions are identical; I like the third one.

```
unsigned long M=1;
unsigned long Random(void){M=1664525*M+1013904223;return(M);}
```

```
unsigned long M=1;
unsigned long
Random(void){
  M = 1664525*M
       +1013904223;
 return(M);
}
```

```
unsigned long M=1;
unsigned long Random(void){
  M = 1664525*M+1013904223;
  return(M);
}
```

Program 5.1. Three equivalent functions that return a random number.

Another situation where spaces, tabs and line breaks matter is string constants. We cannot type tabs or line breaks within a string constant. The characters between the first " and second " define the string constant. A string is a set of ASCII characters terminated with a 0. For example, the following C code will output my name:

```
printf("Jonathan Valvano");
```

The variable **M**, the function **Random**, the operation *****, and the keyword **long** are **tokens** in C Each token must be contained on a single line. We see in the above example that tokens can be separated by white spaces, which include space, tab, line break, or by special characters. Special characters include punctuation marks (Table 5.2) and operations (Table 5.4).

Punctuation	*Meaning*
;	End of statement
:	Defines a label
,	Separates elements of a list
( )	Start and end of a parameter list
{ }	Start and stop of a compound statement
[ ]	Start and stop of a array index
" "	Start and stop of a string
' '	Start and stop of a character constant

Table 5.2. Special characters can be punctuation marks.

Punctuation marks (semicolons, colons, commas, apostrophes, quotation marks, braces, brackets, and parentheses) are very important in C. It is one of the most frequent sources of errors for both the beginning and experienced programmers.

There are four sections of a C program as shown in Program 5.2. The first section is the documentation section, which includes the purpose of the software, the authors, the date, and any copyright information. When the software involves external hardware we will add information about how the external hardware is connected. The second section is the preprocessor directives. We will use the preprocessor directive #include to connect this software with other modules. We use diamond braces to include system libraries, like the standard I/O, and we use quotes to link up with other user code within the project. In this case the **uart** module is software we wrote to perform I/O with the universal asynchronous receiver/transmitter (**uart**). We will discuss modular programming in great detail in this class. The third section is global declarations section. This section will include global variables and function prototypes for functions defined in this module. The last section will be the functions themselves. In this class we will use the terms subroutine, procedure, function, and program interchangeably. Every software system in C has exactly one main program, which define where it begins execution.

```
//**** 0. Documentation Section
//  This program calculates the area of square shaped rooms
//  Author: Ramesh Yerraballi & Jon Valvano
//  Date: 6/28/2013
//
// 1. Pre-processor Directives Section
#include <stdio.h>  // Diamond braces for sys lib: Standard I/O
#include "uart.h"   // Quotes for user lib: UART lib
```

```
// 2. Global Declarations section
// 3. Subroutines Section
// MAIN: Mandatory routine for a C program to be executable
int main(void) {
  UART_Init();  // call subroutine to initialize the uart
  printf("This program calculates areas of square-shaped rooms\n");
}
```

Program 5.2. Software to calculate the area of a square room.

There are two types of comments. The first type explains how to use the software. These comments are usually placed at the top of the file, within the header file, or at the start of a function. The reader of these comments will be writing software that uses or calls these routines. The second type of comments assists a future programmer (ourselves included) in changing, debugging or extending these routines. We usually place these comments within the body of the functions. The comments on the right of each line are examples of the second type.

Preprocessor directives begin with # in the first column. As the name implies preprocessor commands are processed first. I.e., the compiler passes through the program handling the preprocessor directives. Although there are many possibilities (assembly language, conditional compilation, interrupt service routines), I thought I'd mention the two most important ones early in the class. We create a macro using **`#define`** to define constants.

```
#define SIZE 10
```

Basically, wherever **`SIZE`** is found as a token, it is replaced with the **`10`**. A second important directive is the **`#include`**, which allows you to include another entire file at that position within the program. The **`#include`** directive will include the file named **tm4c123ge6pm.h** at this point in the program. This file will define all the I/O port names for the TM4C123.

```
#include "tm4c123ge6pm.h"
```

5.3 Variables and Expressions

Variables are used to hold information. In C, we define a variable by specifying the name of the variable and the type. Table 5.3 lists the possible data types.

Data type	Precision	Range
`unsigned char`	8-bit unsigned	0 to +255
`signed char`	8-bit signed	-128 to +127
`unsigned int`	compiler-dependent	
`int`	compiler-dependent	
`unsigned short`	16-bit unsigned	0 to +65535
`short`	16-bit signed	-32768 to +32767
`unsigned long`	unsigned 32-bit	0 to 4294967295L
`long`	signed 32-bit	-2147483648L to 2147483647L
`float`	32-bit float	±10-38 to ±10+38
`double`	64-bit float	±10-308 to ±10+308

Table 5.2. Data types in C.

On the Keil compiler, there is an option to specify whether **`char`** all by itself without a `signed` or **`unsigned`** before it is considered signed or unsigned. Keil considers int as 32 bits. In this class we will avoid int and use long for 32-bit variables so there is no confusion. We will assume **`char`** is signed, but it is good practice to see exactly how **`char`** and **`int`** are treated by your compiler.

Variables declared outside of a function, like **`M`** in Program 5.1, are properly called external variables because they are defined outside of any function. While this is the standard term for these

variables, it is confusing because there is another class of external variable, one that exists in a separately compiled source file. In this document we will refer to variables in the present source file as globals, and we will refer to variables defined in another file as externals. There are two reasons to employ global variables. The first reason is data permanence. The other reason is information sharing. Normally we pass information from one module to another explicitly using input and output parameters, but there are applications like interrupt programming where this method is unavailable. For these situations, one module can store data into a global while another module can view it.

Local variables are very important in C programming. They contain temporary information that is accessible only within a narrow scope. We can define local variables at the start of a compound statement. We call these local variables since they are known only to the block in which they appear, and to subordinate blocks. The variables `side` and `area` in Program 5.3 are local. In C, local variable must be declared immediately after a brace { that begins a compound statement. Unlike globals, which are said to be static, locals are created dynamically when their block is entered, and they cease to exist when control leaves the block. Furthermore, local names supersede the names of globals and other locals declared at higher levels of nesting. Therefore, locals may be used freely without regard to the names of other variables. Although two global variables cannot use the same name, a local variable of one block can use the same name as a local variable in another block. Programming errors and confusion can be avoided by understanding these conventions. Local variables are implemented in registers or allocated on the stack.

Program 5.3 illustrates the assignment operator. Notice that in the line **`side=3;`** the **`side`** is on the left hand side of the **`=`** . The left side of the assignment specifies the address into which the data transfer will occur. On the other hand, if we were to wrote **`area=side;`** the **`side`** is on the right hand side of the **`=`** . The right side of an assignment statement will evaluate into a value, which specifies the data to be transferred. Notice that the line **`area=side;`** creates two copies of the data. The original value remains in **`side`**, while **`area`** also contains this value. As mentioned above, variables have a type (Table 5.3), and the expression on the right of an assignment statement must evaluate to a value of that same type. If **`side`** has the value 3, the expression **`side*side`** evaluates to a 9, and the 9 is stored into the variable **`area`**. The **`printf`** is used to output the results to the uart port.

```
int main(void) {
  unsigned long side; // room wall meters
  unsigned long area; // size squared meters
  UART_Init();        // call subroutine to initialize the uart
  side = 3;
  area = side*side;
  printf("\nArea of the room with side of %ld m is %ld sqr m\n",side,area);
}
```

Program 5.3. Simple program illustrating variables and assignment statements.

Program 5.4 illustrates the arithmetic operations of addition, subtraction, multiplication and division. In the operation **`x+4*y`**, multiplication has precedence over addition. Table 5.4 lists the operators available in the C language.

```
void main(void){
long x,y,z;    // Three local variables
   x=1; y=2;   // set the values of x and y
   z = x+4*y;  // arithmetic operation
   x++;        // same as x=x+1;
   y--;        // same as y=y-1;
   x = y<<2;   // left shift same as x=4*y;
   z = y>>2;   // right shift same as x=y/4;
```

```
   y += 2;      // same as y=y+2;
}
```

Program 5.4. Simple program illustrating C arithmetic operators.

Operation	*Meaning*
`=`	Assignment statement
`?`	Selection
`<`	Less than
`>`	Greater than
`!`	Logical not (true to false, false to true)
`~`	1's complement
`+`	Addition
`-`	Subtraction
`*`	Multiply or pointer reference
`/`	Divide
`%`	Modulo, division remainder
`|`	Logical or
`&`	Logical and, or address of
`^`	Logical exclusive or
`.`	Used to access parts of a structure

Operation	*Meaning*
`==`	Equal to comparison
`<=`	Less than or equal to
`>=`	Greater than or equal to
`!=`	Not equal to
`<<`	Shift left
`>>`	Shift right
`++`	Increment
`--`	Decrement
`&&`	Boolean and
`||`	Boolean or
`+=`	Add value to
`-=`	Subtract value to
`*=`	Multiply value to
`/=`	Divide value to
`|=`	Or value to
`&=`	And value to
`^=`	Exclusive or value to
`<<=`	Shift value left
`>>=`	Shift value right
`%=`	Modulo divide value to
`->`	Pointer to a structure

Table 5.4. Special characters can be operators; operators can be made from 1, 2, or 3 characters.

As with all programming languages the order of the tokens is important. There are two issues to consider when evaluating complex statements. The **precedence** of the operator determines which operations are performed first. In expression **`z=x+4*y`**, the **`4*y`** is performed first because * has higher precedence than + and =. The addition is performed second because + has higher precedence than =. The assignment = is performed last. Sometimes we use parentheses to clarify the meaning of the expression, even when they are not needed. Therefore, the line **`z=x+4*y;`** could have been written as **`z=(x+4*y);`** **`z=(x+4*y);`** or **`z=(x+(4*y));`**

The second issue is the **associativity**. Associativity determines the left to right or right to left order of evaluation when multiple operations of equal precedence are combined. For example + and - have the same precedence, so how do we evaluate the following?

```
z = y-2+x;
```

We know that + and - associate the left to right, this function is the same as **`z=(y-2)+x;`**. Meaning the subtraction is performed first because it is more to the left than the addition. Most operations associate left to right, but the Table 5.5 illustrates that some operators associate right to left.

Observation: When confused about precedence (and aren't we all) add parentheses to clarify the expression.

Precedence	Operators	Associativity
Highest	`()` `[]` `.` `->` `++`(postfix) `--`(postfix)	Left to right
	`++`(prefix) `--`(prefix) `!` `~` **`sizeof`**(type) +(unary) `-`(unary) `&`(address) `*`(dereference)	Right to left
	`*` `/` `%`	Left to right
	`+` `-`	Left to right
	`<<` `>>`	Left to right

	`<  <=  >  >=`	Left to right
	`==  !=`	Left to right
	`&`	Left to right
	`^`	Left to right
	`|`	Left to right
	`&&`	Left to right
	`||`	Left to right
	`? :`	Right to left
	`=  +=  -=  *=  /=  %=  <<=  >>=  |=  &=  ^=`	Right to left
Lowest	`,`	Left to right

Table 5.5. Precedence and associativity determine the order of operation.

Checkpoint 5.1: Which C data type does one use for numbers in the range of 0 to 200?
Checkpoint 5.2: Which C data type does one use for numbers in the range of -10 to +10?
Checkpoint 5.3: Which C data type does one use for numbers in the range of -1000 to +1000?
Checkpoint 5.4: Which C data type does one use for numbers in the range of zero to a million?
Checkpoint 5.5: What is the range of values possible with a C data type of **`int`**?
Checkpoint 5.6: Add parentheses to clarify this expression **`x&1&&y+1<z*4`**;

5.4 Functions

A **function** is a sequence of operations that can be invoked from other places within the software. We can pass zero or more parameters into a function. A function can have zero or one output parameter. It is important for the C programmer to distinguish the two terms declaration and definition. A function **declaration** specifies its name, its input parameters and its output parameter. Another name for a function declaration is **prototype**. A data structure declaration specifies its type and format. On the other hand, a function definition specifies the exact sequence of operations to execute when it is called. A function **definition** will generate object code, which are machine instructions to be loaded into memory that perform the intended operations. A data structure definition will reserve space in memory for it. The confusing part is that the definition will repeat the declaration specifications. The C compiler performs just one pass through the code, and we must declare data/functions before we can access/invoke them. To run, of course, all data and functions must be defined. A function to calculate the area of a square room is shown in Program 5.5. We can see that the declaration shows us how to use the function, not how the function works. Because the C compilation is a one-pass process, an object must be declared or defined before it can be used in a statement. Actually the preprocessor performs the first pass through the program that handles the preprocessor directives. A top-down approach is to first declare a function, use the function, and lastly define the function as illustrated in Program 5.5.

```
unsigned long Calc_Area(unsigned long s);
int main(void) {
  unsigned long side; // room wall meters
  unsigned long area; // size squared meters
  UART_Init();  // call subroutine to initialize the uart
  printf("This program calculates areas of square-shaped rooms\n");
  side = 3;
  area = Calc_Area(side);
  printf("\nArea of the room with side of %ld m is %ld sqr m\n",side,area);
  side = side+2;
  area = Calc_Area(side);
  printf("\nArea of the room with side of %ld m is %ld sqr m\n",side,area);
}
// Calculates area
// Input:  side of a room (unsigned long) in meters
// Output: area of the room (unsigned long) in square meters
unsigned long Calc_Area(unsigned long s) {
  unsigned long result;
  result = s*s;
```

```
  return(result);
}
```

Program 5.5. A main program that calls a function. In this case the declaration occurs first.

A bottom-down approach is to first define a function, and then use the function as illustrated in Program 5.6. In the bottom up approach, the definition both declares its structure and defines what it does.

```
// Calculates area
// Input:  side of a room (unsigned long) in meters
// Output: area of the room (unsigned long) in square meters
unsigned long Calc_Area(unsigned long s) {
  unsigned long result;
  result = s*s;
  return(result);
}
int main(void) {
  unsigned long side; // room wall meters
  unsigned long area; // size squared meters
  UART_Init();  // call subroutine to initialize the uart
  printf("This program calculates areas of square-shaped rooms\n");
  side = 3;
  area = Calc_Area(side);
  printf("\nArea of the room with side of %ld m is %ld sqr m\n",side,area);
  side = side+2;
  area = Calc_Area(side);
  printf("\nArea of the room with side of %ld m is %ld sqr m\n",side,area);
}
```

Program 5.6. A main program that calls a function. In this case the definition occurs before its use.

Assembly Code

```
Address   Machine Code   Label    Instruction       Comment

0x00000660 EB010200       sum    ADD R2,R1,R0      ;z=x+y
0x00000664 4610                  MOV R0,R2         ;return value
0x00000666 4770                  BX LR
0x00000668 F44F60FA       main   MOV R0,#2000      ;first parameter
0x0000066C F44F61FA              MOV R1,#2000      ;second parameter
0x00000670 F7FFFFF6              BL sum            ;call function
0x00000674 4603                  MOV R3,R0         ;a=sum(2000,2000)
0x00000676 F04F0400              MOV R4,#0x00      ;b=0
0x0000067A 4620           loop   MOV R0,R4         ;first parameter
0x0000067C F04F0101              MOV R1,#0x01      ;second parameter
0x00000680 F7FFFFEE              BL sum            ;call function
0x00000684 4604                  MOV R4,R0         ;b=sum(b,1)
0x00000686 E7F8                  B loop
```

C Code

```
long sum(long x, long y){
  long z;
  z = x+y;
  return(z);
}
void main(void){
  long a,b;
  a = sum(2000,2000);
  b = 0;
  while(1){
    b = sum(b,1);
  }
}
```

Registers

Register	Value
R0	0x00000000
R1	0x00000000
R2	0x00000000
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13(SP)	0x00000000
R14(LR)	0x00000000
R15(PC)	0x00000668

Program 5.7. A function with two inputs and one output.

To specify the absence of a parameter we use the expression **`void`**. The body of a function consists of a statement that performs the work. Normally the body is a compound statement between a {} pair. If the function has a return parameter, then all exit points must specify what to return.

Checkpoint 5.7: What does it mean to say a function as one input parameter?
Checkpoint 5.8: What does it mean to say a function as one output parameter?

5.5 Conditional branching and loops

In Program 5.8 we will introduce a simple conditional control structure. Assume the global variable **`error`** is initialized to zero. The goal is to make sure the function is being used properly. An effective software design approach is to test the input parameters of a function to make sure the values make sense. An unsigned long can represent a number up to 4 billion. Clearly the system is not operating properly if we are trying to calculate the size of a room with 4 billion meter sides. In this case, we define the largest possible room to have a side of 25 meters. The expression **`s<=25`** will return *true* if the side is less than or equal to 25 and will return a *false* if the side is strictly greater than 25. The statement immediately following the **`if`** will be executed if the condition is *true*. The statement immediately following the **`else`** will be executed if the condition is *false*.

```
unsigned long error;
// Calculates area
// Input:  side of a room (unsigned long)
// Output: area of the room (unsigned long)
// Notes: ...
unsigned long Calc_Area(unsigned long s) {
  unsigned long result;
  if(s <= 25){
    result = s*s;
  }else{
    result = 0; // mistake
    error = error +1;
  }
  return(result);
}
```

Program 5.8. Simple program illustrating the C if else control structure.

The goal in program 5.9 is to test the **Calc_Area** function. Like the **`if`** statement, the **`while`** statement has a conditional test (i.e., returns a true/false). The statement immediately following the **`while`** will be executed over and over until the conditional test becomes false.

```
int main(void) {
  unsigned long side; // room wall meters
  unsigned long area; // size squared meters
  UART_Init();  // call subroutine to initialize the uart
  printf("This program calculates areas of square-shaped rooms\n");
  side = 1;
  while(side < 50){
    area = Calc_Area(side);
    printf("\nArea of the room with side of %ld m is %ld sqr m\n",side,area);
    side = side+1;
  }
}
```

Program 5.9. Simple program illustrating the C while control structure.

The **`for`** control structure has three parts and a body.

```
for(part1;part2;part3){body;}
```

In Program 5.10, the first part **side=1** is executed once at the beginning. Before the body is executed, the end-condition part 2 is executed. If the condition is *true*, **side<50** then the body is executed. After the body is executed, the third part is executed, **side=side+1**. The second part is always a conditional that results in a *true* or a *false*. The body and third part are repeated until the conditional is *false*.

```
int main(void) {
  unsigned long side; // room wall meters
  unsigned long area; // size squared meters
  UART_Init();  // call subroutine to initialize the uart
  printf("This program calculates areas of square-shaped rooms\n");
  for(side = 1; side < 50; side = side+1){
    area = Calc_Area(side);
    printf("\nArea of the room with side of %ld m is %ld sqr m\n",side,area);
  }
}
```

Program 5.10. Simple program illustrating the C for-loop control structure.

Although C is a free field language, notice how the indenting has been added to programs in this book. The purpose of this indenting is to make the program easier to read. On the other hand since C is a free field language, the following two statements are quite different

```
if(n1>100) n2=100; n3=0;
if(n1>100) {n2=100; n3=0;}
```

In both cases **n2=100;** is executed if n1>100. In the first case the statement **n3=0;** is always executed, while in the second case **n3=0;** is executed only if n1>100.

5.6 Keyboard input using `scanf`

```
//**** 0. Documentation Section
//  This program calculates the area of square shaped rooms
//  Author: Ramesh Yerraballi& Jon Valvano
//  Date: 6/28/2013
//
// 1. Pre-processor Directives Section
#include< stdio.h>  // Diamond braces for sys lib: Standard I/O
#include "uart.h"   // Quotes for user lib: UART lib
// 2. Global Declarations section
// global variable
unsigned long error;
// Function Prototypes
// Compiler aid for "type checking"
void Initialize(void);
unsigned long Calc_Area(unsigned long s); // Says Calc_Area expects
            // an unsigned long and returns an unsigned long
// 3. Subroutines Section
// MAIN: Mandatory routine for a C program to be executable
int main(void) {
  unsigned long side; // room wall meters
  unsigned long area; // size squared meters
  UART_Init();  // call subroutine to initialize the uart
  Initialize();
  printf("This program calculates areas of square-shaped rooms\n");
  while(side != 0){
    printf("Give room side(zero to quit):");
    scanf("%ld", &side);
    area = Calc_Area(side);
    if(area != 0){
```

```
      printf("\nArea with side of %ld m is %ld sqr m\n",side,area);
    } else {
      printf("\n Size cannot exceed 25 meters\n");
    }
  }
  printf("Goodbye (Mistake count = %ld\n", error);
}
// Initialize global
// Inputs: none
// Outputs: none
// Notes:
void Initialize(void){
  error = 0;
}
// Calculates area
// Input:  side of a room (unsigned long)
// Output: area of the room (unsigned long)
// Notes: ...
unsigned long Calc_Area(unsigned long s) {
  unsigned long result;
  if(s <= 25){
    result = s*s;
  }else{
    result = 0; // mistake
    error = error +1;
  }
  return(result);

}
```

Program 5.11. Software to calculate the area of a square room.

5.7 C Keywords and Punctuation

C has predefined tokens, called **keywords**, which have specific meaning in C programs, as listed in Table 5.6. This section describes keywords and punctuation.

Keyword	*Meaning*
`__asm`	Specify a function is written in assembly code (specific to ARM Keil™ **uVision®**)
`auto`	Specifies a variable as automatic (created on the stack)
`break`	Causes the program control structure to finish
`case`	One possibility within a `switch` statement
`char`	Defines a number with a precision of 8 bits
`const`	Defines parameter as constant in ROM, and defines a local parameter as fixed value
`continue`	Causes the program to go to beginning of loop
`default`	Used in `switch` statement for all other cases
`do`	Used for creating program loops
`double`	Specifies variable as double precision floating point
`else`	Alternative part of a conditional
`extern`	Defined in another module
`float`	Specifies variable as single precision floating point
`for`	Used for creating program loops
`goto`	Causes program to jump to specified location
`if`	Conditional control structure
`int`	Defines a number with a precision that will vary from compiler to compiler
`long`	Defines a number with a precision of 32 bits
`register`	Specifies how to implement a local
`return`	Leave function
`short`	Defines a number with a precision of 16 bits
`signed`	Specifies variable as signed (default)
`sizeof`	Built-in function returns the size of an object
`static`	Stored permanently in memory, accessed locally
`struct`	Used for creating data structures

switch	Complex conditional control structure
typedef	Used to create new data types
unsigned	Always greater than or equal to zero
void	Used in parameter list to mean no parameter
volatile	Can change implicitly outside the direct action of the software.
while	Used for creating program loops

Table 5.6. Keywords have predefined meanings.

The **volatile** keyword disables compiler optimization, forcing the compiler to fetch a new value each time. We will use **volatile** when defining I/O ports because the value of ports can change outside of software action. We will also use **volatile** when sharing a global variable between the main program and an interrupt service routine. It is a good programming practice not to use these keywords for your variable or function names.

Punctuation marks are very important in C. It is one of the most frequent sources of errors for both beginning and experienced programmers.

Semicolons are used as statement terminators. Strange and confusing syntax errors may be generated when you forget a semicolon, so this is one of the first things to check when trying to remove syntax errors. Notice that one semicolon is placed at the end of every simple statement in Program 5.12. When executed, the function **Step** will output the pattern 10, 9, 5, 6 to Port D. The **#define** statement creates a substitution rule, such that every instance of **STEPPER** in the program is replaces with **(*((volatile unsigned long *)0x4000703C))**.

```
#define STEPPER (*((volatile unsigned long *)0x4000703C))
void Step(void){
  STEPPER = 10;
  STEPPER = 9;
  STEPPER = 5;
  STEPPER = 6;
}
```

Program 5.12. Semicolons are used to separate one statement from the next.

Preprocessor directives do not end with a semicolon since they are not actually part of the C language proper. Preprocessor directives begin in the first column with the **#** and conclude at the end of the line. Program 5.13 will fill the array **DataBuffer** with data read from the input Port A. We assume in this example that Port A has been initialized as an input. Notice that semicolons are used to separate the three fields of the for loop statement.

```
unsigned char DataBuffer[100];
#define GPIO_PORTA_DATA_R  (*((volatile unsigned long *)0x400043FC))
void Fill(void){ long j;
  for(j=0; j<100; j++){
    DataBuffer[j] = GPIO_PORTA_DATA_R;
  }
}
```

Program 5.13. Semicolons are used to separate three fields of the for statement.

Colons terminate **case**, and **default** prefixes that appear in **switch** statements. In Program 5.14 one output to the stepper motor produced each time the function **OneStep** is called. The proper stepper motor sequence is 10–9–5–6. The default case is used to restart the pattern. For both applications of the colon (goto and switch), we see that a label is created that is a potential target for a transfer of control. Notice the use of colons in Program 5.14.

```
unsigned char Last=10;
```

```
void OneStep(void){
unsigned char theNext;
  switch(Last){
    case 10: theNext = 9; break;   // 10 to 9
    case 9: theNext = 5;  break;   // 9 to 5
    case 5: theNext = 6;  break;   // 5 to 6
    case 6: theNext = 10; break;   // 6 to 10
    default: theNext = 10;
  }
  GPIO_PORTD_DATA_R = theNext;
  Last = theNext;   // set up for next call
}
```

Program 5.14. Colons are also used with the switch statement.

Commas separate items that appear in lists. We can create multiple variables of the same type using commas.

```
unsigned short beginTime,endTime,elapsedTime;
```

Lists are also used with functions having multiple parameters, both when the function is defined and called. Program 5.15 adds two 16-bit signed numbers, implementing ceiling and floor. Notice the use of commas in Program 5.15.

```
short add(short x, short y){ short z;
  z = x+y;
  if((x>0)&&(y>0)&&(z<0))z = 32767;
  if((x<0)&&(y<0)&&(z>0))z = -32768;
  return(z);
}
void main(void){ short a,b;
  a = add(2000,2000)
  b = 0
  while(1){
    b = add(b,1);
}
```

Program 5.15. Commas separate the parameters of a function.

Lists can also be used in general expressions. Sometimes it adds clarity to a program if related variables are modified at the same place. The value of a list of expressions is always the value of the last expression in the list. In the following example, first **thetime** is incremented, next **thedate** is decremented, and then **x** is set to **k+2**.

```
X = (thetime++, thedate--, k+2);
```

Apostrophes are used to specify character literals. Assuming the function **OutChar** will display a single ASCII character, Program 5.16 will display the lower case alphabet.

```
void Alphabet(void){ unsigned char mych;
  for(mych='a'; mych<='z'; mych++){
    OutChar(mych); // Print next letter
  }
}
```

Program 5.16. Apostrophes are used to specify characters.

Quotation marks are used to specify string literals. Strings are stored as a sequence of ASCII characters followed by a termination code, 0. Program 5.17 will display "Hello World" twice.

```
const unsigned char Msg[]= "Hello World"; // string constant
void OutString(const unsigned char str[]){ int i;
  i = 0;
```

```
  while(str[i]){     // output until the 0 termination
    OutChar(str[i]); // Print next letter
    i = i+1;
  }
}
void PrintHelloWorld(void){
  OutString("Hello World");
  OutString(Msg);
}
```

Program 5.17. Quotation marks are used to specify strings.

Braces {} are used throughout C programs. The most common application is for creating a compound statement. Each open brace { must be matched with a closing brace }. Notice the use of indenting helps to match up braces. Each time an open brace is used, the source code is tabbed over using 2 spaces. In this way, it is easy to see at a glance the brace pairs.

Square **brackets** enclose array dimensions (in declarations) and subscripts (in expressions).

Parentheses enclose argument lists that are associated with function declarations and calls. They are required even if there are no arguments. As with all programming languages, C uses parentheses to control the order in which expressions are evaluated. Thus, (11+3)/2 yields 7, whereas 11+3/2 yields 12. Parentheses are very important when writing expressions.

5.8. Chapter 5 Quiz

5.1 Match the punctuation with its meaning

`;`	sEnd of statement
`:`	Defines a label
`,`	Separates elements of a list
`( )`	Start and end of a parameter list
`{ }`	Start and stop of a compound statement
`[ ]`	Start and stop of a array index
`" "`	Start and stop of a string
`' '`	Start and stop of a character constant

5.2 Match the variable type with its minimum and maximum value

Data type	Range
`unsigned char`	0 to +255
`signed char`	-128 to +127
`unsigned short`	0 to +65535
`short`	-32768 to +32767
`unsigned long`	0 to 4294967295L
`long`	-2147483648L to 2147483647L

5.3 Match the variable type with its precision

Data type	Precision
`char`	8 bits
`short`	16 bits
`long`	32 bits

5.4 Are the variables side and area local or global?

```
int main(void) {
  unsigned long side; // room wall meters
  unsigned long area; // size squared meters
  UART_Init();        // call subroutine to initialize the uart
  side = 3;
  area = side*side;
  printf("\nArea of the room with side of %ld m is %ld sqr m\n",side,area);
}
```

5.5 The goal is to return true if and only if the input is a number between 0x30 and 0x39.

```
long CheckInput(long input) {
  if(( Input xxx 0x30) yyy (Input zzz 0x39)){
    return 1;
  } else{
    return 0; // false
}
```

a) What C code do you need to place in the xxx position of the program?
b) What C code do you need to place in the yyy position of the program?
c) What C code do you need to place in the zzz position of the program?

Chapter 6: Parallel I/O ports
Embedded Systems - Shape the World
Jonathan Valvano and Ramesh Yerraballi

The chapter covers the purpose of parallel ports, how to program them using memory-mapped I/O and initialization rituals. We will learn how to access I/O registers in a friendly manner. We will test the system by single-stepping in the simulator, and we will observe the running system using a logic analyzer.

Learning Objectives:

- Know what is a parallel port
- Know how a pin can be either input or output as specified by the direction register
- Know the steps required to initialize a parallel port
- Know how to access I/O registers in a friendly manner
- Know how to read data from an input port
- Know how to write data to an output port
- Know how to use the logic analyzer in the simulator

6.0. Introduction

Our first input/output interfaces will use the parallel ports or GPIO, allowing us to exchange digital information with the external world. From the very beginning of a project, we must consider how the system will be tested. In this chapter we present some debugging techniques that will be very useful for verifying proper operation of our system. Effective debugging tools are designed into the system becoming part of the system, rather than attached onto the system after it is built.

In this chapter, we present the I/O pin configurations for the TM4C123 microcontrollers. The regular function of a pin is to perform parallel I/O. Most pins, however, have an alternative function. For example, port pins PA1 and PA0 can be either regular parallel port pins or an asynchronous serial port called universal asynchronous receiver/transmitter (UART). The ability to manage time, as an input measurement and an output parameter, has made a significant impact on the market share growth of microcontrollers. Joint Test Action Group (**JTAG**), standardized as the IEEE 1149.1, is a standard test access port used to program and debug the microcontroller board. Each microcontroller uses five port pins for the JTAG interface.

Common Error: Even though it is possible to use the five JTAG pins as general I/O, debugging most microcontroller boards will be more stable if these five pins are left dedicated to the JTAG debugger.

I/O pins on Stellaris and Tiva microcontrollers have a wide range of alternative functions:

- **UART: Universal asynchronous receiver/transmitter**
- **SSI : Synchronous serial interface**
- **I^2C : Inter-integrated circuit**
- **Timer: Periodic interrupts, input capture, and output compare**
- **PWM: Pulse width modulation**
- **ADC : Analog to digital converter, measure analog signals**
- **Analog Comparator Compare two analog signals**
- **QEI: Quadrature encoder interface**
- **USB: Universal serial bus**
- **Ethernet: High-speed network**
- **CAN: Controller area network**

The **UART** can be used for serial communication between computers. It is asynchronous and allows for simultaneous communication in both directions. The **SSI** is alternately called serial peripheral interface (SPI). It is used to interface medium-speed I/O devices. In this book, we will use it to interface a

graphics display. We could use SSI to interface a digital to analog converter (DAC) or a secure digital card (SDC). **I^2C** is a simple I/O bus that we will use to interface low speed peripheral devices. Input capture and output compare will be used to create periodic interrupts and measure period, pulse width, phase, and frequency. **PWM** outputs will be used to apply variable power to motor interfaces. In a typical motor controller, input capture measures rotational speed, and PWM controls power. A PWM output can also be used to create a DAC. The **ADC** will be used to measure the amplitude of analog signals and will be important in data acquisition systems. The analog comparator takes two analog inputs and produces a digital output depending on which analog input is greater. The **QEI** can be used to interface a brushless DC motor. **USB** is a high-speed serial communication channel. The **Ethernet** port can be used to bridge the microcontroller to the Internet or a local area network. The **CAN** creates a high-speed communication channel between microcontrollers and is commonly found in automotive and other distributed control applications.

Observation: The expression **mixed-signal** refers to a system with both analog and digital components. Notice how many I/O ports perform this analog↔digital bridge: ADC, DAC, analog comparator, PWM, QEI, Input capture, and output compare.

6.1. Stellaris LM4F120 and Tiva TM4C123 LaunchPad I/O pins

Figure 6.1 draws the I/O port structure for the LM4F120H5QR and TM4C123GH6PM. These microcontrollers are used on the EK-LM4F120XL and EK-TM4C123GXL LaunchPads. Pins on the LM3S family have two possibilities: digital I/O or an alternative function. However, pins on the LM4F/TM4C family can be assigned to as many as eight different I/O functions. Pins can be configured for digital I/O, analog input, timer I/O, or serial I/O. For example PA0 can be digital I/O or serial input. There are two buses used for I/O. The digital I/O ports are connected to both the advanced peripheral bus and the advanced high-performance bus. Because of the multiple buses, the microcontroller can perform I/O bus cycles simultaneous with instruction fetches from flash ROM. The LM4F120H5QR has eight UART ports, four SSI ports, four I2C ports, two 12-bit ADCs, twelve timers, a CAN port, and a USB interface. The TM4C123GH6PM adds up to 16 PWM outputs. There are 43 I/O lines. There are twelve ADC inputs; each ADC can convert up to 1M samples per second. Table 6.1 lists the regular and alternate names of the port pins.

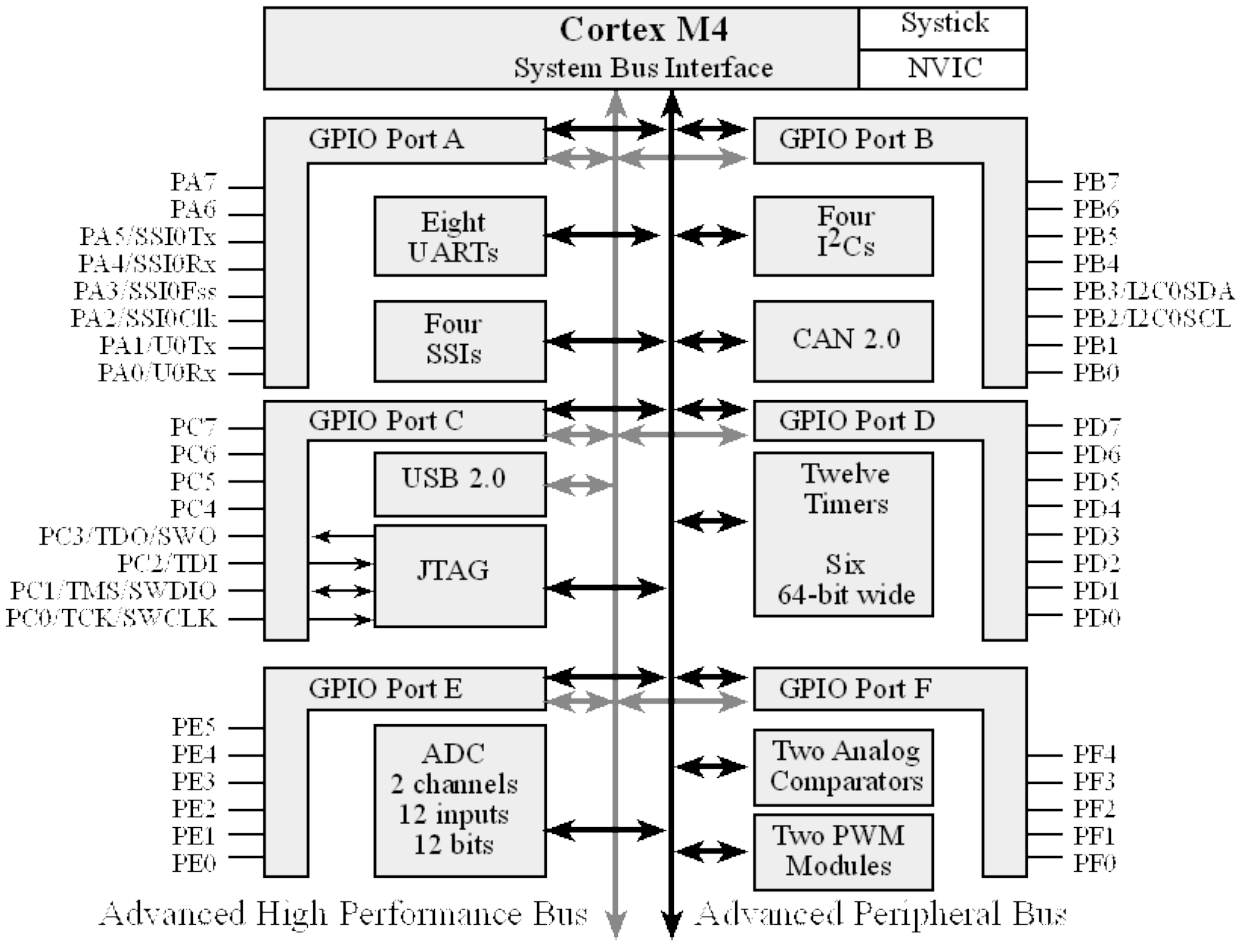


Figure 6.1. I/O port pins for the LM4F120H5QR / TM4C123GH6PM microcontrollers.

Each pin has one configuration bit in the GPIOAMSEL register. We set this bit to connect the port pin to the ADC or analog comparator. For digital functions, each pin also has four bits in the GPIOPCTL register, which we set to specify the alternative function for that pin (0 means regular I/O port). Not every pin can be connected to every alternative function. See Table 6.1.
Pins PC3 – PC0 were left off Table 6.1 because these four pins are reserved for the JTAG debugger and should not be used for regular I/O. Notice, most alternate function modules (e.g., U0Rx) only exist on one pin (PA0). While other functions could be mapped to two or three pins (e.g., CAN0Rx could be mapped to one of the following: PB4, PE4, or PF0.)

For example, if we wished to use UART7 on pins PE0 and PE1, we would set bits 1,0 in the **`GPIO_PORTE_DEN_R`** register (enable digital), clear bits 1,0 in the **`GPIO_PORTE_AMSEL_R`** register (disable analog), set the PMCx bits in the **`GPIO_PORTE_PCTL_R`** register for PE0, PE1 to 0001 (enable UART functionality), and set bits 1,0 in the **`GPIO_PORTE_AFSEL_R`** register (enable alternate function). If we wished to sample an analog signal on PD0, we would set bit 0 in the alternate function select register, clear bit 0 in the digital enable register (disable digital), set bit 0 in the analog mode select register (enable analog), and activate one of the ADCs to sample channel 7.

IO	Ain	0	1	2	3	4	5	6	7	8	9	14
PA0		*Port*	*U0Rx*							CAN1Rx		
PA1		*Port*	*U0Tx*							CAN1Tx		
PA2		Port		SSI0Clk								
PA3		Port		SSI0Fss								
PA4		Port		SSI0Rx								
PA5		Port		SSI0Tx								

PA6		Port			I_2C1SCL		M1PWM2					
PA7		Port			I_2C1SDA		M1PWM3					
PB0		Port	U1Rx						T2CCP0			
PB1		Port	U1Tx						T2CCP1			
PB2		Port			I_2C0SCL				T3CCP0			
PB3		Port			I_2C0SDA				T3CCP1			
PB4	Ain10	Port		SSI2Clk		M0PWM2			T1CCP0	CAN0Rx		
PB5	Ain11	Port		SSI2Fss		M0PWM3			T1CCP1	CAN0Tx		
PB6		Port		SSI2Rx		M0PWM0			T0CCP0			
PB7		Port		SSI2Tx		M0PWM1			T0CCP1			
PC4	C1-	Port	U4Rx	U1Rx		M0PWM6		IDX1	WT0CCP0	U1RTS		
PC5	C1+	Port	U4Tx	U1Tx		M0PWM7		PhA1	WT0CCP1	U1CTS		
PC6	C0+	Port	U3Rx					PhB1	WT1CCP0	USB0epen		
PC7	C0-	Port	U3Tx						WT1CCP1	USB0pflt		
PD0	Ain7	Port	SSI3Clk	SSI1Clk	I_2C3SCL	M0PWM6	M1PWM0		WT2CCP0			
PD1	Ain6	Port	SSI3Fss	SSI1Fss	I_2C3SDA	M0PWM7	M1PWM1		WT2CCP1			
PD2	Ain5	Port	SSI3Rx	SSI1Rx		M0Fault0			WT3CCP0	USB0epen		
PD3	Ain4	Port	SSI3Tx	SSI1Tx				IDX0	WT3CCP1	USB0pflt		
PD4	*USB0DM*	*Port*	*U6Rx*						*WT4CCP0*			
PD5	*USB0DP*	*Port*	*U6Tx*						*WT4CCP1*			
PD6		Port	U2Rx			M0Fault0		PhA0	WT5CCP0			
PD7		Port	U2Tx					PhB0	WT5CCP1	NMI		
PE0	Ain3	Port	U7Rx									
PE1	Ain2	Port	U7Tx									
PE2	Ain1	Port										
PE3	Ain0	Port										
PE4	Ain9	Port	U5Rx		I_2C2SCL	M0PWM4	M1PWM2			CAN0Rx		
PE5	Ain8	Port	U5Tx		I_2C2SDA	M0PWM5	M1PWM3			CAN0Tx		
PF0		Port	U1RTS	SSI1Rx	CAN0Rx		M1PWM4	PhA0	T0CCP0	NMI	C0o	
PF1		Port	U1CTS	SSI1Tx			M1PWM5	PhB0	T0CCP1		C1o	TRD1
PF2		Port		SSI1Clk		M0Fault0	M1PWM6		T1CCP0			TRD0
PF3		Port		SSI1Fss	CAN0Tx		M1PWM7		T1CCP1			TRCLK
PF4		Port					M1Fault0	IDX0	T2CCP0	USB0epen		

Table 6.1. PMCx bits in the GPIOPCTL register on the LM4F/TM4C specify alternate functions. *PD4* and *PD5* are hardwired to the USB device. *PA0* and *PA1* are hardwired to the serial port. PWM not on LM4F120.

The LaunchPad evaluation board (Figure 6.2) is a low-cost development board available as part number EK-LM4F120XL and EK-TM4C123GXL from **https://estore.ti.com/** and in the US from regular electronic distributors like Digikey, Mouser, Arrow, Newark, and Avnet. For detailed instruction for obtaining the lab kit, refer to **http://users.ece.utexas.edu/~valvano/edX/**. The microcontroller board provides an integrated In-Circuit Debug Interface (ICDI), which allows programming and debugging of the onboard LM4F120 or TM4C123 microcontroller. One USB cable is used by the debugger (ICDI), and the other USB allows the user to develop USB applications (device). The user can select board power to come from either the debugger (ICDI) or the USB device (device) by setting the Power selection switch.

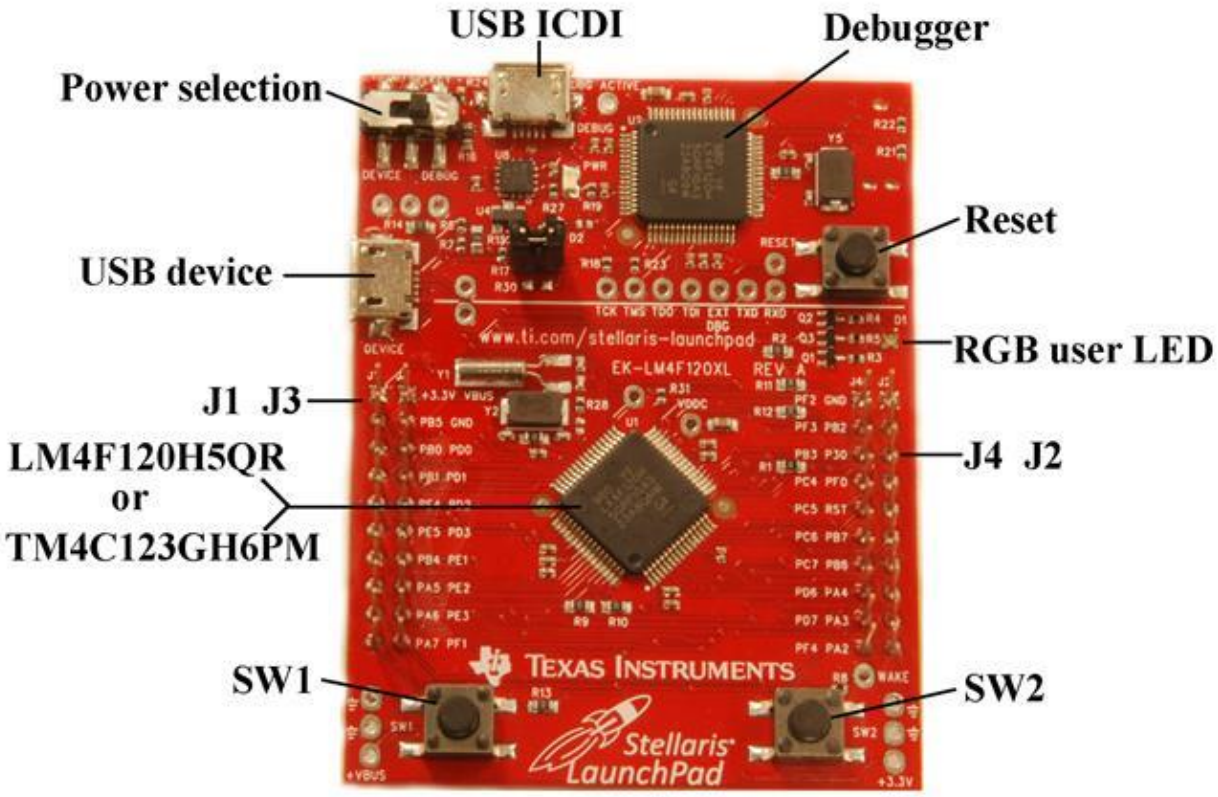


Figure 6.2. Tiva LaunchPad based on the LM4F120H5QR or TM4C123GH6PM.

Pins PA1 – PA0 create a serial port, which is linked through the debugger cable to the PC. The serial link is a physical UART as seen by the LF4F120/TM4C and mapped to a virtual COM port on the PC. The USB device interface uses PD4 and PD5. The JTAG debugger requires pins PC3 – PC0. The LaunchPad connects PB6 to PD0, and PB7 to PD1. If you wish to use both PB6 and PD0 you will need to remove the R9 resistor. Similarly, to use both PB7 and PD1 remove the R10 resistor.

The Tiva LaunchPad evaluation board has two switches and one 3-color LED. See Figure 6.3. The switches are negative logic and will require activation of the internal pull-up resistors. In particular, you will set bits 0 and 4 in **`GPIO_PORTF_PUR_R`** register. The LED interfaces on PF3 – PF1 are positive logic. To use the LED, make the PF3 – PF1 pins an output. To activate the red color, output a one to PF1. The blue color is on PF2, and the green color is controlled by PF3. The 0-Ω resistors (R1, R2, R11, R12, R13, R25, and R29) can be removed to disconnect the corresponding pin from the external hardware.

The LaunchPad has four 10-pin connectors, labeled as J1 J2 J3 J4 in Figures 6.2 and 6.4, to which you can attach your external signals. The top side of these connectors has male pins, and the bottom side has female sockets. The intent is to stack boards together to make a layered system. Texas Instruments also supplies Booster Packs, which are pre-made external devices that will plug into this 40-pin connector. The Booster Packs for the MSP430 LaunchPad are compatible with this board. One simply plugs the 20-pin connectors of the MSP430 booster into the outer two rows. The inner 10-pin headers (connectors J3 and J4) apply only to Stellaris or Tiva Booster Packs.

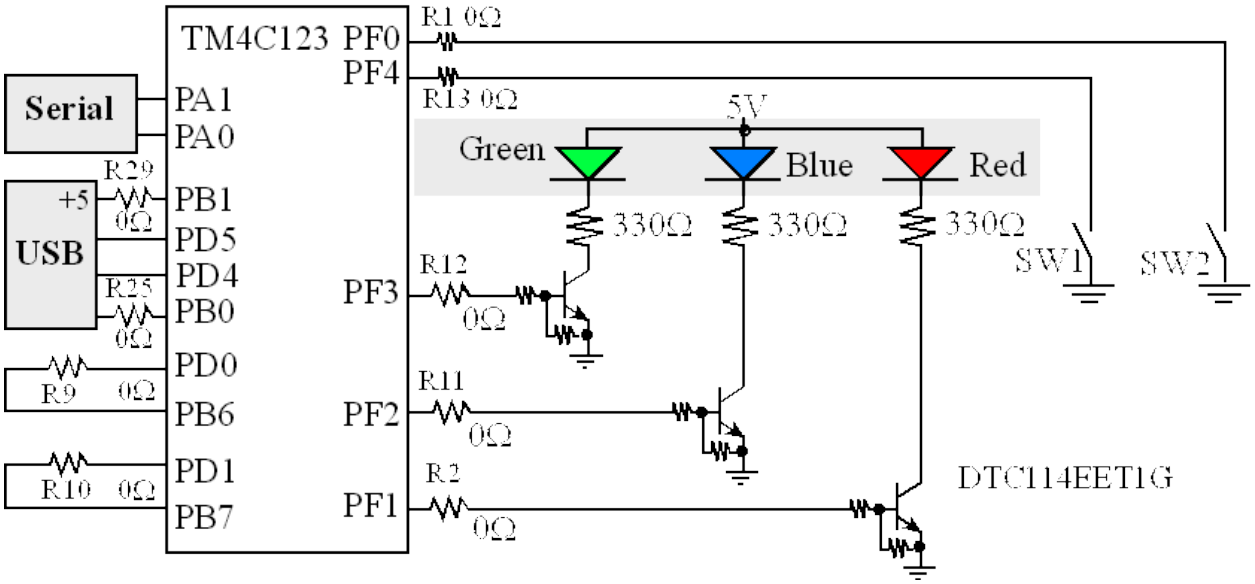


Figure 6.3. Switch and LED interfaces on the Tiva LaunchPad Evaluation Board. The zero ohm resistors can be removed so the corresponding pin can be used for its regular purpose.

There are a number of good methods to connect external circuits to the LaunchPad. One method is to purchase a male to female jumper cable (e.g., item number 826 at **www.adafruit.com**). A second method is to solder a solid wire into a female socket (e.g., Hirose DF11-2428SCA) creating a male to female jumper wire.

Since the LaunchPad has both male and female headers, a very inexpensive method to build systems is to connect solid 24 gauge wire to the female headers on the bottom.

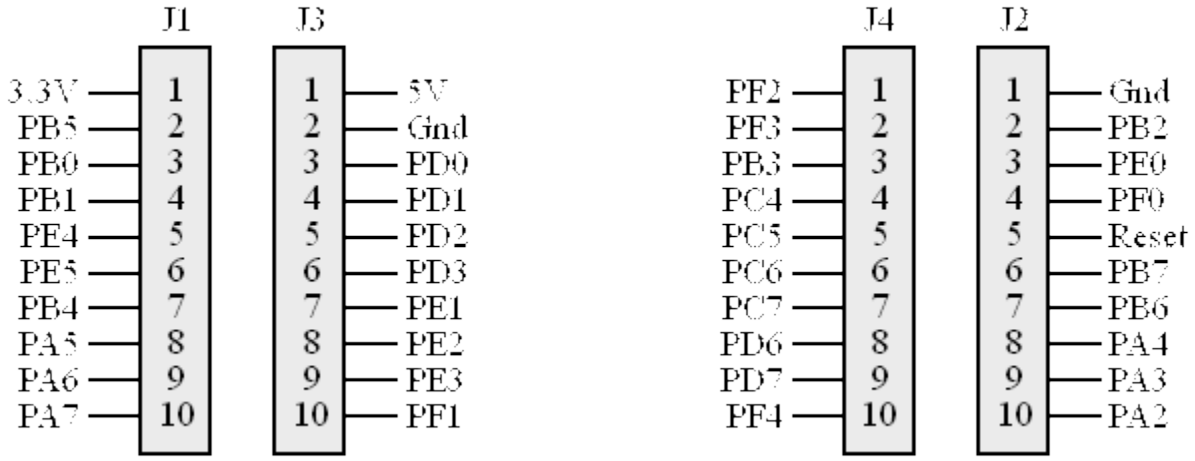


Figure 6.4. Interface connectors on the Tiva LM4F120/TM4C123 LaunchPad Evaluation Board.

Each pin has one configuration bit in the GPIOAMSEL register. We set this bit to connect the port pin to the ADC or analog comparator. For digital functions, each pin also has four bits in the GPIOPCTL register, which we set to specify the alternative function for that pin (0 means regular I/O port). Not every pin can be connected to every alternative function. See Table 6.1.

6.2. Basic Concepts of Input and Output Ports

The simplest I/O port on a microcontroller is the parallel port. A parallel I/O port is a simple mechanism that allows the software to interact with external devices. It is called parallel because multiple signals can be accessed all at once. An **input port**, which allows the software to read external digital signals, is read only. That means a read cycle access from the port address returns the values existing on the inputs

at that time. In particular, the tristate driver (triangle shaped circuit in Figure 6.5) will drive the input signals onto the data bus during a read cycle from the port address. A write cycle access to an input port usually produces no effect. The digital values existing on the input pins are copied into the microcontroller when the software executes a read from the port address. There are no digital input-only ports on the LM4F/TM4C family of microcontrollers. The LM4F/TM4C family of microcontrollers has 5V-tolerant digital inputs, meaning an input high signal can be any voltage from 2.0 to 5.0 V. On the STMicroelectronics STM32F10xx family, some inputs are 5-V tolerant and others are not.

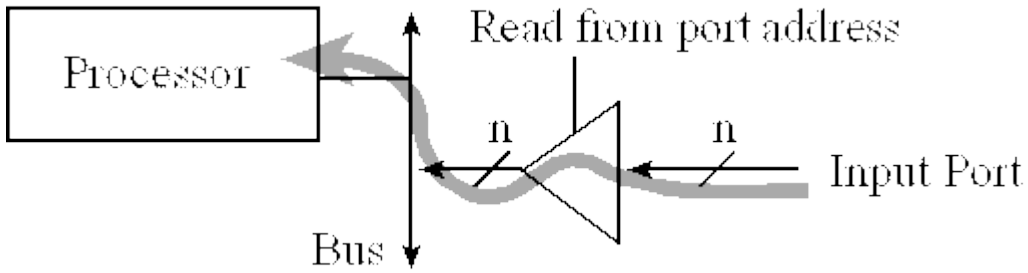


Figure 6.5. A read only input port allows the software to sense external digital signals.

Checkpoint 6.1: What happens if the software writes to an input port like Figure 6.5?

While an input device usually just involves the software reading the port, an output port can participate in both the read and write cycles very much like a regular memory. Figure 6.6 describes a **readable output port**. A write cycle to the port address will affect the values on the output pins. In particular, the microcontroller places information on the data bus and that information is clocked into the D flip-flops. Since it is a readable output, a read cycle access from the port address returns the current values existing on the port pins. There are no output-only ports on the LM4F/TM4C family of microcontrollers.

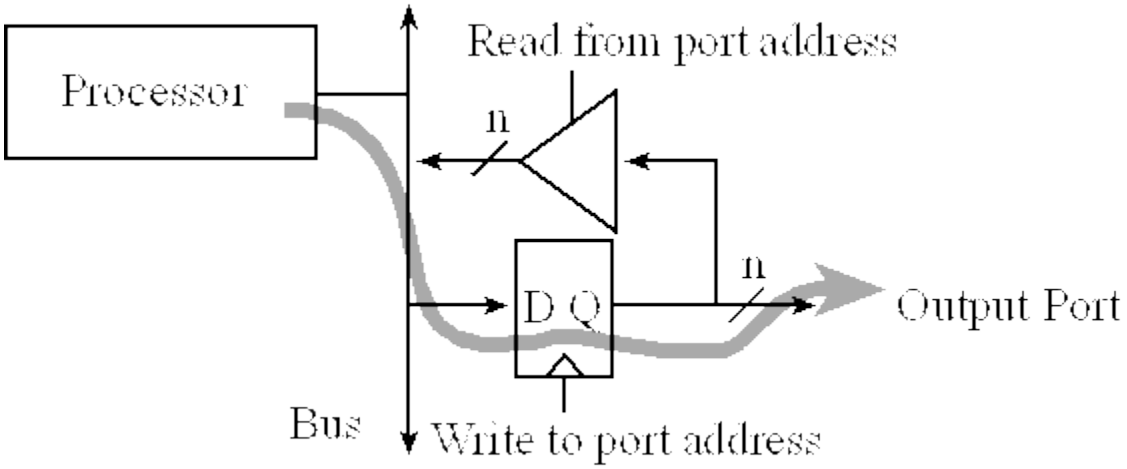


Figure 6.6. A readable output port allows the software to generate external digital signals.

Checkpoint 6.2: What happens if the software reads from an output port like Figure 6.6?

To make the microcontroller more marketable, most ports can be software-specified to be either inputs or outputs. Microcontrollers use the concept of a **direction register** to determine whether a pin is an input (direction register bit is 0) or an output (direction register bit is 1), as shown in Figure 6.7. We define an initialization **ritual** as a program executed during start up that initializes hardware and software. If the ritual software makes direction bit zero, the port behaves like a simple input, and if it makes the direction bit one, it becomes a readable output port. Each digital port pin has a direction bit. This means some pins on a port may be inputs while others are outputs. The digital port pins on most microcontrollers are bidirectional, operating similar to Figure 6.7.

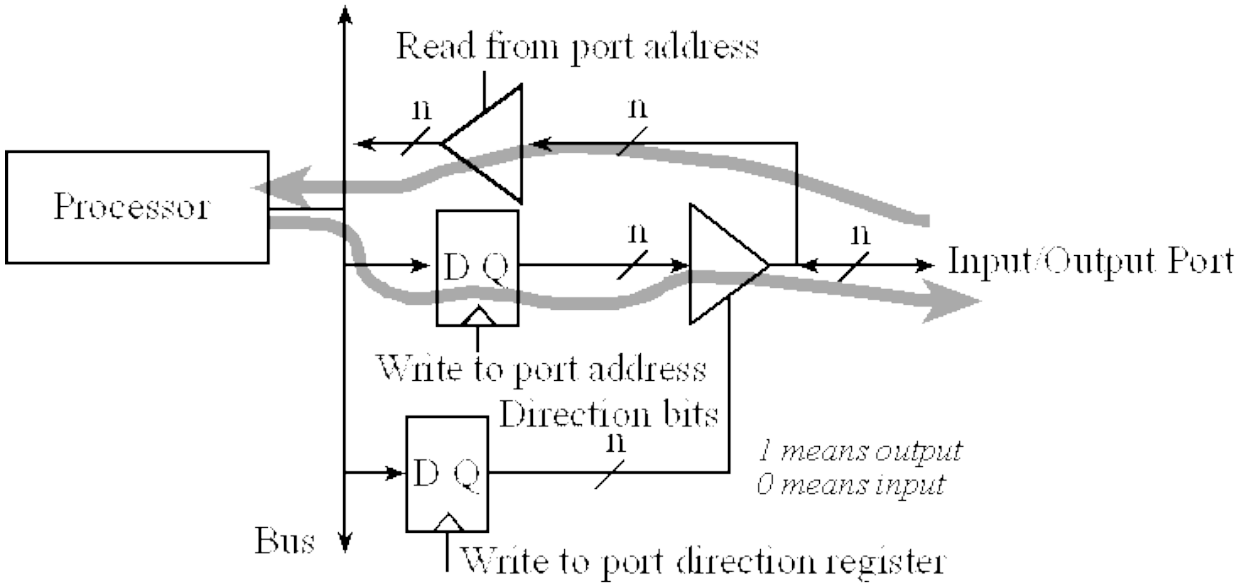


Figure 6.7. A bidirectional port can be configured as a read-only input port or a readable output port.

Common Error: Many program errors can be traced to confusion between I/O ports and regular memory. For example, you should not write to an input port, and sometimes we cannot read from an output port.

6.3. I/O Programming and the Direction Register

On most embedded microcontrollers, the I/O ports are memory mapped. This means the software can access an input/output port simply by reading from or writing to the appropriate address. It is important to realize that even though I/O operations "look" like reads and writes to memory variables, the I/O ports often DO NOT act like memory. For example, some bits are read-only, some are write-only, some can only be cleared, others can only be set, and some bits cannot be modified. To make our software more readable we include symbolic definitions for the I/O ports. We set the direction register (e.g., **`GPIO_PORTF_DIR_R`**) to specify which pins are input and which are output. Individual port pins can be general purpose I/O (GPIO) or have an alternate function. We will set bits in the alternate function register (e.g., **`GPIO_PORTF_AFSEL_R`**) when we wish to activate the alternate functions listed in Table 6.1. For each I/O pin we wish to use whether GPIO or alternate function we must enable the digital circuits by setting the bit in the enable register (e.g., **`GPIO_PORTF_DEN_R`**). Typically, we write to the direction and alternate function registers once during the initialization phase. We use the data register (e.g., **`GPIO_PORTF_DATA_R`**) to perform input/output on the port. Conversely, we read and write the data register multiple times to perform input and output respectively during the running phase. Table 6.2 shows some of the parallel port registers for the LM4F120/TM4C123. The only differences among the Stellaris and Tiva families are the number of ports and available pins in each port.

Address	7	6	5	4	3	2	1	0	Name
$400F.E108	--	--	GPIOF	GPIOE	GPIOD	GPIOC	GPIOB	GPIOA	SYSCTL_RCGC2_R
$4000.43FC	DATA	DATA	DATA	DATA	DATA	DATA	DATA	DATA	GPIO_PORTA_DATA_R
$4000.4400	DIR	DIR	DIR	DIR	DIR	DIR	DIR	DIR	GPIO_PORTA_DIR_R
$4000.4420	SEL	SEL	SEL	SEL	SEL	SEL	SEL	SEL	GPIO_PORTA_AFSEL_R
$4000.4510	PUE	PUE	PUE	PUE	PUE	PUE	PUE	PUE	GPIO_PORTA_PUR_R
$4000.451C	DEN	DEN	DEN	DEN	DEN	DEN	DEN	DEN	GPIO_PORTA_DEN_R
$4000.4524	1	1	1	1	1	1	1	1	GPIO_PORTA_CR_R
$4000.4528	0	0	0	0	0	0	0	0	GPIO_PORTA_AMSEL_R
$4000.53FC	DATA	DATA	DATA	DATA	DATA	DATA	DATA	DATA	GPIO_PORTB_DATA_R
$4000.5400	DIR	DIR	DIR	DIR	DIR	DIR	DIR	DIR	GPIO_PORTB_DIR_R
$4000.5420	SEL	SEL	SEL	SEL	SEL	SEL	SEL	SEL	GPIO_PORTB_AFSEL_R
$4000.5510	PUE	PUE	PUE	PUE	PUE	PUE	PUE	PUE	GPIO_PORTB_PUR_R
$4000.551C	DEN	DEN	DEN	DEN	DEN	DEN	DEN	DEN	GPIO_PORTB_DEN_R

$4000.5524	1	1	1	1	1	1	1	1	GPIO_PORTB_CR_R
$4000.5528	0	0	AMSEL	AMSEL	0	0	0	0	GPIO_PORTB_AMSEL_R
$4000.63FC	DATA	DATA	DATA	DATA	JTAG	JTAG	JTAG	JTAG	GPIO_PORTC_DATA_R
$4000.6400	DIR	DIR	DIR	DIR	JTAG	JTAG	JTAG	JTAG	GPIO_PORTC_DIR_R
$4000.6420	SEL	SEL	SEL	SEL	JTAG	JTAG	JTAG	JTAG	GPIO_PORTC_AFSEL_R
$4000.6510	PUE	PUE	PUE	PUE	JTAG	JTAG	JTAG	JTAG	GPIO_PORTC_PUR_R
$4000.651C	DEN	DEN	DEN	DEN	JTAG	JTAG	JTAG	JTAG	GPIO_PORTC_DEN_R
$4000.6524	1	1	1	1	JTAG	JTAG	JTAG	JTAG	GPIO_PORTC_CR_R
$4000.6528	AMSEL	AMSEL	AMSEL	AMSEL	JTAG	JTAG	JTAG	JTAG	GPIO_PORTC_AMSEL_R
$4000.73FC	DATA	DATA	DATA	DATA	DATA	DATA	DATA	DATA	GPIO_PORTD_DATA_R
$4000.7400	DIR	DIR	DIR	DIR	DIR	DIR	DIR	DIR	GPIO_PORTD_DIR_R
$4000.7420	SEL	SEL	SEL	SEL	SEL	SEL	SEL	SEL	GPIO_PORTD_AFSEL_R
$4000.7510	PUE	PUE	PUE	PUE	PUE	PUE	PUE	PUE	GPIO_PORTD_PUR_R
$4000.751C	DEN	DEN	DEN	DEN	DEN	DEN	DEN	DEN	GPIO_PORTD_DEN_R
$4000.7524	CR	1	1	1	1	1	1	1	GPIO_PORTD_CR_R
$4000.7528	0	0	AMSEL	AMSEL	AMSEL	AMSEL	AMSEL	AMSEL	GPIO_PORTD_AMSEL_R
$4002.43FC			DATA	DATA	DATA	DATA	DATA	DATA	GPIO_PORTE_DATA_R
$4002.4400			DIR	DIR	DIR	DIR	DIR	DIR	GPIO_PORTE_DIR_R
$4002.4420			SEL	SEL	SEL	SEL	SEL	SEL	GPIO_PORTE_AFSEL_R
$4002.4510			PUE	PUE	PUE	PUE	PUE	PUE	GPIO_PORTE_PUR_R
$4002.451C			DEN	DEN	DEN	DEN	DEN	DEN	GPIO_PORTE_DEN_R
$4002.4524			1	1	1	1	1	1	GPIO_PORTE_CR_R
$4002.4528			AMSEL	AMSEL	AMSEL	AMSEL	AMSEL	AMSEL	GPIO_PORTE_AMSEL_R
$4002.53FC				DATA	DATA	DATA	DATA	DATA	GPIO_PORTF_DATA_R
$4002.5400				DIR	DIR	DIR	DIR	DIR	GPIO_PORTF_DIR_R
$4002.5420				SEL	SEL	SEL	SEL	SEL	GPIO_PORTF_AFSEL_R
$4002.5510				PUE	PUE	PUE	PUE	PUE	GPIO_PORTF_PUR_R
$4002.551C				DEN	DEN	DEN	DEN	DEN	GPIO_PORTF_DEN_R
$4002.5524				1	1	1	1	CR	GPIO_PORTF_CR_R
$4002.5528				0	0	0	0	0	GPIO_PORTF_AMSEL_R

	31-28	27-24	23-20	19-16	15-12	11-8	7-4	3-0	
$4000.452C	PMC7	PMC6	PMC5	PMC4	PMC3	PMC2	PMC1	PMC0	GPIO_PORTA_PCTL_R
$4000.552C	PMC7	PMC6	PMC5	PMC4	PMC3	PMC2	PMC1	PMC0	GPIO_PORTB_PCTL_R
$4000.652C	PMC7	PMC6	PMC5	PMC4	0x1	0x1	0x1	0x1	GPIO_PORTC_PCTL_R
$4000.752C	PMC7	PMC6	PMC5	PMC4	PMC3	PMC2	PMC1	PMC0	GPIO_PORTD_PCTL_R
$4002.452C			PMC5	PMC4	PMC3	PMC2	PMC1	PMC0	GPIO_PORTE_PCTL_R
$4002.552C				PMC4	PMC3	PMC2	PMC1	PMC0	GPIO_PORTF_PCTL_R
$4000.6520	LOCK (write 0x4C4F434B to unlock, other locks) (reads 1 if locked, 0 if unlocked)								GPIO_PORTC_LOCK_R
$4000.7520	LOCK (write 0x4C4F434B to unlock, other locks) (reads 1 if locked, 0 if unlocked)								GPIO_PORTD_LOCK_R
$4002.5520	LOCK (write 0x4C4F434B to unlock, other locks) (reads 1 if locked, 0 if unlocked)								GPIO_PORTF_LOCK_R

Table 6.2 Some TM4C123 parallel ports. Each register is 32 bits wide. For PMCx bits, see Table 6.1. JTAG means do not use these pins and do not change any of these bits.

To initialize an I/O port for general use we perform seven steps. Steps two through four are needed only for the LM4F/TM4C microcontrollers. First, we activate the clock for the port. Second, we unlock the port; unlocking is needed only for pins PC3-0, PD7, PF0 on the LM4F and TM4C. Third, we disable the analog function of the pin, because we will be using the pin for digital I/O. Fourth, we clear bits in the **PCTL** (Table 6.1) to select regular digital function. Fifth, we set its direction register. Sixth, we clear bits in the alternate function register, and lastly, we enable the digital port. We need to add a short delay between activating the clock and accessing the port registers. The direction register specifies bit for bit whether the corresponding pins are input or output. A **DIR** bit of 0 means input and 1 means output.

Common Error: You will get a bus fault if you access a port without enabling its clock.

In this first example we will make PF4 and PF0 input, and we will make PF3 PF2 and PF1 output, as shown in Program 6.1. To use a port we first must activate its clock in the **`SYSCTL_RCGC2_R`** register. The second step is to unlock the port, by writing a special value to the **LOCK** register, followed by setting bits in the **CR** register. Only PC3-0, PD7, and PF0 on the TM4C need to be unlocked. All the

other bits on the TM4C are always unlocked. The third step is to disable the analog functionality, by clearing bits in the **AMSEL** register. The fourth step is to select GPIO functionality, by clearing bits in the **PCTL** register, as described in Table 6.1. The fifth step is to specify whether the pin is an input or an output by clearing or setting bits in the **DIR** register. Because we are using the pins as regular digital I/O, the sixth step is to clear the corresponding bits in the **AFSEL** register. The last step is to enable the corresponding I/O pins by writing ones to the **DEN** register. To run this example on the LaunchPad, we also set bits in the **PUR** register for the two switch inputs (Figure 6.3) to have an internal pull-up resistor.

When the software reads from location 0x400253FC, the bottom 8 bits are returned with the current values on Port F. The top 24 bits are returned zero. As shown in Figure 6.7, when reading an I/O port, the input pins show the current digital state, and the output pins show the value last written to the port. The function **`PortF_Input`** will read from the five input pins, and return a value depending on the current status of the inputs. As shown in Figure 6.7, when writing to an I/O port, the input pins are not affected, and the output pins are changed to the value written to the port. The function **`PortF_Output`** will write new values to the three output pins. The **`#include`** will define symbolic names for all the I/O ports for that microcontroller. The header file **tm4c123ge6pm.h** can be found in the **inc** folder.

```
#include "tm4c123ge6pm.h"
unsigned long In; // input from PF4
unsigned long Out; // output to PF2 (blue LED)
//Function Prototypes
void PortF_Init(void);
// 3. Subroutines Section
// MAIN: Mandatory for a C Program to be executable
int main(void){ // initialize PF0 and PF4 and make them inputs
PortF_Init(); // make PF3-1 out (PF3-1 built-in LEDs)
while(1){
In = GPIO_PORTF_DATA_R&0x10; // read PF4 into Sw1
In = In>>2; // shift into position PF2
Out = GPIO_PORTF_DATA_R;
Out = Out&0xFB;
Out = Out|In;
GPIO_PORTF_DATA_R = Out; // output
}
}
// Subroutine to initialize port F pins for input and output
// PF4 is input SW1 and PF2 is output Blue LED
// Inputs: None
// Outputs: None
// Notes: ...
void PortF_Init(void){ volatile unsigned long delay;
SYSCTL_RCGC2_R |= 0x00000020; // 1) activate clock for Port F
delay = SYSCTL_RCGC2_R; // allow time for clock to start
GPIO_PORTF_LOCK_R = 0x4C4F434B; // 2) unlock GPIO Port F
GPIO_PORTF_CR_R = 0x1F; // allow changes to PF4-0
// only PF0 needs to be unlocked, other bits can't be locked
GPIO_PORTF_AMSEL_R = 0x00; // 3) disable analog on PF
GPIO_PORTF_PCTL_R = 0x00000000; // 4) PCTL GPIO on PF4-0
GPIO_PORTF_DIR_R = 0x0E; // 5) PF4,PF0 in, PF3-1 out
GPIO_PORTF_AFSEL_R = 0x00; // 6) disable alt funct on PF7-0
GPIO_PORTF_PUR_R = 0x11; // enable pull-up on PF0 and PF4
GPIO_PORTF_DEN_R = 0x1F; // 7) enable digital I/O on PF4-0
}
```

Program 6.1. A set of functions using PF4,PF0 as inputs and PF3-1 as outputs (C6_InputOutputxxx.zip).

Checkpoint 6.3: Does the entire port need to be defined as input or output, or can some pins be input while others are output

Checkpoint 6.4: How do we change Program 6.1 to run using Port A

In Program 6.1 the assumption was the software module had access to all of Port F. In other words, this software owned all pins of Port F. In most cases, a software module needs access to only some of the port pins. If two or more software modules access the same port, a conflict will occur if one module changes modes or output values set by another module. It is good software design to write **friendly** software, which only affects the individual pins as needed. Friendly software does not change the other bits in a shared register. Conversely, **unfriendly** software modifies more bits of a register than it needs to. The difficulty of unfriendly code is each module will run properly when tested by itself, but weird bugs result when two or more modules are combined.

Consider the problem that a software module needs to output to just Port A bit 7. After enabling the clock for Port A, we use read-modify-write software to initialize just pin 7. The following initialization does not modify the configurations for the other 7 bits in Port A. Unlocking is not required for PA7 (just PD7 and PF0 require unlocking)

```
SYSCTL_RCGC2_R |= 0x00000001; // 1) activate clock for Port A
delay = SYSCTL_RCGC2_R; // allow time for clock to start
GPIO_PORTA_AMSEL_R &= ~0x80; // 3) disable analog on PA7
GPIO_PORTA_PCTL_R &= ~0xF0000000; // 4) PCTL GPIO on PA7
GPIO_PORTA_DIR_R |= 0x80; // 5) PA7 out
GPIO_PORTA_AFSEL_R &= ~0x80; // 6) disable alt funct on PA7
GPIO_PORTA_DEN_R |= 0x80; // 7) enable digital I/O on PA7
```

There is no conflict if two or more modules enable the clock for Port A. There are two ways on LM4F/TM4C microcontrollers to access individual port bits. The first method is to use read-modify-write software to change just one pin. A read-or-write sequence can be used to set bits.

```
LDR R1, =GPIO_PORTA_DATA_R
LDR R0, [R1]       ; previous
ORR R0, R0, #0x80 ; set bit 1
STR R0, [R1]
```

```
// make PA7 high
GPIO_PORTA_DATA_R |= 0x80;
```

A read-and-write sequence can be used to clear one or more bits.

```
LDR R1, =GPIO_PORTA_DATA_R
LDR R0, [R1]       ; previous
BIC R0, R0, #0x80 ; clear bit 1
STR R0, [R1]
```

```
// make PA7 low
GPIO_PORTA_DATA_R& = ~0x80;
```

The second method uses the **bit-specific addressing**. The LM4F/TM4C family implements a flexible way to access port pins. This bit-specific addressing doesn't work for all the I/O registers, just the parallel port data registers. The mechanism allows collective access to 1 to 8 bits in a data port. We define eight address offset constants in Table 6.3. Basically, if we are interested in bit *b*, the constant is $4*2^b$. There 256 possible bit combinations we might be interested in accessing, from all of them to none of them. Each possible bit combination has a separate address for accessing that combination. For each bit we are interested in, we add up the corresponding constants from Table 6.3 and then add that sum to the base address for the port. The base addresses for the data ports can be found Table 6.2. For example, assume we are interested in Port A bits 1, 2, and 3. The base address for Port A is 0x4000.4000, and the constants are 0x0008, 0x0010, and 0x0020. The sum of 0x4000.4000+0x0008+0x0010 +0x0020 is the address 0x4000.4038. If we read from 0x4000.4038 only bits 1, 2, and 3 will be returned. If we write to this address only bits 1, 2, and 3 will be modified.

If we wish to access bit	*Constant*
7	0x0200
6	0x0100
5	0x0080
4	0x0040
3	0x0020
2	0x0010
1	0x0008
0	0x0004

Table 6.3. Address offsets used to specify individual data port bits.

The base address for Port A is 0x4000.4000. If we want to read and write all 8 bits of this port, the constants will add up to 0x03FC. Notice that the sum of the base address and the constants yields the 0x4000.43FC address used in Table 6.2 and Program 6.1. In other words, read and write operations to **GPIO_PORTA_DATA_R** will access all 8 bits of Port A. If we are interested in just bit 5 of Port A, we add 0x0080 to 0x4000.4000, and we can define this in C and in assembly as

```
#define PA5   (*((volatile unsigned long *)0x40004080))
PA5 EQU 0x40004080
```

Now, a simple write operation can be used to set **PA5**. The following code is friendly because it does not modify the other 7 bits of Port A.

```
PA5 = 0x20; // make PA5 high
```

A simple write sequence will clear **PA5**. The following code is also friendly.

```
PA5 = 0x00; // make PA5 low
```

A read from **PA5** will return 0x20 or 0x00 depending on whether the pin is high or low, respectively. If **PA5** is an output, the following code is also friendly.

```
PA5 = PA5^0x20; // toggle PA5
```

Note that the base address when computing the bit-specific address for PortA is 0x40004000, the following table lists the base addresses for the other ports.

Port	**Base address**
PortA	0x40004000
PortB	0x40005000
PortC	0x40006000
PortD	0x40007000
PortE	0x40024000
PortF	0x40025000

Table 6.4. Base Addresses for bit-specific addressing of ports A-F

Checkpoint 6.5: What happens if we write to location 0x4000.4000?
Checkpoint 6.6: Specify a #define that allows us to access bits 7 and 1 of Port A. Use this #define to make both bits 7 and 1 of Port A high.
Checkpoint 6.7: Specify a #define that allows us to access bits 6, 1, 0 of Port B. Use this #define to make bits 6, 1 and 0 of Port B high.

To understand the port definitions in C, we remember **#define** is simply a copy paste. E.g.,

```
data = PA5;
```

becomes
```
data = (*((volatile unsigned long *)0x40004080));
```

To understand why we define ports this way, let's break this port definition into pieces. First, 0x40004080 is the address of Port A bit 5. If we write just **`#define PA5 0x40004080`** it will create
```
data = 0x40004080;
```

which does not read the contents of PA5 as desired. This means we need to dereference the address. If we write **`#define PA5 (*0x40004080)`** it will create
```
data = (*0x40004080);
```

This will attempt to read the contents at 0x40004080, but doesn't know whether to read 8 16 or 32 bits. So the compiler gives a syntax error because the type of data does not match the type of (*0x40004080). To solve a type mismatch in C we **typecast**, placing a (new type) in front of the object we wish to convert. We wish force the type conversion to unsigned 32 bits, so we modify the definition to include the typecast,
```
#define PA5   (*((volatile unsigned long *)0x40004080))
```

The **`volatile`** is added because the value of a port can change beyond the direct action of the software. It forces the C compiler to read a new value each time through a loop and not rely on the previous value.

6.4. Debugging monitor using an LED

One of the important tasks in debugging a system is to observe when and where our software is executing. A debugging tool that works well for real-time systems is the monitor. In a real-time system, we need the execution time of the debugging tool to be small compared to the execution time of the program itself. **Intrusiveness** is defined as the degree to which the debugging code itself alters the performance of the system being tested. A monitor is an independent output process, somewhat similar to the print statement, but one that executes much faster and thus is much less intrusive. An LED attached to an output port of the microcontroller is an example of a BOOLEAN monitor. You can place LEDs on unused output pins. Software toggles these LEDs to let you know where and when your program is running. Assume an LED is attached to Port F bit 2. Program 6.2 will toggle the LED. We create a bit-specific address constant to access just PF2:

```
PF2 EQU 0x40025010
Toggle
  LDR R1, =PF2
  LDR R0, [R1]
  EOR R0, R0, #0x04
  STR R0, [R1]
  BX  LR
```

```
#define PF2 (*((volatile unsigned long *)0x40025010))
void Toggle(void){
  PF2 ^= 0x04; // toggle LED
}
```

Program 6.2. An LED monitor.

A **heartbeat** is a pulsing output that is not required for the correct operation of the system, but it is useful to see while the program is running. In particular, you add **`BL Toggle`** statements at strategic places within your system. It only takes 13 bus cycles to execute. Port G must be initialized so that bit 2 is an output before the debugging begins. You can either observe the LED directly or look at the LED control signals with a high-speed oscilloscope or logic analyzer. An LCD can be an effective monitor for small amounts of information. Inexpensive LCDs can display from 8 to 160 characters. Unfortunately, it takes about 50 μs to output each character, so the use of an LCD monitor might be intrusive. When using LED monitors it is better to modify just the one bit, leaving the other 7 as is. In this way, you can have additional LED monitors.

6.5. Hardware debugging tools

Microcomputer related problems often require the use of specialized equipment to debug the system hardware and software. Two very useful tools are the **logic analyzer** and the oscilloscope. A logic analyzer is essentially a multiple channel digital storage scope with many ways to trigger, see Figure 6.8. As a troubleshooting aid, it allows the experimenter to observe numerous digital signals at various points in time and thus make decisions based upon such observations. As with any debugging process, it is necessary to select which information to observe out of a vast set of possibilities. Any digital signal in the system can be connected to the logic analyzer. Figure 6.8 shows an 8-channel logic analyzer, but real devices can support 128 or more channels. One problem with logic analyzers is the massive amount of information that it generates. With logic analyzers (similar to other debugging techniques) we must strategically select which signals in the digital interfaces to observe and when to observe them. In particular, the triggering mechanism can be used to capture data at appropriate times eliminating the need to sift through volumes of output. Sometimes there are extra I/O pins on the microcontroller, not needed for the normal operation of the system (shown as the bottom two wires in Figure 6.8). In this case, we can connect the pins to a logic analyzer, and add software debugging instruments that set and clear these pins at strategic times within the software. In this way we can visualize the hardware/software timing.

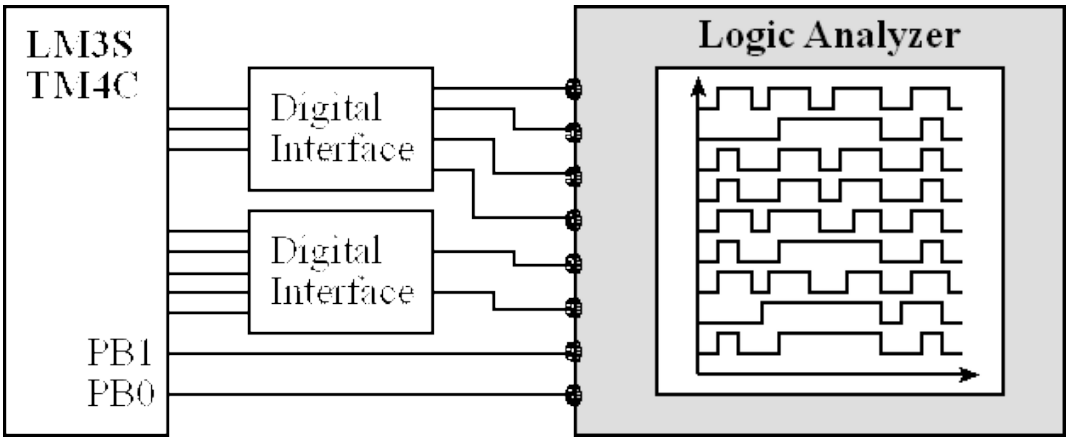


Figure 6.8. A logic analyzer and example output.

An **oscilloscope** can be used to capture voltage versus time data. You can adjust the voltage range and time scale. The oscilloscope trigger is how and when the data will be capture. In normal mode, we measure patterns that repeat over and over, and we use the trigger (e.g., rising edge of channel 1) to freeze the image. In single shot mode, the display is initially blank, and once the trigger occurs, one trace is captured and display.

6.6. Chapter 6 Quiz

6.1 To make a pin a digital input, what value do you load into corresponding bits the following registers. Assume it does not need an internal pullup

DIR
PUR
PCTL
AFSEL
AMSEL
DEN

6.2 To make a pin a digital output, what value do you load into corresponding bits the following registers. Assume it does not need an internal pullup

DIR
PUR
PCTL
AFSEL
AMSEL
DEN

6.3 Which line of C code is a friendly way to set Port B bit 2 assuming this pin has already been initialized as an output

```
GPIO_PORTB_DATA_R = 0x00;
GPIO_PORTB_DATA_R = 0x02;
GPIO_PORTB_DATA_R = 0x04;
GPIO_PORTB_DATA_R |= 0x02;
GPIO_PORTB_DATA_R |= 0x04;
GPIO_PORTB_DATA_R &= 0x02;
GPIO_PORTB_DATA_R &= 0x04;
GPIO_PORTB_DATA_R &= ~0x02;
GPIO_PORTB_DATA_R &= ~0x04;
```

6.4 Which line of C code is a friendly way to clear Port B bit 2 assuming this pin has already been initialized as an output

```
GPIO_PORTB_DATA_R = 0x00;
GPIO_PORTB_DATA_R = 0x02;
GPIO_PORTB_DATA_R = 0x04;
GPIO_PORTB_DATA_R |= 0x02;
GPIO_PORTB_DATA_R |= 0x04;
GPIO_PORTB_DATA_R &= 0x02;
GPIO_PORTB_DATA_R &= 0x04;
GPIO_PORTB_DATA_R &= ~0x02;
GPIO_PORTB_DATA_R &= ~0x04;
```

6.5 Which debugging instrument can measure voltage versus time?

Heart Beat
Oscilloscope
Logic analyzer
LED

6.6 Which debugging instrument can measure multiple digital signals versus time?

Heart beat
Oscilloscope
Logic analyzer
LED

Chapter 7: Design and Development
Embedded Systems - Shape the World
Jonathan Valvano and Ramesh Yerraballi

In this chapter, we will begin by presenting a general approach to modular design. In specific, we will discuss how to organize software blocks in an effective manner. The ultimate success of an embedded system project depends both on its software and hardware. Computer scientists pride themselves in their ability to develop quality software. Similarly electrical engineers are well-trained in the processes to design both digital and analog electronics. Manufacturers, in an attempt to get designers to use their products, provide application notes for their hardware devices. The main objective of this class is to combine effective design processes together with practical software techniques in order to develop quality embedded systems. As the size and especially the complexity of the software increase, the software development changes from simple "coding" to "software engineering", and the required skills also vary along this spectrum. These software skills include modular design, layered architecture, abstraction, and verification. Real-time embedded systems are usually on the small end of the size scale, but never the less these systems can be quite complex. Therefore, both hardware and software skills are essential for developing embedded systems. Writing good software is an art that must be developed, and cannot be added on at the end of a project. Just like any other discipline (e.g., music, art, science, religion), expertise comes from a combination of study and practice. The watchful eye of a good mentor can be invaluable, so take the risk and show your software to others inviting praise and criticism. Good software combined with average hardware will always outperform average software on good hardware. In this chapter we will introduce some techniques for developing quality software.

Learning Objectives:

- Understand system development process as a life cycle
- Take Requirements and formulate a problem statement.
- Learn that an algorithm is a formal way to describe a solution
- Define an algorithm with pseudo code or visually as a flowchart
- Translate flowchart to code
- Test in simulator (Test → Write code → Test → Write code … cycle)
- Run on real board

7.1. Product Life 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 7.1, the development of a product follows an analysis-design-implementation-testing-deployment cycle. For complex systems with long life-spans, we transverse multiple times around the life cycle. For simple systems, a one-time pass may suffice.

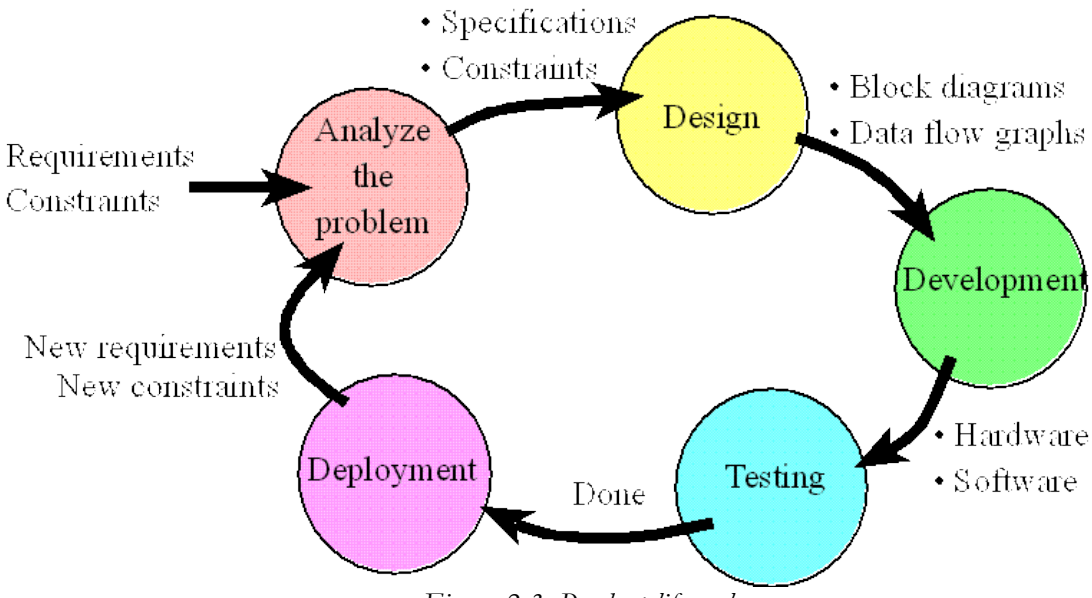


Figure 2.3. Product life cycle.

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 motor controller. During the analysis phase, we would determine obvious specifications such as range, stability, accuracy, and response time. There may be less obvious requirements to satisfy, such as weight, size, battery life, product life, ease of operation, display readability, and reliability. Often, improving the performance on one parameter can be achieved only by decreasing the performance of another. This art of compromise defines the tradeoffs an engineer must make when designing a product. A **constraint** is a limitation, within which the system must operate. The system may be constrained to such factors as cost, safety, compatibility with other products, use of specific electronic and mechanical parts as other devices, interfaces with other instruments and test equipment, and development schedule. The following measures are often considered during the analysis phase of a project:

Safety: The risk to humans or the environment
Accuracy: The difference between the expected truth and the actual parameter
Precision: The number of distinguishable measurements
Resolution: The smallest change that can be reliably detected
Response time: The time between a triggering event and the resulting action
Bandwidth: The amount of information processed per time
Maintainability: The flexibility with which the device can be modified
Testability: The ease with which proper operation of the device can be verified
Compatibility: The conformance of the device to existing standards
Mean time between failure: The reliability of the device, the life of a product
Size and weight: The physical space required by the system
Power: The amount of energy it takes to operate the system
Nonrecurring engineering cost (NRE cost): The one-time cost to design and test
Unit cost: The cost required to manufacture one additional product
Time-to-prototype: The time required to design, build, and test an example system
Time-to-market: The time required to deliver the product to the customer
Human factors: The degree to which our customers like/appreciate the product

Checkpoint 7.1: What's the difference between a requirement and a specification?

The following is one possible outline of a **Requirements Document**. IEEE publishes a number of templates that can be used to define a project (IEEE STD 830-1998). A requirements document states what the system will do. It does not state how the system will do it. The main purpose of a requirements document is to serve as an agreement between you and your clients describing what the system will do. This agreement can become a legally binding contract. Write the document so that it is easy to read and understand by others. It should be unambiguous, complete, verifiable, and modifiable.

1. Overview
   1.1. Objectives: Why are we doing this project? What is the purpose?
   1.2. Process: How will the project be developed?
   1.3. Roles and Responsibilities: Who will do what? Who are the clients?
   1.4. Interactions with Existing Systems: How will it fit in?
   1.5. Terminology: Define terms used in the document.
   1.6. Security: How will intellectual property be managed?
2. Function Description
   2.1. Functionality: What will the system do precisely?
   2.2. Scope: List the phases and what will be delivered in each phase.
   2.3. Prototypes: How will intermediate progress be demonstrated?
   2.4. Performance: Define the measures and describe how they will be determined.
   2.5. Usability: Describe the interfaces. Be quantitative if possible.
   2.6. Safety: Explain any safety requirements and how they will be measured.
3. Deliverables
   3.1. Reports: How will the system be described?
   3.2. Audits: How will the clients evaluate progress?
   3.3. Outcomes: What are the deliverables? How do we know when it is done?

Observation: To build a system without a requirements document means you are never wrong, but never done.

When we begin the **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 into modules or subcomponents. 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 hiding the details of how it works. Issues such as safety (e.g., Isaac Asimov's first Law of Robotics "*A robot may not harm a human being, or, through inaction, allow a human being to come to harm*") and testing (e.g., we need to verify our system is operational) should be addressed during the high-level design. A data flow graph for a simple position measurement system is shown in Figure 7.2. The sensor converts position in an electrical resistance. The analog circuit converts resistance into the 0 to +3V voltage range required by the ADC. The ADC converts analog voltage into a digital sample. The **ADC driver**, using the ADC and timer hardware, collects samples and calculates voltages. The software converts voltage to position. Voltage and position data are represented as fixed-point numbers within the computer. The position data is passed to the **OLED driver** creating ASCII strings, which will be sent to the **organic light emitting diode** (OLED) module.

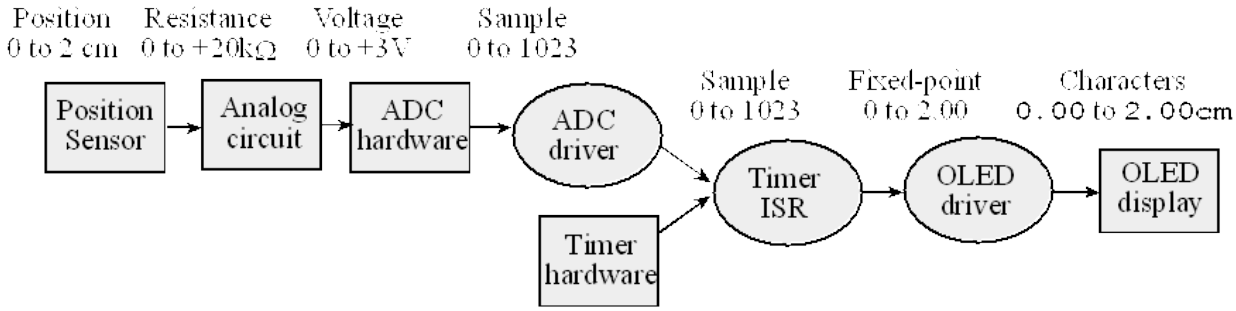


Figure 7.2. A data flow graph showing how the position signal passes through the system.

A preliminary design 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 build 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 throughout the class, include both the organization of information and mechanisms to access the data. Again safety and testing should be addressed during this low-level design.

A call graph for a simple position measurement system is shown in Figure 7.3. 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 7.3, 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 this book, 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 timer interrupt service routine (ISR) gets the next sample from the **ADC** software, converts it to position, and displays the result by calling the **OLED** interface software. The double-headed arrow between the ISR and the hardware means the hardware triggers the interrupt and the software accesses the hardware.

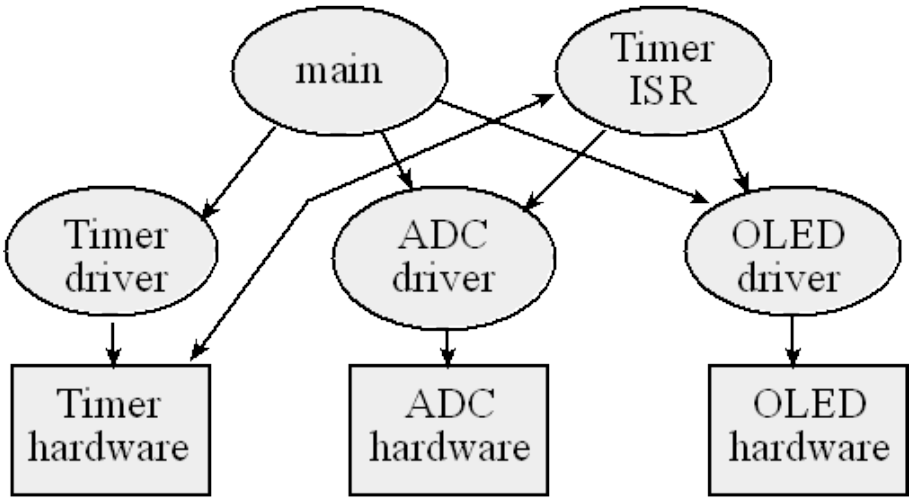


Figure 7.3. A call graph for a simple position measurement system.

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 involves **developing an implementation**. An advantage of a top-down design is that implementation of subcomponents can occur simultaneously. During the initial iterations of the life 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-deployment 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 though 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 evaluate 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 expected truth and measured), and stability (consistent operation.) Debugging techniques will be presented at the end of most chapters.

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 life cycle.

Figure 7.1 describes top-down design as a cyclic process, beginning with a problem statement and ending up with a solution. With a **bottom-up** design we begin with solutions and build up to a problem statement. Many innovations begin with an idea, "what if…?" In a bottom-up design, one begins with designing, building, and testing low-level components. The low-level designs can be developed in parallel. Bottom-up design may be inefficient because some subsystems may be designed, built, and tested, but never used. As the design progresses the components are fit together to make the system more and more complex. Only after the system is completely built and tested does one define the overall system specifications. The bottom-up design process allows creative ideas to drive the products a company develops. It also allows one to quickly test the feasibility of an idea. If one fully understands a problem area and the scope of potential solutions, then a top-down design will arrive at an effective solution most quickly. On the other hand, if one doesn't really understand the problem or the scope of its solutions, a bottom-up approach allows one to start off by learning about the problem.

7.2. Successive Refinement

Throughout the book in general, we discuss how to solve problems on the computer. In this section, we discuss the process of converting a problem statement into an algorithm. Later in the book, we will show how to map algorithms into assembly language. We begin with a set of general specifications, and then create a list of requirements and constraints. The general specifications describe the problem statement in an overview fashion, requirements define the specific things the system must do, and constraints are the specific things the system must not do. These requirements and constraints will guide us as we develop and test our system.

Observation: Sometimes the specifications are ambiguous, conflicting, or incomplete.
There are two approaches to the situation of ambiguous, conflicting, or incomplete specifications. The best approach is to resolve the issue with your supervisor or customer. The second approach is to make a decision and document the decision.

Performance Tip: If you feel a system specification is wrong, discuss it with your supervisor. We can save a lot of time and money by solving the correct problem in the first place.

Successive refinement, **stepwise refinement**, and **systematic decomposition** are three equivalent terms for a technique to convert a problem statement into a software algorithm. We start with a task and decompose the task into a set of simpler subtasks. Then, the subtasks are decomposed into even simpler sub-subtasks. We make progress as long as each subtask is simpler than the task itself. During the task decomposition we must make design decisions as the details of exactly how the task will be performed are put into place. Eventually, a subtask is so simple that it can be converted to software code. We can decompose a task in four ways, as shown in Figure 2.6. The **sequence**, **conditional**, and **iteration** are the three building blocks of structured programming. Because embedded systems often have real-time requirements, they employ a fourth building block called interrupts. We will implement time-critical tasks using interrupts, which are hardware-triggered software functions. Interrupts will be discussed in more detail in Chapters 9, 10, and 11. When we solve problems on the computer, we need to answer these questions:

• What does being in a state mean?	List the parameters of the state
• What is the starting state of the system?	Define the initial state
• What information do we need to collect?	List the input data
• What information do we need to generate?	List the output data
• How do we move from one state to another?	Specify actions we could perform
• What is the desired ending state?	Define the ultimate goal

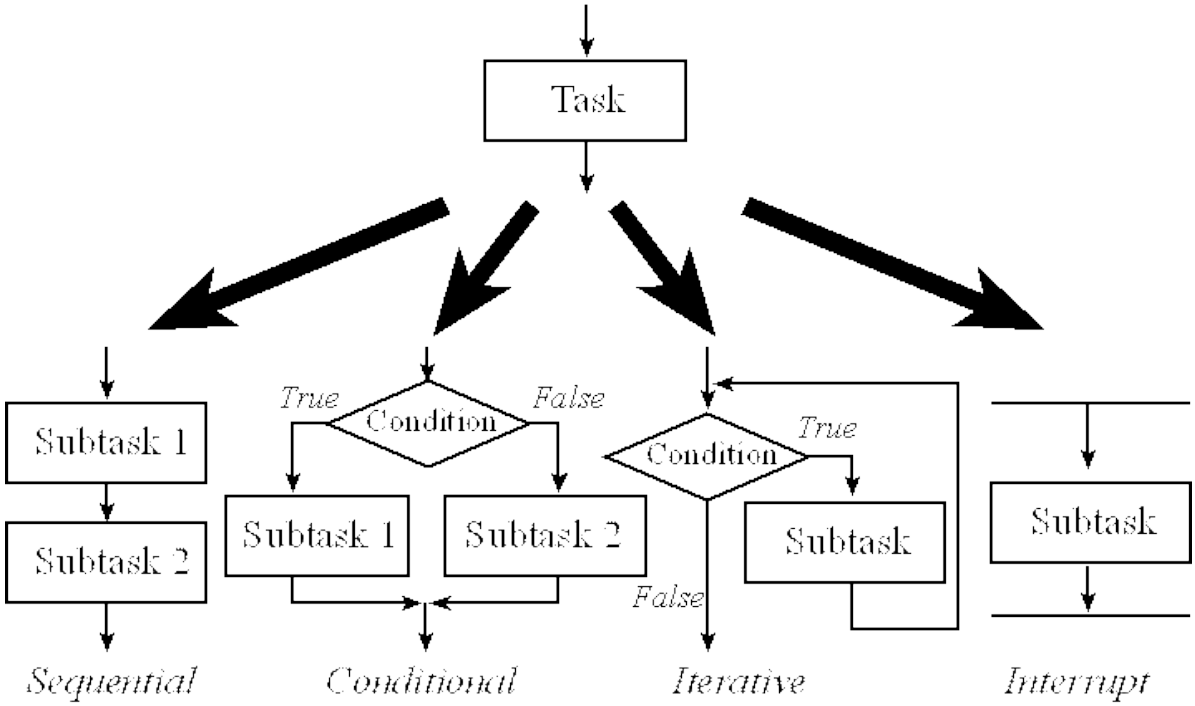


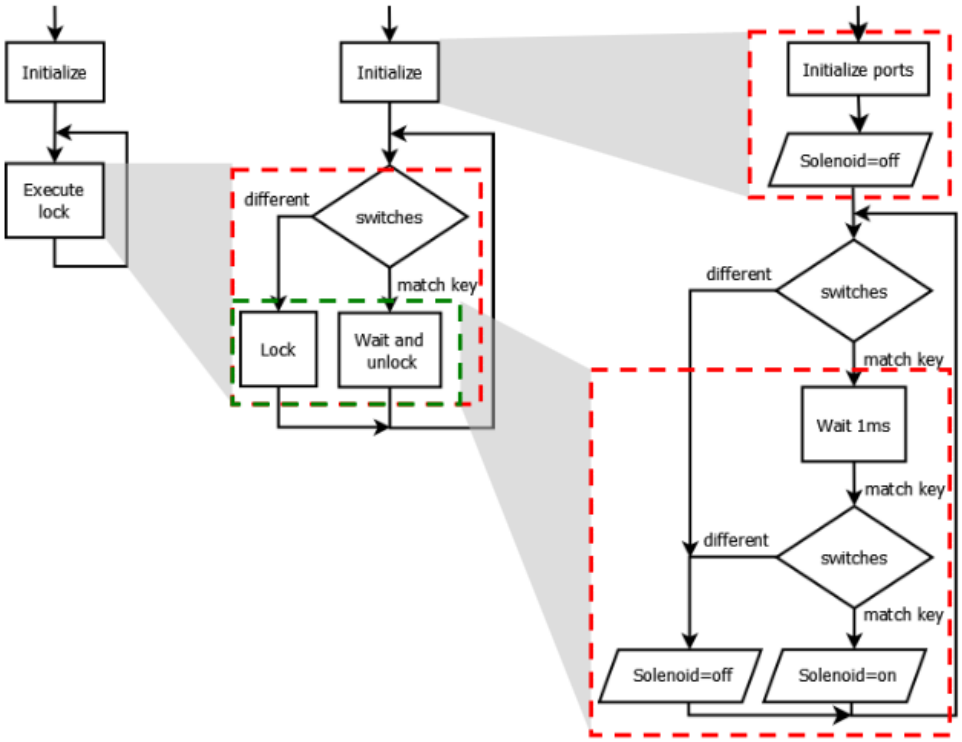
Figure 7.4. We can decompose a task using the building blocks of structured programming.

We need to recognize these phrases that translate to four basic building blocks:

- "do A then do B" → sequential
- "do A and B in either order" → sequential
- "if A, then do B" → conditional
- "for each A, do B" → iterative
- "do A until B" → iterative
- "repeat A over and over forever" → iterative (condition always true)
- "on external event do B" → interrupt
- "every t msec do B" → interrupt

Example 7.0. Build a digital door lock using seven switches.

The system has seven binary inputs from the switches and one binary output to the door lock. The **state** of this system is defined as "door locked" and "door unlocked". Initially, we want the door to be locked, which we can make happen by turning a solenoid off (make binary output low). If the 7-bit binary pattern on the switches matches a pre-defined keycode, then we want to unlock the door (make binary output high). Because the switches might bounce (flicker on and off) when changed, we will make sure the switches match the pre-defined keycode for at least 1 ms before unlocking the door. We can change states by writing to the output port for the solenoid. Like most embedded systems, there is no ending state. Once the switches no longer match the keycode the door will lock again. The first step in successive refinement is to divide the tasks into those performed once (Initialization), and those tasks repeated over and over (Execute lock), as shown as the left flowchart in the Interactive tool 7.0 As shown in the middle flow chart, we implement if the switches match the key, then unlock. If the switches do not match we will lock the door. To verify the user entered the proper keycode the switches must match, then match again after 1ms. There are two considerations when designing a system: security and safety. Notice that the system will lock the door if power is removed, because power applied to the solenoid will unlock the door. For safety reasons, there should be a mechanical way to unlock the door from the inside in case of emergency.



7.3. 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.

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.
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.

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 class focuses on real-time embedded systems written in 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 7.2: 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 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.

7.4. Functions, Procedures, Methods, and Subroutines

A program module that performs a well-defined task can be packaged up and defined as a single entity. Functions in that module can be invoked whenever a task needs to be performed. Object-oriented high-level languages like C++ and Java define program modules as methods. Functions and procedures are defined in some high-level languages like Pascal, FORTRAN, and Ada. In these languages, functions return a parameter and procedures do not. Most high-level languages however define program modules as functions, whether they return a parameter or not. A subroutine is the assembly language version of a function. Consequently, subroutines may or may not have input or output parameters. Formally, there are two components to a subroutine: definition and invocation. The subroutine **definition** specifies the task to be performed. In other words, it defines what will happen when executed. The syntax for an assembly subroutine begins with a label, which will be the name of the subroutine and ends with a return instruction. The definition of a subroutine includes a formal specification its input parameters and output parameters. In well-written software, the task performed by a subroutine will be well-defined and logically complete. The subroutine **invocation** is inserted to the software system at places when and where the task should be performed. We define software that invokes the subroutine as "the calling program" because it calls the subroutine. There are three parts to a subroutine invocation: pass input parameters, subroutine call, and accept output parameters. If there are input parameters, the calling program must establish the values for input parameters before it calls the subroutine. A **`BL`** instruction is used to call the subroutine. After the subroutine finishes, and if there are output parameters, the calling program accepts the return value(s). In this chapter, we will pass parameters using the registers. If the register contains a value, the parameter is classified as **call by value**. If the register contains an address, which points to the value, then the parameter is classified as **call by reference**.

Checkpoint 7.3: What is the difference between call by value and call by reference?

For example, consider a subroutine that samples the 12-bit ADC, as drawn in Figure 7.1. An analog input signal is connected to ADC0. The details of how the ADC works will be presented later in the class, but for now we focus on the defining and invoking subroutines. The execution sequence begins with the calling program setting up the input parameters. In this case, the calling program sets Register R0 equal to the channel number, **`MOV R0,#0`**. The instruction **`BL ADC_In`** will save the return address in the LR register and jump to the **`ADC_In`** subroutine. The subroutine performs a well-defined task. In this case, it takes the channel number in Register R0 and performs an analog to digital conversion, placing the digital representation of the analog input into Register R0. The **`BX LR`** instruction will move the return address into the PC, returning the execution thread to the instruction after the **`BL`** in the calling program. In this case, the output parameter in Register R0 contains the result of the ADC conversion. It is the responsibility of the calling program to accept the return parameter. In this case, it simply stores the result into variable **`n`**. In this example, both the input and output parameters are call by value.

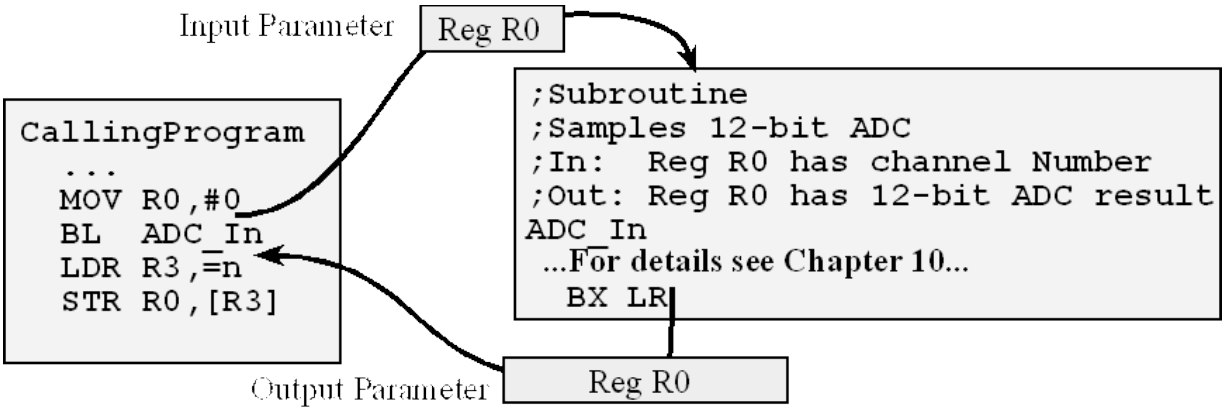


Figure 7.1. The calling program invokes the `ADC_In` subroutine passing parameters in registers.

The overall goal of modular programming is to enhance clarity. The smaller the task, the easier it will be to understand. **Coupling** is defined as the influence one module's behavior has on another module. In order to make modules more independent we strive to minimize coupling. Obvious and appropriate examples of coupling are the input/output parameters explicitly passed from one module to another. A quantitative measure of coupling is the number of bytes per second (bandwidth) that are transferred from one module to another. On the other hand, information stored in public global variables can be quite difficult to track. In a similar way, shared accesses to I/O ports can also introduce unnecessary complexity. Public global variables cause coupling between modules that complicate the debugging process because now the modules may not be able to be separately tested. On the other hand, we must use global variables to pass information into and out of an interrupt service routine and from one call to an interrupt service routine to the next call. When passing data into or out of an interrupt service routine, we group the functions that access the global into the same module, thereby making the global variable private. Another problem specific to embedded systems is the need for fast execution, coupled with the limited support for local variables. On many microcontrollers it is inefficient to implement local variables on the stack. Consequently, many programmers opt for the less elegant yet faster approach of global variables. Again, if we restrict access to these globals to function in the same module, the global becomes private. It is poor design to pass data between modules through public global variables; it is better to use a well-defined abstract technique like a FIFO queue.

We should assign a logically complete task to each module. The module is logically complete when it can be separated from the rest of the system and placed into another application. The interface design is extremely important. The interface to a module is the set of public functions that can be called and the formats for the input/output parameters of these functions. The interfaces determine the policies of our modules: "What does the module do?" In other words, the interfaces define the set of actions that can be initiated. The interfaces also define the coupling between modules. In general we wish to minimize the bandwidth of data passing between the modules yet maximize the number of modules. Of the following three objectives when dividing a software project into subtasks, it is really only the first one that matters

- Make the software project easier to understand
- Increase the number of modules
- Decrease the interdependency (minimize bandwidth between modules).

Checkpoint 7.4: List some examples of coupling.

We will illustrate the process of dividing a software task into modules with an abstract but realistic example. The overall goal of the example shown in Figure 7.2 is to sample data using an ADC, perform calculations on the data, and output results. The organic light emitting diode (OLED) could be used to display data to the external world. Notice the typical format of an embedded system in that it has some tasks performed once at the beginning, and it has a long sequence of tasks performed over and over. The

structure of this example applies to many embedded systems such as a diagnostic medical instrument, an intruder alarm system, a heating/AC controller, a voice recognition module, automotive emissions controller, or military surveillance system. The left side of Figure 7.2 shows the complex software system defined as a linear sequence of ten steps, where each step represents many lines of assembly code. The linear approach to this program follows closely to linear sequence of the processor as it executes instructions. This linear code, however close to the actual processor, is difficult to understand, hard to debug, and impossible to reuse for other projects. Therefore, we will attempt a modular approach considering the issues of functional abstraction, complexity abstraction, and portability in this example. The modular approach to this problem divides the software into three modules containing seven subroutines. In this example, assume the sequence Step4-Step5-Step6 causes data to be sorted. Notice that this sorting task is executed twice.

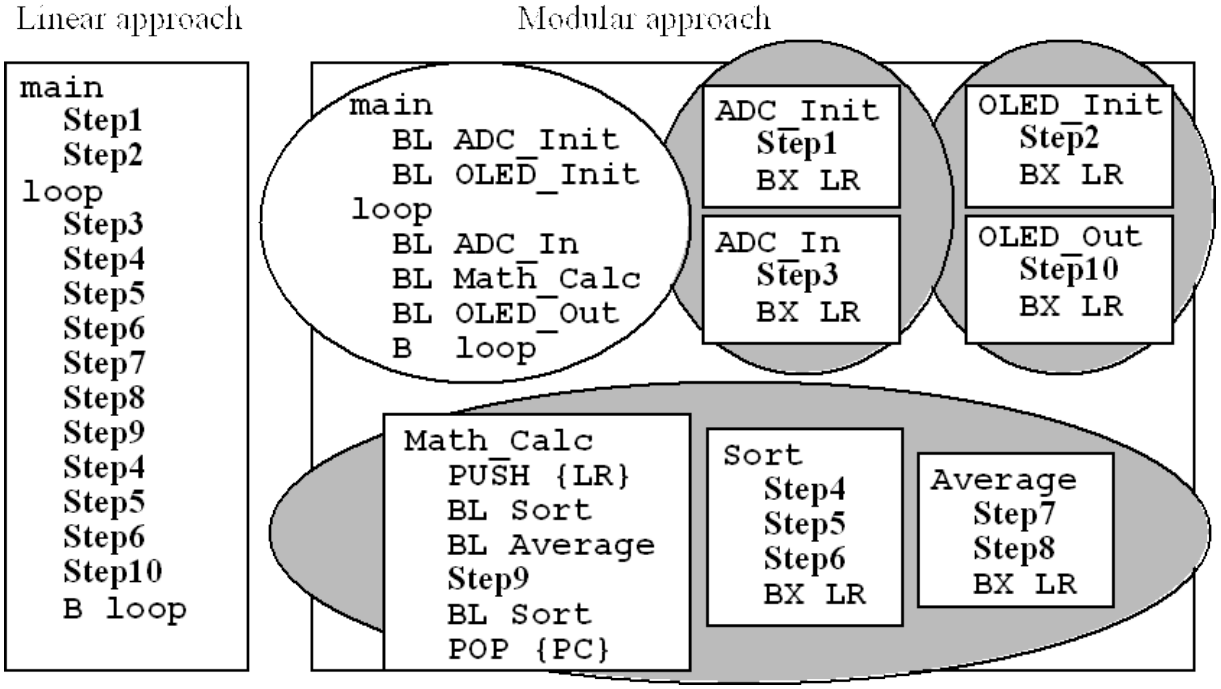


Figure 7.2. A complex software system is broken into three modules containing seven subroutines.

Functional abstraction encourages us to create a **`Sort`** subroutine allowing us to write the software once, but execute it from different locations. Complexity abstraction encourages us to organize the ten-step software into a main program with multiple modules, where each module has multiple subroutines. For example, assume the assembly instructions in Step1 cause the ADC to be initialized. Even though this code is executed only once, complexity abstraction encourages us to create an **`ADC_Init`** subroutine so the system is easier to understand and easier to debug. In a similar way assume Step2 initializes the OLED port, Step3 samples the ADC, the sequence Step7-Step8 performs an average, and Step10 outputs to the OLED. Therefore, each well-defined task is defined as a separate subroutine. The subroutines are then grouped into modules. For example, the ADC module is a collection of subroutines that operate the ADC. The complex behavior of the ADC is now abstracted into two easy to understand tasks: turn it on, and use it. In a similar way, the OLED module includes all functions that access the OLED. Again, at the abstract level of the main program, understanding how to use the OLED is a matter knowing we first turn it on then we transmit data. The math module is a collection of subroutines to perform necessary calculations on the data. In this example, we assume sort and average will be private subroutines, meaning they can be called only by software within the math module and not by software outside the module. Making private subroutines is an example of "information hiding", separating what the module does from how the module works. When we **port** a system, it means we take a working system and redesign it with some minor but critical change. The OLED device is used in this system to output results. We might be asked to port this system onto a device that uses an LCD in place of the OLED for its output. In this case, all we need to do is design, implement and test an LCD module with

two subroutines **LCD_Init** and **LCD_Out** that function in a similar manner as the existing OLED routines. The modular approach performs the exact same ten steps in the exact same order. However, the modular approach is easier to debug, because first we debug each subroutine, then we debug each module, and finally we debug the entire system. The modular approach clearly supports code reuse. For example, if another system needs an ADC, we can simply use the ADC module software without having to debug it again.

Observation: When writing modular code, notice its two-dimensional aspect. Down the y-axis still represents time as the program is executed, but along the x-axis we now visualize a functional block diagram of the system showing its data flow: input, calculate, output.

7.5. Making Decisions

The previous section presented fundamental concepts and general approaches to solving problems on the computer. In the subsequent sections, detailed implementations will be presented.

7.5.1. Conditional if-then Statements

Decision making is an important aspect of software programming. Two values are compared and certain blocks of program are executed or skipped depending on the results of the comparison. In assembly language it is important to know the precision (e.g., 8-bit, 16-bit, 32-bit) and the format of the two values (e.g., unsigned, signed). It takes three steps to perform a comparison. You begin by reading the first value into a register. If the second value is not a constant, it must be read into a register, too. The second step is to compare the first value with the second value. You can use either a subtract instruction with the **S** (**SUBS**) or a compare instruction (**CMP CMN**). The **CMP CMN SUBS** instructions set the condition code bits. The last step is a conditional branch.

Observation: Think of the three steps 1) bring first value into a register, 2) compare to second value, 3) conditional branch, **bxx** (where `xx` is `eq ne lo ls hi hs gt ge lt` or `le`). The branch will occur if (first is `xx` second).

In Programs 71 and 7.2, we assume **G** is a 32-bit unsigned variable. Program 7.1 contains two separate if-then structures involving testing for equal or not equal. It will call **GEqual7** if **G** equals **7**, and **GNotEqual7** if **G** does not equal **7**. When testing for equal or not equal it doesn't matter whether the numbers are signed or unsigned. However, it does matter if they are 8-bit or 16-bit. To convert these examples to 16 bits, use the **LDRH R0,[R2]** instruction instead of the **LDR R0,[R2]** instruction. To convert these examples to 8 bits, use the **LDRB R0,[R2]** instruction instead of the **LDR R0,[R2]** instruction.

Assembly code	C code

`    LDR R2, =G     ; R2 = &G`
`    LDR R0, [R2]   ; R0 = G`
`    CMP R0, #7     ; is G == 7 ?`
`    BNE next1      ; if not, skip`
`    BL  GEqual7    ; G == 7`
`next1`
	`unsigned long G;`
	`if(G ==  7){`
	`  GEqual7();`
	`}`

`    LDR R2, =G     ; R2 = &G`
`    LDR R0, [R2]   ; R0 = G`
`    CMP R0, #7     ; is G != 7 ?`
`    BEQ next2      ; if not, skip`
`    BL  GNotEqual7 ; G != 7`
`next2`
	`if(G != 7){`
	`  GNotEqual7();`
	`}`

Program 7.1. Conditional structures that test for equality (this works with signed and unsigned numbers).

When testing for greater than or less than, it does matter whether the numbers are signed or unsigned. Program 7.2 contains four separate unsigned if-then structures. In each case, the first step is to bring the first value in R0; the second step is to compare the first value with a second value; and the third step is to execute an unsigned branch **Bxx**. The branch will occur if the first unsigned value is **xx** the second unsigned value.

Assembly code	C code

`    LDR R2, =G      ; R2 =& G`
`    LDR R0, [R2]    ; R0 = G`
`    CMP R0, #7      ; is G > 7?`
`    BLS next1       ; if not, skip`
`    BL  GGreater7   ; G > 7`
`next1`
	`unsigned long G;`
	`if(G> 7){`
	`  GGreater7();`
	`}`

`    LDR R2, =G      ; R2 =& G`
`    LDR R0, [R2]    ; R0 = G`
`    CMP R0, #7      ; is G >= 7?`
`    BLO next2       ; if not, skip`
`    BL  GGreaterEq7 ; G >= 7`
`next2`
	`if(G> = 7){`
	`  GGreaterEq7();`
	`}`

`    LDR R2, =G      ; R2 =& G`
`    LDR R0, [R2]    ; R0 = G`
`    CMP R0, #7      ; is G < 7?`
`    BHS next3       ; if not, skip`
`    BL  GLess7      ; G < 7`
`next3`
	`if(G< 7){`
	`  GLess7();`
	`}`

`    LDR R2, =G      ; R2 =& G`
`    LDR R0, [R2]    ; R0 = G`
`    CMP R0, #7      ; is G <= 7?`
`    BHI next4       ; if not, skip`
`    BL  GLessEq7    ; G <= 7`
`next4`
	`if(G< = 7){`
	`  GLessEq7();`
	`}`

Program 7.2. Unsigned conditional structures.

It will call **GGreater7** if **G** is greater than **7**, **GGreaterEq7** if **G** is greater than or equal to **7**, **GLess7** if **G** is less than **7**, and **GLessEq7** if **G** is less than or equal to **7**. When comparing unsigned values, the instructions **BHI BLO BHS** and **BLS** should follow the subtraction or comparison instruction. A conditional if-then is implemented by bringing the first number in a register, subtracting the second number, then using the branch instruction with complementary logic to skip over the body of the if-then. To convert these examples to 16 bits, use the **LDRH R0,[R2]** instruction instead of the **LDR R0,[R2]** instruction. To convert these examples to 8 bits, use the **LDRB R0,[R2]** instruction instead of the **LDR R0,[R2]** instruction.

Example 7.1. Assuming G1 is 8-bit unsigned, write software that sets G1=50 if G1 is greater than 50. In other words, we will force G1 into the range 0 to 50

Solution: First, we draw a flowchart describing the desired algorithm, see Figure 7.3. Next, we restate the conditional as "skip over if G1 is less than or equal to 50". To implement the assembly code we bring G1 into Register R0 using **LDRB** to load an unsigned byte, subtract 50, then branch to next if G1 is less than or equal to 50, as presented in Program 7.3. We will use an unsigned conditional branch because the data format is unsigned.

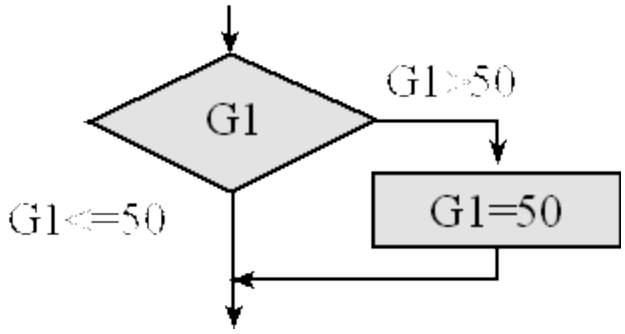


Figure 7.3. Flowchart of an if-then structure.

```
    LDR R2, =G1     ; R2 = &G1
    LDRB R0, [R2]   ; R0 = G1
    CMP R0, #50     ; is G1 > 50?
    BLS next        ; if not, skip to end
    MOV R1, #50     ; R1 = 50
    STRB R1, [R2]   ; G1 = 50
next
```

```
unsigned char G1;
if(G1>50){
  G1 = 50;
}
```

Program 7.3. An unsigned if-then structure. LDRB used because 8-bit, BLS used because it is unsigned.

Checkpoint 7.5: Assume you have an 8-bit unsigned global variable **N**. Write C code that executes the function **isTen**, if N is equal to 10.
Checkpoint 7.6: Assume **H1** and **H2** are two 16-bit unsigned variables. Write C code that executes the function **isEqual** if H1 equals H2.

Program 7.4 contains four separate signed if-then structures, where **G** is signed 32 bits. In each case, the first step is to bring the first value in R0; the second step is to compare the first value with a second value; and the third step is to execute a signed branch **Bxx**. The branch will occur if the first signed value is **xx** the second signed value.

Assembly code	C code

`LDR R2, =G ; R2 =& G`
`LDR R0, [R2] ; R0 = G`
`CMP R0, #7 ; is G > 7?`
`BLE next1 ; if not, skip`
`BL GGreater7 ; G > 7`
`next1`
	`long G;`
	`if(G> 7){`
	`  GGreater7();`
	`}`

`LDR R2, =G ; R2 =& G`
`LDR R0, [R2] ; R0 = G`
`CMP R0, #7 ; is G >= 7?`
`BLT next2 ; if not, skip`
`BL GGreaterEq7 ; G >= 7`
`next2`
	`if(G> = 7){`
	`  GGreaterEq7();`
	`}`

`LDR R2, =G ; R2 =& G`
`LDR R0, [R2] ; R0 = G`
`CMP R0, #7 ; is G < 7?`
`BGE next3 ; if not, skip`
`BL GLess7 ; G < 7`
`next3`
	`if(G< 7){`
	`  GLess7();`
	`}`

`LDR R2, =G ; R2 =& G`
`LDR R0, [R2] ; R0 = G`
`CMP R0, #7 ; is G <= 7?`
`BGT next4 ; if not, skip`
`BL GLessEq7 ; G <= 7`
`next4`
	`if(G< = 7){`
	`  GLessEq7();`
	`}`

Program 7.4. Signed conditional structures.

Similar to Program 7.2, Program 7.4 will call **GGreater7** if **G** is greater than **7**, **GGreaterEq7** if **G** is greater than or equal to **7**, **GLess7** if **G** is less than **7**, and **GLessEq7** if **G** is less than or equal to **7**. When comparing signed values, the instructions **BGT BLT BGE** and **BLE** should follow the subtraction or comparison instruction. A conditional if-then is implemented by bringing the first number in a register, subtracting the second number, then using the branch instruction with complementary logic to skip over the body of the if-then. To convert these examples to 16 bits, use the **LDRSH R0,[R2]** instruction instead of the **LDR R0,[R2]** instruction. To convert these examples to 8 bits, use the **LDRSB R0,[R2]** instruction instead of the **LDR R0,[R2]** instruction.

Checkpoint 7.7: Assume you have a 32-bit signed global variable **N**. Write C code that executes the function **isNeg**, if N is negative.
Common error: It is an error to use an unsigned conditional branch when comparing two signed values. Similarly, it is a mistake to use a signed conditional branch when comparing two unsigned values.

Observation: One cannot directly compare a signed number to an unsigned number. The proper method is to first convert both numbers to signed numbers of a higher precision and then compare.

Example 7.2. Redesign the Example 7.1 code assuming G1 is 8-bit signed. In particular we restrict the range to -128 to +50.

Solution: We can use the same flowchart shown previously in Figure 7.3. The way to compare two values is to subtract them from each other and check if that subtraction resulted in a positive number, zero, or negative number. If the subtraction yields a zero, then the numbers are obviously equal and the Z bit will be set. If it is positive, that means the first value is bigger than the second value and the N bit will be 0. If it is negative, then the first value is smaller than the second one and the N bit will be 1. In this case we bring G1 into Register R0 this time using **LDRSB** to load a signed byte (first value), and subtract 50 (second value). The **CMP** instruction subtracts 50 from R0 but doesn't save the result, it just sets the condition codes. The BLE uses the condition codes to branch to next if G1 is less than or equal to 50, as presented in Program 7.5. However, we will use a signed conditional branch (**BLE**) because the data format is signed..

```
    LDR   R2, =G1   ; R2 = &G1                  signed char G1;
    LDRSB R0, [R2]  ; R0 = G1 (signed)          if(G1>50){
    CMP   R0, #50   ; is G1 > 50?                 G1 = 50;
    BLE   next      ; if not, skip to end       }
    MOV   R1, #50   ; R1 = 1
    STRB  R1, [R2]  ; G1 = 50
next
```

Program 7.5. A signed if-then structure LDRSB is a signed 8-bit load. BLE is a signed branch.

Notice that the C code for Program 7.2 looks similar to Program 7.4, and the C code for Program 7.3 looks similar to Program 7.5. This is because the compiler knows the type of variables G1 and G2; therefore, it knows whether to utilize unsigned or signed branches. Unfortunately, this similarity can be deceiving. When writing code whether it be assembly or C, you still need to keep track of whether your variables are signed or unsigned. Furthermore, when comparing two objects, they must have comparable types. E.g., "Which is bigger, 2 unsigned apples or –3 signed dollars?" The compiler does not seem to reject comparisons between signed and unsigned variables as an error. However, I recommend that you do not compare a signed variable to an unsigned variable. When comparing objects of different types, it is best to first convert both objects to the same format, and then perform the comparison. Conversely, we see that all numbers are converted to 32 bits before they are compared. This means there is no difficulty comparing variables of differing precisions: e.g., 8-bit, 16-bit, and 32-bit as long as both are signed or both are unsigned.

We can use the unconditional branch to add an **`else`** clause to any of the previous **`if then`** structures. A simple example of an unsigned conditional is illustrated in the Figure 7.4 and presented in Program 7.6. The first three lines test the condition **`G1>G2`**. If **`G1>G2`**, the software branches to **`high`**. Once at high, the software calls the **`isGreater`** subroutine then continues. Conversely, if **`G1≤G2`**, the software does not branch and the **`isLessEq`** subroutine is executed. After executing the **`isLessEq`** subroutine, there is an unconditional branch, so that only one and not both subroutines are called.

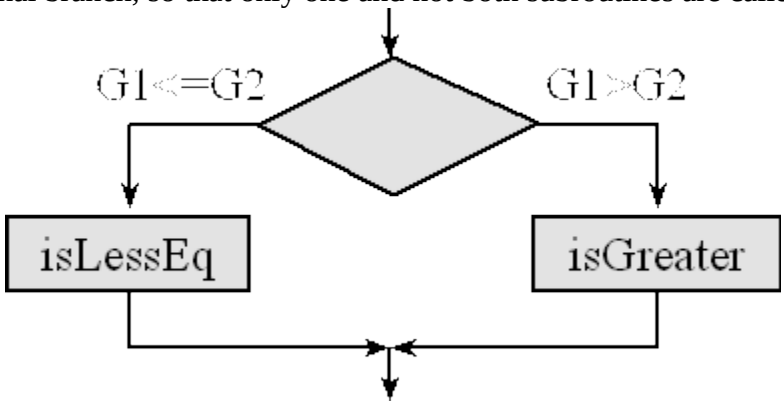


Figure 7.4. Flowchart of an if-then-else structure.

```
     LDR R2, =G1    ; R2 = &G1
     LDR R0, [R2]   ; R0 = G1
     LDR R2, =G2    ; R2 = &G2
     LDR R1, [R2]   ; R1 = G2
     CMP R0, R1     ; is G1 > G2 ?
     BHI high       ; if so, skip to high
low  BL  isLessEq   ; G1 <= G2
     B   next       ; unconditional
high BL  isGreater  ; G1 > G2
next
```

```
unsigned long G1,G2;
if(G1>G2){
  isGreater();
}
else{
  isLessEq();
}
```

Program 7.6. An unsigned if-then-else structure (unsigned 32-bit).

The **selection operator** takes three input parameters and yields one output result. The format is

`Expr1 ? Expr2 : Expr3`

The first input parameter is an expression, **`Expr1`**, which yields a Boolean (0 for false, not zero for true). **`Expr2`** and **`Expr3`** return values that are regular numbers. The selection operator will return the result of **`Expr2`** if the value of **`Expr1`** is true, and will return the result of **`Expr3`** if the value of **`Expr1`** is false. The type of the expression is determined by the types of **`Expr2`** and **`Expr3`**. If **`Expr2`** and **`Expr3`** have different types, then promotion is applied. The left and right side perform identical functions. If **`b`** is 1 set **`a`** equal to 10, otherwise set **`a`** to 1.

```
a = (b==1) ? 10 : 1;
```

```
if(b==1)
   a=10;
else
   a=1;
```

A 3-wide median filter can be designed using if-else conditional statements.

```
short Median(short u1,short u2,short u3){ short result;
if(u1>u2)
if(u2>u3) result=u2; // u1>u2,u2>u3 u1>u2>u3
else
if(u1>u3) result=u3; // u1>u2,u3>u2,u1>u3 u1>u3>u2
else result=u1;   // u1>u2,u3>u2,u3>u1 u3>u1>u2
else
if(u3>u2) result=u2; // u2>u1,u3>u2       u3>u2>u1
```

```
else
if(u1>u3) result=u1; // u2>u1,u2>u3,u1>u3 u2>u1>u3
else result=u3; // u2>u1,u2>u3,u3>u1 u2>u3>u1
return(result);
}
```

Program 7.7. A 3-wide median function.

Checkpoint 7.8: Assume you have a 32-bit unsigned global variable **`N`**. Write C code that changes N to 65535 if N is initially greater than 65535.

7.5.2. `switch` Statements

Switch statements provide a non-iterative choice between any number of paths based on specified conditions. They compare an expression to a set of constant values. Selected statements are then executed depending on which value, if any, matches the expression. The expression between the parentheses following **`switch`** is evaluated to a number and compared one by one to the explicit cases. Figure 7.5 draws a flowchart describes software that performs one output each time the function **`OneStep`** is called. The **`break`** causes execution to exit the switch statement. The **`default`** case is run if none of the explicit case statements match. The operation of the switch statement performs this list of actions:

If **`Last`** is equal to 10, then **`theNext`** is set to 9.
If **`Last`** is equal to 9, then **`theNext`** is set to 5.
If **`Last`** is equal to 5, then **`theNext`** is set to 6.
If **`Last`** is equal to 6, then **`theNext`** is set to 10.
If **`Last`** is not equal any of the above, then **`theNext`** is set to 10.

When using **`break`**, only the first matching case will be invoked. In other words, once a match is found, no other tests are performed. The body of the switch is not a normal compound statement since local declarations are not allowed in it or in subordinate blocks.

Assume the output port is connected to a stepper motor, and the motor has 24 steps per rotation. Calling **`OneStep`** will cause the motor to rotate by exactly 15 degrees. 15 degrees is 360 degrees divided by 24.

```
unsigned char Last=10;
void OneStep(void){
unsigned char next;
  switch(Last){
    case 10: next = 9; break;
    case 9:  next = 5;  break;
    case 5:  next = 6;  break;
    case 6:  next = 10; break;
    default: next = 10;
  }
  GPIO_PORTD_DATA_R = next;
  Last = next;
}
```

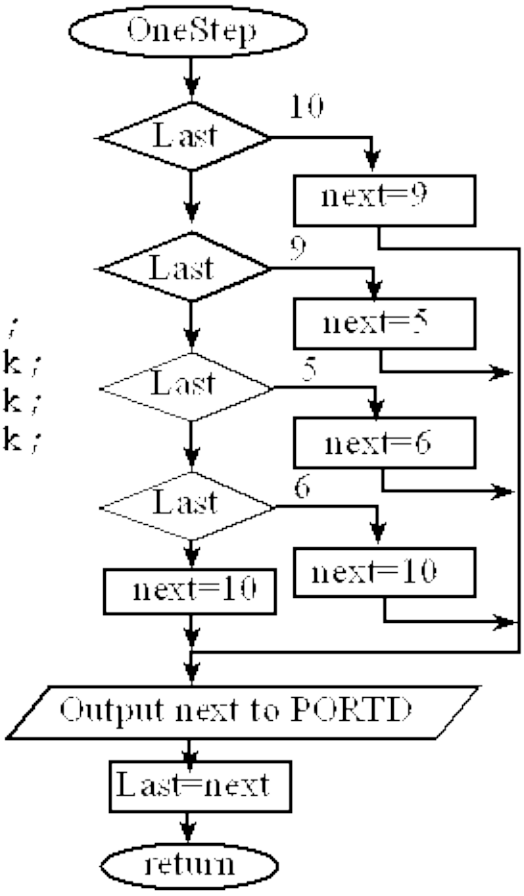


Figure 7.5. The switch statement is used to make multiple comparisons.

Program 7.8 converts an ASCII character to the equivalent decimal value. This example of a switch statement shows that the multiple tests can be performed for the same condition.

```
unsigned char Convert(unsigned char letter){
unsigned char digit;
  switch (letter) {
    case 'A':
    case 'B':
    case 'C':
    case 'D':
    case 'E':
    case 'F':
         digit = letter+10-'A';  break;
    case 'a':
    case 'b':
    case 'c':
    case 'd':
    case 'e':
    case 'f':
         digit = letter+10-'a';  break;
    default:
         digit = letter-'0';
  }
  return digit;
}
```

Program 7.8. A switch statement is used to convert an ASCII character to numeric value.

Checkpoint 7.9: Write a C function that converts lower case ASCII to uppercase. If the input is between 'a' to 'z', then the output equals the input minus -0x20. If the input is not between 'a' to 'z', then the output equals the input.

7.5.3. While Loops

Quite often the microcomputer is asked to wait for events or to search for objects. Both of these operations are solved using the **`while`** or **`do-while`** structure. A simple example of while loop is illustrated in the Figure 7.6 and presented in Program 7.9. Assume Port A bit 3 is an input. The operation is defined by the C code

```
while(GPIO_PORTA_DATA_R&0x08){Body();}
```

specifies the function **`Body()`** will be executed over and over as long as Port A bit 3 is high. .

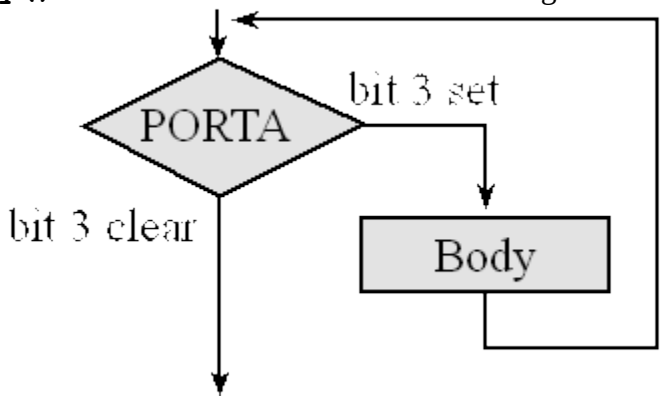


Figure 7.6. Flowchart of a while structure. Execute Body() over and over bit 3 of G1 is high.

Program 7.9 begins with a test of Port A bit 3. If bit 3 is low then the body of the while loop is skipped. The unconditional branch (**`B loop`**) after the body causes Port A bit 3 to be tested each time through the loop. In this way, the body is executed repeatedly until Port A bit 3 is low.

```
     LDR R4, =GPIO_PORTA_DATA_R
loop LDR R0, [R4]  ; R0 = Port A
     AND R0, #0x08 ; test bit 3
     BEQ next      ; if so, quit
     BL  Body      ; body of the loop
     B   loop
next
```

```
unsigned long G1,G2;

while(GPIO_PORTA_DATA_R&0x08){
  Body();
}
```

Program 7.9. A while loop structure.

Observation: The body of a while loop may execute zero or more times, but the body of a do-while loop is executed at least once.

One of the conventions when writing assembly is whether or not subroutines should save registers. According to AAPCS, we will allow subroutines to freely modify R0–R3 and R12. Conversely, if a subroutine wishes to use R4 through R11, it will preserve the values using the stack. Similarly, if the subroutine wishes to use LR (e.g., to call another subroutine) it must save and restore LR. This means address pointers R4 and R5 only need to be set once in Program 7.9, because we can assume that the call to **`Body()`** will not corrupt them. However, since the variables themselves are held in RAM and may therefore be changed by some other piece of code, it does make sense to reload the values of the variables each time through the loop.

Checkpoint 7.10: Assume you have a 16-bit unsigned global variable **`N`**. Write C code that calls the function body over and over as long as bit 0 of N is a 1.

7.5.4. Do-while Loops

A do-while loop performs the body first, and the test for completion second. It will execute the body at least once. Assume **`PF1`** and **`PA5`** are definitions of I/O pins using bit-specific addressing. Program 7.10 will toggle the output PF1 as long as input PA5 is low.

```
      LDR  R1, =PF1   ; R1 = &PF1
      LDR  R5, =PA5   ; R5 = &PA5
loop  LDR  R0, [R1]
      EOR  R0, #2     ; toggle bit 1
      STR  R0, [R1]
      LDR  R2, [R5]   ; R2 = PA5
      ANDS R2, #0x20  ; bit 5 set?
      BEQ  loop       ; spin while low
next
```

```
// toggle PF1 while PA5 low
do{
  PF1 = PF1^0x02;
} while((PA5&0x20)==0);
```

Program 7.10. A do-while loop structure.

Checkpoint 7.11: Assume PF4 PF3 and PF0 are bit-specific addresses for Port F pins 4, 3, 0 respectively. PF3 is an output, and PF4 and PF0 are inputs. Toggle PF3 once, and then keep toggling PF3 as long as both PF4 and PF0 are high.

7.5.5. For Loops

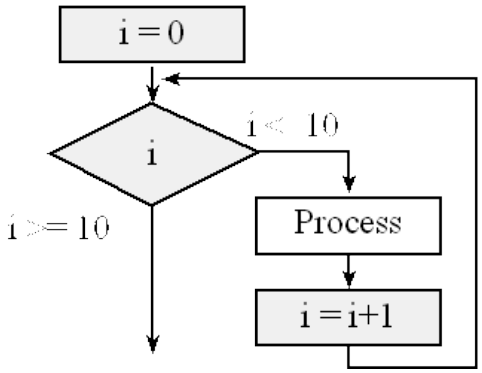
A **`for-loop`** control structure is a special case of the while loop. For loops can iterate up or down. To show the similarity between the while loop and for loop these two C functions are identical. The `<init>` code is executed once. The `<test>` code returns a true/false and is tested before each iteration. The `<body>` and `<end>` codes are executed each iteration.

```
<init>;
while(<test>){
  <body>;
 < end>;
}
```

```
for(<init>; <test>; <end>){
  <body>;
}
```

For-loops are a convenient way to perform repetitive tasks. As an example, we write code that calls **`Process()`** 10 times. Two possible solutions are illustrated in Figure 7.7. The solution on the left starts at 0 and counts up to 10, while the solution on the right starts at 10 and counts down to 0. The first field is the initialization task (e.g., **`i=0`**) which is performed only once at the beginning of the for loop. The next field specifies the conditions with which to continue execution (e.g., **`i<10`**), that is, we check this condition before deciding to repeat the loop another time or not. If the condition evaluates to false we end the for loop, otherwise we continue another repetition before checking again. The last field is the operation to perform after each iteration/repetition (e.g., **`i++`**), this is is the update task that is performed each iteration before the condition is checked. Similar to a while loop, the test occurs before each execution of the body. The order is as follows: initialize->condition_check(is true)->body->update->condition_check(is true)->body->update...update->condition_check(is false)->end for loop.

```
for(i=0; i<10; i++){
  Process();
}
```




```
for(i=10; i!=0; i--){
  Process();
}
```

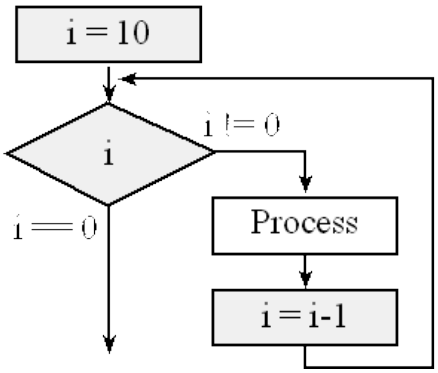


Figure 7.7. Two flowcharts of a `for-loop` structure.

The count-up implementation places the loop counter in the Register R4, as shown in Program 7.11. As mentioned earlier, we assume the subroutine **`Process`** preserves the value in R4.

```
     MOV R4, #0      ; R4 = 0
loop CMP R4, #10     ; index >= 10?
     BHS done        ; if so, skip to done
     BL  Process     ; process function
     ADD R4, R4, #1  ; R4 = R4 + 1
     B   loop
done
```

```
for(i=0; i<10; i++){
  Process();
}
```

Program 7.11. A simple for-loop.

If we assume the body will execute at least once, we can execute a little faster, as shown in Program 7.12, by counting down. Counting down is one instruction faster than counting up.

```
     MOV  R4, #10     ; R4 = 10
loop BL   Process     ; body
     SUBS R4, R4, #1  ; R4 = R4-1
     BNE  loop
done
```

```
     MOV R4, #0      ; R4 = 0
loop BL  Process     ; body
     ADD R4, R4, #1  ; R4 = R4+1
     CMP R4, #10     ; done?
     BLO loop        ; if not,repeat
```

Checkpoint 7.12: Assume PF2 is a bit-specific address for Port F pin 2, and assume PF2 is anoutput. Write a C function that toggles PF2 one million times.

7.6. Functional debugging

Functional debugging involves the verification of input/output parameters. Functional debugging is a static process where inputs are supplied, the system is run, and the outputs are compared against the expected results. Four methods of functional debugging are presented in this section, and two more functional debugging methods are presented in the next chapter after indexed addressing mode is presented. There are two important aspects of debugging: control and observability. The first step of debugging is to **stabilize** the system. In the debugging context, we stabilize the problem by creating a test routine that fixes (or stabilizes) all the inputs. In this way, we can reproduce the exact inputs over and over again. Stabilization is an effective approach to debugging because we can control exactly what software is being executed. Once stabilized, if we modify the program, we are sure that the change in our outputs is a function of the modification we made in our software and not due to a change in the input parameters. When a system has a small number of possible inputs (e.g., less than a million), it makes sense to test them all. When the number of possible inputs is large we need to choose a set of inputs. There are many ways to make this choice. We can select values:

Near the extremes and in the middle
Most typical of how our clients will properly use the system
Most typical of how our clients will improperly attempt to use the system
That differ by one
You know your system will find difficult
Using a random number generator

To stabilize the system we define a fixed set of inputs to test, run the system on these inputs, and record the outputs. Debugging is a process of finding patterns in the differences between recorded behavior and expected results. The advantage of modular programming is that we can perform modular debugging. We make a list of modules that might be causing the bug. We can then create new test routines to stabilize these modules and debug them one at a time. Unfortunately, sometimes all the modules seem to work, but the combination of modules does not. In this case we study the interfaces between the modules, looking for intended and unintended (e.g., unfriendly code) interactions.

Many debuggers allow you to set the program counter to a specific address then execute one instruction at a time. The debugger provides three stepping commands **Step**, **StepOver** and **StepOut** commands.

Step is the usual execute one assembly instruction. However, when debugging C we can also execute one line of C. **StepOver** will execute one assembly instruction, unless that instruction is a subroutine call, in which case the debugger will execute the entire subroutine and stop at the instruction following the subroutine call. **StepOut** assumes the execution has already entered a subroutine, and will finish execution of the subroutine and stop at the instruction following the subroutine call.

A **breakpoint** is a mechanism to tag places in our software, which when executed will cause the software to stop. Normally, you can break on any line of your program.

One of the problems with breakpoints is that sometimes we have to observe many breakpoints before the error occurs. One way to deal with this problem is the conditional breakpoint. To illustrate the implementation of conditional breakpoints, add a global variable called **`Count`** and initialize it to 32 in the initialization ritual. Add the following conditional breakpoint to the appropriate location in your software. Using the debugger, we set a regular breakpoint at **`bkpt`**. We run the system again (you can change the 32 to match the situation that causes the error.)

```
            PUSH {R1, R2}   ; save R1 and R2
            LDR  R2, =Count ; R2 = Count
            LDR  R1, [R2]   ; R1 = Count
            SUBS R1, R1, #1 ; Count = Count - 1
            STR  R1, [R2]   ; store to Count
            BNE  DEBUG_skip ; if Count != 0, skip
DEBUG_bkpt  NOP             ; put breakpoint here
DEBUG_skip  POP {R1, R2}    ; restore R1 and R2
```

```
if(--Count==0)
   bkpt
```

Notice that the breakpoint occurs only on the 32nd time the break is encountered. Any appropriate condition can be substituted. Most modern debuggers allow you to set breakpoints that will trigger on a count. However, this method allows flexibility of letting you choose the exact conditions that cause the break.

The use of print statements is a popular and effective means for functional debugging. One difficulty with print statements in embedded systems is that a standard "printer" may not be available. Another problem with printing is that most embedded systems involve time-dependent interactions with its external environment. The print statement itself may be so slow, that the debugging process itself causes the system to fail. In this regard, the print statement is **intrusive**. Therefore, throughout this book we will utilize debugging methods that do not rely on the availability of a standard output device.

7.7. Design Example

Say, we are shipwrecked on an island and we want to send an SOS to aircraft and other ships passing by to get their attention. We have a LaunchPad and a battery that we can use to design a solution. Program 7.2 shows the solution developed in the videos.

```
// 0.Documentation Section
// C7_SOS, main.c
// Runs on LM4F120 or TM4C123 LaunchPad
// Input from PF4(SW1),PF0(SW2), output to PF3 (Green LED)
// Pressing SW1 starts SOS (Green LED flashes SOS).
//    S: Toggle light 3 times with 1/2 sec gap between ON..1/2sec..OFF
//    O: Toggle light 3 times with 2 sec gap between ON..2sec..OFF
//    S: Toggle light 3 times with 1/2 sec gap between ON..1/2sec..OFF
//    5 second delay between SOS
// Pressing SW2 stops SOS
// Authors: Daniel Valvano, Jonathan Valvano and Ramesh Yerraballi
// Date: July 15, 2013
// 1. Pre-processor Directives Section
```

```
// Constant declarations to access port registers using
// symbolic names instead of addresses
#define GPIO_PORTF_DATA_R    (*((volatile unsigned long *)0x400253FC))
#define GPIO_PORTF_DIR_R     (*((volatile unsigned long *)0x40025400))
#define GPIO_PORTF_AFSEL_R   (*((volatile unsigned long *)0x40025420))
#define GPIO_PORTF_PUR_R     (*((volatile unsigned long *)0x40025510))
#define GPIO_PORTF_DEN_R     (*((volatile unsigned long *)0x4002551C))
#define GPIO_PORTF_LOCK_R    (*((volatile unsigned long *)0x40025520))
#define GPIO_PORTF_CR_R      (*((volatile unsigned long *)0x40025524))
#define GPIO_PORTF_AMSEL_R   (*((volatile unsigned long *)0x40025528))
#define GPIO_PORTF_PCTL_R    (*((volatile unsigned long *)0x4002552C))
#define SYSCTL_RCGC2_R       (*((volatile unsigned long *)0x400FE108))

// 2. Declarations Section
//   Global Variables
unsigned long SW1; // input from PF4
unsigned long SW2; // input from PF0
//   Function Prototypes
void PortF_Init(void);
void FlashSOS(void);
void delay(unsigned long halfsecs);

// 3. Subroutines Section
// MAIN: Mandatory for a C Program to be executable
int main(void){
  PortF_Init(); // Init port PF4 PF2 PF0
  while(1){
    do{
      SW1 = GPIO_PORTF_DATA_R&0x10; // PF4 into SW1
    }while(SW1 == 0x10);
    do{
      FlashSOS();
              SW2 = GPIO_PORTF_DATA_R&0x01; // PF0 into SW2
            }while(SW2 == 0x01);
  }
}

// Subroutine to initialize port F pins for input and output
// PF4 is input SW1 and PF2 is output Blue LED
// Inputs: None
// Outputs: None
// Notes: ...
void PortF_Init(void){ volatile unsigned long delay;
  SYSCTL_RCGC2_R |= 0x00000020;   // 1) F clock
  delay = SYSCTL_RCGC2_R;         // delay
  GPIO_PORTF_LOCK_R = 0x4C4F434B; // 2) unlock PortF PF0
  GPIO_PORTF_CR_R |= 0x1F;        // allow changes to PF4-0
  GPIO_PORTF_AMSEL_R &= 0x00;     // 3) disable analog function
  GPIO_PORTF_PCTL_R &= 0x00000000; // 4) GPIO clear bit PCTL
  GPIO_PORTF_DIR_R &= ~0x11;      // 5.1) PF4,PF0 input,
  GPIO_PORTF_DIR_R |= 0x08;       // 5.2) PF3 output
  GPIO_PORTF_AFSEL_R& = 0x00;     // 6) no alternate function
  GPIO_PORTF_PUR_R |= 0x11;       // enable pullup resistors on PF4,PF0
  GPIO_PORTF_DEN_R |= 0x1F;       // 7) enable digital pins PF4-PF0
}
// Color    LED(s) PortF
// dark     ---    0
// red      R--    0x02
// blue     --B    0x04
// green    -G-    0x08
// yellow   RG-    0x0A
// sky blue -GB    0x0C
// white    RGB    0x0E
```

```
// Subroutine to Flash a green LED SOS once
// PF3 is green LED: SOS
//    S: Toggle light 3 times with 1/2 sec gap between ON..1/2sec..OFF
//    O: Toggle light 3 times with 2 sec gap between ON..2sec..OFF
//    S: Toggle light 3 times with 1/2 sec gap between ON..1/2sec..OFF
// Inputs: None
// Outputs: None
// Notes: ...
void FlashSOS(void){
  //S
  GPIO_PORTF_DATA_R |= 0x08;  delay(1);
  GPIO_PORTF_DATA_R &= ~0x08; delay(1);
  GPIO_PORTF_DATA_R |= 0x08;  delay(1);
  GPIO_PORTF_DATA_R &= ~0x08; delay(1);
  GPIO_PORTF_DATA_R |= 0x08;  delay(1);
  GPIO_PORTF_DATA_R &= ~0x08; delay(1);
  //O
  GPIO_PORTF_DATA_R |= 0x08; delay(4);
  GPIO_PORTF_DATA_R &= ~0x08;delay(4);
  GPIO_PORTF_DATA_R |= 0x08; delay(4);
  GPIO_PORTF_DATA_R &= ~0x08;delay(4);
  GPIO_PORTF_DATA_R |= 0x08; delay(4);
  GPIO_PORTF_DATA_R &= ~0x08;delay(4);
  //S
  GPIO_PORTF_DATA_R |= 0x08; delay(1);
  GPIO_PORTF_DATA_R &= ~0x08;delay(1);
  GPIO_PORTF_DATA_R |= 0x08; delay(1);
  GPIO_PORTF_DATA_R &= ~0x08;delay(1);
  GPIO_PORTF_DATA_R |= 0x08; delay(1);
  GPIO_PORTF_DATA_R &= ~0x08;delay(1);
  delay(10); // Delay for 5 secs in between flashes
}

// Subroutine to delay in units of half seconds
// We will make a precise estimate later:
//   For now we assume it takes 1/2 sec to count down
//   from 2,000,000 down to zero
// Inputs: Number of half seconds to delay
// Outputs: None
// Notes: ...
void delay(unsigned long halfsecs){
  unsigned long count;

  while(halfsecs > 0 ) { // repeat while still halfsecs to delay
    count = 1538460;
// originally count was 400000, which took 0.13 sec to complete
// later we change it to 400000*0.5/0.13=1538460 that it takes 0.5 sec
    while (count > 0) {
      count--;
    } // This while loop takes approximately 3 cycles
    halfsecs--;
  }
}
```

Program 7.13. Software solution that implements the rescue device (C7_SOS).

Chapter 8: Switch and LED Interfacing
Embedded Systems - Shape the World
Jonathan Valvano and Ramesh Yerraballi

We couldn't wait to show you how much fun it is to make the microcontroller interact with real physical devices. So in this section we will take a short side step from the business of concepts and theories to teach you how to connect switches and LEDs to the microcontroller. We will use switches to input data and use LEDs to output results. In this chapter, we will illustrate the design, construction and testing of an embedded system. The switch LED and LaunchPad will then be combined to create a system. You will need a solid understanding of Ohm's Law, so you may need to review current, voltage, power, and resistance from Chapter 3.

Learning Objectives:

- Understanding basic circuit elements like source, ground, and resistors.
- Understanding how switches and LEDs work.
- Application of Ohm's Law
- Analog circuit design and construction on a solderless breadboard
- Interfacing switches and LEDs to a microcontroller
- Programming simple logic.

8.1. Breadboard

To build circuits, we'll use a solderless breadboard, also referred to as a protoboard. The holes in the protoboard are internally connected in a systematic manner, as shown in Figure 8.1. The long rows of holes along the outer sides of the protoboard are electrically connected. Some protoboards like the one in Figure 8.1 have four long rows (two on each side), while others have just two long rows (one on each side). We refer to the long rows as power buses. If your protoboard has only two long rows (one on each side, we will connect one row to +3.3V and another row to ground. If your protoboard has two long rows on each side, then two rows will be ground, one row will be +3.3V. Use a black marker and label the voltage on each row. In the middle of the protoboard, you'll find two groups of holes placed in a 0.1 inch grid. Each adjacent row of five pins is electrically connected. We usually insert components into these holes. If integrated circuits (IC) are to be placed on the protoboard, it is done such that the two rows of pins straddle the center valley.

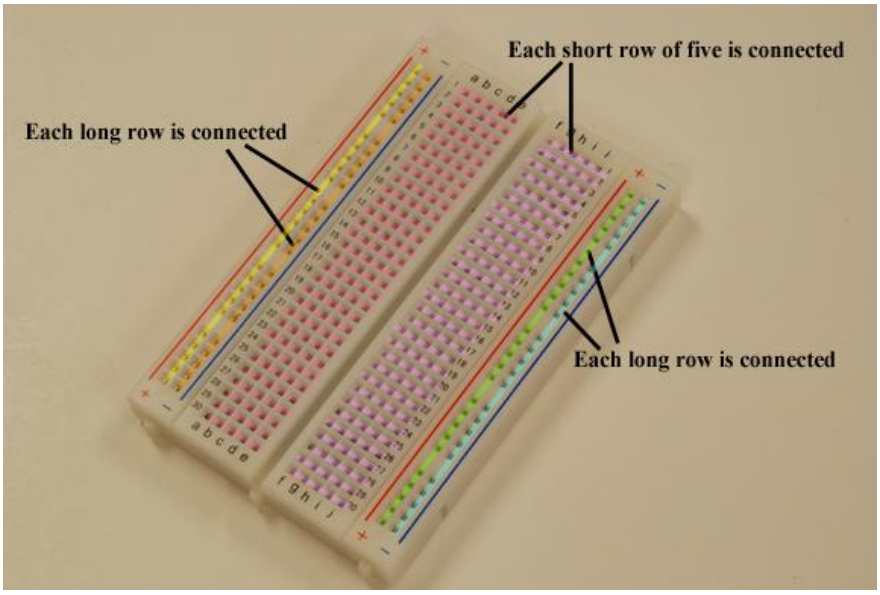


Figure 8.1. The pins on each of the four long rows are connected. The 5 pins in each short row are connected. Place a +3.3V wire to one long row. Place a ground wire from the LaunchPad to another long row.

To build a circuit we will use 22-gauge or 24-gauge solid wire. We strip off about 1/4 to 1/2 inch of insulation so the bare wire can be pushed straight into a hole. We only push one wire into a hole and remember which rows are internally connected as described in Figure 8.1.

8.2. Switch Interfaces

Input/output devices are critical components of an embedded system. The first input device we will study is the **switch**. It allows the human to input binary information into the computer. Typically we define the asserted state, or logic true, when the switch is pressed. Contact switches can also be used in machines to detect mechanical contact (e.g., two parts touching, paper present in the printer, or wheels on the ground etc.) A single pole single throw (SPST) switch has two connections. The switches are shown as little open circles in Figure 8.2. In a normally open switch (NO), the resistance between the connections is infinite (over 100 MΩ on the B3F tactile switch) if the switch is not pressed and zero (under 0.1 Ω on the B3F tactile switch) if the switch is pressed.

To convert the infinite/zero resistance into a digital signal, we can use a pull-down resistor to ground or a pull-up resistor to +3.3V as shown in Figure 8.2. Notice that 10 kΩ is 100,000 times larger than the on-resistance of the switch and 10,000 times smaller than its off-resistance. Another way to choose the pull-down or pull-up resistor is to consider the input current of the microcontroller input pin. The current into the microcontroller will be less than 2µA (shown as I_{IL} and I_{IH} in the data sheet). So, if the current into microcontroller is 2µA, then the voltage drop across the 10 kΩ resistor will be 0.02 V, which is negligibly small. With a pull-down resistor, the digital signal will be low if the switch is not pressed and high if the switch is pressed (right Figure 8.2). This is defined as **positive logic** because the asserted state is a logic high. Conversely, with a pull-up resistor, the digital signal will be high if the switch is not pressed and low if the switch is pressed (middle of Figure 8.2). This is defined as **negative logic** because the asserted state is a logic low.

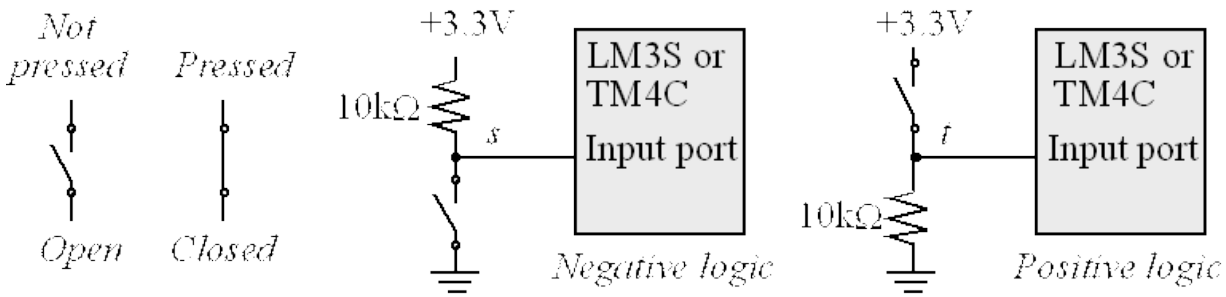


Figure 8.2. Single Pole Single Throw (SPST) Switch interface.

One of the complicating issues with mechanical switches is they can bounce (oscillate on and off) when touched and when released. The contact bounce varies from switch to switch and from time to time, but usually bouncing is a transient event lasting less than 5 ms. We can eliminate the effect of bounce if we design software that waits at least 10 ms between times we read the switch values.

To interface a switch we connect it to a pin (e.g., Figure 8.3) and initialize the pin as an input. The initialization function will enable the clock, set the direction register to input, turn off the alternative function, and enable the pin. Notice the software is friendly because it just affects PA5 without affecting the other bits in Port A. The input function reads Port A and returns a true (0x20) if the switch is pressed and returns a false (0) if the switch is not pressed. Figure 8.4 shows how we could build this circuit with a protoboard and a LaunchPad.

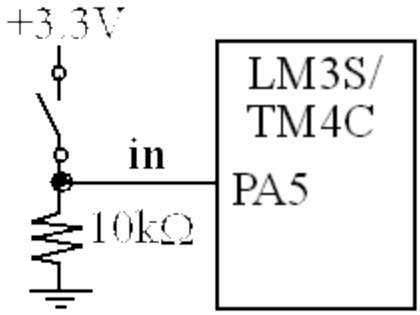


Figure 8.3. Interface of a switch to a microcomputer input.

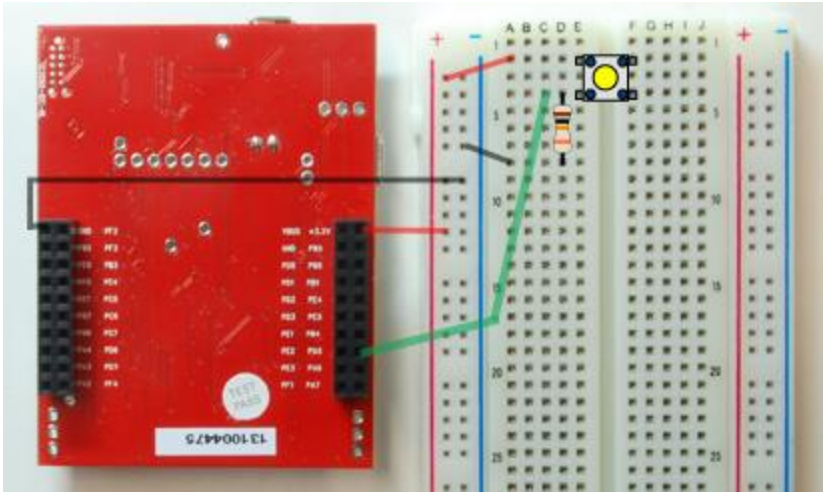


Figure 8.4. Construction of the interface of a switch to a microcomputer input. The brown-black-orange resistor is 10k. The switches in the lab-kit should plug into the protoboard. The switch is across the two pins that are closer to each other. It doesn't matter what color the wires are, but in this figure the wires are black, red and green. The two black wires are ground, the red wire is +3.3V, and the green wire is the signal ***in****, which connects the switch to PA5 of the microcontroller.*

The software in Program 8.1 is called a driver, and it includes an initialization, which is called once, and a second function that can be called to read the current position of the switch. Writing software this way is called an abstraction, because it separates what the switch does (Init, On, Off) from how it works (PortA, bit 5, TM4C123). The first input function uses the bit-specific address to get just PA5, while the second reads the entire port and selects bit 5 using a logical AND.

```
#define PA5   (*((volatile unsigned long *)0x40004080))
void Switch_Init(void){ volatile unsigned long delay;
  SYSCTL_RCGC2_R |= 0x00000001;     // 1) activate clock for Port A
  delay = SYSCTL_RCGC2_R;           // allow time for clock to start
                                    // 2) no need to unlock GPIO Port A
  GPIO_PORTA_AMSEL_R &= ~0x20;      // 3) disable analog on PA5
  GPIO_PORTA_PCTL_R &= ~0x00F00000; // 4) PCTL GPIO on PA5
  GPIO_PORTA_DIR_R &= ~0x20;        // 5) direction PA5 input
  GPIO_PORTA_AFSEL_R& = ~0x20;      // 6) PA5 regular port function
  GPIO_PORTA_DEN_R |= 0x20;         // 7) enable PA5 digital port
}
unsigned long Switch_Input(void){
  return PA5; // return 0x20(pressed) or 0(not pressed)
}
unsigned long Switch_Input2(void){
  return (GPIO_PORTA_DATA_R&0x20); // 0x20(pressed) or 0(not pressed)
}
```

Program 8.1. Software interface for a switch on PA5 (C8_Switch).

Maintenance Tip: When interacting with just some of the bits of an I/O register it is better to modify just the bits of interest, leaving the other bits unchanged. In this way, the action of one piece of software does not undo the action of another piece.

8.3. LED Interfaces

A **light emitting diode** (LED) emits light when an electric current passes through it. LEDs have polarity, meaning current must pass from anode to cathode to activate. The anode is labelled **a** or **+** , and cathode is labelled **k** or **-**. The cathode is the short lead and there may be a slight flat spot on the body of round LEDs. Thus, the anode is the longer lead. The brightness of an LED depends on the applied electrical power (*P=I*V*). Since the LED voltage is approximately constant in the active region (see left side of Figure 8.5), we can establish the desired brightness by setting the current.

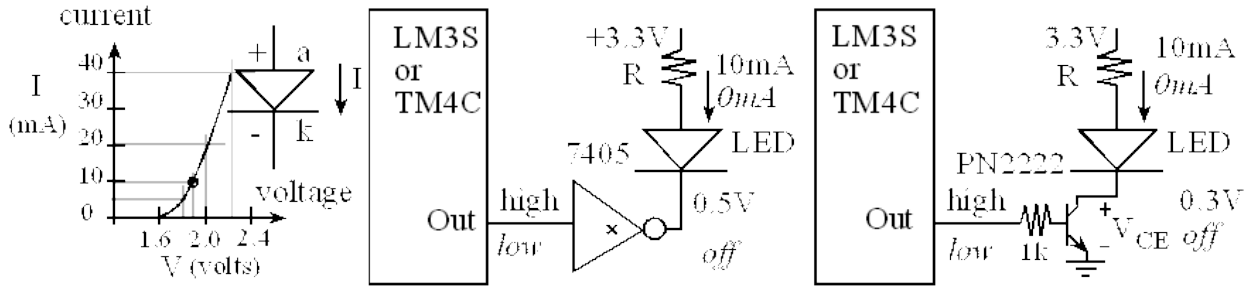


Figure 8.5. Positive logic LED interface (Lite-On LTL-10223W).

Checkpoint 8.1: What resistor value in Figure 8.5 is needed if the desired LED operating point is 1.7V and 11 mA?

If the LED current is above 8 mA, we cannot connect it directly to the microcontroller because the high currents may damage the chip. Figure 8.5 shows two possible interface circuits we could use. In both circuits if the software makes its output high the LED will be on. If the software makes its output low the LED will be off (shown in Figure 8.5 with *italics*). When the software writes a logic 1 to the output port, the input to the 7405/PN2222 becomes high, output from the 7405/PN2222 becomes low, 10 mA travels through the LED, and the LED is on. When the software writes a logic 0 to the output port, the input to the 7405/PN2222 becomes low, output from the 7405/PN2222 floats (neither high nor low), no current travels through the LED, and the LED is dark. The value of the resistor is selected to establish the proper LED current. When active, the LED voltage will be about 2 V, and the power delivered to the LED will be controlled by its current. If the desired brightness requires an operating point of 1.9 V at 10 mA, then the resistor value should be

$$R = \frac{3.3 - V_d - V_{OL}}{I_d} = \frac{3.3 - 1.9 - 0.5}{0.01} = 90\Omega$$

Where, V_d, I_d is the desired LED operating point, and V_{OL} is the output low voltage of the LED driver. If we use a standard resistor value of 100Ω in place of the 90Ω, then the current will be (3.3-1.9-0.5V)/100Ω, which is about 9 mA. This slightly lower current is usually acceptable.

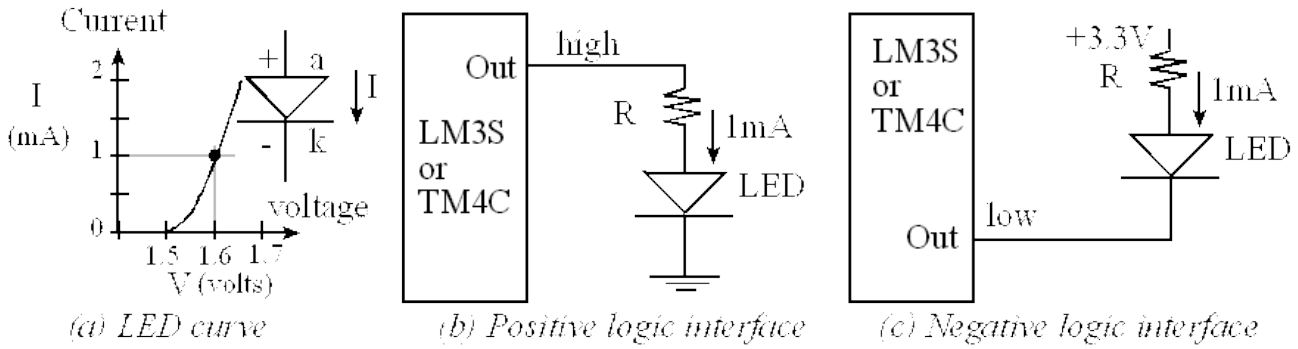


Figure 8.6. Low current LED interface (Agilent HLMP-D150).

When the LED current is less than 8 mA, we can interface it directly to an output pin without using a driver. The LED shown in Figure 8.6a has an operating point of 1.7 V and 1 mA. For the positive logic interface (Figure 8.6b) we calculate the resistor value based on the desired LED voltage and current

$$R = \frac{V_{OH} - V_d}{I_d} = \frac{2.4 - 1.6}{0.001} = 800\,\Omega$$

Where, V_{OH} is the output high voltage of the microcontroller output pin. Since V_{OH} can vary from 2.4 to 3.3 V, it makes sense to choose a resistor from a measured value of V_{OH}, rather than the minimum value of 2.4 V. Negative logic means the LED is activated when the software outputs a zero. For the negative logic interface (Figure 8.6c) we use a similar equation to determine the resistor value

$$R = \frac{3.3 - V_d - V_{OL}}{I_d} = \frac{3.3 - 1.6 - 0.4}{0.001} = 1.3\,\text{k}\Omega$$

Where, V_{OL} is the output low voltage of the microcontroller output pin.

If we use a 1.2 kΩ in place of the 1.3 kΩ, then the current will be (3.3-1.6-0.4V)/1.2kΩ, which is about 1.08 mA. This slightly higher current is usually acceptable. If we use a standard resistor value of 1.5 kΩ in place of the 1.3 kΩ, then the current will be (3.3-1.6-0.4V)/1.5kΩ, which is about 0.87 mA. This slightly lower current is usually acceptable.

The software in Program 8.2 is called a driver, and it includes an initialization, which is called once, and two functions that can be called to turn on and off the LED. Writing software this way is called an abstraction, because it separates what the LED does (Init, On, Off) from how it works (PortA, TM4C123).

Checkpoint 8.2: What resistor value in of Figure 8.6 is needed if the desired LED operating point is 1.7V and 2 mA? Use the negative logic interface and, V_{OL} of 0.4V.

```
void LED_Init(void){ volatile unsigned long delay;
   SYSCTL_RCGC2_R |= 0x01;            // 1) activate clock for Port A
   delay = SYSCTL_RCGC2_R;            // allow time for clock to start
                                      // 2) no need to unlock PA2
   GPIO_PORTA_PCTL_R &= ~0x00000F00; // 3) regular GPIO
   GPIO_PORTA_AMSEL_R &= ~0x04;       // 4) disable analog function on PA2
   GPIO_PORTA_DIR_R |= 0x04;          // 5) set direction to output
   GPIO_PORTA_AFSEL_R &= ~0x04;       // 6) regular port function
   GPIO_PORTA_DEN_R |= 0x04;          // 7) enable digital port
 }
 // Make PA2 high
```

```
void LED_On(void){
  GPIO_PORTA_DATA_R |= 0x04;
}
// Make PA2 low
void LED_Off(void){
  GPIO_PORTA_DATA_R &= ~0x04;
}
```

Program 8.2. Software interface for an LED on PF2

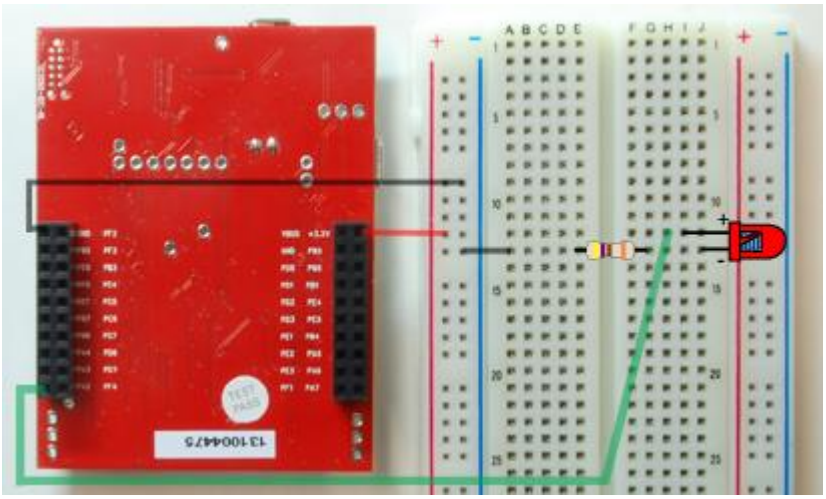


Figure 8.7. Construction of the interface of an LED to a microcomputer output (Figure 8.6b). The yellow-purple-brown resistor is 470ohm. It doesn't matter what color the wires are, but in this figure the wires are black, red and green. The two black wires are ground, the red wire is +3.3V, and the green wire is the signal ***Out****, which connects PA2 of the microcontroller to the positive side of the LED.*

8.4. Design Example

Some problems are so unique that they require the engineer to invent completely original solutions. Most of the time, however, the engineer can solve even complex problems by building the system from components that already exist. Creativity will still be required in selecting the proper components, making small changes in their behavior (tweaking), arranging them in an effective and efficient manner, and then verifying the system satisfies both the requirements and constraints. When young engineers begin their first job, they are sometimes surprised to see that education does not stop with college graduation, but rather is a life-long activity. In fact, it is the educational goal of all engineers to continue to learn both processes (rules about how to solve problems) and products (hardware and software components). As the engineer becomes more experienced, he or she has a larger toolbox from which processes and components can be selected.

The hardest step for most new engineers is the first one: **where to begin**? We begin by analyzing the problem to create a set of specifications and constraints in the form of a requirements document. Next, we look for components, in the form of previously debugged solutions, which are similar to our needs. Often during the design process, additional questions or concerns arise. We at that point consult with our clients to clarify the problem. Next we rewrite the requirements document and get it reapproved by the clients.

It is often difficult to distinguish whether a parameter is a specification or a constraint. In actuality, when designing a system it often doesn't matter into which category a parameter falls, because the system must satisfy all specifications and constraints. Nevertheless, when documenting the device it is better to categorize parameters properly. **Specifications** generally define in a quantitative manner the overall system objectives as given to us by our customers.

Constraints, on the other hand, generally define the boundary space within which we must search for a solution to the problem. If we must use a particular component, it is often considered a constraint. In this class, we constrain most designs to include a Tiva LaunchPad. Constraints also are often defined as an inequality, such as the cost must be less than $50, or the battery must last for at least one week. Specifications on the other hand are often defined as a quantitative number, and the system satisfies the requirement if the system operates within a specified tolerance of that parameter. Tolerance can be defined as a percentage error or as a range with minimum and maximum values.

In engineering everything is either a system or an interface between systems. For example a switch can be considered a system. When we interface it to the LaunchPad the switch-LaunchPad combination is a new system. Therefore, we begin by collecting the components required to build the system. We then combine the components and debug the system. As the components are combined we create new more powerful components. When writing software, we can use flowcharts to develop new algorithms. The more we can simulate the system, the more design possibilities we can evaluate, and the quicker we can make changes. Debugging involves both making sure it works, together with satisfying all requirements and constraints.

8.4.1 Requirements

First, lets develop a requirements document. The main purpose of a requirements document is to serve as an agreement between you and your clients describing what the system will do. This agreement can become a legally binding contract. Write the document so that it is easy to read and understand by others. It should be *unambiguous, complete, verifiable,* and *modifiable*.

The requirements document should not include how the system will be designed. This allows the engineer to make choices during the design to minimize cost and maximize performance. Rather it should describe the problem being solved and what the system actually does. It can include some constraints placed on the development process. Ideally, it is co-written by both the engineers and the non-technical clients. However, it is imperative that both the engineers and the clients understand and agree on the specifics in the document.

Requirements document for the security system

1. **Overview**
    1. *Objectives*: Design a security system with two sensors and one alarm. Contact switches will be used for the sensors. If the window is secure the switch will be pressed, and if the window is open, the switch will be released. The alarm condition will be a flashing LED. The system can be activated/deactivated with a toggle switch.
    2. *Process*: We will first prototype it on a breadboard using a LaunchPad, two momentary switches, a toggle switch, and an LED. Hardware and software designs will be developed and tested. Once the prototype is tested, the momentary switches can be replaced with switches that fit into the windows and doors, and the flashing LED can be replaced with an audio alarm. The number of sensors will need to be increased to allow one sensor to be placed in each window and door of the house.
    3. *Roles and Responsibilities*: Dr. Yerraballi will develop the hardware and Dr. Valvano will develop the software, both will test.
    4. *Interactions with Existing Systems*: We will use a LaunchPad. It will be powered through the USB cable either with AC outlet or stand-alone battery. The components of the prototype will be selected, if possible, from the parts in the lab kit.
    5. *Terminology*: A momentary switch has a spring, such that if no force is applied the contact will be open. Applying force continuously will close the contact on the momentary switch. A toggle switch can be placed in either the open or closed state, and force is required to change from open to closed or from closed to open. See Figure 8.8.

6. *Security*: The wires between the sensors and the microcontroller will need to be hidden. However, cutting the wires should constitute a breach, setting off the alarm.
7.

2. **Function Description**
    1. *Functionality*: There are two contact switches that measure the status of your home. Each contact switch is positioned in the frame of a window or a door. If the switch is closed or pressed the window is secure, but if either switch is open it means the window is open and the home is unsecure.
    2. *Scope*: The scope of this design will involve hardware/software prototype design, but will not include legal, marketing, or financial aspects.
    3. *Prototypes*: We will use a breadboard and LaunchPad.
    4. *Performance*: If either sensor shows the window is insecure and the system is activated the alarm should occur. The performance is defined as functionally correct. There is no particular specification for the minimum response time between the arrival of an insecure state and the alarm signal.
    5. *Usability*: There is also a toggle switch with which the user can use to activate or deactivate the alarm. If the alarm is activated, the LED will flash at 5 Hz if either switch is not pressed.
    6. *Safety*: No concerns
    7.
3. **Deliverables**
    1. *Reports*: Videos will be recorded and converted to a design chapter in the MOOC Embedded Systems – Shape the World.
    2. *Audits*: None
    3. *Outcomes*: We will create a prototype system
    4.

8.4.2 Components

Before we get into the actual design of the Security system, let's take stock of the components we will use in building the prototype.

8.4.3 Hardware Design

A **data flow graph** is a block diagram of the system, showing the flow of information. Arrows point from source to destination. Notice that a data flow graph looks like a block diagram of the system. In fact we draw a data flow graph by showing how the components are connected together. By visualizing the flow of data we are able to identify the components of the system and the nature of the data they work with.

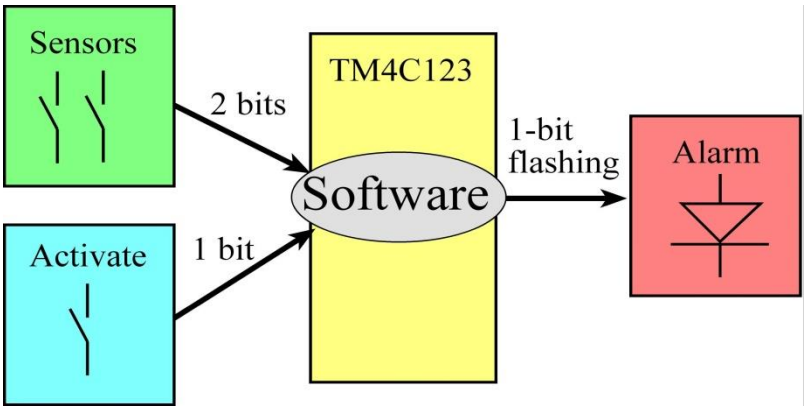


Figure 8.8. Data Flow Graph for Security System

The data-flow diagram gives us a blueprint for both the hardware circuit we are going to build and the software we are going to write. Let's first build the circuit:

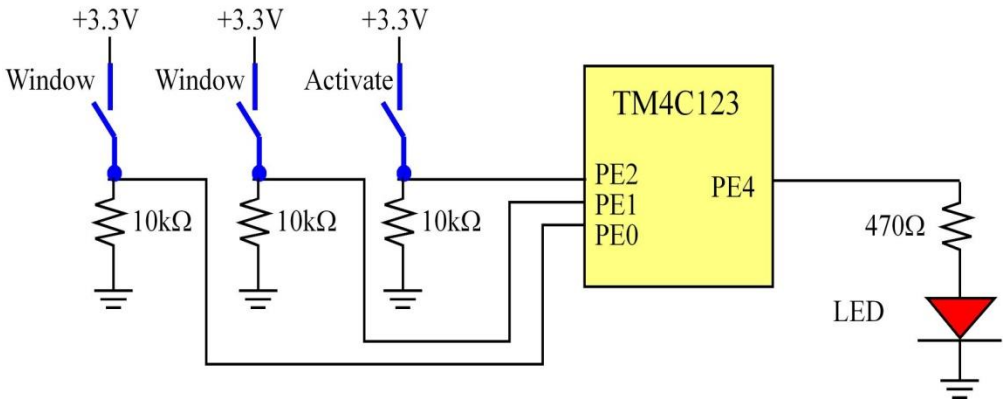


Figure 8.9. Hardware circuit

8.4.4 Software Design

Programs themselves are written in a linear or one-dimensional fashion. In other words, we type one line of software after another in a sequential 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. Even the simple systems have multiple software tasks. Furthermore, a complex application will require multiple microcontrollers. Therefore, we need a multi-dimensional way to visualize software behavior. Flowcharts are one way to describe software in a two-dimensional format, specifically providing convenient mechanisms to visualize multi-tasking, branching, and function calls. Flowcharts are very useful in the initial design stage of a software system to define complex algorithms. As an added benefit, flowcharts can be used in the final documentation stage of a project in order to assist in its use or modification.

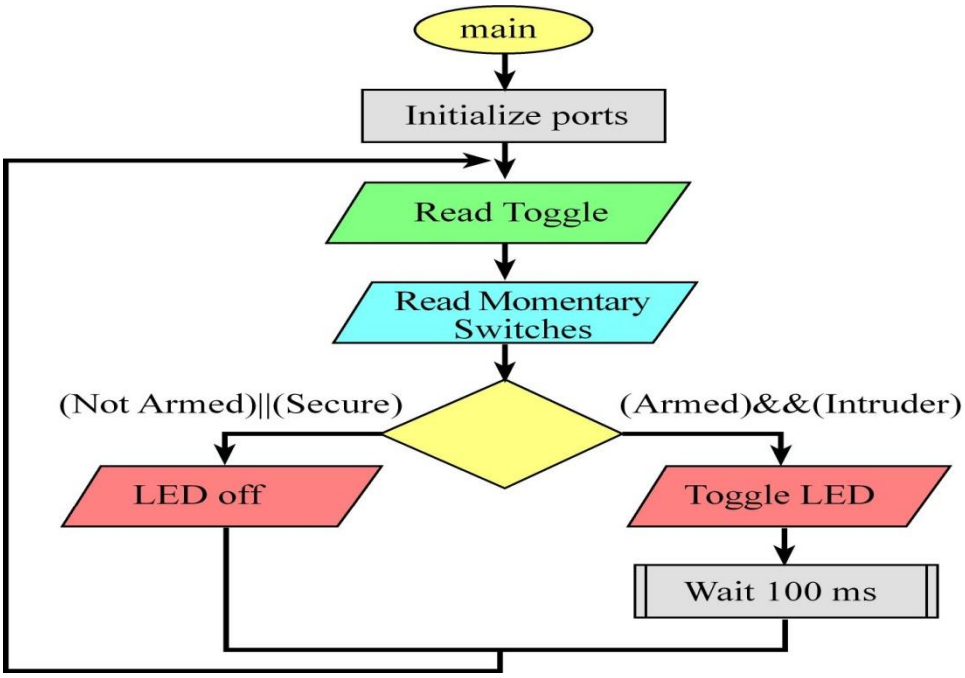


Figure 8.10. Flowchart for the Software

The code that implements the flowchart design described above is given below. This solution uses a friendly approach to accessing Port E. Which implies that we could use the other pins in Port E without changing this code.

```
unsigned long arm,sensor;
```

```
void delayms(unsigned long ms);
void EnableInterrupts(void);
int main(void){ unsigned long volatile delay;
  TExaS_Init();  // activate multimeter, 80 MHz
  SYSCTL_RCGC2_R |= 0x10;           // Port E clock
  delay = SYSCTL_RCGC2_R;           // wait 3-5 bus cycles
  GPIO_PORTE_DIR_R |= 0x10;         // PE4 output
  GPIO_PORTE_DIR_R &= ~0x07;        // PE2,1,0 input
  GPIO_PORTE_AFSEL_R &= ~0x17;      // not alternative
  GPIO_PORTE_AMSEL_R &= ~0x17;      // no analog
  GPIO_PORTE_PCTL_R &= ~0x000F0FFF; // bits for PE4,PE2,PE1,PE0
  GPIO_PORTE_DEN_R |= 0x17;         // enable PE4,PE2,PE1,PE0
  EnableInterrupts();
  while(1){
    arm = GPIO_PORTE_DATA_R&0x04;    // arm 0 if deactivated, 1 if activated
    sensor = GPIO_PORTE_DATA_R&0x03; // 1 means ok, 0 means break in
    if((arm==0x04)&&(sensor != 0x03)){
      GPIO_PORTE_DATA_R ^= 0x10;   // toggle output for alarm
      delayms(100);  // 100ms delay makes a 5Hz period
    }else{
      GPIO_PORTE_DATA_R &= ~0x10;   // LED off if deactivated
    }
  }
}
```

Program 8.3. Software system that flashes the LED if it is armed and if there is an intruder.

8.4.5 Testing

As a general practice embedded systems developers start with first testing their solutions in a simulated environment (if possible) before running it on the real board with real bardware. Note that, just because your testing proves successful in simulation it does not mean it will succeed on the real board. However, failure to run in simulation almost always guarantees that it will fail on the real board.

8.4.6 Conclusion

We have successfully designed, built and tested a Security system. As a last step, we'll look back at the requirements and see if we met the timing specifications. In particular, we will check to see if our calculations used for sizing the resistors in the switch and LED interfaces match actual observations.

Chapter 9: Arrays, Time, and Functional Debugging
Embedded Systems - Shape the World
Jonathan Valvano and Ramesh Yerraballi

In this chapter, we will illustrate a formal method for testing. Because embedded systems are deployed in safety-critical systems, we need to be rigorous in our methods that evaluate if the deployed system performs its tasks as required. Embedded systems not only need to arrive at the correct answer, they need to arrive at it at the correct time. We will introduce a simple hardware counter that we can use to measure time. The testing method we will develop in this chapter will be to create a data logger to store when and what our system is doing.

Learning Objectives:

- Understand the concept of minimally intrusive debugging
- Learn how the SysTick counter works
- Learn about arrays
- Learn how to use indexing to access arrays
- Understand precision, length and origin
- Learn how to create a debugging dump
- Understand that a formal method to verify program correctness
- Learn how to use the dump to collect debugging information in real time

9.1. Debugging Theory

Every programmer is faced with the need to debug and verify the correctness of his or her software. A debugging **instrument** is hardware or software used for the purpose of debugging. In this class, we will study hardware-level probes like the logic analyzer, oscilloscope, and Joint Test Action Group (JTAG standardized as the IEEE 1149.1) interface; software-level tools like simulators, monitors, and profilers; and manual tools like inspection and print statements. **Nonintrusiveness** is the characteristic or quality of a debugger that allows the software/hardware system to operate normally as if the debugger did not exist. Intrusiveness is used as a measure of the degree of perturbation caused in system performance by the debugging instrument itself. For example, a print statement added to your source code is very intrusive because it significantly affects the real-time interaction of the hardware and software. It is important to quantify the intrusiveness of an instrument. Let t be the average time it takes to run the software code comprising the debugging instrument. This time t is how much less time the system has, in order to perform its regular duties. Let Δt be the average time between executions of the instrument. A quantitative measure of intrusiveness is $t/\Delta t$, which is the fraction of the time consumed by the process of debugging itself. A debugging instrument is classified as **minimally intrusive** if it has a negligible effect on the system being debugged. In other words, if $t/\Delta t$ is so small that the debugging activities have a finite but inconsequential effect on the system behavior, we classify it as minimally intrusive. In a real microcomputer system, breakpoints and single-stepping are intrusive, because the real hardware continues to change while the software has stopped. When a program interacts with real-time events, the performance can be significantly altered when using intrusive debugging tools. On the other hand, we will learn later in this chapter that dumps, dumps with filter, and monitors (e.g., which output strategic information on LEDs or an OLED display) are much less intrusive. A logic analyzer that passively monitors the activity of the software is completely nonintrusive. Interestingly, breakpoints and single-stepping on a mixed hardware/software simulator are often nonintrusive, because the simulated hardware and the simulated software are affected together.

For example, the heartbeat code GPIO_PORTF_DATA_R ^= 0x02; requires only 6 bus cycles to execute. If the heartbeat runs every 1ms, and the bus clock is 80 MHz, then is equal to 6/80000. Normally, if this ratio is less than 1/1000 we classify it minimally intrusive.

Checkpoint 9.1: What does it mean for a debugging instrument to be minimally intrusive? Give both a general answer and a specific criterion.

Research in the area of program monitoring and debugging mirrors the rapid pace of developments in other areas of computer architecture and software systems. Because of the complexity explosion in computer systems, effective debugging tools are essential. The critical aspect of debugging an embedded system is the ability to see what the software is doing, where it is executing, and when it did it, without the debugger itself modifying system behavior. Terms such as program testing, diagnostics, performance debugging, functional debugging, tracing, profiling, instrumentation, visualization, optimization, verification, performance measurement, and execution measurement have specialized meanings, but they are also used interchangeably, and they often describe overlapping functions. For example, the terms profiling, tracing, performance measurement, or execution measurement may be used to describe the process of examining a program from a time perspective. But, tracing is also a term that may be used to describe the process of monitoring a program state or history for functional errors, or to describe the process of stepping through a program with a debugger. Usage of these terms among researchers and users vary.

Furthermore, the meaning and scope of the term debugging itself is not clear. In this class the goal of debugging is to maintain and improve software, and the role of a debugger is to support this endeavor. The debugging process is defined as testing, stabilizing, localizing, and correcting errors. Although testing, stabilizing, and localizing errors are important and essential to debugging, they are auxiliary processes: the primary goal of debugging is to remedy faults or to correct errors in a program.

Stabilization is process of fixing the inputs so that the system can be run over and over again yielding repeatable outputs.

Although, a wide variety of program monitoring and debugging tools are available today, in practice it is found that an overwhelming majority of users either still prefer or rely mainly on “rough and ready” manual methods for locating and correcting program errors. These methods include desk-checking, dumps, and print statements, with print statements being one of the most popular manual methods. Manual methods are useful because they are readily available, and they are relatively simple to use. But, the usefulness of manual methods is limited: they tend to be highly intrusive, and they do not provide adequate control over repeatability, event selection, or event isolation. A real-time system, where software execution timing is critical, usually cannot be debugged with simple print statements, because the print statement itself will require too much time to execute.

Black-box testing is simply observing the inputs and outputs without looking inside. Black-box testing has an important place in debugging a module for its functionality. On the other hand, **white-box testing** allows you to control and observe the internal workings of a system. A common mistake made by new engineers is to just perform black box testing. Effective debugging uses both. One must always start with black-box testing by subjecting a hardware or software module to appropriate test-cases. Once we document the failed test-cases, we can use them to aid us in effectively performing the task of white-box testing.

A print statement is a common example of a debugging instrument. Using the editor, one adds print statements to the code that either verify proper operation or illustrate the programming errors. If we test a system, then remove the instruments, the system may actually stop working, because of the importance of timing in embedded systems. If we leave debugging instruments in the final product, we can use the instruments to test systems on the production line, or test systems returned for repair. On the other hand, sometimes we wish to provide a mechanism to reliably and efficiently remove all instruments when the debugging is done. Consider the following mechanisms as you develop your own unique debugging style.

• Place all instruments in a unique column, so you can easily distinguish instruments from regular programs.
• Define all debugging instruments as functions that all have a specific pattern in their names. In this way, the find/replace mechanism of the editor can be used to find all the calls to the instruments.
• Define the instruments so that they test a run time global flag. When this flag is turned off, the instruments perform no function. Notice that this method leaves a permanent copy of the debugging code in the final system, causing it to suffer a runtime overhead, but the debugging code can be activated dynamically without recompiling. Many commercial software applications utilize this method because it simplifies "on-site" customer support.
• Use conditional compilation (or conditional assembly) to turn on and off the instruments when the software is compiled. When the assembler or compiler supports this feature, it can provide both performance and effectiveness.

The emergence of concurrent languages and the increasing use of embedded real-time systems place further demands on debuggers. The complexities introduced by the interaction of multiple events or time dependent processes are much more difficult to debug than errors associated with sequential programs. The behavior of non-real-time sequential programs is reproducible: for a given set of inputs their outputs remain the same. In the case of concurrent or real-time programs this does not hold true. Control over repeatability, event selection, and event isolation is even more important for concurrent or real-time environments.

Checkpoint 9.2: Consider the difference between a runtime flag that activates a debugging command versus an assembly/compile-time flag. In both cases it is easy to activate/deactivate the debugging statements. For each method, list one factor for which that method is superior to the other.
Checkpoint 9.3: What is the advantage of leaving debugging instruments in a final delivered product?
Observation: There are two important components of debugging: having control over events and being able to see what is happening. Remember: **control** and **observability**!
Common Error: The most common debugging mistake new programmers make is to simply observe the overall inputs and outputs system without looking inside the device. Then they go to their professor and say, "My program gives incorrect output. Do you know why?"

9.2. SysTick Timer

Before we describe the process of instumentation, we will discus a feature that exists on all Cortex™-M microcontrollers, a timer, called **SysTick**. Therefore, the use of SysTick in designing your system will assure you that your system will easily port to other Cortex-M microcontrollers. SysTick is a simple counter that we can use to create time delays and generate periodic interrupts. Table 9.1 shows the register definitions for SysTick. The basis of SysTick is a 24-bit down counter that runs at the bus clock frequency. There are four steps involved in the initialization of the SysTick timer. First, we clear the **ENABLE** bit to turn off SysTick during initialization. Second, we set the **RELOAD** register. Third, we write to the **`NVIC_ST_CURRENT_R`** value to clear the counter. Lastly, we write the desired mode to the control register, **`NVIC_ST_CTRL_R`**. The mode involves the **CLK_SRC** and the **INTEN** bits. We set the **CLK_SRC** bit specifying the core clock will be used. We will set **CLK_SRC**=1, so the counter runs off the system clock. Later, we will set **INTEN** to enable interrupts, but in this first example we clear **INTEN** so interrupts will not be requested. We need to set the **ENABLE** bit so the counter will run. Once the initialization is complete, the timer starts to count down, i.e., **CURRENT** is decremented once every clock tick.

When the **CURRENT** value counts down from 1 to 0, the **COUNT** flag is set. On the next clock, the **CURRENT** is loaded with the **RELOAD** value. In this way, the SysTick counter (**CURRENT**) is continuously decrementing. If the **RELOAD** value is *n*, then the SysTick counter operates at modulo *n*+1 (…*n*, *n*-1, *n*-2 … 1, 0, *n*, *n*-1, …). In other words, it rolls over every *n*+1 counts. In this chapter we set **RELOAD** to 0x00FFFFFF, so the **CURRENT** value is a simple indicator of what count is now.

Noting what the count was at some point and then what it is now, allows us to calculate the time that has elapsed

Address	31-24	23-17	16	15-3	2	1	0	Name
$E000E010	0	0	COUNT	0	CLK_SRC	INTEN	ENABLE	NVIC_ST_CTRL_R
$E000E014	0	24-bit RELOAD value						NVIC_ST_RELOAD_R
$E000E018	0	24-bit CURRENT value of SysTick counter						NVIC_ST_CURRENT_R

Table 9.1. SysTick registers.

Without activating the phase-lock-loop (PLL), our TM4C123 LaunchPad will run at 16 MHz, meaning the SysTick counter decrements every 62.5 ns. If we activate the PLL to run the microcontroller at 80 MHz, then the SysTick counter decrements every 12.5 ns. In general, if the period of the core bus clock is *t*, then the **COUNT** flag will be set every (*n*+1)*t*. Note that, reading the **NVIC_ST_CTRL_R** control register will return the **COUNT** flag in bit 16. The act of reading this register when the **COUNT** flag is set will automatically clear it (post read). Also, writing any value to the **NVIC_ST_CURRENT_R** register will reset the counter to zero and clear the **COUNT** flag. Program 9.1 initializes the SysTick. To determine the time, one simply reads the **NVIC_ST_CURRENT_R** register.

```
#define NVIC_ST_CTRL_R      (*((volatile unsigned long *)0xE000E010))
#define NVIC_ST_RELOAD_R    (*((volatile unsigned long *)0xE000E014))
#define NVIC_ST_CURRENT_R   (*((volatile unsigned long *)0xE000E018))
void SysTick_Init(void){
  NVIC_ST_CTRL_R = 0;               // 1) disable SysTick during setup
  NVIC_ST_RELOAD_R = 0x00FFFFFF;    // 2) maximum reload value
  NVIC_ST_CURRENT_R = 0;            // 3) any write to current clears it
  NVIC_ST_CTRL_R = 0x00000005;      // 4) enable SysTick with core clock
}
```

Program 9.1. Initialization of SysTick.

Program 9.2 shows how to measure the elapsed time between calls to a function. Assume the system calls the function **Action()** over and over. The variable **Now** is the time (in 12.5ns units) when the function has been called. The variable **Last** is the time (also in 12.5ns units) when the function was called previously. To measure elapsed time, we perform a time subtraction. Since the SysTick counts down we subtract **Last-Now**. Since the time is only 24 bits and the software variables are 32 bits we "**and"** with 0x00FFFFFF to create a 24-bit difference.

```
unsigned long Now;      // 24-bit time at this call (12.5ns)
unsigned long Last;     // 24-bit time at previous call (12.5ns)
unsigned long Elapsed;  // 24-bit time between calls (12.5ns)
void Action(void){      // function under test
  Now = NVIC_ST_CURRENT_R;          // what time is it now?
  Elapsed = (Last-Now)&0x00FFFFFF; // 24-bit difference
  Last = Now;                       // set up for next
  ...
}
```

Program 9.2. Use of SysTick to measure elapsed time.

The first measurement will be wrong because there is no previous execution from which to measure. The system will be accurate as long as the elapsed time is less than 0.209 second. More precisely, as long as the elapsed time is less than 2^{24}*12.5ns. This is similar to the problem of using an analog clock to measure elapsed time. For example you notice the clock says 10:00 when you go to sleep, and you notice it says 7:00 when you wake up. As long as you are sure you slept less than 12 hours, you are confident you slept for 9 hours.

Our TM4C123 microcontroller has some 32-bit and some 64-bit timers, but we will use SysTick because it is much simpler to configure. We just have to be aware that we are limited to 24 bits.

Checkpoint 9.4. If we activate the PLL and change the bus clock to 50 MHz (20ns), what is the longest elapsed time we could measure with Program 9.2?

9.3. Arrays

In developing our instrument we will need a place to put data. One of the simplest and fastest places to store data is in RAM memory. The TM4C123 has 32 kibibytes of RAM, and we can use it to store temporary data. If the information is a constant, and we know its values at compile time, we can place the data in ROM. The TM4C123 has 256 kibibytes of flash ROM, and we can use it to store constant data.

Random access means one can read and write any element in any order. Random access is allowed for all indexable data structures. An indexed data structure has elements of the same size and can be accessed knowing the name of the structure, the size of each element, and the element number. In C, we use the syntax **`[]`** to access an indexed structure. Arrays, matrices, and tables are examples of indexed structures available in C.

Sequential access means one reads and writes the elements in order. Pointers are usually employed in these types of data structures. Strings, linked-lists, stacks, queues, and trees are examples of sequential structures. The first in first out circular queue (**FIFO**) is useful for data flow problems.

An **array** is made of elements of equal precision and allows random access. The **precision** is the size of each element. Typically, precision is expressed in bits or bytes. The **length** is the number of elements. The **origin** is the index of the first element. A data structure with the first element existing at index zero is called **zero-origin indexing**. In C, zero-origin index is the default. For example, **`Data[0]`** is the first element of the array **`Data`**.

Just like any variable, arrays must be declared before they can be accessed. The number of elements in an array is determined by its declaration. Appending a constant expression in square brackets to a name in a declaration identifies the name as the name of an array with the number of elements indicated. Multi-dimensional arrays require multiple sets of brackets. These examples illustrate valid declarations of arrays. Because there is no **`const`** modifier, the arrays will be defined in RAM. The C compiler will add code that runs before your main, which will initialize all RAM-based variables to zero.

```
short Data[5];      // allocate space for 5 16-bit integers
long scores[20];    // allocate space for 20 32-bit integers
int Width[6];       // 6 signed, precision depends on compiler
char Image[5][10];  // allocate space for 50 8-bit integers
short Points[5][5][5];// allocate space for 125 16-bit ints
```

If you would like the compiler to initialize to something other than zero, you can explicitly define the initial values. Again, this initialization will occur before your main is executed. Even though they have nonzero initial values, these arrays are in RAM, and thus can be modified at run time.

```
short Data[5]={1,-1,2,-2,6};
long scores[20]={-2,-1,0,1,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2}
int Width[6]={-10,-20,10,100,2000,40};
short Image[5][10]={
  {0,0,1,0,1,1,1,0,1,0},
  {0,0,1,0,0,1,0,0,1,0},
  {0,0,1,0,0,1,0,0,1,0},
  {0,0,1,0,0,0,0,0,1,0},
  {0,0,0,1,1,1,1,1,0,0}};
```

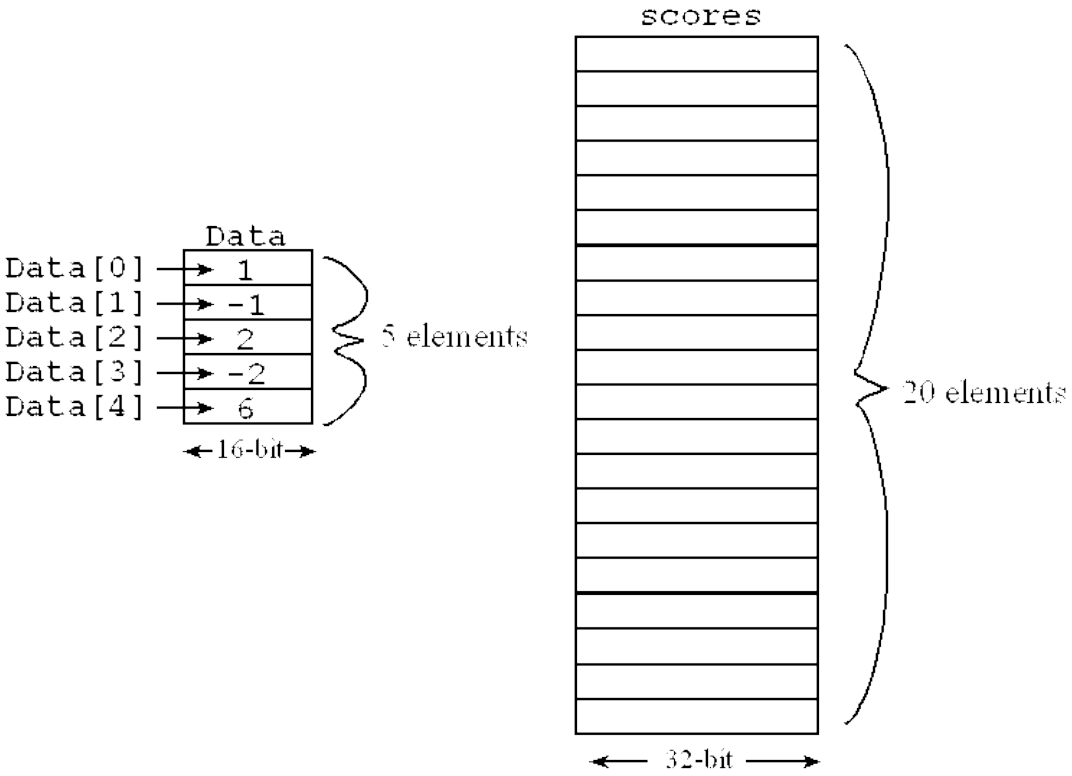


Figure 9.1. The ***Data*** *array has 5 16-bit elements, and the* ***scores*** *array has 20 32-bit elements.*

If the array contains constants, and we know the values at compile time, we can place the data in ROM using the **const** modifier. For ROM-based data, we must define the value explicitly in the software. These arrays are in ROM, and thus cannot be modified at run time.

```
const short Data2[5]={1,2,3,4,5};
const long scores2[20]={-2,-1,0,1,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2}
const int Width2[6]={-10,-20,10,100,2000,40};
const short Image2[5][10]={
  {0,0,1,0,1,1,1,0,1,0},
  {0,0,1,0,0,1,0,0,1,0},
  {0,0,1,0,0,1,0,0,1,0},
  {0,0,1,0,0,0,0,0,1,0},
  {0,0,0,1,1,1,1,1,0,0}};
```

In a formal way, **const** actually means “constant”. More specifically it means the executing software cannot change the value at run time. This means in C programs not implemented on an embedded system, **const** data could be stored in RAM, initialized at startup. In this case, the **const** modifier simply means the running program cannot change the values. However, in the class we are writing C code that will be deployed into a microcontroller as part of an embedded system, so **const** data will be stored in ROM.

When an array element is referenced, the subscript expression designates the desired element by its position in the data. The first element occupies position zero, the second position one, and so on. It follows that the last element is subscripted by [N-1] where N is the number of elements in the array. The statement

```
scores[9] = 0;
```

for instance, sets the tenth element of the array to zero. The array subscript can be any expression that results in an integer. The following for-loop clears 20 elements of the array data to zero.

```
for(j=0;j<20;j++) scores[j] = 0;
```

If the array has two dimensions, then two subscripts are specified when referencing. As programmers we may any assign logical meaning to the first and second subscripts. For example we could consider the first subscript as the row and the second as the column. Then, the statement

```
TheValue = Image[3][5];
```

copies the information from the 4th row 6th column into the variable **TheValue**. If the array has three dimensions, then three subscripts are specified when referencing. Again we may assign any logical meaning to the various subscripts. For example we could consider the first subscript as the x coordinate, the second subscript as the y coordinate and the third subscript as the z coordinate. Then, the statement

```
Points[2][3][4]=100;
```

sets the values at position (2,3,4) to 100. Array subscripts are treated as signed integers. It is the programmer's responsibility to see that only positive values are produced, since a negative subscript would refer to some point in memory preceding the array. One must be particularly careful about assuming what existing either in front of or behind our arrays in memory.

Example 9.1. Assume there are 50 students in the class, and their grades on the first exam are stored in an array. Write a function that calculates and returns class average. Write a second function that finds and returns the highest grade on the exam.

Solution: Assuming the grades vary from 0 to 100, the data could be stored in **char**, **short** or **long** format. To be compatible with the ARM architecture, we will create a 32-bit array. In this example, we will not worry about how grades are entered, but assume there is data in this array.

```
long Grades[50];
```

To calculate average, we sum the values and divide by the number of students. The return statement will return the 32-bit calculation of sum/50. This will never create an error, because the sum of 50 numbers ranging from 0 to 100 will always be less than 4 trillion (2^{32}) and the average of these numbers will never go below 0 or above 100.

```
long Average(void){long sum,i;
  sum = 0;
  for(i=0; i<50; i++){
    sum = sum+Grades[i];
  }
  return (sum/50);
}
```

Program 9.3. A function that calculates the average of data stored in an array.

To make this more general, we will pass in the array and the size.

```
long Average(long class[],long size){
long sum,i;
  sum = 0;
  for(i=0; i<size; i++){
    sum = sum+class[i];  // add up all values
  }
  return (sum/size);
}
```

The array parameter is actually a pointer to the array. So to use this new function we call

```
MyClassAverage = Average(Grades, 50);
```

Similarly to find the largest we will search the array. We initialized the result parameter (**largest**) to the smallest possible answer, and then every time we find one larger, we replace the result parameter.

```
long Max(long class[],long size){
long largest,i;
  largest = 0;  // smallest possible value
  for(i=0; i<size; i++){
    if(class[i] > largest){
      largest = class[i];  // new maximum
    }
  }
  return (largest);
}
```

To use this new function we call

```
HighestsScore = Max(Grades, 50);
```

Example 9.2. Design an exponential function, $y = 10^x$, with a 32-bit output.

Solution: Since the output is less than 4,294,967,295, the input must be between 0 and 9. One simple solution is to employ a constant word array, as shown in Figure 9.3. Each element is 32 bits. In assembly, we define a word constant using **DCD**, making sure in exists in ROM.

In C, the syntax for accessing all array types is independent of precision. See Program 9.4. The compiler automatically performs the correct address correction. We will assume the input is less than or equal to 9. If **x** is the index and **Base** is the base address of the array, then the address of the element at **x** is **Base+4*x**. In assembly, we can access the array using indexed addressing. We will assume the input (which is in Register R0) is less than or equal to 9. We show the asembly equivalent of the C code below (and examples that follow) just to pique your curiosity, you may ignore it if you choose.

0x00000134	1
0x00000138	10
0x0000013C	100
0x00000140	1,000
0x00000144	10,000
0x00000148	100,000
0x0000014C	1,000,000
0x00000150	10,000,000
0x00000154	100,000,000
0x00000158	1,000,000,000

Figure 9.2. A word array with 10 elements. Addresses illustrate the array is stored in ROM as 4 bytes each.

```
      AREA  |.text|,CODE,READONLY,ALIGN=2
Powers DCD  1, 10, 100, 1000, 10000
       DCD  100000, 1000000, 10000000
       DCD  100000000, 1000000000
; Input: R0=x    Output: R0=10^x
power  LSL R0, R0, #2   ; x = x*4
       LDR R1, =Powers  ; R1 = &Powers
       LDR R0, [R0, R1] ; y=Powers[x]
       BX  LR
```

```
const unsigned long Powers[10]
  ={1,10,100,1000,10000,
    100000,1000000,10000000,
    100000000,1000000000};

unsigned long power(unsigned long x){
  return Powers[x];
}
```

Program 9.4. Array implementation of a nonlinear function.

In general, let **n** be the precision of a zero-origin indexed array in bytes. If **I** is the index and **Base** is the beginning address of the array, then the address of the element at **I** is

```
Base+n*I
```

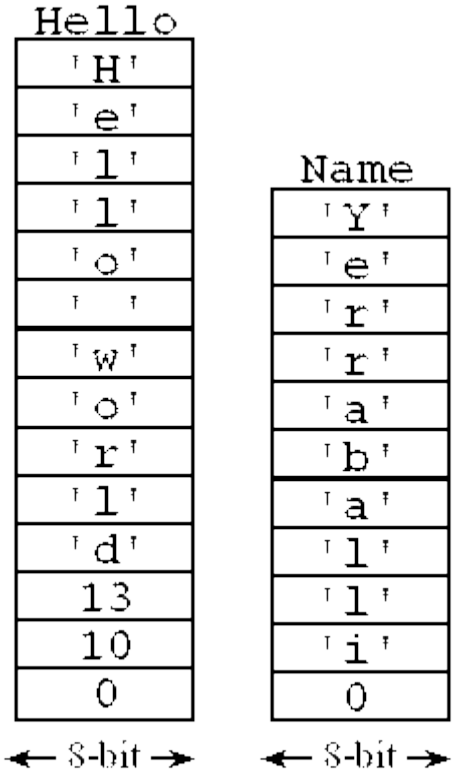
The origin of an array is the index of the first element. The origin of a zero-origin indexed array is zero. In general, if **o** is the origin of the array, then the address of the element at **I** is

```
Base+n*(I-o)
```

9.4. Strings

A **string** is simply an array of ASCII characters. In C, strings are automatically null-terminated by the compiler:

```
const char Hello[] = "Hello world\n\r";
const char Name[] = "Yerraballi";
```



When defining constant strings or arrays we must specify their value because the data will be loaded into ROM and cannot be changed at run time. In the previous constant arrays we specified the size; however for constant arrays the size can be left off and the compiler will calculate the size automatically. Notice also the string can contain special characters, some of which are listed in Table 9.1. The **Hello** string has 13 characters followed by a null (0) termination.

Character	*Escape Sequence*
alert (beep)	**\a**
backslash	****
backspace	**\b**
carriage return	**\r**
double quote	**\"**
form feed	**\f**
horizontal tab	**\t**
newline	**\n**

null character	`\0`
single quote	`\'`
vertical tab	`\v`
question mark	`\?`

Table 9.1. Escape sequences.

Note, that in C, ASCII strings are stored with null-termination. So, the C compiler automatically adds the zero at the end, but in assembly, the zero must be explicitly defined.

Example 9.3. Write software to output an ASCII string an output device.

Solution: Because the length of the string may be too long to place all the ASCII characters into the registers at the same time, call by reference parameter passing will be used. With call by reference, a pointer to the string will be passed. The function **`OutString`**, shown in Program 9.5, will output the string data to the display. We will assume the function **`OutChar`** is given to us, which outputs a single ASCII character. In C, we process one character (***pt** gives us the current character) of the string in each iteration of the while loop, which ends when the null value is reached. Each iteration advances the pointer by incrementing it (**pt**++). In the assembly version, R4 is used as a pointer to the string; one is added to the pointer each time through the loop because each element in the string is one byte. Since this function calls a subfunction it must save its original return address (LR). The POP PC operation will perform the function return.

```
;Input: R0 points to string
OutString
     PUSH {R4, LR}
                MOV             R4,
R0
loop LDRB R0, [R4]
     ADD  R4, #1  ;next
                CMP             R0,
#0  ;done?
          BEQ       done          ;0
termination
     BL   OutChar ;print character
     B    loop
done POP  {R4, PC}
```

```
// displays a string
void OutString(char *pt){
  while(*pt){
    OutChar(*pt); // output
    pt++;         // next
  }
}
```

Program 9.5. A variable length string contains ASCII data.

Observation: Most C compilers have standard libraries. If you include "string.h" you will have access to many convenient string operations.

When dealing with strings we must remember that they are arrays of characters with null termination. In C, we can pass a string as a parameter, but doing so creates a constant string and implements call by reference. Assuming **`Hello`** is as defined above, these three invocations are identical:

```
OutString(Hello);
OutString(&Hello[0]);
OutString("Hello world\n\r");
```

Previously we dealt with constant strings. With string variables, we do not know the length at compile time, so we must allocate space for the largest possible size the string could be. E.g., if we know the string size could vary from 0 to 19 characters, we would allocate 20 bytes.

```
char String1[20];
char String2[20];
```

In C, we cannot assign one string to another. I.e., these are illegal

```
String1 = "Hello";   //********illegal************
String2 = String1;   //********illegal************
```

We can make this operation occur by calling a function called **strcpy**, which copies one string to another. This function takes two pointers. We must however make sure the destination string has enough space to hold the string being copied.

```
strcpy(String1,"Hello");  // copies "Hello" into String1
strcpy(String2,String1);  // copies String1 into String2
```

Program 9.6 shows two implementations of this string copy function. R0 and R1 are pointers, and R2 contains the data as it is being copied. In this case, **dest++;** is implemented as an "add 1" because the data is one byte each (char). In other non-string situations, the increment pointer would be "add 2" for halfword data (short) and would be "add 4" for word data(long). Again, the C compiler does this automatically, but when writing in assembly one has to do this explicitly.

```
; Input: R0=&dest    R1=&source
strcpy LDRB R2,[R1]  ;source data
       STRB R2,[R0]  ;copy
       CMP  R2,#0    ;termination?
       BEQ  done
       ADD  R1,#1    ;next
       ADD  R0,#1
       B    strcpy
done   BX   LR
;faster version
strcpy LDRB R2,[R1],#1 ;source data
       STRB R2,[R0],#1 ;copy
       CMP  R2,#0      ;termination?
       BEQ  done
       B    strcpy
done   BX   LR
```

```
// copy string from source to dest
void    strcpy(char    *dest,    char
*source){
  while(*source){
    *dest = *source;  // copy
    dest++;           // next
    source++;
  }
  *dest = *source;   // termination
}
// another version
void    strcpy(char    *dest,    char
*source){
char data;
  do{
   data = *dest++ = *source++;
  } while(data);
}
```

Program 9.6. Simple string copy functions.

9.5. Functional debugging

9.5.1. Stabilization

Functional debugging involves the verification of input/output parameters. Functional debugging is a static process where inputs are supplied, the system is run, and the outputs are compared against the expected results. Four methods of functional debugging are presented in this section. There are two important aspects of debugging: control and observability. The first step of debugging is to **stabilize** the system. In the debugging context, we stabilize the system by creating a test routine that fixes (or stabilizes) all the inputs. In this way, we can reproduce the exact inputs over and over again. Stabilization is an effective approach to debugging because we can control exactly what software is being executed. Once stabilized, if we modify the program, we are sure that the change in our outputs is a function of the modification we made in our software and not due to a change in the input parameters. When a system has a small number of possible inputs (e.g., less than a million), it makes sense to test them all. When the number of possible inputs is large we need to choose a set of inputs. There are many ways to make this choice. We can select values:

Near the extremes and in the middle
Most typical of how our clients will properly use the system
Most typical of how our clients will improperly attempt to use the system
That differ by one

You know your system will find difficult
Using a random number generator

To stabilize the system we define a fixed set of inputs to test, run the system on these inputs, and record the outputs. Debugging is a process of finding patterns in the differences between recorded behavior and expected results. The advantage of modular programming is that we can perform modular debugging. We make a list of modules that might be causing the bug. We can then create new test routines to stabilize these modules and debug them one at a time. Unfortunately, sometimes all the modules seem to work, but the combination of modules does not. In this case we study the interfaces between the modules, looking for intended and unintended (e.g., unfriendly code) interactions.

9.5.2. Single Stepping

Many debuggers allow you to set the program counter to a specific address then execute one instruction at a time. The debugger provides three stepping commands **Step**, **StepOver** and **StepOut** commands. **Step** is the usual execute one assembly instruction. However, when debugging C we can also execute one line of C. **StepOver** will execute one assembly instruction, unless that instruction is a subroutine call, in which case the debugger will execute the entire subroutine and stop at the instruction following the subroutine call. **StepOut** assumes the execution has already entered a subroutine, and will finish execution of the subroutine and stop at the instruction following the subroutine call.

9.5.3. Breakpoints

A **breakpoint** is a mechanism to tag places in our software, which when executed will cause the software to stop. Normally, you can break on any line of your program.

One of the problems with breakpoints is that sometimes we have to observe many breakpoints before the error occurs. One way to deal with this problem is the conditional breakpoint. To illustrate the implementation of conditional breakpoints, add a global variable called **`Count`** and initialize it to 32 in the initialization ritual. Add the following conditional breakpoint to the appropriate location in your software. Using the debugger, we set a regular breakpoint at **`bkpt`**. We run the system again (you can change the 32 to match the situation that causes the error.)

```
            PUSH {R1, R2}    ; save R1 and R2
            LDR  R2, =Count  ; R2 = Count
            LDR  R1, [R2]    ; R1 = Count
            SUBS R1, R1, #1  ; Count = Count - 1
            STR  R1, [R2]    ; store to Count
            BNE  DEBUG_skip  ; if Count != 0, skip
DEBUG_bkpt  NOP              ; put breakpoint here
DEBUG_skip  POP {R1, R2}     ; restore R1 and R2
```

```
if(--Count==0)
   bkpt
```

Notice that the breakpoint occurs only on the 32^{nd} time the break is encountered. Any appropriate condition can be substituted. Most modern debuggers allow you to set breakpoints that will trigger on a count. However, this method allows flexibility of letting you choose the exact conditions that cause the break.

9.5.4. Instrumentation: Print Statements

The use of print statements is a popular and effective means for functional debugging. One difficulty with print statements in embedded systems is that a standard "printer" may not be available. Another problem with printing is that most embedded systems involve time-dependent interactions with its external environment. The print statement itself may be so slow, that the debugging process itself causes the system to fail. In this regard, the print statement is **intrusive**. Therefore, throughout this book we will utilize debugging methods that do not rely on the availability of a standard output device.

9.5.5. Instrumentation: Dump into Array without Filtering

There are three limitations of using print statements to debug. First, many embedded systems do not have a standard output device onto which we could stream debugging information. A second difficulty with print statements is that they can significantly slow down the execution speed in real-time systems. The bandwidth of the print functions often cannot keep pace with the real-time execution. For example, our system may wish to call a function 1000 times a second (or every 1 ms). If we add print statements to it that require more than 1 ms to perform, the presence of the print statements will cause the system to crash. In this situation, the print statements would be considered extremely intrusive. Another problem with print statements occurs when the system is using the same output hardware for its normal operation, as is required to perform the print function. For example, your watch may have an LCD, but that display is used to implement the watch functionality. If we output debugging information to the LCD, the debugger output and normal system output are intertwined.

To solve these limitations, we can add a debugging instrument that dumps strategic information into an array at run time. We can then observe the contents of the array at a later time. One of the advantages of dumping is that the JTAG debugger allows you to visualize memory even when the program is running. So this technique will be quite useful in systems with a JTAG debugger. Assume **happy** and **sad** are strategic 8-bit variables. The first step when instrumenting a dump is to define a buffer in RAM to save the debugging measurements.

```
SIZE     EQU 20
HappyBuf SPACE SIZE
SadBuf   SPACE SIZE
Cnt      SPACE 4
```

```
#define SIZE 20
unsigned char HappyBuf[SIZE];
unsigned char SadBuf[SIZE];
unsigned long Cnt;
```

The **Cnt** will be used to index into the buffers. **Cnt** must be initialized to zero, before the debugging begins. The debugging instrument, shown in Program 9.7, dumps the strategic variables into the buffers. When writing debugging instruments it is good style to preserve all registers.

```
Save PUSH {R0-R3,LR}
     LDR  R0,=Cnt  ;R0 = &Cnt
     LDR  R1,[R0]  ;R1 = Cnt
     CMP  R1,#SIZE
     BHS  done     ;full?
     LDR  R3,=Happy
     LDRB R3,[R3]  ;R3 is happy
     LDR  R2,=HappyBuf
     STRB R3,[R2,R1] ;save happy
     LDR  R3,=Sad
     LDRB R3,[R3]  ;R3 is sad
     LDR  R2,=SadBuf
     STRB R3,[R2,R1] ;save sad
     ADD  R1,#1
     STR  R1,[R0] ;save Cnt
done POP  {R0-R3,PC}
```

```
void Save(void){
  if(Cnt < SIZE){
    HappyBuf[Cnt] = happy;
    SadBuf[Cnt] = sad;
    Cnt++;
  }
}
```

Program 9.7. Instrumentation dump.

Next, you add **BL Save** statements at strategic places within the system. You can either use the debugger to display the results, or add software that prints the results after the program has run and stopped.

9.5.6. Instrumentation: Dump into Array with Filtering.

One problem with dumps is that they can generate a tremendous amount of information. If you suspect a certain situation is causing the error, you can add a **filter** to the instrument. A filter is a software/hardware condition that must be true in order to place data into the array. In this situation, if we

suspect the error occurs when another variable gets large, we could add a filter that saves in the array only when the variable is above a certain value. In the example shown in Program 9.8, the instrument dumps only when **sad** is greater than 100.

```
Save PUSH {R0-R3,LR}
     LDR  R3,=Sad
     LDRB R3,[R3]  ;R3 is sad
     CMP  R3,#100
       BLS    done       ;assuming
unsigned
     LDR  R0,=Cnt  ;R0 = &Cnt
     LDR  R1,[R0]  ;R1 = Cnt
     CMP  R1,#SIZE
     BHS  done     ;full?
     LDR  R2,=SadBuf
     STRB R3,[R2,R1] ;save sad
     LDR  R3,=Happy
     LDRB R3,[R3]  ;R3 is happy
     LDR  R2,=HappyBuf
     STRB R3,[R2,R1] ;save happy
     ADD  R1,#1
     STR  R1,[R0] ;save Cnt
done POP  {R0-R3,PC}
```

```
void Save(void){
  if(sad > 100){
    if(Cnt < SIZE*2){
      HappyBuf[Cnt] = happy;
      SadBuf[Cnt] = sad;
      Cnt++;
    }
  }
}
```

Program 9.8. Instrumentation dump with filter.

9.5.7. Prove that our system is functioning as intended.

When embedded systems are deployed in safety critical situations we need to document how the system was tested, and provide proof it is functioning as intended. We will illustrate the process with the example from module 2. In Lab02 the goal was to create a system that toggled the light at 5 Hz. This means the red LED should be on for 0.1 sec and off for 0.1 sec. If we look at the LED with our eyes it looks like it is running correctly. If we look at the signal on the oscilloscope or logic analyzer, again it looks correct. But can we prove it? In this first debugging process we will dump the value of PF1 in one array and the time difference in a second array, see Program 9.9. When we run this program, it reveals all 49 measurements where the average time difference is 1,599,996 bus cycles, which is 0.09999975 seconds, which is close to the desired time of 0.1 second.

```
// first data point is wrong, the other 49 will be correct
unsigned long Time[50];
unsigned long Data[50];
int main(void){  unsigned long i,last,now;
                 // initialize PF0 and PF4 and make them inputs
  PortF_Init();  // make PF3-1 out (PF3-1 built-in LEDs)
  SysTick_Init(); // initialize SysTick, runs at 16 MHz
  i = 0;         // array index
  last = NVIC_ST_CURRENT_R;
  while(1){
    Led = GPIO_PORTF_DATA_R;   // read previous
    Led = Led^0x02;            // toggle red LED
    GPIO_PORTF_DATA_R = Led;   // output
    if(i<50){
      now = NVIC_ST_CURRENT_R;
      Time[i] = (last-now)&0x00FFFFFF;  // 24-bit time difference
      Data[i] = GPIO_PORTF_DATA_R&0x02; // record PF1
      last = now;
      i++;
    }
    Delay();
  }
}
```

Program 9.9. Instrumentation to record the first 49 time differences (debugging shown in bold).

However compelling this data is, it doesn't prove it is always 0.1 sec. Let's further specify the desire is to create a system that is accurate to ±0.01%. This means any time difference 1,600,000±160 cycles is acceptable. In Program 9.10 we will count the number of times the time difference is unacceptable. If we run this program for a month we can observe its behavior over 25 million times. Furthermore, we can leave this debugging code into the deployed system, and verify the system is running as expected for the entire life of the system.

```
// first data point is wrong, the others will be correct
long Errors;
#define CORRECT 1600000
#define TOLERANCE 160
int main(void){  unsigned long last,now,diff;
                 // initialize PF0 and PF4 and make them inputs
  PortF_Init();   // make PF3-1 out (PF3-1 built-in LEDs)
  SysTick_Init(); // initialize SysTick, runs at 16 MHz
  Errors = -1;    // no errors (ignore first measurement)
  last = NVIC_ST_CURRENT_R;
  while(1){
    Led = GPIO_PORTF_DATA_R;   // read previous
    Led = Led^0x02;            // toggle red LED
    GPIO_PORTF_DATA_R = Led;   // output
    now = NVIC_ST_CURRENT_R;
    diff = (last-now)&0x00FFFFFF;  // 24-bit time difference
    if((diff<(CORRECT-TOLERANCE))||(diff>(CORRECT+TOLERANCE)){
      Error++;
    }
    last = now;
    Delay();
  }
}
```

Program 9.10. Instrumentation to count the number of mistakes (debugging shown in bold).

One interesting feature we could add to this system is to implement the **`Error`** count in ROM. The flash ROM allows the running program to change its value. The disadvantage of storing data in ROM is it takes over 1ms to cause the change. In situations where we have infrequent but important data, this 1ms overhead is not significant. The advantage of storing infrequent but important debugging information in ROM is that this data is available even if power is removed and restored. For example, if the embedded system were to be involved in a loss of life accident, the data stored in ROM could be recovered to determine if any run-time errors might have contributed to the accident. Conversely, this data stored in ROM could verify that no errors in the operation of the embedded system had occurred prior to the accident. If you wish to write to flash ROM, look at example projects called "flash" on http://users.ece.utexas.edu/~valvano/arm/.

Chapter 10: Finite State Machines
Embedded Systems - Shape the World
Jonathan Valvano and Ramesh Yerraballi

Time is a critical parameter in an embedded system. In this chapter, we will further develop SysTick as a means to control time in our embedded system. We will activate the phase-lock-loop (PLL) for two reasons. First, by selecting the bus frequency we can tradeoff power for speed. Second, by creating a bus clock based on an external crystal, system time will be very accurate. An effective development process will be to separate what the system does from how it works. This abstraction will be illustrated during the design of finite state machines (FSM). All embedded systems have inputs and outputs, but FSMs have states. We will embody knowledge, "what we know" or "where we've been", by being in a state. A traffic light and vending machine will be implemented using FSMs. Finally, we will introduce stepper motors and show how to use a FSM to control the motors.

Learning Objectives:

- Learn how to activate the PLL so the microcontroller has an accurate time base
- Use SysTick to produce accurate time delays
- Learn how to organize data on the computer using structures
- Develop a design strategy for building Finite State Machines
- Explain how stepper motors work using two motors to make an autonomous robot

10.1. Phase-Lock-Loop

Normally, the execution speed of a microcontroller is determined by an external crystal. The Stellaris® EK-LM4F120XL and EK-TM4C123GXL boards have a 16 MHz crystal. Most microcontrollers include a phase-lock-loop (PLL) that allows the software to adjust the execution speed of the computer. Typically, the choice of frequency involves the tradeoff between software execution speed and electrical power. In other words, slowing down the bus clock will require less power to operate and generate less heat. Speeding up the bus clock obviously allows for more calculations per second, at the cost of requiring more power to operate and generating more heat.

The default bus speed for the LM4F/TM4C internal oscillator is 16 MHz ±1%. The internal oscillator is significantly less precise than the crystal, but it requires less power and does not need an external crystal. The TExaS real-board grader has been turning on the PLL, and in this section we will explain how it work. If we wish to have accurate control of time, we will activate the external crystal (called the main oscillator) use the PLL to select the desired bus speed.

There are two ways to activate the PLL. We could call a library function, or we could access the clock registers directly. In general, using library functions creates a better design because the solution will be more stable (less bugs) and will be more portable (easier to switch microcontrollers). However, the objective of the class is to present microcontroller fundamentals. Showing the direct access does illustrate some concepts of the PLL.

An external crystal is attached to the LM4F/TM4C microcontroller, as shown in Figure 10.1. Table 10.1 shows the clock registers used to define what speed the processor operates. The output of the main oscillator (Main Osc) is a clock at the same frequency as the crystal. By setting the OSCSRC bits to 0, the multiplexer control will select the main oscillator as the clock source.

The main oscillator for the LM4F/TM4C LaunchPad will be 16 MHz. This means the reference clock (Ref Clk) input to the phase/frequency detector will be 16 MHz. For a 16 MHz crystal, we set the XTAL bits to 10101 (see Table 10.1). In this way, a 400 MHz output of the voltage controlled oscillator (VCO) will yield a 16 MHz clock at the other input of the phase/frequency detector. If the 400 MHz clock is too slow, the **up** signal will add to the charge pump, increasing the input to the VCO, leading to an increase

in the 400 MHz frequency. If the 400 MHz clock is too fast, **down** signal will subtract from the charge pump, decreasing the input to the VCO, leading to a decrease in the 400 MHz frequency. Because the reference clock is stable, the feedback loop in the PLL will drive the output to a stable 400 MHz frequency.

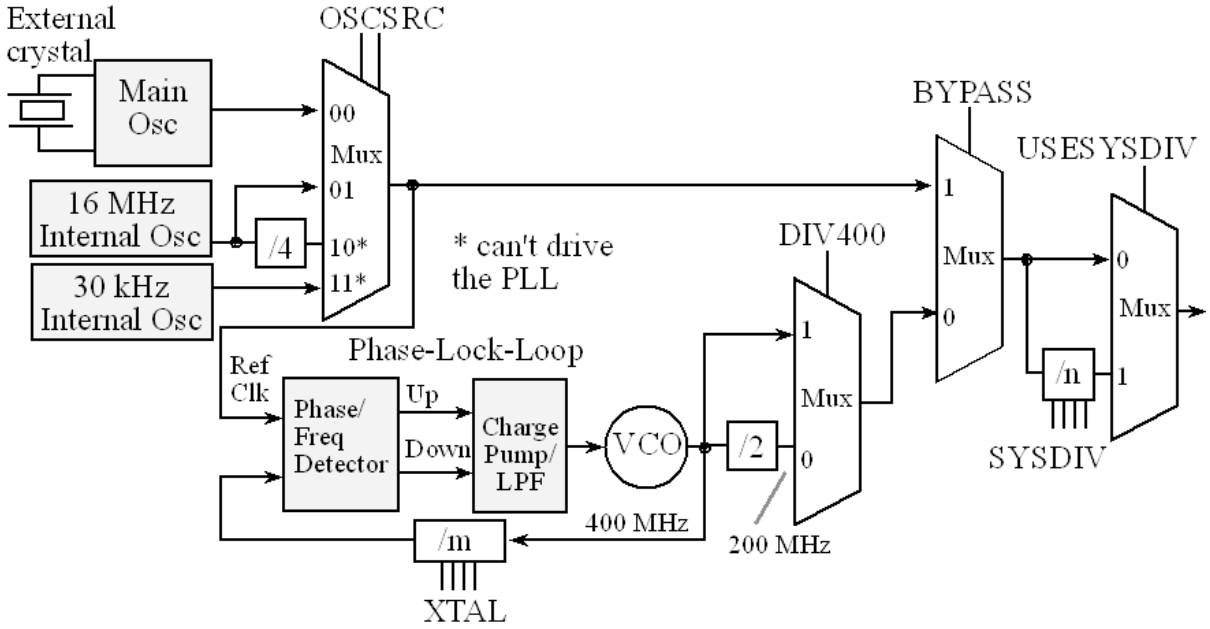


Figure 10.1. Block diagram of the main clock tree on the LM4F/TM4C including the PLL.

XTAL	*Crystal Freq (MHz)*
0x0	Reserved
0x1	Reserved
0x2	Reserved
0x3	Reserved
0x4	3.579545 MHz
0x5	3.6864 MHz
0x6	4 MHz
0x7	4.096 MHz
0x8	4.9152 MHz
0x9	5 MHz
0xA	5.12 MHz
0xB	6 MHz (reset value)
0xC	6.144 MHz
0xD	7.3728 MHz
0xE	8 MHz
0xF	8.192 MHz

XTAL	*Crystal Freq (MHz)*
0x10	10.0 MHz
0x11	12.0 MHz
0x12	12.288 MHz
0x13	13.56 MHz
0x14	14.31818 MHz
0x15	16.0 MHz
0x16	16.384 MHz
0x17	18.0 MHz
0x18	20.0 MHz
0x19	24.0 MHz
0x1A	25.0 MHz
0x1B	Reserved
0x1C	Reserved
0x1D	Reserved
0x1E	Reserved
0x1F	Reserved

Address	26-23	22	13	11	10-6	5-4	Name
$400FE060	SYSDIV	USESYSDIV	PWRDN	BYPASS	XTAL	OSCSRC	SYSCTL_RCC_R
$400FE050					PLLRIS		SYSCTL_RIS_R

	31	30	28-22	13	11	6-4	
$400FE070	USERCC2	DIV400	SYSDIV2	PWRDN2	BYPASS2	OSCSRC2	SYSCTL_RCC2_R

Table 10.1. Main clock registers (RCC2 in LM4F/TM4C only).

Program 10.1 shows the steps 0 to 6 to activate the LM4F123/TM4C123 Launchpad with a 16 MHz main oscillator to run at 80 MHz. 0) Use RCC2 because it provides for more options. 1) The first step is to set BYPASS2 (bit 11). At this point the PLL is bypassed and there is no system clock divider. 2) The

second step is to specify the crystal frequency in the four XTAL bits using the code in Table 10.1. The OSCSRC2 bits are cleared to select the main oscillator as the oscillator clock source. 3) The third step is to clear PWRDN2 (bit 13) to activate the PLL. 4) The fourth step is to configure and enable the clock divider using the 7-bit SYSDIV2 field. If the 7-bit SYSDIV2 is **n**, then the clock will be divided by **n**+1. To get the desired 80 MHz from the 400 MHz PLL, we need to divide by 5. So, we place a 4 into the SYSDIV2 field. 5) The fifth step is to wait for the PLL to stabilize by waiting for PLLRIS (bit 6) in the **SYSCTL_RIS_R** to become high. 6) The last step is to connect the PLL by clearing the BYPASS2 bit. To modify this program to operate on other microcontrollers, you will need to change the crystal frequency and the system clock divider.

Checkpoint 10.1: How would you change Program 10.1 if your microcontroller had an 8 MHz crystal and you wish to run at 50 MHz?

```
void PLL_Init(void){
  // 0) Use RCC2
  SYSCTL_RCC2_R |=  0x80000000;  // USERCC2
  // 1) bypass PLL while initializing
  SYSCTL_RCC2_R |=  0x00000800;  // BYPASS2, PLL bypass
  // 2) select the crystal value and oscillator source
  SYSCTL_RCC_R = (SYSCTL_RCC_R &~0x000007C0)   // clear XTAL field, bits 10-6
                 + 0x00000540;   // 10101, configure for 16 MHz crystal
  SYSCTL_RCC2_R &= ~0x00000070;  // configure for main oscillator source
  // 3) activate PLL by clearing PWRDN
  SYSCTL_RCC2_R &= ~0x00002000;
  // 4) set the desired system divider
  SYSCTL_RCC2_R |= 0x40000000;   // use 400 MHz PLL
  SYSCTL_RCC2_R = (SYSCTL_RCC2_R&~0x1FC00000)  // clear system clock divider
                  + (4<<22);      // configure for 80 MHz clock
  // 5) wait for the PLL to lock by polling PLLLRIS
  while((SYSCTL_RIS_R&0x00000040)==0){};  // wait for PLLRIS bit
  // 6) enable use of PLL by clearing BYPASS
  SYSCTL_RCC2_R& = ~0x00000800;
}
```

Program 10.1. Activate the LM4F/TM4C with a 16 MHz crystal to run at 80 MHz (PLL_xxx.zip).

We can make a first order estimate of the relationship between work done in the software and electrical power required to run the system. There are two factors involved in the performance of software: efficiency of the software and the number of instructions being executed per second

Software Work = algorithm * instructions/sec

In other words, if we want to improve software performance we can write better software or increase the rate at which we execute instructions. Recall that the compiler converts our C software into Cortex M machine code, so the efficiency of the compiler will also affect this relationship. Furthermore, most compilers have optimization settings that allow you to make your software run faster at the expense of using more memory. On the Cortex M, most instructions execute in 1 or 2 bus cycles. See section 3.3 in **CortexM4_TRM_r0p1.pdf** for more details. In CMOS logic, most of the electrical power required to run the system occurs in making signals change. It takes power to make a digital signal rise from 0 to 1, or fall from 1 to 0. It takes some power to run independent of frequency. Simplifying things greatly, we see a simple and linear relationship between bus frequency and electrical power. Let *m* be the slope of this linear relationship

$$\text{Power} = m * f_{Bus}$$

Some of the factors that affect the slope m are operating voltage and fundamental behavior of how the CMOS transistors are designed. If we approximate the Cortex M processor as being able to execute one

instruction every two bus cycles, we can combine the above two equations to see the speed-power tradeoff.

Software Work = algorithm * ½ f_{Bus} = algorithm * ½ Power/*m*

Observation: To save power, we slow down the bus frequency removing as much of the wasted bus cycles while still performing all of the required tasks.

For battery-powered systems the consumed power is a critical factor. For these systems we need to measure power. Since there are so many factors that determine power, the data sheets for the devices will only be approximate. Since power equals voltage times current, and we know the voltage (in our case 3.3V), we need to measure supply current. Figure 10.2 shows a current sense amplifier that can measure current to a Target system. We made a special printed circuit board placing a fixed resistor (R1=1 ohm in this circuit) between the 3.3V supply and the target system. The voltage across R1 is a linear function of the current to the target. The current sense amplifier (LT1187) amplifies this signal and the output of the amplifier is measured by an analog to digital converter.

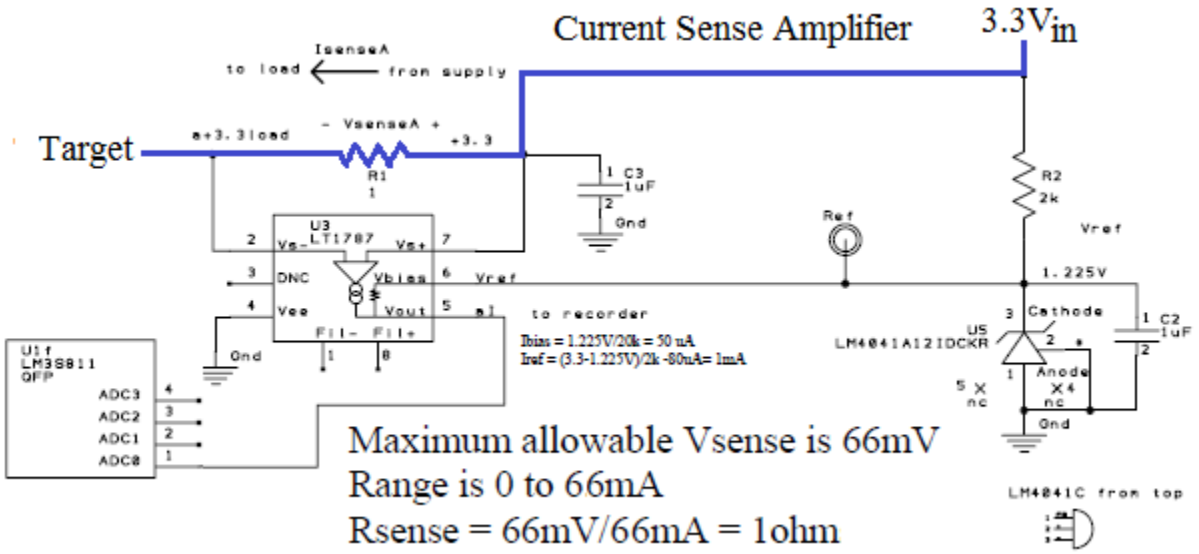


Figure 10.2. A current sense amplifier can be used to measure instantaneous current to the target. The blue trace shows the current path from supply to target.

Figure 10.3 shows a current measurement for a battery-powered system. In sleep mode all the clocks are turned off and the current drops to less than 1 µA. Just like the TM4C123 the software has control over which I/O devices are active. You can see from the data, the I/O devices are turned on one at a time (sample from ADC, transmit using RF, and receive using RF). The total energy needed to collect one measurement on this system can be found by multiplying current*voltage*time, where current and time are measured in Figure 10.3. In this calculation, assume we take on measurement every hour (sleeps for 1 hour, wakes up samples, transmits, receives, and then goes back to sleep).

(14mA*0.005s+31mA*0.007s+21mA*0.0046s+0.001mA*3600s)*3.3V/hr = 0.0013J/hr

A 3.3V battery with 100mA-hr has 0.33J of energy. This battery will run the system for 0.33J/0.0013J/hr = 25 hours

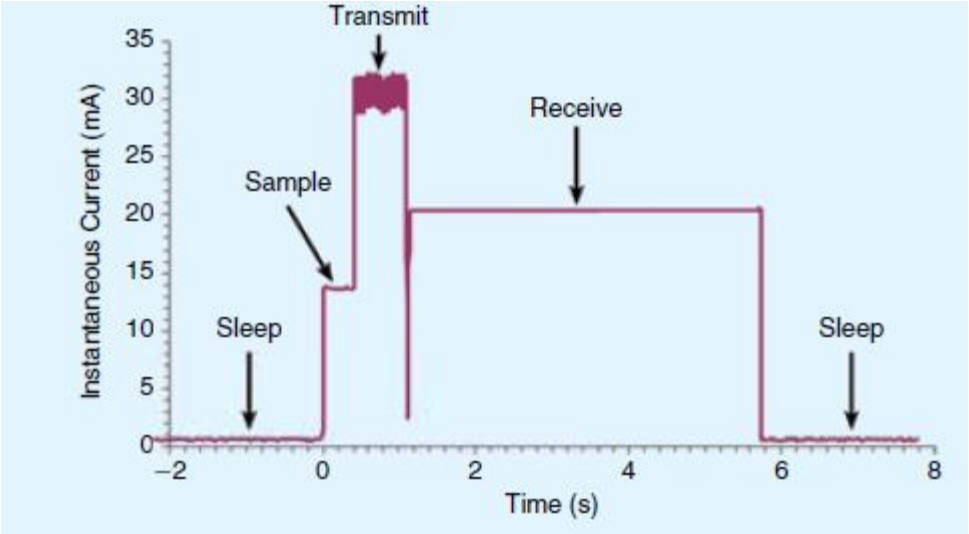


Figure 10.3. Instantaneous current measured on a battery powered system.

Observation: Being able to dynamically control bus frequency and I/O devices is important for low-power design.

10.2. Accurate Time Delays using SysTick

The accuracy of SysTick depends on the accuracy of the clock. We use the PLL to derive a bus clock based on the 16 MHz crystal, the time measured or generated using SysTick will be very accurate. More specifically, the accuracy of the NX5032GA crystal on the LaunchPad board is ±50 parts per million (PPM), which translates to 0.005%, which is about ±5 seconds per day. One could spend more money on the crystal and improve the accuracy by a factor of 10. Not only are crystals accurate, they are stable. The NX5032GA crystal will vary only ±150 PPM as temperature varies from -40 to +150 ºC. Crystals are more stable than they are accurate, typically varying by less than 5 PPM per year.

Program 10.2 shows a simple function to implement time delays based on SysTick. The **RELOAD** register is set to the number of bus cycles one wishes to wait. If the PLL function of Program 10.1 has been executed, then the units of this delay will be 12.5 ns. Writing to **CURRENT** will clear the counter and will clear the count flag (bit 16) of the **CTRL** register. After SysTick has been decremented **`delay`** times, the count flag will be set and the **`while`** loop will terminate. Since SysTick is only 24 bits, the maximum time one can wait with **`SysTick_Wait`** is 2^{24}*12.5ns, which is about 200 ms. To provide for longer delays, the function **`SysTick_Wait10ms`** calls the function **`SysTick_Wait`** repeatedly. Notice that 800,000*12.5ns is 10ms.

```
#define NVIC_ST_CTRL_R      (*((volatile unsigned long *)0xE000E010))
#define NVIC_ST_RELOAD_R    (*((volatile unsigned long *)0xE000E014))
#define NVIC_ST_CURRENT_R   (*((volatile unsigned long *)0xE000E018))
void SysTick_Init(void){
  NVIC_ST_CTRL_R = 0;               // disable SysTick during setup
  NVIC_ST_CTRL_R = 0x00000005;      // enable SysTick with core clock
}
// The delay parameter is in units of the 80 MHz core clock. (12.5 ns)
void SysTick_Wait(unsigned long delay){
  NVIC_ST_RELOAD_R = delay-1;  // number of counts to wait
  NVIC_ST_CURRENT_R = 0;       // any value written to CURRENT clears
  while((NVIC_ST_CTRL_R&0x00010000)==0){ // wait for count flag
  }
}
// 800000*12.5ns equals 10ms
void SysTick_Wait10ms(unsigned long delay){
  unsigned long i;
  for(i=0; i<delay; i++){
```

```
    SysTick_Wait(800000);  // wait 10ms
  }
}
```

Program 10.2. Use of SysTick to delay for a specified amount of time (SysTick_Wait_xxx.zip).

Checkpoint 10.2: What is the longest time one could wait using **`SysTick_Wait10ms`**?

10.3. Structures

A **structure** has elements with different types and/or precisions. In C, we use **`struct`** to define a structure. The **`const`** modifier causes the structure to be allocated in ROM. Without the **`const`**, the C compiler will place the structure in RAM, allowing it to be dynamically changed. If the structure were to contain an ASCII string of variable length, then we must allocate space to handle its maximum size. In this first example, the structure will be allocated in RAM so no **`const`** is included. The following code defines a structure with three elements. We give separate names to each element. In this example the elements are **`Xpos Ypos Score`**. The **`typedef`** command creates a new data type based on the structure, but no memory is allocated.

```
struct player{
  unsigned char Xpos;      // first element
  unsigned char Ypos;      // second element
  unsigned short Score;    // third element
};
typedef struct player playerType;
```

We can allocate a variable called **`Sprite`** of this type, which will occupy four bytes in RAM:

```
playerType Sprite;
```

We can access the individual elements of this variable using the syntax **`name.element`**. After these three lines are executed we have the data structure as shown in Figure 10.4 (assuming the variable occupies the four bytes starting at 0x2000.0250.

```
Sprite.Xpos = 10;
Sprite.Ypos = 20;
Sprite.Score = 12000;
```

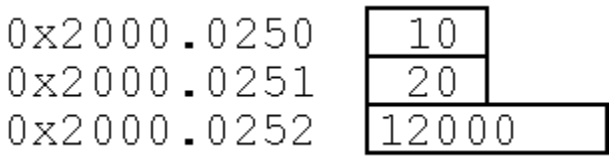


Figure 10.4. A structure collects elements of different sizes and/or types into one object.

We can also have an array of structures. We define structure array in a similar way as other arrays

```
playerType Ships[10];
unsigned long i;
```

While we are accessing an array, we must make sure the index is valid. In this case, the variable **`i`** must be between 0 and 9. We can read and write the individual fields using the syntax combining array access and structure access.

```
Ships[i].Xpos = 10;
Ships[i].Ypos = 20;
Ships[i].Score = 12000;
```

The C function in Program 10.3 takes a player, moves it to location 50,50 and adds one point. For example, we execute **`MoveCenter(6);`** to move the 7th ship to location 50,50 and increase its score.

```
// move to center and add to score
void MoveCenter(unsigned long i){
  Ships[i].Xpos = 50;
  Ships[i].Ypos = 50;
  if(Ships[i].Score < 65535){
    Ships[i].Score++;
  }
}
```

Program 10.3. A function that accesses a structure.

Observation: Most C compilers will align 16-bit elements within structures to an even address and will align 32-bit elements to a word-aligned address.

Without the **`const`**, the C compiler will place the structure in RAM, allowing it to be dynamically changed. If the structure resides in RAM, then the system will have to initialize the data structure explicitly by executing software. If the structure is in ROM, we must initialize it at compile time. The next section shows examples of ROM-based structures.

10.4. Finite State Machines with Indexed Structures

Software abstraction allows us to define a complex problem with a set of basic abstract principles. If we can construct our software system using these abstract building blocks, then we have a better understanding of both the problem and its solution. This is because we can separate what we are doing (policies) from the details of how we are getting it done (mechanisms). This separation also makes it easier to optimize. Abstraction provides for a proof of correct function and simplifies both extensions and customization. The abstraction presented in this section is the Finite State Machine (FSM). The abstract principles of FSM development are the inputs, outputs, states, and state transitions. The FSM state graph defines the time-dependent relationship between its inputs and outputs. If we can take a complex problem and map it into a FSM model, then we can solve it with simple FSM software tools. Our FSM software implementation will be easy to understand, debug, and modify. Other examples of software abstraction include Proportional Integral Derivative digital controllers, fuzzy logic digital controllers, neural networks, and linear systems of differential equations. In each case, the problem is mapped into a well-defined model with a set of abstract yet powerful rules. Then, the software solution is a matter of implementing the rules of the model. In our case, once we prove our software correctly solves one FSM, then we can make changes to the state graph and be confident that our software solution correctly implements the new FSM.

A Finite State Machine (FSM) is an abstraction that describes the solution to a problem very much like an Algorithm. Unlike an algorithm which gives a sequence of steps that need to be followed to realize the solution to a problem, a FSM describes the system (the solution being a realization of the system's behavior) as a machine that changes states in reaction to inputs and produces appropriate outputs. Many systems in engineering can be described using an FSM. First let's define what are the essential elements that constitute an FSM. A Finite Statement Machine can be described by these five essential elements:

1. A finite set of states that you can find the system in. One of these states has to be identified as the initial state
2. A finite set of external inputs to the system
3. A finite set of external outputs that the system generates
4. An explicit specification of all state transitions. That is, for every state, what happens (as in, which state will the system transition to) when you are in that state and a specific input occurs?
5. An explicit specification of how the outputs are determined. That is, when does a specific output get generated?

A representation of a system's behavior involves describing all five of these essential elements. Elements 4 and 5 are visually described using a State Transition Graph. We can also state 4 and 5 mathematically as follows:

- Element 4: The next state that the system goes into is a function of the input received and the current state. i.e.,
  NextState = f(Input, CurrentState)
- Element 5: The output that the system generates is a function of only the current state. i.e.,
  Output = g(CurrentState)

A State Transition Graph (STG) has nodes and edges, where the nodes relate to the states of the FSM and the edges represent the transitions from one state to another when a particular input is received. Edges are accordingly labeled with the input that caused the transition. The output can also be captured in the Graph. Note that a FSM where the output is only dependent on the current state and not the input is called a *Moore* FSM. FSMs where the output is dependent on both the current state and the input are called *Mealy* FSMs i.e.,

Output = h(Input, CurrentState)

We note that some systems lend themselves better to a Mealy description while others are more naturally expressed as Moore machines. However, both machine descriptions are equivalent in that any system that can be described using a Mealy machine can also be expressed equivalently as a Moore machine and vice versa. See Figure 10.5.

The FSM controller employs a well-defined model or framework with which we solve our problem. STG will be specified using either a linked or table data structure. An important aspect of this method is to create a 1-1 mapping from the STG into the data structure. The three advantages of this abstraction are 1) it can be faster to develop because many of the building blocks preexist; 2) it is easier to debug (prove correct) because it separates conceptual issues from implementation; and 3) it is easier to change. When designing a FSM, we begin by defining what constitutes a state. In a simple system like a single intersection traffic light, a state might be defined as the pattern of lights (i.e., which lights are on and which are off). In a more sophisticated traffic controller, what it means to be in a state might also include information about traffic volume at this and other adjacent intersections. The next step is to make a list of the various states in which the system might exist. As in all designs, we add outputs so the system can affect the external environment, and inputs so the system can collect information about its environment or receive commands as needed. The execution of a Moore FSM repeats this sequence over and over

1. Perform output, which depends on the current state
2. Wait a prescribed amount of time (optional)
3. Input
4. Go to next state, which depends on the input and the current state

The execution of a Mealy FSM repeats this sequence over and over

1. Wait a prescribed amount of time (optional)
2. Input
3. Perform output, which depends on the input and the current state
4. Go to next state, which depends on the input and the current state

There are other possible execution sequences. Therefore, it is important to document the sequence before the state graph is drawn. The high-level behavior of the system is defined by the state graph. The states are drawn as circles. Descriptive states names help explain what the machine is doing. Arrows are drawn from one state to another, and labeled with the input value causing that state transition.

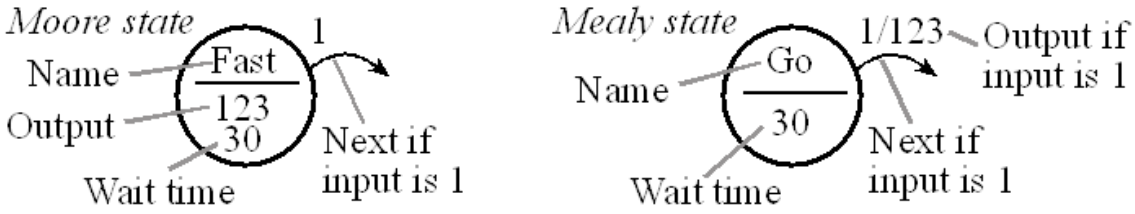


Figure 10.5. The output in a Moore depends just on the state. In a Mealy the output depends on state and input.

Observation: If the machine is such that a specific output value is necessary "to be a state", then a Moore implementation will be more appropriate.

Observation: If the machine is such that no specific output value is necessary "to be a state", but rather the output is required to transition the machine from one state to the next, then a Mealy implementation will be more appropriate.

A linked structure consists of multiple identically-structured nodes. Each node of the linked structure defines one state. One or more of the entries in the node is a link to other nodes. In an embedded system, we usually use statically-allocated fixed-size linked structures, which are defined at compile time and exist throughout the life of the software. In a simple embedded system the state graph is fixed, so we can store the linked data structure in nonvolatile memory. For complex systems where the control functions change dynamically (e.g., the state graph itself varies over time), we could implement dynamically-allocated linked structures, which are constructed at run time and number of nodes can grow and shrink in time. We will use a table structure to define the state graph, which consists of contiguous multiple identically-structured elements. Each element of the table defines one state. One or more of the entries is an index to other elements. The index is essentially a link to another state. An important factor when implementing FSMs is that there should be a clear and one-to-one mapping between the FSM state graph and the data structure. I.e., there should be one element of the structure for each state. If each state has four arrows, then each node of the linked structure should have four links.

The outputs of Moore FSM are only a function of the current state. In contrast, the outputs are a function of both the input and the current state in a Mealy FSM. Often, in a Moore FSM, the specific output pattern defines what it means to be in the current state. In the following videos we take a simplistic system where we wish to detect if a input stream has a odd number of 1s so far. It does not lend itself to an easy implementation because we do not have a notion of how long a bit has to persist for it to be inferred as the same bit as opposed to a new bit. It serves though as a simple example. The later examples are more rigorously specified and therefore lend themselves to proper implementations.

Example 10.1. Design a traffic light controller for the intersection of two equally busy one-way streets. The goal is to maximize traffic flow, minimize waiting time at a red light, and avoid accidents.

Solution: The intersection has two one-ways roads with the same amount of traffic: North and East, as shown in Figure 10.6. Controlling traffic is a good example because we all know what is supposed to happen at the intersection of two busy one-way streets. We begin the design defining what constitutes a state. In this system, a state describes which road has authority to cross the intersection. The basic idea, of course, is to prevent southbound cars to enter the intersection at the same time as westbound cars. In this system, the light pattern defines which road has right of way over the other. Since an output pattern to the lights is necessary to remain in a state, we will solve this system with a Moore FSM. It will have two inputs (car sensors on North and East roads) and six outputs (one for each light in the traffic signal.) The six traffic lights are interfaced to Port B bits 5–0, and the two sensors are connected to Port E bits 1–0,

PE1=0, PE0=0 means no cars exist on either road
PE1=0, PE0=1 means there are cars on the East road
PE1=1, PE0=0 means there are cars on the North road
PE1=1, PE0=1 means there are cars on both roads

The next step in designing the FSM is to create some states. Again, the Moore implementation was chosen because the output pattern (which lights are on) defines which state we are in. Each state is given a symbolic name:

goN, PB5-0 = 100001 makes it green on North and red on East
waitN, PB5-0 = 100010 makes it yellow on North and red on East
goE, PB5-0 = 001100 makes it red on North and green on East
waitE, PB5-0 = 010100 makes it red on North and yellow on East

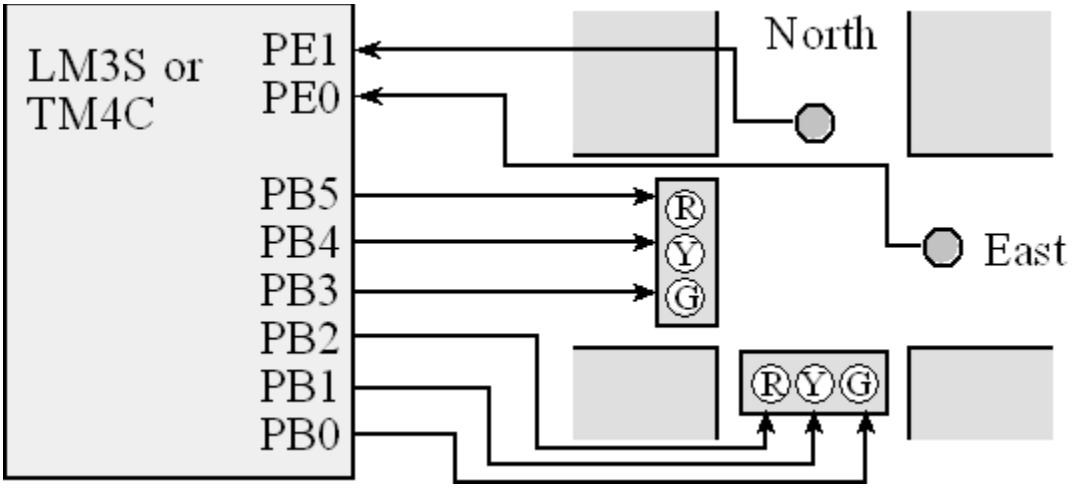


Figure 10.6. Traffic light interface with two sensors and 6 lights.

The output pattern for each state is drawn inside the state circle. The time to wait for each state is also included. How the machine operates will be dictated by the input-dependent state transitions. We create decision rules defining what to do for each possible input and for each state. For this design we can list heuristics describing how the traffic light is to operate:

If no cars are coming, stay in a green state, but which one doesn't matter.
To change from green to red, implement a yellow light of exactly 5 seconds.
Green lights will last at least 30 seconds.
If cars are only coming in one direction, move to and stay green in that direction.
If cars are coming in both directions, cycle through all four states.

Before we draw the state graph, we need to decide on the sequence of operations.

1. Initialize timer and direction registers
2. Specify initial state
3. Perform FSM controller
    a) Output to traffic lights, which depends on the state
    b) Delay, which depends on the state
    c) Input from sensors
    d) Change states, which depends on the state and the input

We implement the heuristics by defining the state transitions, as illustrated in Figure 10.7. Instead of using a graph to define the finite state machine, we could have used a table, as shown in Table 10.2.

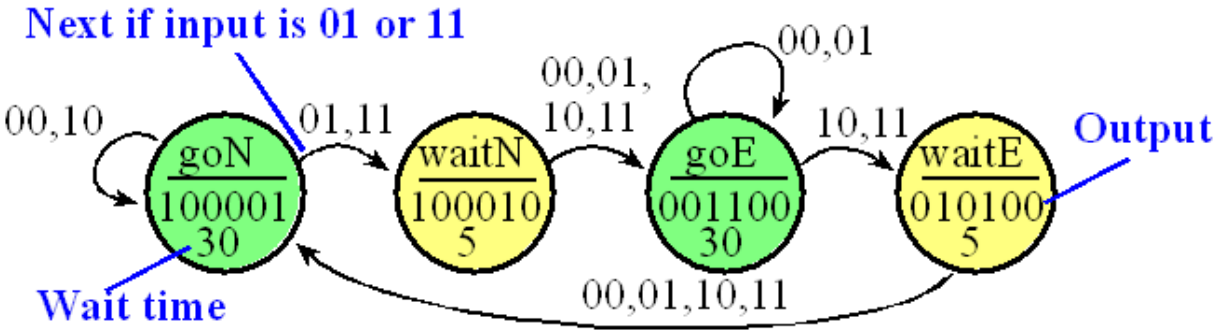


Figure 10.7. Graphical form of a Moore FSM that implements a traffic light.

A **state transition table** has exactly the same information as the state transition graph, but in tabular form. The first column specifies the state number, which we will number sequentially from 0. Each state has a descriptive name. The "Lights" column defines the output patterns for six traffic lights. The "Time" column is the time to wait with this output. The last four columns will be the next states for each possible input pattern.

Num	Name	Lights	Time	In=0	In=1	In=2	In=3
0	goN	100001	30	goN	waitN	goN	waitN
1	waitN	100010	5	goE	goE	goE	goE
2	goE	001100	30	goE	goE	waitE	waitE
3	waitE	010100	5	goN	goN	goN	goN

Table 10.2. Tabular form of a Moore FSM that implements a traffic light.

The next step is to map the FSM graph onto a data structure that can be stored in ROM. Program 10.4 uses an array of structures, where each state is an element of the array, and state transitions are defined as indices to other nodes. The four **`Next`** parameters define the input-dependent state transitions. The wait times are defined in the software as decimal numbers with units of 10ms, giving a range of 10 ms to about 10 minutes. Using good labels makes the program easier to understand, in other words **`goN`** is more descriptive than **`0`**.

The main program begins by specifying the Port E bits 1 and 0 to be inputs and Port B bits 5–0 to be outputs. The initial state is defined as **`goN`**. The main loop of our controller first outputs the desired light pattern to the six LEDs, waits for the specified amount of time, reads the sensor inputs from Port E, and then switches to the next state depending on the input data. The timer functions were presented earlier as Program 10.2. The function **`SysTick_Wait10ms`** will wait 10ms times the parameter. Bit-specific addressing will facilitate friendly access to Ports B and E. **`SENSOR`** accesses PE1–PE0, and **`LIGHT`** accesses PB5–PB0.

```
#define SENSOR  (*((volatile unsigned long *)0x4002400C))
#define LIGHT   (*((volatile unsigned long *)0x400050FC))
// Linked data structure
struct State {
  unsigned long Out;
  unsigned long Time;
  unsigned long Next[4];};
typedef const struct State STyp;
#define goN   0
#define waitN 1
#define goE   2
#define waitE 3
STyp FSM[4]={
```

```
 {0x21,3000,{goN,waitN,goN,waitN}},
 {0x22, 500,{goE,goE,goE,goE}},
 {0x0C,3000,{goE,goE,waitE,waitE}},
 {0x14, 500,{goN,goN,goN,goN}}};
unsigned long S;  // index to the current state
unsigned long Input;
int main(void){ volatile unsigned long delay;
  PLL_Init();       // 80 MHz, Program 10.1
  SysTick_Init();   // Program 10.2
  SYSCTL_RCGC2_R |= 0x12;      // 1) B E
  delay = SYSCTL_RCGC2_R;      // 2) no need to unlock
  GPIO_PORTE_AMSEL_R &= ~0x03; // 3) disable analog function on PE1-0
  GPIO_PORTE_PCTL_R& = ~0x000000FF; // 4) enable regular GPIO
  GPIO_PORTE_DIR_R &= ~0x03;   // 5) inputs on PE1-0
  GPIO_PORTE_AFSEL_R& = ~0x03; // 6) regular function on PE1-0
  GPIO_PORTE_DEN_R |= 0x03;    // 7) enable digital on PE1-0
  GPIO_PORTB_AMSEL_R &= ~0x3F; // 3) disable analog function on PB5-0
  GPIO_PORTB_PCTL_R &= ~0x00FFFFFF; // 4) enable regular GPIO
  GPIO_PORTB_DIR_R |= 0x3F;    // 5) outputs on PB5-0
  GPIO_PORTB_AFSEL_R &= ~0x3F; // 6) regular function on PB5-0
  GPIO_PORTB_DEN_R |= 0x3F;    // 7) enable digital on PB5-0
  S = goN;
  while(1){
    LIGHT = FSM[S].Out;  // set lights
    SysTick_Wait10ms(FSM[S].Time);
    Input = SENSOR;     // read sensors
    S = FSM[S].Next[Input];
  }
}
```

Program 10.4. Linked data structure implementation of the traffic light controller (TableTrafficLightxxx.zip).

In order to make it easier to understand, which will simplify verification and modification, we have made a 1-to-1 correspondence between the state graph in Figure 10.7 and the **`FSM[4]`** data structure in Program 10.4. Notice also how this implementation separates the civil engineering policies (the data structure specifies what the machine does), from the computer engineering mechanisms (the executing software specifies how it is done.) Once we have proven the executing software to be operational, we can modify the policies and be confident that the mechanisms will still work. When an accident occurs, we can blame the civil engineer that designed the state graph.

On microcontrollers that have flash, we can place the **`FSM`** data structure in flash. This allows us to make minor modifications to the finite state machine (add/delete states, change input/output values) by changing the data structure. In this way small modifications/upgrades/options to the finite state machine can be made by reprogramming the flash reusing the hardware external to the microcontroller.
The FSM approach makes it easy to change. To change the wait time for a state, we simply change the value in the data structure. To add more states (e.g., put a red/red state after each yellow state, which will reduce accidents caused by bad drivers running the yellow light), we simply increase the size of the **`fsm[]`** structure and define the **`Out`**, **`Time`**, and **`Next`** fields for these new states.

To add more output signals (e.g., walk and left turn lights), we simply increase the precision of the **`Out`** field. To add two more input lines (e.g., wait button, left turn car sensor), we increase the size of the next field to **`Next[16]`**. Because now there are four input lines, there are 16 possible combinations, where each input possibility requires a **`Next`** value specifying where to go if this combination occurs. In this simple scheme, the size of the **`Next[]`** field will be 2 raised to the power of the number of input signals.

Checkpoint 10.3: Why is it good to use labels for the states? E.g., why is **`goN`** is better than **`0`**.

Observation: In order to make the FSM respond quicker, we could implement a time delay function that returns immediately if an alarm condition occurs. If no alarm exists, it waits the specified delay.

Example 10.2. Design vending machine with two outputs (soda, change) and two inputs (dime, nickel).

Solution: This vending machine example illustrates additional flexibility that we can build into our FSM implementations. In particular, rather than simple digital inputs, we will create an input function that returns the current values of the inputs. Similarly, rather than simple digital outputs, we will implement general functions for each state. We could have solved this particular vending machine using the approach in the previous example, but this approach provides an alternative mechanism when the input and/or output operations become complex. Our simple vending machine has two coin sensors, one for dimes and one for nickels, see Figure 10.8. When a coin falls through a slot in the front of the machine, light from the QEB1134 sensor reflects off the coin and is recognized back at the sensor. An op amp (OPA2350) creates a digital high at the Port B input whenever a coin is reflecting light. So as the coin passes the sensor, a pulse (V2) is created. The two coin sensors will be inputs to the FSM. If the digital input is high (1), this means there is a coin currently falling through the slot. When a coin is inserted into the machine, the sensor goes high, then low. Because of the nature of vending machines we will assume there cannot be both a nickel and a dime at the same time. This means the FSM input can be 0, 1, or 2. To implement the soda and change dispensers, we will interface two solenoids to Port E. The coil current of the solenoids is less than 40 mA, so we can use the 7406 open collector driver. If the software makes PE0 high, waits 10ms, then makes PE0 low, one soda will be dispensed. If the software makes PE1 high, waits 10ms, then makes PE1 low, one nickel will be returned.

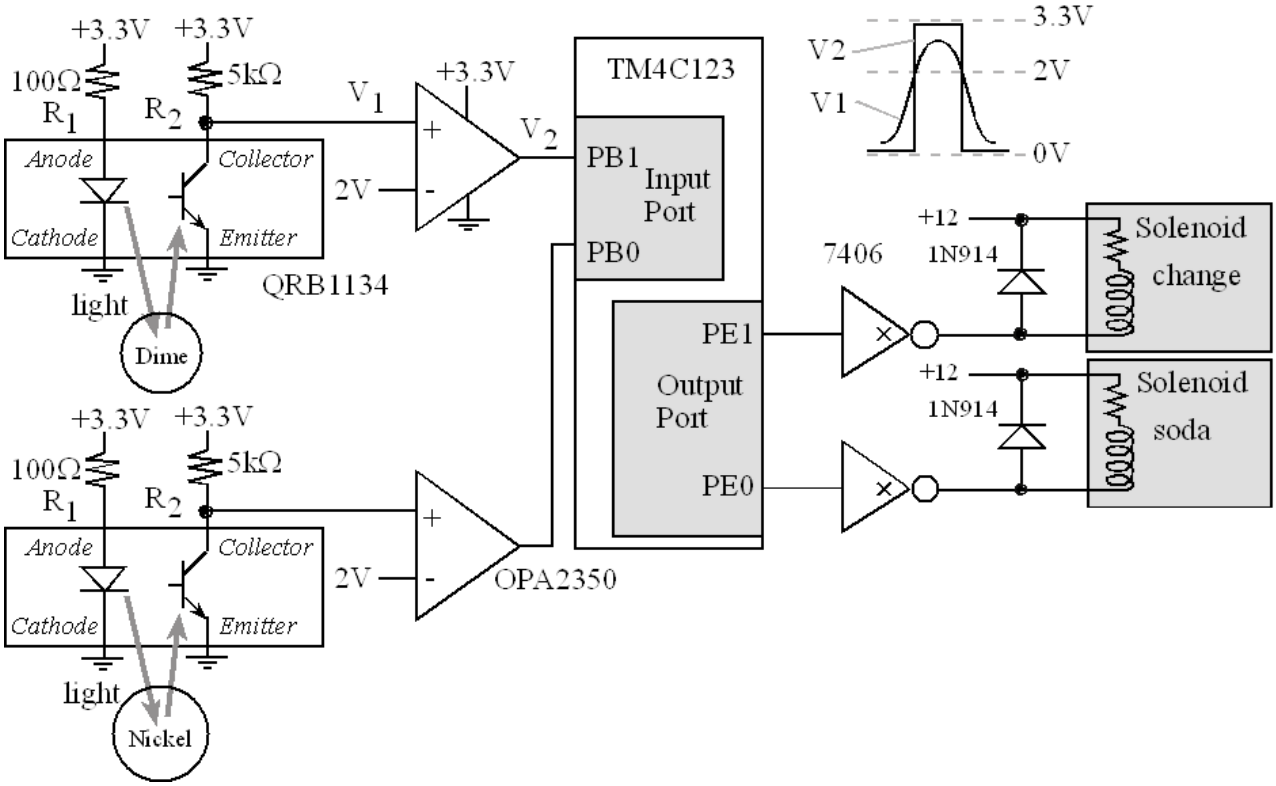


Figure 10.8. A vending machine interfaced to a microcontroller.

We need to decide on the sequence of operations before we draw the state graph.

1) Initialize timer and directions registers
2) Specify initial state
3) Perform FSM controller
    a) Call an output function, which depends on the state

b) Delay, which depends on the state
c) Call an input function to get the status of the coin sensors
d) Change states, which depends on the state and the input.

Figure 10.9 shows the Moore FSM that implements the vending machine. A soda costs 15 cents, and the machine accepts nickels (5 cents) and dimes (10 cents). We have an input sensor to detect nickels (bit 0) and an input sensor to detect dimes (bit 1.) We choose the wait time in each state to be 20ms, which smaller than the time it takes the coin to pass by the sensor. Waiting in each state will debounce the sensor, preventing multiple counting of a single event. Notice that we wait in all states, because the sensor may bounce both on touch and release. Each state also has a function to execute. The function **`Soda`** will trigger the Port E output so that a soda is dispensed. Similarly, the function **`Change`** will trigger the Port E output so that a nickel is returned. The **`M`** states refer to the amount of collected money. When we are in a **`W`** state, we have collected that much money, but we're still waiting for the last coin to pass the sensor. For example, we start with no money in state **`M0`**. If we insert a dime, the input will go 10_2, and our state machine will jump to state **`W10`**. We will stay in state **`W10`** until the dime passes by the coin sensor. In particular when the input goes to 00, then we go to state **`M10`**. If we insert a second dime, the input will go 10_2, and our state machine will jump to state **`W20`**. Again, we will stay in state **`W20`** until this dime passes. When the input goes to 00, then we go to state **`M20`**. Now we call the function **`change`** and jump to state **`M15`**. Lastly, we call the function **`Soda`** and jump back to state **`M0`**.

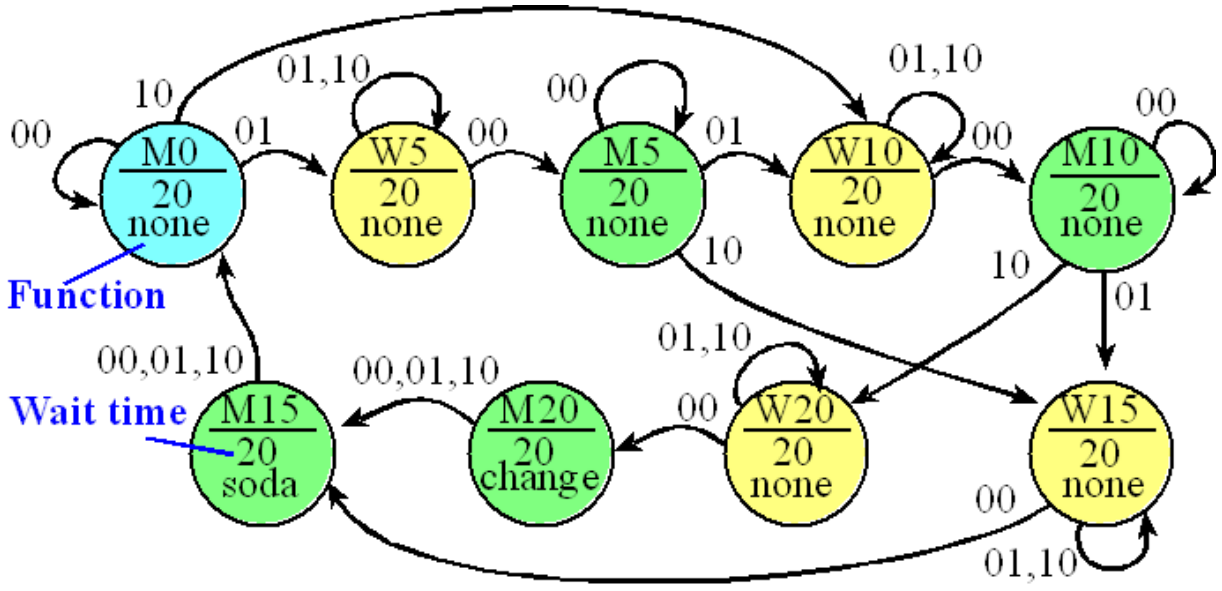


Figure 10.9. This Moore FSM implements a vending machine.

Since this is a layered system, we will begin by designing the low-level input/output functions that handle the operation of the sensors and solenoid, see Program 10.5. The bit-specific addressing **`COINS`** provides friendly access to PB1 and PB0, **`CHANGE`** provides friendly access to PE1, and **`SODA`** provides friendly access to PE0. The initialization specifies Port B bits 1 and 0 to be input and the Port E bits 1 and 0 to be outputs. The PLL and SysTick are also initialized.

```
#define T10ms 800000
#define T20ms 1600000
#define COINS   (*((volatile unsigned long *)0x4002400C))
#define SODA    (*((volatile unsigned long *)0x40005004))
#define CHANGE  (*((volatile unsigned long *)0x40005008))
void FSM_Init(void){ volatile unsigned long delay;
  PLL_Init();       // 80 MHz, Program 10.1
  SysTick_Init();   // Program 10.2
  SYSCTL_RCGC2_R |= 0x12;      // 1) B E
```

```
  delay = SYSCTL_RCGC2_R;      // 2) no need to unlock
  GPIO_PORTE_AMSEL_R &= ~0x03; // 3) disable analog function on PE1-0
  GPIO_PORTE_PCTL_R &= ~0x000000FF; // 4) enable regular GPIO
  GPIO_PORTE_DIR_R &= ~0x03;   // 5) inputs on PE1-0
  GPIO_PORTE_AFSEL_R& = ~0x03; // 6) regular function on PE1-0
  GPIO_PORTE_DEN_R |= 0x03;    // 7) enable digital on PE1-0
  GPIO_PORTB_AMSEL_R &= ~0x3F; // 3) disable analog function on PB5-0
  GPIO_PORTB_PCTL_R &= ~0x000000FF; // 4) enable regular GPIO
  GPIO_PORTB_DIR_R |= 0x03;    // 5) outputs on PB1-0
  GPIO_PORTB_AFSEL_R &= ~0x03; // 6) regular function on PB1-0
  GPIO_PORTB_DEN_R |= 0x03;    // 7) enable digital on PB1-0
  SODA = 0; CHANGE = 0;
}
unsigned long Coin_Input(void){
  return COINS;  // PB1,0 can be 0, 1, or 2
}
void Solenoid_None(void){
};
void Solenoid_Soda(void){
  SODA = 0x01;          // activate solenoid
  SysTick_Wait(T10ms);  // 10 msec, dispenses a delicious soda
  SODA = 0x00;          // deactivate
}
void Solenoid_Change(void){
  CHANGE = 0x02;        // activate solenoid
  SysTick_Wait(T10ms);  // 10 msec, return 5 cents
  CHANGE = 0x00;        // deactivate
}
```

Program 10.5. Low-level input/output functions for the vending machine.

The initial state is defined as **M0**. Our controller software first calls the function for this state, waits for the specified amount of time, reads the sensor inputs from Port B, then switches to the next state depending on the input data. Notice again the 1-to-1 correspondence between the state graph in Figure 10.9 and the data structure in Program 10.6.

```
struct State {
  void (*CmdPt)(void);   // output function
  unsigned long Time;    // wait time, 12.5ns units
  unsigned long Next[3];};
typedef const struct State StateType;
#define M0  0
#define W5  1
#define M5  2
#define W10 3
#define M10 4
#define W15 5
#define M15 6
#define W20 7
#define M20 8
StateType FSM[9]={
  {&Solenoid_None,  T20ms,{M0,W5,W10}},      // M0, no money
  {&Solenoid_None,  T20ms,{M5,W5,W5}},       // W5, seeing a nickel
  {&Solenoid_None,  T20ms,{M5,W10,W15}},     // M5, have 5 cents
  {&Solenoid_None,  T20ms,{M10,W10,W10}},    // W10, seeing a dime
  {&Solenoid_None,  T20ms,{M10,W15,W20}},    // M10, have 10 cents
  {&Solenoid_None,  T20ms,{M15,W15,W15}},    // W15, seeing something
  {&Solenoid_Soda,  T20ms,{M0,M0,M0}},       // M15, have 15 cents
  {&Solenoid_None,  T20ms,{M20,W20,W20}},    // W20, seeing dime
  {&Solenoid_Change,T20ms,{M15,M15,M15}}};   // M20, have 20 cents
unsigned long S; // index into current state
unsigned long Input;
```

```
int main(void){
  FSM_Init();
  S = M0;       // Initial State
  while(1){
    (FSM[S].CmdPt)();           // call output function
    SysTick_Wait(FSM[S].Time);  // wait Program 10.2
    Input = Coin_Input();       // input can be 0,1,2
    S = FSM[S].Next[Input];     // next
  }
}
```

Program 10.6. Vending machine controller.

10.5. Stepper motors

A motor can be evaluated in terms of its maximum speed (RPM), its torque (N-m), and the efficiency in which it translates electrical power into mechanical power. Sometimes however, we wish to use a motor to control the rotational position (θ=motor shaft angle) rather than to control the rotational speed ($\omega=d\theta/dt$). Stepper motors are used in applications where precise positioning is more important than high RPM, high torque, or high efficiency. Stepper motors are very popular for microcontroller-based embedded systems because of their inherent digital interface. This next video shows a stepper motor controlled by a FSM. The first button makes it spin one way, the second button makes it spin the other way, and the third button makes it step just once. If both the first two buttons are pressed it wiggles back and forth.

Larger motors provide more torque, but require more current. It is easy for a computer to control both the position and velocity of a stepper motor in an open-loop fashion. Although the cost of a stepper motor is typically higher than an equivalent DC permanent magnetic field motor, the overall system cost is reduced because stepper motors may not require feedback sensors. They are used in printers to move paper and print heads, tapes/disks to position read/write heads, and high-precision robots.

A bipolar stepper motor has two coils on the stator (the frame of the motor), labeled **A** and **B** in Figure 10.10. Typically, there is always current flowing through both coils. When current flows through both coils, the motor does not spin (it remains locked at that shaft angle). Stepper motors are rated in their holding torque, which is their ability to hold stationary against a rotational force (torque) when current is constantly flowing through both coils. To move a bipolar stepper, we reverse the direction of current through one (not both) of the coils, see Figure 10.10. To move it again we reverse the direction of current in the other coil. Remember, current is always flowing through both coils. Let the direction of the current be signified by up and down arrows in Figure 10.10. To make the current go up, the microcontroller outputs a binary 01 to the interface. To make the current go down, it outputs a binary 10. Since there are 2 coils, four outputs will be required (e.g., 0101_2 means up/up). To spin the motor, we output the sequence 0101_2, 0110_2, 1010_2, 1001_2… over and over. Each output causes the motor to rotate a fixed angle. To rotate the other direction, we reverse the sequence (0101_2, 1001_2, 1010_2, 0110_2…).

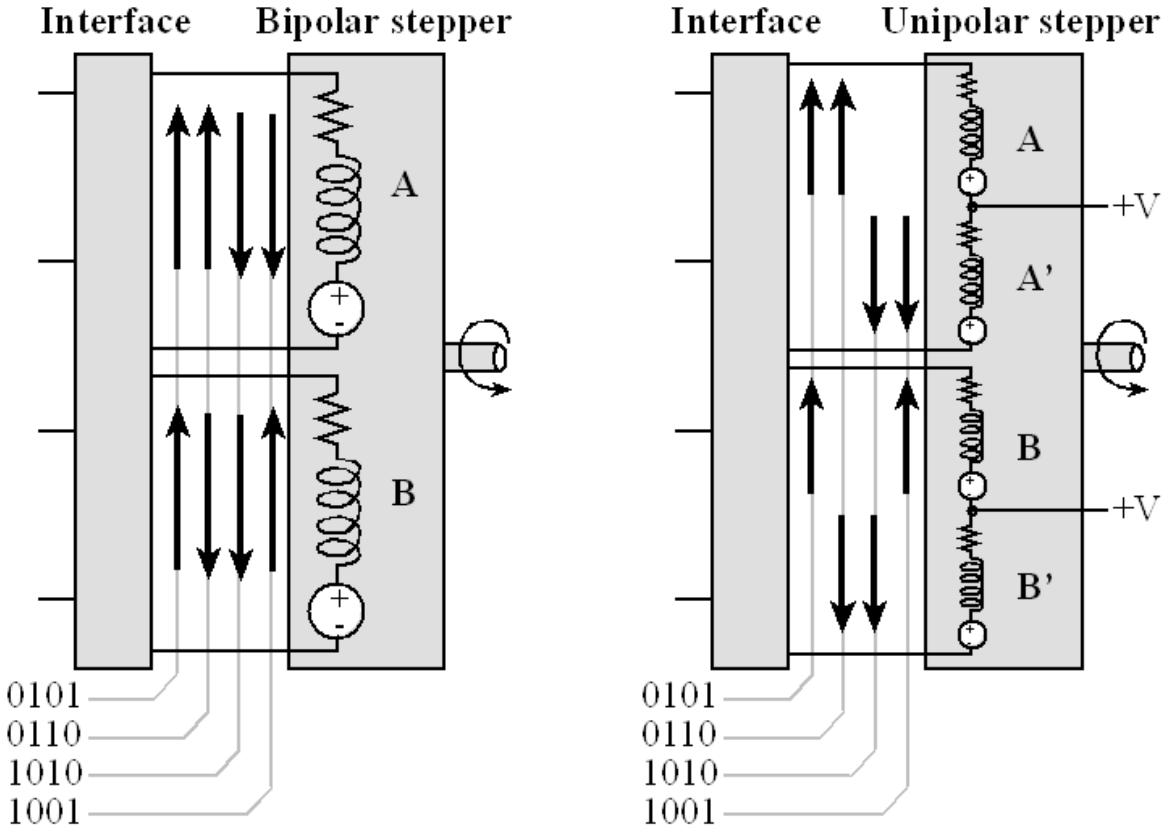


Figure 10.10. A bipolar stepper has 2 coils, but a unipolar stepper divides those two coils into four parts.

There is a North and a South permanent magnet on the rotor (the part that spins). The amount of rotation caused by each current reversal is a fixed angle depending on the number of teeth on the permanent magnets. For example, the rotor in Figure 10.11 is drawn with one North tooth and one South tooth. If there are *n* teeth on the South magnet (also *n* teeth on the North magnet), then the stepper will move at *90/n* degrees. This means there will be *4n* steps per rotation. Because moving the motor involves accelerating a mass (rotational inertia) against a load friction, after we output a value, we must wait an amount of time before we can output again. If we output too fast, the motor does not have time to respond. The speed of the motor is related to the number of steps per rotation and the time in between outputs. For information on stepper motors see the data sheets
http://users.ece.utexas.edu/~valvano/Datasheets/StepperBasic.pdf
http://users.ece.utexas.edu/~valvano/Datasheets/StepperDriveBasic.pdf
http://users.ece.utexas.edu/~valvano/Datasheets/StepperSelection.pdf
http://users.ece.utexas.edu/~valvano/Datasheets/Stepper.pdf
http://users.ece.utexas.edu/~valvano/Datasheets/Stepper_ST.pdf

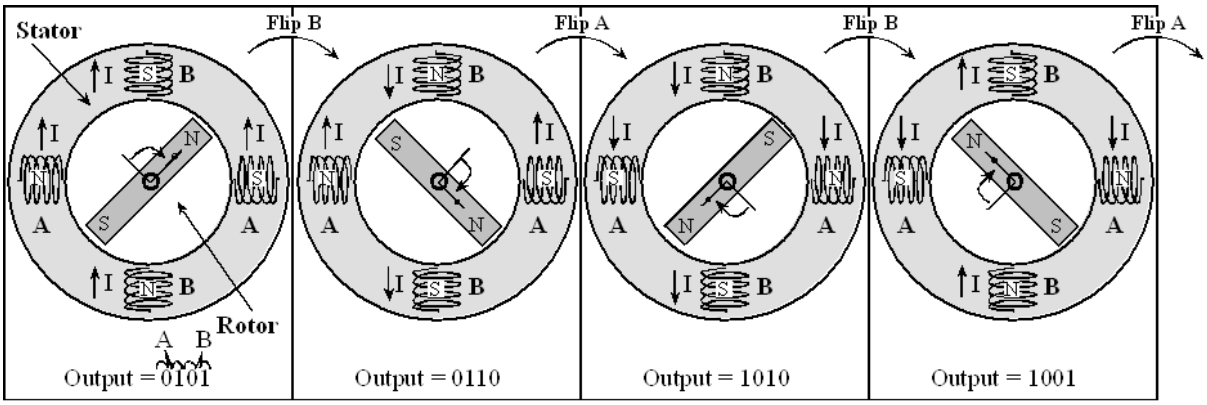


Figure 10.11. To rotate this stepper by 18°, the interface flips the direction of one of the currents.

Questions:
What is the output sequence that produces one complete counterclockwise rotation in the motor?
What is the output sequence that results in one complete clockwise rotation in the motor?
Identify two sets of output values in the sequence that will never occur one after the other.
How can you implement a stepper motor with a finite state machine? Draw the complete state graph with all transition arrows.

The unipolar stepper motor provides for bi-directional currents by using a center tap, dividing each coil into two parts. In particular, coil **A** is split into coil **A** and **A'**, and coil **B** is split into coil **B** and **B'**. The center tap is connected to the +V power source and the four ends of the coils can be controlled with open collector drivers. Because only half of the electro-magnets are energized at one time, a unipolar stepper has less torque than an equivalent-sized bipolar stepper. However, unipolar steppers are easier to interface. For more information on interfacing stepper motors see Volume 2, Embedded Systems: Real-Time Interfacing to ARM® Cortex™-M Microcontrollers.

Figure 10.12 shows a circular linked graph containing the output commands to control a stepper motor. This simple FSM has no inputs, four output bits and four states. There is one state for each output pattern in the usual stepper sequence 5,6,10,9... The circular FSM is used to spin the motor is a clockwise direction.

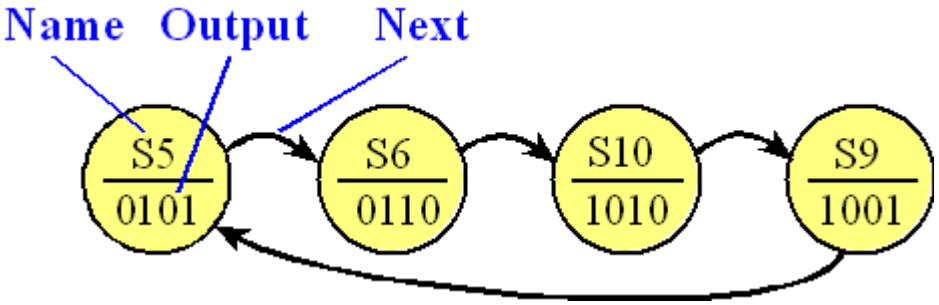


Figure 10.12. This stepper motor FSM has four states. The 4-bit outputs are given in binary.

Example 10.3. Design an autonomous robot using FSMs and stepper motors. Make the robot avoid walls.

Solution: We choose a stepper motor according to the speed and torque requirements of the system. A stepper with 200 steps/rotation will provide a very smooth rotation while it spins. Just like the DC motor, we need an interface that can handle the currents required by the coils. We can use a L293 to interface either unipolar or bipolar steppers that require less than 1 A per coil. In general, the output current of a driver must be large enough to energize the stepper coils. We control the interface using an output port of the microcontroller, as shown in Figure 10.13. Motors require interface circuits, like the L293, because of the large currents required for the motor. Furthermore, most motors will require more current than the USB or LaunchPad can supply. In this system the motors are powered directly from an 8.4V battery. Motor current flows from the battery to the L293, out 1Y, across the coil of the stepper motor, into the 2Y-pin of the L293, and finally back to the battery. Notice the motor currents do not flow across the LaunchPad or in/out the USB. The circuit shows the interface of two bipolar steppers, but the unipolar stepper interface is similar except there would be a direct connection of +8.4V to the motor (see Figure 10.14). The front of the robot has two bumper switches. The 7805 regulator allows the LaunchPad to be powered from the battery.

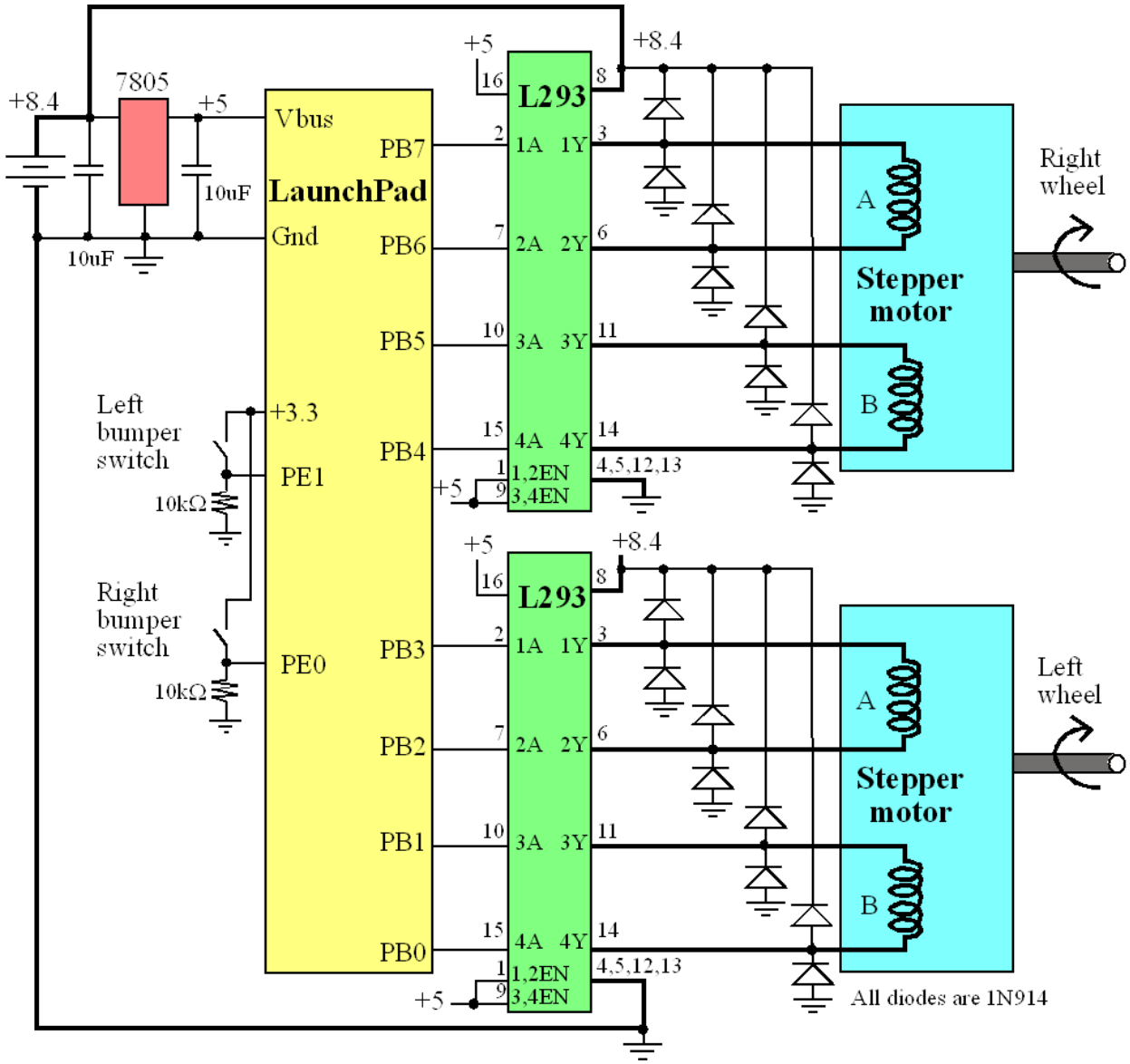


Figure 10.13. Two bipolar stepper motors interfaced to a microcontroller allow the robot to move (make sure R9, R10, R25, and R29 are removed from your LaunchPad). The dark lines carry large currents. Bipolar stepper motors have 4 wires.

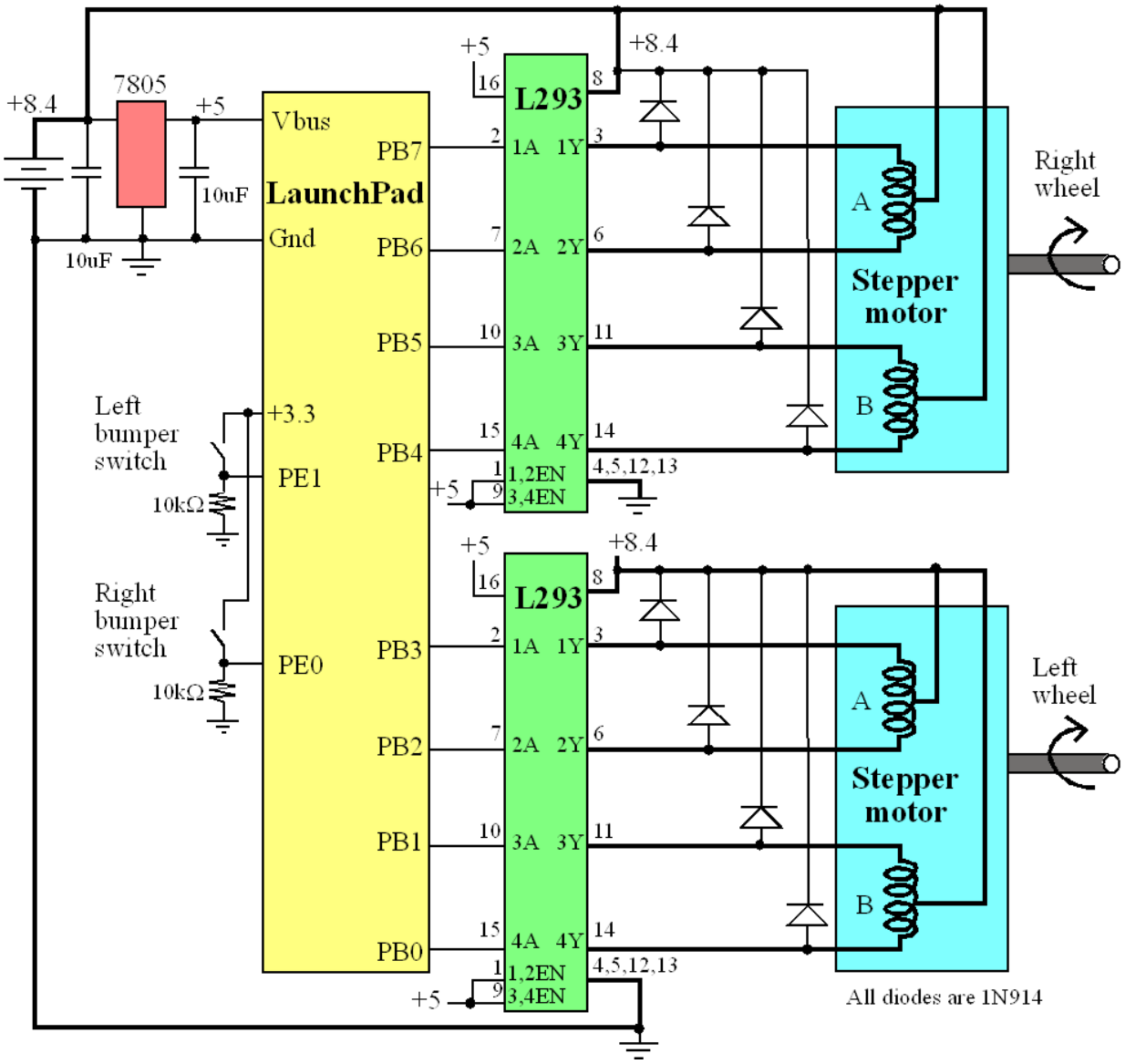


Figure 10.14. Two unipolar stepper motors interfaced to a microcontroller allow the robot to move. The dark lines carry large currents. Unipolar stepper motors have 5 or 6 wires.

To make the robot move forward (states 0,1,2,3) we spin both motors. To satisfy Isaac Asimov's first law of robotics "A robot may not injure a human being or, through inaction, allow a human being to come to harm", we will add two bumper switches in the front that will turn the robot if it detects an object in its path. To make the robot move backward (states S3, S2, S1, S0), we step both motors the other direction. We turn right by stepping the right motor back and the left motor forward (states S0, S7, S8, S9). We turn left by stepping the left motor back and the right motor forward (states S0, S4, S5, S6). Figure 10.15 shows the FSM to control this simple robot. This FSM has two inputs, so each state has four next states. Notice the 1-to-1 correspondence between the state graph in Figure 10.15 and the **`FSM[10]`** data structure in Program 10.7.

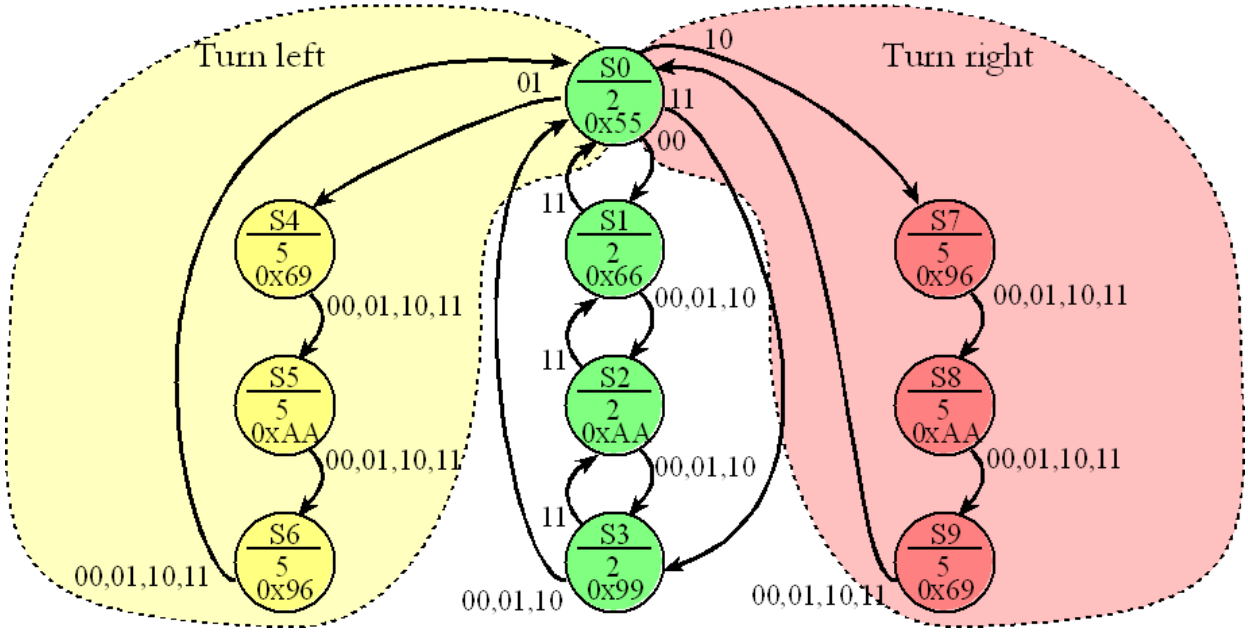


Figure 10.15. If the bumpers are not active (00) the both motor spin at 15 RPM, if both bumpers are active the robot backs up, if just the right bumper is active it turns left, and if just the left bumper is active, it turns right.

The main program, Program 10.7, begins by initializing all of Ports B to be an output and PE1,PE0 to be an input. There are ten states in this robot. If the bumper switch activates, it attempts to turn away from the object. Every 20 ms the program outputs a new stepper commands to both motors. The function **`SysTick_Wait10ms()`** generates an appropriate delay between outputs to the stepper. For a 200 step/rotation stepper, if we wait 20 ms between outputs, there will be 50 outputs/sec, or 3000 outputs/min, causing the stepper motor to spin at 15 RPM. When calculating speed, it is important to keep track of the units.

```
// represents a State of the FSM
struct State{
  unsigned char out;      // PB7-4 to right motor, PB3-0 to left
  unsigned short wait;    // in 10ms units
  unsigned char next[4]; // input 0x00 means ok,
                         //       0x01 means right side bumped something,
                         //       0x02 means left side bumped something,
                         //        0x03 means head-on collision (both sides bumped
something)
};
typedef const struct State StateType;
StateType Fsm[10] = {
   {0x55, 2, {1, 4, 7, 3} }, // S0) initial state and state where bumpers are
checked
  {0x66, 2, {2, 2, 2, 0} }, // S1) both forward [1]
  {0xAA, 2, {3, 3, 3, 1} }, // S2) both forward [2]
  {0x99, 2, {0, 0, 0, 2} }, // S3) both forward [3]
  {0x69, 5, {5, 5, 5, 5} }, // S4) left forward; right reverse [1] turn left
  {0xAA, 5, {6, 6, 6, 6} }, // S5) left forward; right reverse [2] turn left
  {0x96, 5, {0, 0, 0, 0} }, // S6) left forward; right reverse [3] turn left
  {0x96, 5, {8, 8, 8, 8} }, // S7) left reverse; right forward [1] turn right
  {0xAA, 5, {9, 9, 9, 9} }, // S8) left reverse; right forward [2] turn right
  {0x69, 5, {0, 0, 0, 0} }  // S9) left reverse; right forward [3] turn right
};

void PortB_Init(void){ volatile unsigned long delay;
  SYSCTL_RCGC2_R |= 0x02;          // 1) activate Port B
  delay = SYSCTL_RCGC2_R;          // allow time for clock to stabilize
                                   // 2) no need to unlock PB7-0
```

```
  GPIO_PORTB_AMSEL_R = 0x00;       // 3) disable analog functionality on PB7-0
  GPIO_PORTB_PCTL_R = 0x00000000;  // 4) configure PB7-0 as GPIO
  GPIO_PORTB_DIR_R = 0xFF;         // 5) make PB7-0 out
  GPIO_PORTB_AFSEL_R = 0x00;       // 6) disable alt funct on PB7-0
  GPIO_PORTB_DR8R_R = 0xFF;        // enable 8 mA drive on PB7-0
  GPIO_PORTB_DEN_R = 0xFF;         // 7) enable digital I/O on PB7-0
}
void PortE_Init(void){ volatile unsigned long delay;
  SYSCTL_RCGC2_R |= 0x10;          // 1) activate Port E
  delay = SYSCTL_RCGC2_R;          // allow time for clock to stabilize
                                   // 2) no need to unlock PE1-0
  GPIO_PORTE_AMSEL_R &= ~0x03;     // 3) disable analog function on PE1-0
  GPIO_PORTE_PCTL_R &= ~0x000000FF;// 4) configure PE1-0 as GPIO
  GPIO_PORTE_DIR_R &= ~0x03;       // 5) make PE1-0 in
  GPIO_PORTE_AFSEL_R &= ~0x03;     // 6) disable alt funct on PE1-0
  GPIO_PORTE_DEN_R |= 0x03;        // 7) enable digital I/O on PE1-0
}
unsigned char cState;      // current State (0 to 9)
int main(void){
  unsigned char input;
  PLL_Init();                      // Program 10.1
  SysTick_Init();                  // Program 10.2
  PortB_Init();                    // initialize motor outputs on Port B
  PortE_Init();                    // initialize sensor inputs on Port E
  cState = 0;                      // initial state = 0
  while(1){
    // output based on current state
    GPIO_PORTB_DATA_R = Fsm[cState].out; // step motor
    // wait for time according to state
    SysTick_Wait10ms(Fsm[cState].wait);
    // get input
    input = GPIO_PORTE_DATA_R&0x03; // Input 0,1,2,3
    // change the state based on input and current state
    cState = Fsm[cState].next[input];
  }
}
```

Program 10.7. Stepper motor controller.

To illustrate how easy it is to make changes to this implementation, let's consider these three modifications. To make it spin in the other direction, we simply change pointers to sequence in the other direction. We could also add additional input pins and make it perform other commands. To make it travel at a different speed, we change the wait time.

Checkpoint 10.4: If the stepper motor were to have 36 steps per rotation, how fast would the two motors spin using Program 10.7?
Checkpoint 10.5: What would you change in Program 10.7 to make the motor spin at 30 RPM?
Checkpoint 10.6: Does the robot in the previous example satisfy Asimov's second law of robotics?

Performance tip: Use a DC motor for applications requiring high torque or high speed, and use a stepper motor for applications requiring accurate positioning at low speed.
Performance tip: To get high torque at low speed, use a geared DC motor (the motor spins at high speed, but the shaft spins slowly).

Chapter 11: Serial Interfacing
Embedded Systems - Shape the World
Jonathan Valvano and Ramesh Yerraballi

This chapter provides an introduction to serial interfacing, which means we send one bit at time. Serial communication is prevalent in both the computer industry in general and the embedded industry in specific. There are many serial protocols, but in this course we will show you one of the first and simplest protocols that transmit one bit at a time. We will show the theory and details of the universal asynchronous receiver/transmitter (UART) and then use it as an example for developing an I/O driver. We will use busy-wait to synchronize the software with the hardware.

Learning Objectives:

- I/O synchronization.
- Models of I/O devices (busy, done, off).
- Learn how to program the UART.
- Build a distributed system by connecting two systems together.
- Learn how to convert between numbers and ASCII strings

11.1. I/O Synchronization

Before we begin define serial communication, let's begin by introducing some performance measures. As engineers and scientists we are constantly making choices as we design new product or upgrade existing systems. A **performance measure** is a quantitative metric that the goodness of the system. The metrics and synchronization algorithms presented in this section will apply to all I/O communication.

Latency is the time between when the I/O device indicated service is required and the time when service is initiated. Latency includes hardware delays in the digital hardware plus computer software delays. For an input device, software latency (or software response time) is the time between new input data ready and the software reading the data. For an output device, latency is the delay from output device idle and the software giving the device new data to output. In this book, we will also have periodic events. For example, in our data acquisition systems, we wish to invoke the analog to digital converter (ADC) at a fixed time interval. In this way we can collect a sequence of digital values that approximate the continuous analog signal. Software latency in this case is the time between when the ADC conversion is supposed to be started, and when it is actually started. The microcomputer-based control system also employs periodic software processing. Similar to the data acquisition system, the latency in a control system is the time between when the control software is supposed to be run, and when it is actually run. A **real-time** system is one that can guarantee a worst case latency. In other words, the software response time is small and bounded. Furthermore, this bound is small enough to satisfy overall specification of the system, such as no lost data. **Throughput** or **bandwidth** is the maximum data flow in bytes/second that can be processed by the system. Sometimes the bandwidth is limited by the I/O device, while other times it is limited by computer software. Bandwidth can be reported as an overall average or a short-term maximum. **Priority** determines the order of service when two or more requests are made simultaneously. Priority also determines if a high-priority request should be allowed to suspend a low priority request that is currently being processed. We may also wish to implement equal priority, so that no one device can monopolize the computer. In some computer literature, the term "soft-real-time" is used to describe a system that supports priority.

The purpose of our interface is to allow the microcontroller to interact with its external I/O device. One of the choices the designer must make is the algorithm for how the software synchronizes with the hardware. There are five mechanisms to synchronize the microcontroller with the I/O device. Each mechanism synchronizes the I/O data transfer to the busy to done transition. The methods are discussed in the following paragraphs.

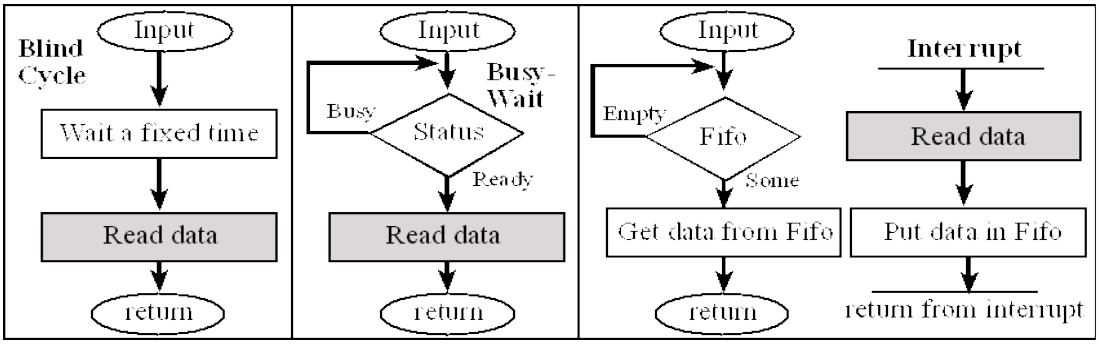


Figure 11.1. Synchronization Mechanisms

Blind cycle is a method where the software simply waits a fixed amount of time and assumes the I/O will complete before that fixed delay has elapsed. For an input device, the software triggers (starts) the external input hardware, waits a specified time, then reads data from device. Blind cycle synchronization for an input device is shown on the left part of Figure 11.1. For an output device, shown on the left part of Figure 11.2, the software writes data to the output device, triggers (starts) the device, then waits a specified time. We call this method **blind**, because there is no status information about the I/O device reported to the software. It is appropriate to use this method in situations where the I/O speed is short and predictable. We can ask the LCD to display an ASCII character, wait 37 µs, and then we are sure the operation is complete. This method works because the LCD speed is short and predictable. Another good example of blind-cycle synchronization is spinning a stepper motor. If we repeat this 8-step sequence over and over 1) output a 0x05, 2) wait 1ms, 3) output a 0x06, 4) wait 1ms, 5) output a 0x0A, 6) wait 1ms, 7) output a 0x09, 8) wait 1ms, the motor will spin at a constant speed.

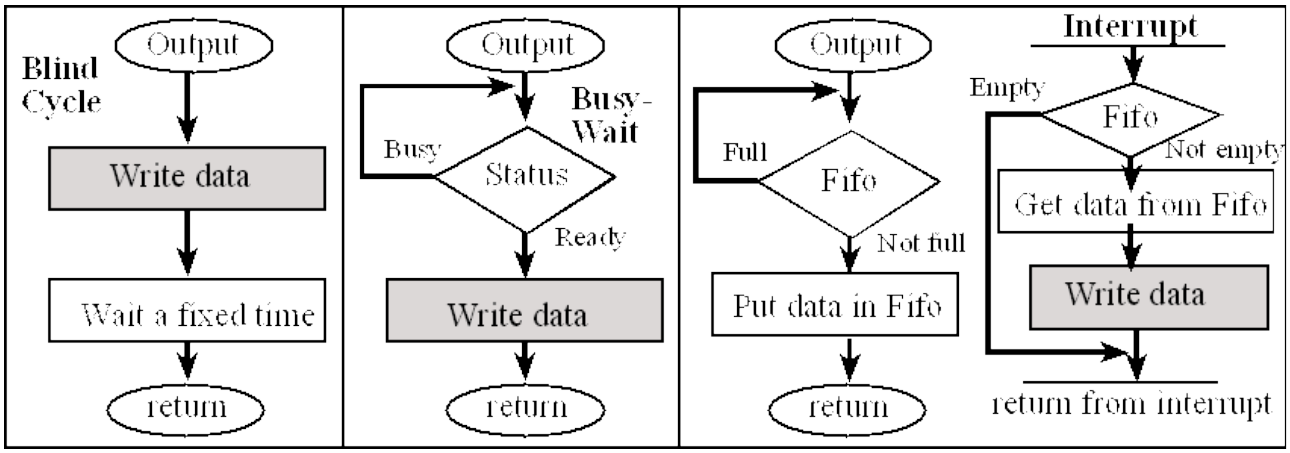


Figure 11.2. The output device sets a flag when it has finished outputting the last data.

Busy Wait is a software loop that checks the I/O status waiting for the done state. For an input device, the software waits until the input device has new data, and then reads it from the input device, see the middle parts of Figures 11.1 and 11.2. For an output device, the software writes data, triggers the output device then waits until the device is finished. Another approach to output device interfacing is for the software to wait until the output device has finished the previous output, write data, and then trigger the device. Busy-wait synchronization will be used in situations where the software system is relatively simple and real-time response is not important. The UART software in this chapter will use busy-wait synchronization.

An **interrupt** uses hardware to cause special software execution. With an input device, the hardware will request an interrupt when input device has new data. The software interrupt service will read from the input device and save in global RAM, see the right parts of Figures 11.1 and 11.2. With an output device, the hardware will request an interrupt when the output device is idle. The software interrupt

service will get data from a global structure, and then write to the device. Sometimes we configure the hardware timer to request interrupts on a periodic basis. The software interrupt service will perform a special function. A data acquisition system needs to read the ADC at a regular rate. Interrupt synchronization will be used in situations where the system is fairly complex (e.g., a lot of I/O devices) or when real-time response is important. Interrupts will be presented in Chapter 12.

Periodic Polling uses a clock interrupt to periodically check the I/O status. At the time of the interrupt the software will check the I/O status, performing actions as needed. With an input device, a ready flag is set when the input device has new data. At the next periodic interrupt after an input flag is set, the software will read the data and save them in global RAM. With an output device, a ready flag is set when the output device is idle. At the next periodic interrupt after an output flag is set, the software will get data from a global structure, and write it. Periodic polling will be used in situations that require interrupts, but the I/O device does not support interrupt requests directly.

DMA, or direct memory access, is an interfacing approach that transfers data directly to/from memory. With an input device, the hardware will request a DMA transfer when the input device has new data. Without the software's knowledge or permission the DMA controller will read data from the input device and save it in memory. With an output device, the hardware will request a DMA transfer when the output device is idle. The DMA controller will get data from memory, and then write it to the device. Sometimes we configure the hardware timer to request DMA transfers on a periodic basis. DMA can be used to implement a high-speed data acquisition system. DMA synchronization will be used in situations where high bandwidth and low latency are important. DMA will not be covered in this introductory class. For details on how to implement DMA on the LM4F120/TM4C123, see Embedded Systems: Real-Time Operating Systems for ARM Cortex-M Microcontrollers.

One can think of the hardware being in one of three states. The **idle** state is when the device is disabled or inactive. No I/O occurs in the idle state. When active (not idle) the hardware toggles between the **busy** and **ready** states. The interface includes a **flag** specifying either busy (0) or ready (1) status. Hardware-software synchronization revolves around this flag:

- The hardware will set the flag when the hardware component is complete.
- The software can read the flag to determine if the device is busy or ready.
- The software can clear the flag, signifying the software component is complete.
- This flag serves as the hardware triggering event for an interrupt.

For an input device, a status flag is set when new input data is available. The "busy to ready" state transition will cause a busy-wait loop to complete, see middle of Figure 11.1. Once the software recognizes the input device has new data, it will read the data and ask the input device to create more data. It is the **busy to ready** state transition that signals to the software that the hardware task is complete, and now software service is required. When the hardware is in the ready state the I/O transaction is complete. Often the simple process of reading the data will clear the flag and request another input.

The problem with I/O devices is that they are usually much slower than software execution. Therefore, we need synchronization, which is the process of the hardware and software waiting for each other in a manner such that data is properly transmitted. A way to visualize this synchronization is to draw a state versus time plot of the activities of the hardware and software. For an input device, the software begins by waiting for new input. When the input device is busy it is in the process of creating new input. When the input device is ready, new data is available. When the input device makes the transition from busy to ready, it releases the software to go forward. In a similar way, when the software accepts the input, it can release the input device hardware. The arrows in Figure 11.3 represent the synchronizing events. In this example, the time for the software to read and process the data is less than the time for the input

device to create new input. This situation is called **I/O bound**, meaning the bandwidth is limited by the speed of the I/O hardware.

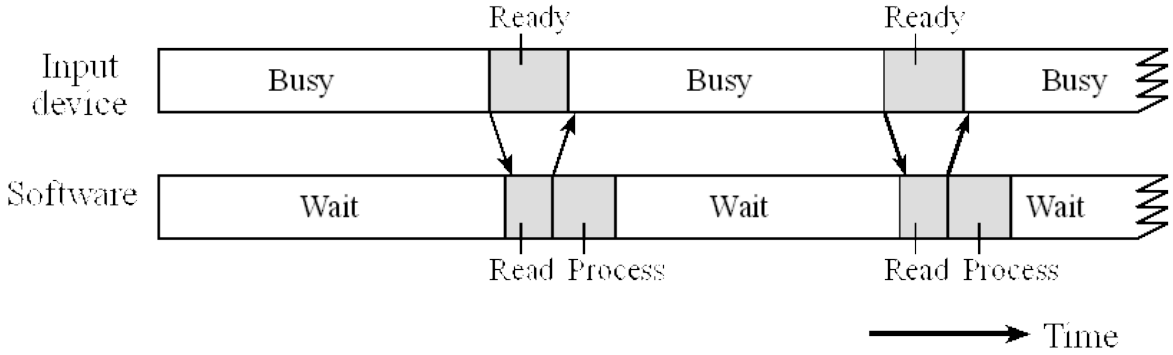


Figure 11.3. The software must wait for the input device to be ready (I/O bound input interface).

If the input device were faster than the software, then the software waiting time would be zero. This situation is called **CPU bound** (meaning the bandwidth is limited by the speed of the executing software). In real systems the bandwidth depends on both the hardware and the software. Another characteristic of real systems is the data can vary over time, like car traffic arriving and leaving a road intersection. In other words, the same I/O channel can sometimes be I/O bound, but at other times the channel could be CPU bound.

We can store or buffer data in a **first in first out** (FIFO) queue, see Figure 11.4, while passing the data from one module to another. These modules may be input devices, output devices or software. Because the buffer separates the generation of data from the consumption of data, it is very efficient, and hence it is prevalent in I/O communication. In particular, it can handle situations where there is an increase or decrease in the rates at which data is produced or consumed. Other names for this important interfacing mechanism include **bounded buffer**, **producer-consumer**, and **buffered I/O**. Data are entered into the FIFO as they arrive; we call **Put** to store data in the FIFO. Data are removed as they leave; we call **Get** to remove data from the FIFO. The FIFO maintains the order of the data, as it passes through the buffer. We can think of a FIFO like a line at the post office. There is space in the lobby for a finite number of people to wait. As customers enter the post office they get in line at the end (put onto FIFO). As the postal worker services the customers, people at the front leave the line (get from the FIFO). It is bad situation (a serious error) if the waiting room becomes full and there is no room for people to wait (full FIFO). However, if there are no customers waiting (empty FIFO) the postal worker sits idle. An empty FIFO may be inefficient, but it is not considered an error.

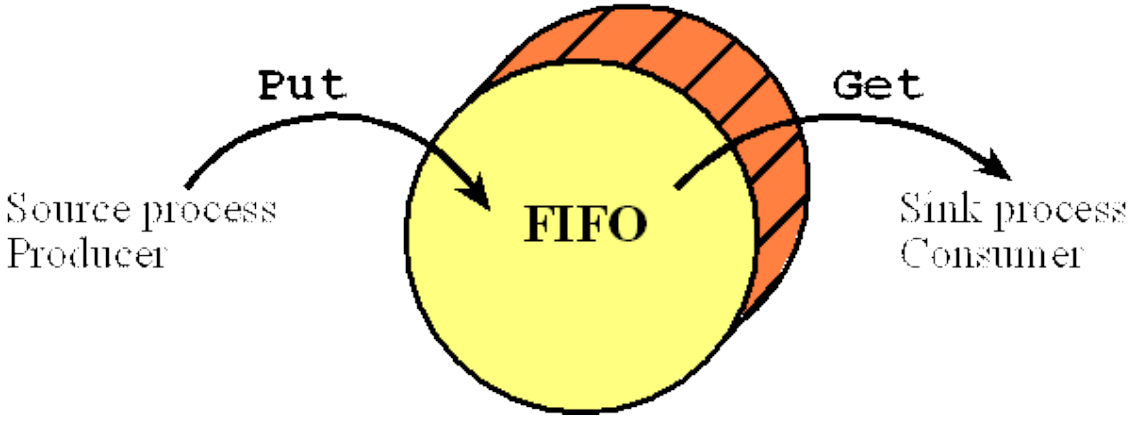


Figure 11.4. A FIFO queue can be used to pass data from a producer to a consumer. At any given time there can be a variable number of elements stored in the FIFO. The order in which data are removed is the same as the order the data are entered.

The busy-wait method is classified as unbuffered because the hardware and software must wait for each other during the transmission of each piece of data. The interrupt solution (shown in the right part of Figure 11.1) is classified as buffered, because the system allows the input device to run continuously, filling a FIFO with data as fast as it can. In the same way, the software can empty the buffer whenever it is ready and whenever there is data in the buffer. The buffering used in an interrupt interface may be a hardware FIFO, a software FIFO, or both hardware and software FIFOs. We will see the FIFO queues will allow the I/O interface to operate during both situations: I/O bound and CPU bound.

For an output device, a status flag is set when the output is idle and ready to accept more data. The "busy to ready" state transition causes a busy-wait loop to complete, see the middle part of Figure 11.2. Once the software recognizes the output is idle, it gives the output device another piece of data to output. It will be important to make sure the software clears the flag each time new output is started. Figure 11.5 contains a state versus time plot of the activities of the output device hardware and software. For an output device, the software begins by generating data then sending it to the output device. When the output device is busy it is processing the data. Normally when the software writes data to an output port, that only starts the output process. The time it takes an output device to process data is usually longer than the software execution time. When the output device is done, it is ready for new data. When the output device makes the transition from busy to ready, it releases the software to go forward. In a similar way, when the software writes data to the output, it releases the output device hardware. The output interface illustrated in Figure 11.5 is also I/O bound because the time for the output device to process data is longer than the time for the software to generate and write it. Again, I/O bound means the bandwidth is limited by the speed of the I/O hardware.

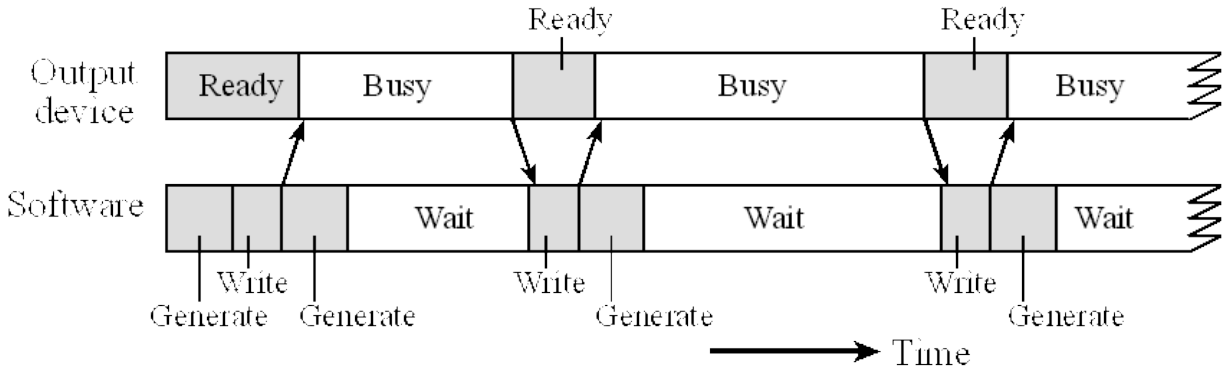


Figure 11.5. The software must wait for the output device to finish the previous operation (I/O bound).

The busy-wait solution for this output interface is also unbuffered, because when the hardware is done, it will wait for the software and after the software generates data, it waits for the hardware. On the other hand, the interrupt solution (shown as the right part of Figure 11.2) is buffered, because the system allows the software to run continuously, filling a FIFO as fast as it wishes. In the same way, the hardware can empty the buffer whenever it is ready and whenever there is data in the FIFO. Again, FIFO queues allow the I/O interface to operate during both situations: I/O bound and CPU bound.

On some systems an interrupt will be generated on a hardware failure. Examples include power failure, temperature too high, memory failure, and mechanical tampering of secure systems. Usually, these events are extremely important and require immediate attention. The Cortex™-M processor will execute special software (**`fault`**) when it tries to execute an illegal instruction, access an illegal memory location, or attempt an illegal I/O operation.

11.2. Universal Asynchronous Receiver Transmitter (UART)

In this section we will develop a simple device driver using the Universal Asynchronous Receiver/Transmitter (UART). This serial port allows the microcontroller to communicate with devices

such as other computers, printers, input sensors, and LCDs. Serial transmission involves sending one bit at a time, such that the data is spread out over time. The total number of bits transmitted per second is called the **baud rate**. The reciprocal of the baud rate is the **bit time**, which is the time to send one bit. Most microcontrollers have at least one UART. The LM4F120/TM4C123 has 8 UARTs. Before discussing the detailed operation on the TM4C, we will begin with general features common to all devices. Each UART will have a baud rate control register, which we use to select the transmission rate. Each device is capable of creating its own serial clock with a transmission frequency approximately equal to the serial clock in the computer with which it is communicating. A **frame** is the smallest complete unit of serial transmission. Figure 11.6 plots the signal versus time on a serial port, showing a single frame, which includes a **start bit** (which is 0), 8 bits of data (least significant bit first), and a **stop bit** (which is 1). There is always only one start bit, but the Stellaris® UARTs allow us to select the 5 to 8 data bits and 1 or 2 stop bits. The UART can add even, odd, or no parity bit. However, we will employ the typical protocol of 1 start bit, 8 data bits, no parity, and 1 stop bit. This protocol is used for both transmitting and receiving. The information rate, or **bandwidth**, is defined as the amount of data or useful information transmitted per second. From Figure 11.6, we see that 10 bits are sent for every byte of usual data. Therefore, the bandwidth of the serial channel (in bytes/second) is the baud rate (in bits/sec) divided by 10.

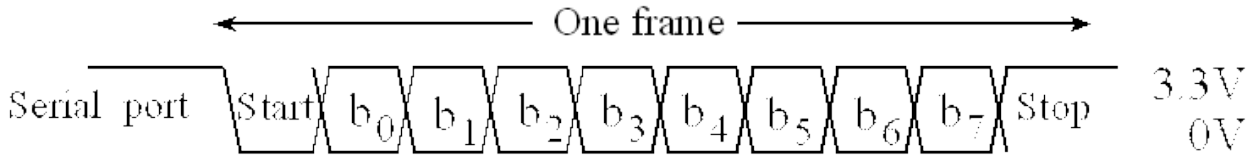


Figure 11.6. A serial data frame with 8-bit data, 1 start bit, 1 stop bit, and no parity bit.

Common Error: If you change the bus clock frequency without changing the baud rate register, the UART will operate at an incorrect baud rate.

Checkpoint 11.1: Assuming the protocol drawn in Figure 11.6 and a baud rate of 1000 bits/sec, what is the bandwidth in bytes/sec?

Table 11.1 shows the three most commonly used RS232 signals. The EIA-574 standard uses RS232 voltage levels and a DB9 connector that has only 9 pins. The most commonly used signals of the full RS232 standard are available with the EIA-574 protocols. Only **TxD**, **RxD**, and **SG** are required to implement a simple bidirectional serial channel, thus the other signals are not shown (Figure 11.7). We define the **data terminal equipment** (DTE) as the computer or a terminal and the **data communication equipment** (DCE) as the modem or printer.

DB9 Pin	EIA-574 Name	Signal	Description	True	DTE	DCE
3	103	TxD	Transmit Data	-5.5V	out	in
2	104	RxD	Receive Data	-5.5V	in	out
5	102	SG	Signal Ground			

Table 11.1. The commonly-used signals on the EIA-574 protocol.

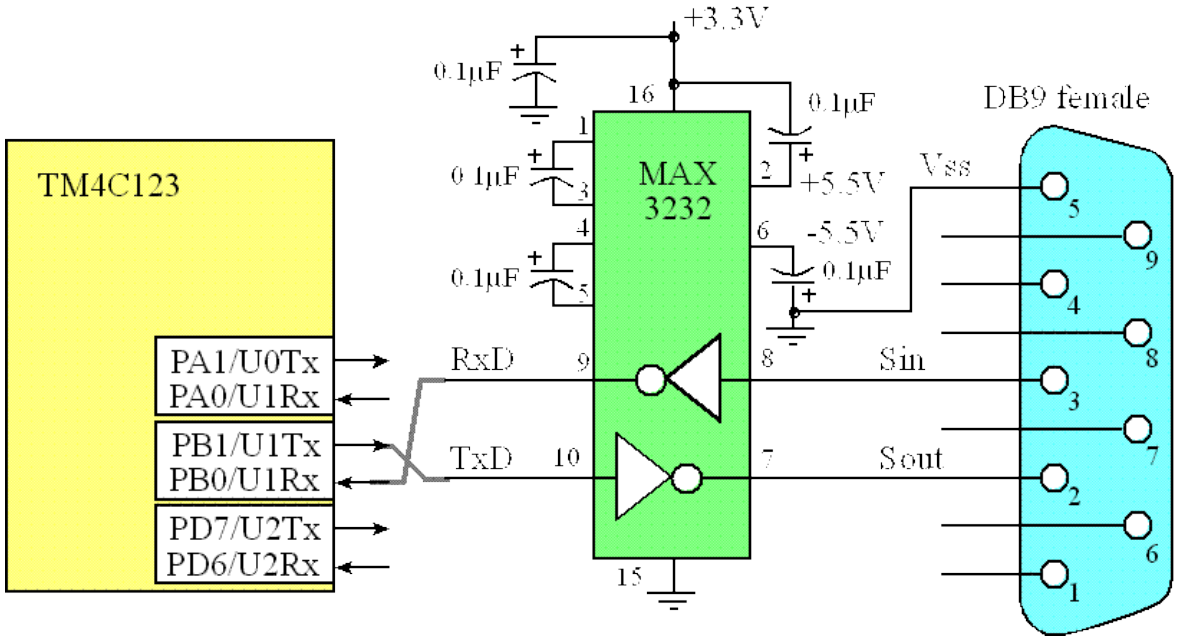


Figure 11.7. Hardware interface implementing an asynchronous RS232 channel. The TM4C123 has eight UART ports.

Observation: The LaunchPad sends UART0 channel through the USB cable, so the circuit shown in Figure 11.7 will not be needed. On the PC side of the cable, the serial channel becomes a virtual COM port.

RS232 is a non-return-to-zero (NRZ) protocol with true signified as a voltage between -5 and -15 V. False is signified by a voltage between +5 and +15 V. A MAX3232 converter chip is used to translate between the +5.5/-5.5 V RS232 levels and the 0/+3.3 V digital levels. The capacitors in this circuit are important, because they form a charge pump used to create the ±5.5 voltages from the +3.3 V supply. The RS232 timing is generated automatically by the UART. During transmission, the MAX3232 translates a digital high on microcontroller side to -5.5V on the RS232/EIA-574 cable, and a digital low is translated to +5.5V. During receiving, the MAX3232 translates negative voltages on RS232/EIA-574 cable to a digital high on the microcontroller side, and a positive voltage is translated to a digital low. The computer is classified as DTE, so its serial output is pin 3 in the EIA-574 cable, and its serial input is pin 2 in the EIA-574 cable. When connecting a DTE to another DTE, we use a cable with pins 2 and 3 crossed. I.e., pin 2 on one DTE is connected to pin 3 on the other DTE and pin 3 on one DTE is connected to pin 2 on the other DTE. When connecting a DTE to a DCE, then the cable passes the signals straight across. In all situations, the grounds are connected together using the SG wire in the cable. This channel is classified as **full-duplex**, because transmission can occur in both directions simultaneously.

11.2.1. Asynchronous Communication

We will begin with transmission, because it is simple. The transmitter portion of the UART includes a data output pin, with digital logic levels as drawn in the following interactive tool. The transmitter has a 16-element FIFO and a 10-bit shift register, which cannot be directly accessed by the programmer. The FIFO and shift register in the transmitter are separate from the FIFO and shift register associated with the receiver. In other words each UART has a receiver and a transmitter, but the interactive tool just shows the transmitter on one microcontroller and the receiver on the other. To output data using the UART, the transmitter software will first check to make sure the transmit FIFO is not full (it will wait if **TXFF** is 1) and then write to the transmit data register (e.g., **`UART0_DR_R`**). The bits are shifted out in this order: start, $\mathbf{b_0}$, $\mathbf{b_1}$, $\mathbf{b_2}$, $\mathbf{b_3}$, $\mathbf{b_4}$, $\mathbf{b_5}$, $\mathbf{b_6}$, $\mathbf{b_7}$, and then stop, where $\mathbf{b_0}$ is the LSB and $\mathbf{b_7}$ is the MSB. The transmit data register is write only, which means the software can write to it (to start a new transmission) but cannot read from it. Even though the transmit data register is at the same address as the receive data register, the transmit and receive data registers are two separate registers. The transmission software can

write to its data register if its TXFF (transmit FIFO full) flag is zero. TXFF equal to zero means the FIFO is not full and has room. The receiving software can read from its data register if its RXFE (receive FIFO empty) flag is zero. RXFE equal to zero means the FIFO is not empty and has some data. While playing the following interactive tool, watch the behavior of the TXFF and RXFE flags.

When a new byte is written to **`UART0_DR_R`**, it is put into the transmit FIFO. Byte by byte, the UART gets data from the FIFO and loads them into the 10-bit transmit shift register. The 10-bit shift register includes a start bit, 8 data bits, and 1 stop bit. Then, the frame is shifted out one bit at a time at a rate specified by the baud rate register. If there are already data in the FIFO or in the shift register when the **`UART0_DR_R`** is written, the new frame will wait until the previous frames have been transmitted, before it too is transmitted. The FIFO guarantees the data are transmitted in the order they were written. The serial port hardware is actually controlled by a clock that is 16 times faster than the baud rate, referred to in the datasheet as **Baud16**. When the data are being shifted out, the digital hardware in the UART counts 16 times in between changes to the **U0Tx** output line.

The software can actually write 16 bytes to the **`UART0_DR_R`**, and the hardware will send them all one at a time in the proper order. This FIFO reduces the software response time requirements of the operating system to service the serial port hardware. Unfortunately, it does complicate the hardware/software timing. At 9600 bits/sec, it takes 1.04 ms to send a frame. Therefore, there will be a delay ranging from 1.04 and 16.7 ms between writing to the data register and the completion of the data transmission. This delay depends on how much data are already in the FIFO at the time the software writes to **`UART0_DR_R`**.

Receiving data frames is a little trickier than transmission because we have to synchronize the receive shift register with the incoming data. The receiver portion of the UART includes a **U0Rx** data input pin with digital logic levels. At the input of the microcontroller, true is 3.3V and false is 0V. There is also a 16-element FIFO and a 10-bit shift register, which cannot be directly accessed by the programmer (shown on the right side of the interactive tool). The receive shift register is 10 bits wide, but the FIFO is 12 bits, 8 bits of data and 4 error flags. Again the receive shift register and receive FIFO are separate from those in the transmitter. The receive data register, **`UART0_DR_R`**, is read only, which means write operations to this address have no effect on this register (recall write operations activate the transmitter). The receiver obviously cannot start a transmission, but it recognizes a new frame by its start bit. The bits are shifted in using the same order as the transmitter shifted them out: start, $\mathbf{b_0}$, $\mathbf{b_1}$, $\mathbf{b_2}$, $\mathbf{b_3}$, $\mathbf{b_4}$, $\mathbf{b_5}$, $\mathbf{b_6}$, $\mathbf{b_7}$, and then stop.

There are six status bits generated by receiver activity. The Receive FIFO empty flag, **RXFE**, is clear when new input data are in the receive FIFO. When the software reads from **`UART0_DR_R`**, data are removed from the FIFO. When the FIFO becomes empty, the **RXFE** flag will be set, meaning there are no more input data. There are other flags associated with the receiver. There is a Receive FIFO full flag **RXFF**, which is set when the FIFO is full. There are four status bits associated with each byte of data. For this reason, the receive FIFO is 12 bits wide. The overrun error, **OE**, is set when input data are lost because the FIFO is full and more input frames are arriving at the receiver. An overrun error is caused when the receiver interface latency is too large. The break error, **BE**, is set when the input is held low for more than a frame. Parity is a mechanism to send one extra bit so the receiver can detect if there were any errors in transmission. With even parity the number of 1's in the data plus parity will be an even number. The **PE** bit is set on a parity error. Because the error rate is so low, most systems do not implement parity. We will not use parity in this class. The framing error, **FE**, is set when the stop bit is incorrect. Framing errors are probably caused by a mismatch in baud rate.

The receiver waits for the 1 to 0 edge signifying a start bit, then shifts in 10 bits of data one at a time from the **U0Rx** line. The internal clock is 16 times faster than the baud rate. After the 1 to 0 edge, the

receiver waits 8 internal clocks and samples the start bit. 16 internal clocks later it samples $\mathbf{b_0}$. Every 16 internal clocks it samples another bit until it reaches the stop bit. The UART needs an internal clock faster than the baud rate so it can wait the half a bit time between the 1 to 0 edge beginning the start bit and the middle of the bit window needed for sampling. The start and stop bits are removed (checked for framing errors), the 8 bits of data and 4 bits of status are put into the receive FIFO. The hardware FIFO implements buffering so data is safely stored in the receiver hardware if the software is performing other tasks while data is arriving.

Observation: If the receiving UART device has a baud rate mismatch of more than 5%, then a framing error can occur when the stop bit is incorrectly captured.

An overrun occurs when there are 16 elements in the receive FIFO, and a 17^{th} frame comes into the receiver. In order to avoid overrun, we can design a real-time system, i.e., one with a maximum latency. The latency of a UART receiver is the delay between the time when new data arrives in the receiver (**RXFE**=0) and the time the software reads the data register. If the latency is always less than 160 bit times, then overrun will never occur.

Observation: With a serial port that has a shift register and one data register (no FIFO buffering), the latency requirement of the input interface is the time it takes to transmit one data frame.

11.2.2. TM4C UART Details

Next we will overview the specific UART functions on the TM4C microcontroller. This section is intended to supplement rather than replace the Texas Instruments manuals. When designing systems with any I/O module, you must also refer to the reference manual of your specific microcontroller. It is also good design practice to review the errata for your microcontroller to see if any quirks (mistakes) exist in your microcontroller that might apply to the system you are designing.

Stellaris TM4C microcontrollers have eight UARTs. The specific port pins used to implement the UARTs vary from one chip to the next. To find which pins your microcontroller uses, you will need to consult its datasheet. Table 11.2 shows some of the registers for the UART0 and UART1. For the other UARTs, the register names will replace the 0 with a 1 – 7. For the exact register addresses, you should include the appropriate header file (e.g., **tm4c123gh6pm.h**). To activate a UART you will need to turn on the UART clock in the **RCGC1** register. You should also turn on the clock for the digital port in the **RCGC2** register. You need to enable the transmit and receive pins as digital signals. The alternative function for these pins must also be selected. In particular we set bits in both the AFSEL and PCTL registers.

The OE, BE, PE, and FE are error flags associated with the receiver. You can see these flags in two places: associated with each data byte in **`UART0_DR_R`** or as a separate error register in **`UART0_RSR_R`**. The overrun error (**OE**) is set if data has been lost because the input driver latency is too long. **BE** is a break error, meaning the other device has sent a break. **PE** is a parity error (however, we will not be using parity). The framing error (**FE**) will get set if the baud rates do not match. The software can clear these four error flags by writing any value to **`UART0_RSR_R`**.

The status of the two FIFOs can be seen in the **`UART0_FR_R`** register. The **BUSY** flag is set while the transmitter still has unsent bits, even if the transmitter is disabled. It will become zero when the transmit FIFO is empty and the last stop bit has been sent. If you implement busy-wait output by first outputting then waiting for **BUSY** to become 0 (right flowchart of Figure 11.10), then the routine will write new data and return after that particular data has been completely transmitted.

The **`UART0_CTL_R`** control register contains the bits that turn on the UART. **TXE** is the Transmitter Enable bit, and **RXE** is the Receiver Enable bit. We set **TXE**, **RXE**, and **UARTEN** equal to 1 in order to activate the UART device. However, we should clear **UARTEN** during the initialization sequence.

	31–12	11	10	9	8	7–0	Name
$4000.C000		OE	BE	PE	FE	DATA	UART0_DR_R

	31–3	3	2	1	0	
$4000.C004		OE	BE	PE	FE	UART0_RSR_R

	31–8	7	6	5	4	3	2–0	
$4000.C018		TXFE	RXFF	TXFF	RXFE	BUSY		UART0_FR_R

	31–16	15–0	
$4000.C024		DIVINT	UART0_IBRD_R

	31–6	5–0	
$4000.C028		DIVFRAC	UART0_FBRD_R

		7	6 – 5	4	3	2	1	0	
$4000.C02C		SPS	WPEN	FEN	STP2	EPS	PEN	BRK	UART0_LCRH_R

	31–10	9	8	7	6–3	2	1	0	
$4000.C030		RXE	TXE	LBE		SIRLP	SIREN	UARTEN	UART0_CTL_R

	31–12	11	10	9	8	7–0	
$4000.D000		OE	BE	PE	FE	DATA	UART1_DR_R

	31–3	3	2	1	0	
$4000.D004		OE	BE	PE	FE	UART1_RSR_R

	31–8	7	6	5	4	3	2–0	
$4000.D018		TXFE	RXFF	TXFF	RXFE	BUSY		UART1_FR_R

	31–16	15–0	
$4000.D024		DIVINT	UART1_IBRD_R

	31–6	5–0	
$4000.D028		DIVFRAC	UART1_FBRD_R

	31–8	7	6 – 5	4	3	2	1	0	
$4000.D02C		SPS	WPEN	FEN	STP2	EPS	PEN	BRK	UART1_LCRH_R

	31–10	9	8	7	6–3	2	1	0	
$4000.D030		RXE	TXE	LBE		SIRLP	SIREN	UARTEN	UART1_CTL_R

Table 11.2. Some UART registers. Each register is 32 bits wide. Shaded bits are zero.

The **`IBRD`** and **`FBRD`** registers specify the baud rate. The baud rate **divider** is a 22-bit binary fixed-point value with a resolution of 2^{-6}. The **Baud16** clock is created from the system bus clock, with a frequency of (Bus clock frequency)/**divider**. The baud rate is 16 times slower than **Baud16**

Baud rate = **Baud16**/**16** = (Bus clock frequency)/(16*__divider__)

For example, if the bus clock is 80 MHz and the desired baud rate is 19200 bits/sec, then the **divider** should be 80,000,000/16/19200 or 260.4167. Let *m* be the integer part, without rounding. We store the integer part (*m*=260) in **`IBRD`**. For the fraction, we find an integer *n*, such that *n*/64 is about 0.4167. More simply, we multiply 0.4167*64 = 26.6688 and round to the closest integer, 27. We store this fraction part (*n*=27) in **`FBRD`**. We did approximate the divider, so it is interesting to determine the actual baud rate. Assume the bus clock is 80 MHz.

Baud rate = (80 MHz)/(16* (*m*+*n*/64)) = (80 MHz)/(16* (260+27/64)) = 19199.616 bits/sec

The baud rates in the transmitter and receiver must match within 5% for the channel to operate properly. The error for this example is 0.002%.

The three registers **LCRH**, **IBRD**, and **FBRD** form an internal 30-bit register. This internal register is only updated when a write operation to **LCRH** is performed, so any changes to the baud-rate divisor must be followed by a write to the **LCRH** register for the changes to take effect. Out of reset, both FIFOs are disabled and act as 1-byte-deep holding registers. The FIFOs are enabled by setting the **FEN** bit in **LCRH**.

Checkpoint 11.2: Assume the bus clock is 10 MHz. What is the baud rate if UART0_IBRD_R equals 2 and UART0_FBRD_R equals 32?
Checkpoint 11.3: Assume the bus clock is 50 MHz. What values should you put in UART0_IBRD_R and UART0_FBRD_R to make a baud rate of 38400 bits/sec?

11.2.3. UART1 Device Driver on PC5 and PC4

Software that sends and receives data must implement a mechanism to synchronize the software with the hardware. In particular, the software should read data from the input device only when data is indeed ready. Similarly, software should write data to an output device only when the device is ready to accept new data. With busy-wait synchronization, the software continuously checks the hardware status waiting for it to be ready. In this section, we will use busy-wait synchronization to write I/O programs that send and receive data using the UART. After a frame is received, the receive FIFO will be not empty (**RXFE** becomes 0) and the 8-bit data is available to be read. To get new data from the serial port, the software first waits for **RXFE** to be zero, then reads the result from **UART1_DR_R**. Recall that when the software reads **UART1_DR_R** it gets data from the receive FIFO. This operation is illustrated in Figure 11.8 and shown in Program 11.1. In a similar fashion, when the software wishes to output via the serial port, it first waits for **TXFF** to be clear, then performs the output. When the software writes **UART1_DR_R** it puts data into the transmit FIFO.

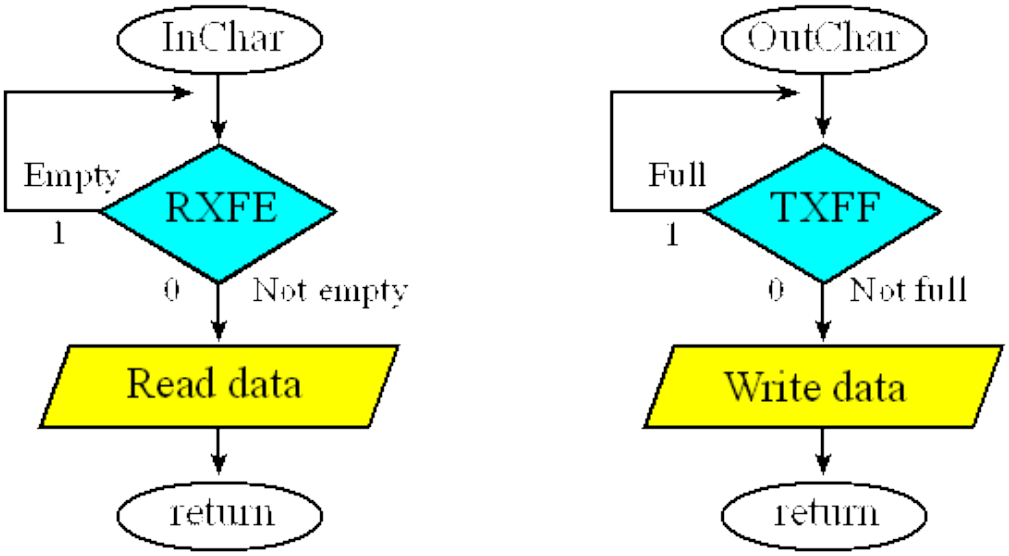


Figure 11.8. Flowcharts of InChar and OutChar using busy-wait synchronization.

The initialization program, **UART_Init**, enables the UART1 device and selects the baud rate. The **PCTL** bits were defined back in Chapter 6, and repeated as Table 11.3. **PCTL** bits 5-4 are set to 0x22 to select U1Tx and U1Rx on PC5 and PC4. The input routine waits in a loop until **RXFE** is 0 (FIFO not empty), then reads the data register. The output routine first waits in a loop until **TXFF** is 0 (FIFO not full), then writes data to the data register. Polling before writing data is an efficient way to perform output. **UART2_xxx.zip** is the interrupt-driven version. Be careful when using Port C to be friendly; the pins PC3-PC0 are used by the debugger and you should not modify their configurations.

IO	Ain	0	1	2	3	4	5	6	7	8	9	14
PA0		*Port*	*U0Rx*							CAN1Rx		
PA1		*Port*	*U0Tx*							CAN1Tx		
PA2		Port		SSI0Clk								
PA3		Port		SSI0Fss								
PA4		Port		SSI0Rx								
PA5		Port		SSI0Tx								
PA6		Port			I_2C1SCL		M1PWM2					
PA7		Port			I_2C1SDA		M1PWM3					
PB0		Port	U1Rx						T2CCP0			
PB1		Port	U1Tx						T2CCP1			
PB2		Port			I_2C0SCL				T3CCP0			
PB3		Port			I_2C0SDA				T3CCP1			
PB4	Ain10	Port		SSI2Clk		M0PWM2			T1CCP0	CAN0Rx		
PB5	Ain11	Port		SSI2Fss		M0PWM3			T1CCP1	CAN0Tx		
PB6		Port		SSI2Rx		M0PWM0			T0CCP0			
PB7		Port		SSI2Tx		M0PWM1			T0CCP1			
PC4	C1-	Port	U4Rx	U1Rx		M0PWM6		IDX1	WT0CCP0	U1RTS		
PC5	C1+	Port	U4Tx	U1Tx		M0PWM7		PhA1	WT0CCP1	U1CTS		
PC6	C0+	Port	U3Rx					PhB1	WT1CCP0	USB0epen		
PC7	C0-	Port	U3Tx						WT1CCP1	USB0pflt		
PD0	Ain7	Port	SSI3Clk	SSI1Clk	I_2C3SCL	M0PWM6	M1PWM0		WT2CCP0			
PD1	Ain6	Port	SSI3Fss	SSI1Fss	I_2C3SDA	M0PWM7	M1PWM1		WT2CCP1			
PD2	Ain5	Port	SSI3Rx	SSI1Rx		M0Fault0			WT3CCP0	USB0epen		
PD3	Ain4	Port	SSI3Tx	SSI1Tx				IDX0	WT3CCP1	USB0pflt		
PD4	*USB0DM*	*Port*	*U6Rx*						*WT4CCP0*			
PD5	*USB0DP*	*Port*	*U6Tx*						*WT4CCP1*			
PD6		Port	U2Rx			M0Fault0		PhA0	WT5CCP0			
PD7		Port	U2Tx					PhB0	WT5CCP1	NMI		
PE0	Ain3	Port	U7Rx									
PE1	Ain2	Port	U7Tx									
PE2	Ain1	Port										
PE3	Ain0	Port										
PE4	Ain9	Port	U5Rx		I_2C2SCL	M0PWM4	M1PWM2			CAN0Rx		
PE5	Ain8	Port	U5Tx		I_2C2SDA	M0PWM5	M1PWM3			CAN0Tx		
PF0		Port	U1RTS	SSI1Rx	CAN0Rx		M1PWM4	PhA0	T0CCP0	NMI	C0o	
PF1		Port	U1CTS	SSI1Tx			M1PWM5	PhB0	T0CCP1		C1o	TRD1
PF2		Port		SSI1Clk		M0Fault0	M1PWM6		T1CCP0			TRD0
PF3		Port		SSI1Fss	CAN0Tx		M1PWM7		T1CCP1			TRCLK
PF4		Port					M1Fault0	IDX0	T2CCP0	USB0epen		

Table 11.3. PMCx bits in the GPIOPCTL register on the LM4F/TM4C specify alternate functions. *PD4* and *PD5* are hardwired to the USB device. *PA0* and *PA1* are hardwired to the serial port. PWM does not exist on LM4F120.

```
// Assumes a 80 MHz bus clock, creates 115200 baud rate
void UART_Init(void){              // should be called only once
  SYSCTL_RCGC1_R |= 0x00000002;  // activate UART1
  SYSCTL_RCGC2_R |= 0x00000004;  // activate port C
  UART1_CTL_R &= ~0x00000001;    // disable UART
  UART1_IBRD_R = 43;     // IBRD = int(80,000,000/(16*115,200)) = int(43.40278)
  UART1_FBRD_R = 26;     // FBRD = round(0.40278 * 64) = 26
  UART1_LCRH_R = 0x00000070;  // 8 bit, no parity bits, one stop, FIFOs
  UART1_CTL_R |= 0x00000001;     // enable UART
  GPIO_PORTC_AFSEL_R |= 0x30;    // enable alt funct on PC5-4
  GPIO_PORTC_DEN_R |= 0x30;      // configure PC5-4 as UART1
  GPIO_PORTC_PCTL_R = (GPIO_PORTC_PCTL_R&0xFF00FFFF)+0x00220000;
  GPIO_PORTC_AMSEL_R &= ~0x30;   // disable analog on PC5-4
}
// Wait for new input, then return ASCII code
unsigned char UART_InChar(void){
  while((UART1_FR_R&0x0010) != 0);      // wait until RXFE is 0
  return((unsigned char)(UART1_DR_R&0xFF));
}
// Wait for buffer to be not full, then output
void UART_OutChar(unsigned char data){
  while((UART1_FR_R&0x0020) != 0);      // wait until TXFF is 0
```

```
  UART1_DR_R = data;
}
// Immediately return input or 0 if no input
unsigned char UART_InCharNonBlocking(void){
  if((UART1_FR_R&UART_FR_RXFE) == 0){
    return((unsigned char)(UART1_DR_R&0xFF));
  } else{
    return 0;
  }
}
```

Program 11.1. Device driver functions that implement serial I/O (CC11_UART and C11_Network).

Checkpoint 11.4: How does the software clear RXFE?
Checkpoint 11.5: How does the software clear TXFF?
Checkpoint 11.6: Describe what happens if the receiving computer is operating on a baud rate that is twice as fast as the transmitting computer?
Checkpoint 11.7: Describe what happens if the transmitting computer is operating on a baud rate that is twice as fast as the receiving computer?
Checkpoint 11.8: How do you change Program 11.1 to run at the same baud rate, but the system clock is now 10 MHz.

11.3. Conversions

In this section we will develop methods to convert between ASCII strings and binary numbers. Let's begin with a simple example. Let **Data** be a fixed length string of three ASCII characters. Each entry of **Data** is an ASCII character 0 to 9. Let **Data[0]** be the ASCII code for the hundred's digit, **Data[1]** be the ten's digit and **Data[2]** be the one's digit. Let **n** be an unsigned 32-bit integer. We will also need an index, **i**. The decimal digits 0 to 9 are encoded in ASCII as 0x30 to 0x39. So, to convert a single ASCII digit to a decimal number, we simply subtract 0x30. To convert this string of 3 decimal digits into binary we can simply calculate

```
n = 100*(Data[0]-0x30) + 10*(Data[1]-0x30) + (Data[2]-0x30);
```

Adding parentheses, we can convert this 3-digit ASCII string as

```
n = (Data[2]-0x30) + 10*((Data[1]-0x30) + 10*(Data[0]-0x30));
```

This second method could be used for converting any string of known and fixed length. If **Data** were a string of 9 decimal digits we could put the above function into a loop

```
n = 0;
for (i=0; i<9 ;i++){
  n = 10*n + (Data[i]-0x30);
}
```

If the length were variable, we can replace the for-loop with a while-loop. If **Data** were a variable length string of ASCII characters terminated with a null character (0), we could convert it to binary using a while loop, as shown in Program 11.2. A pointer to the string is passed using call by reference. In the assembly version, the pointer R0 is incremented as the string is parsed. R1 contains the local variable **n**, R2 contains the data from the string, and R3 contains the constant 10.

```
// Convert ASCII string to
//    unsigned 32-bit decimal
// string is null-terminated
unsigned long Str2UDec(unsigned char string[]){
  unsigned long i = 0;  // index
  unsigned long n = 0;  // number
  while(string[i] != 0){
```

```
    n = 10*n +(string[i]-0x30);
    i++;
  }
  return n;
}
```

Program 11.2. Unsigned ASCII string to decimal conversion.

The example, shown in Program 11.3, uses an I/O device capable of sending and receiving ASCII characters. When using a development board, we can send serial data to/from the PC using the UART. The function **UART_InChar()** returns an ASCII character from the I/O device. The function **UART_OutChar()** sends an ASCII character to the I/O device. The function **UART_InUDec()** will accept number characters (0x30 to 0x39) from the device until any non-number is typed. All input characters are echoed.

```
#define CR 0x0D
// Accept ASCII input in unsigned decimal format, up to 4294967295
// If n>4294967295, it will truncate without reporting the error
unsigned long UART_InUDec(void){
unsigned long n=0;            // this will be the return value
unsigned char character;      // this is the input ASCII typed
  while(1){
   character = UART_InChar(); // accepts input
   UART_OutChar(character);   // echo this character
   if((character < '0') || (character > '9')){  // check for non-number
     return n;                // quit if not a number
   }
   n = 10*n+(character-0x30); // overflows if above 4294967295
  }
}
```

Program 11.3. Input an unsigned decimal number.

If the ASCII characters were to contain optional "+" and "-" signs, we could look for the presence of the sign character in the first position. If there is a minus sign, then set a flag. Next use our unsigned conversion routine to process the rest of the ASCII characters and generate the unsigned number, **n**. If the flag was previously set, we can negate the value **n**. Be careful to guarantee the + and – are only processed as the first character.

To convert an unsigned integer into a fixed length string of ASCII characters, we could use the integer divide. Assume **n** is an unsigned integer less than or equal to 999. In Program 11.4, the number 0 is converted to the string "000". The first program stores the conversion in an array and the second function outputs the conversion to the UART.

```
unsigned char Data[4]; // 4-byte empty buffer
// n is the input 0 to 999
void UDec2Str(unsigned short n){
  Data[0] = n/100 + 0x30; // hundreds digit
  n = n%100;              // n is now between 0 and 99
  Data[1] = n/10 + 0x30;  // tens digit
  n = n%10;               // n is now between 0 and 9
  Data[2] = n + 0x30;     // ones digit
  Data[3] = 0;            // null termination
}
// n is the input 0 to 999
void UART_OutUDec3(unsigned short n){
  UART_OutChar(0x30+n/100); // hundreds digit
  n = n%100;                // 0 to 99
  UART_OutChar(0x30+n/10);  // tens digit
  n = n%10;                 // 0 to 9
```

```
  UART_OutChar(0x30+n);      // ones digit
```
Program 11.4. Unsigned decimal to ASCII string conversion.

Sometimes we represent noninteger values using integers. For example the system could store the value 1.23 as the integer 123. In this example, the voltage ranges from 0.00 to 9.99V, but the values in the computer are stored as integers 0 to 999. If the system wishes to display those values in volts, we simply add a decimal point to the output while converting, as shown in Program 11.5. For example calling **`OutVolt(123)`** will output the string "1.23V". Instead of creating an output string, this function outputs each character to display device by calling **`UART_OutChar`**.

```
//----------------------UART_OutUDec----------------------
// Output a voltage to the UART
// Input: n is an integer from 0 to 999 meaning 0.00 to 9.99V
// Output: none
// Fixed format: for example n=8 displayed as "0.08V"
void OutVolt(unsigned long n){  // each integer means 0.01V
  UART_OutChar(0x30+n/100);      // digit to the left of the decimal
  n = n%100;                     // 0 to 99
  UART_OutChar('.');             // decimal point
  UART_OutChar(0x30+n/10);       // tenths digit
  n = n%10;                      // 0 to 9
  UART_OutChar(0x30+n);          // hundredths digit
  UART_OutChar('V');}            // units
```
Program 11.5. Print the voltage value to an output device ($0 \le n \le 999$).

To convert an unsigned integer into a variable length string of ASCII characters, we convert the digits in reverse order, and then switch them.

```
//----------------------UART_OutUDec----------------------
// Output a 32-bit number in unsigned decimal format
// Input: 32-bit number to be transferred
// Output: none
// Variable format 1-10 digits with no space before or after
void UART_OutUDec(unsigned long n){
  if(n> = 10){
    UART_OutUDec(n/10);
    n = n%10;
  }
  UART_OutChar(n+'0'); /* n is between 0 and 9 */
}
```
Program 11.6. Print unsigned 32-bit decimal number to an output device.

Example 11.1. You are given a subroutine, **`UART_OutChar`**, which outputs one ASCII character. Design a function that outputs a 32-bit unsigned integer.

Solution: We will solve this iteratively. As always, we ask "what is our starting point?", "how do we make progress?", and "when are we done?" The input, n, is a 32-bit unsigned number, and we are done when 1 to 10 ASCII characters are displayed, representing the value of n. Figure 11.9 demonstrates the successive refinement approach to solving this problem iteratively. The iterative solution has three phases: initialization, creation of digits, and output of the ASCII characters. The digits are created from the remainders occurring by dividing the input, n by 10. To get all the digits we divide by 10 until the quotient is 0. Because the digits are created in the opposite order, each digit will be saved in a buffer during the creation phase and retrieved from the buffer during the output stage. The counter is needed so the output stage knows how many digits are in the buffer.

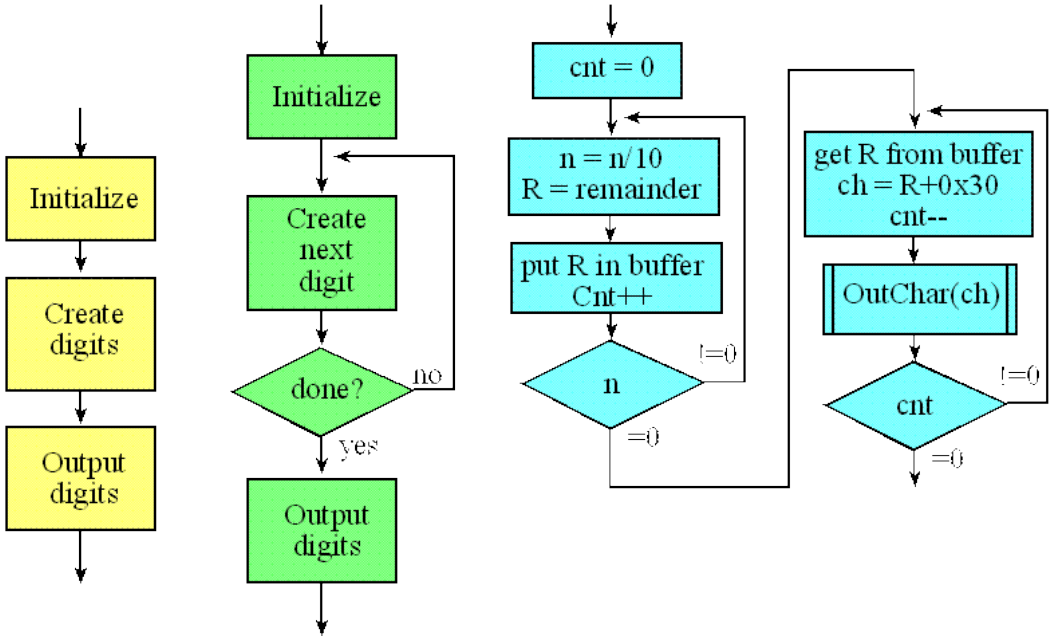


Figure 11.9. Successive refinement method for the iterative solution.

The iteration solution requires two loops; the first loop determines the digits in opposite order, and the second loop outputs the digits in proper order.

```
// iterative method
void OutUDec(unsigned long n){
unsigned cnt=0;
unsigned char buffer[11];
  do{
    buffer[cnt] = n%10;// digit
    n = n/10;
    cnt++;
  }
  while(n);// repeat until n==0
  for(; cnt; cnt--){
    OutChar(buffer[cnt-1]+'0');
  }
}
```

Program 11.7. Iterative implementation of output decimal.

11.4. Distributed Systems

A **network** is a collection of interfaces that share a physical medium and a data protocol. A network allows software tasks in one computer to communicate and synchronize with software tasks running on another computer. For an embedded system, the network provides a means for distributed computing. The **topology** of a network defines how the components are interconnected. Examples topologies include rings, buses and multi-hop. Figure 11.10 shows a **ring** network of three microcontrollers. The advantage of this ring network is low cost and can be implemented on any microcontroller with a serial port. Notice that the microcontrollers need not be the same type or speed.

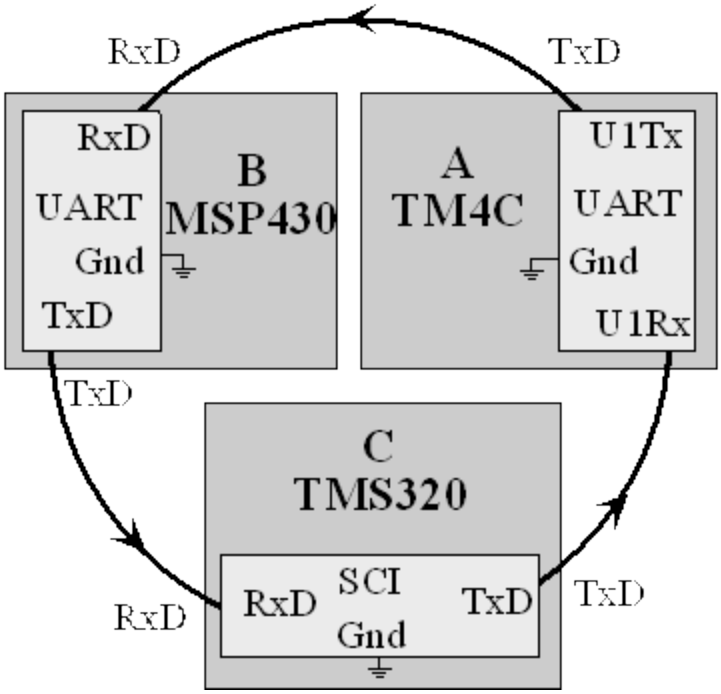


Figure 11.10. A simple ring network with three nodes, linked using the serial ports.

In this chapter we presented the hardware and software interfaces for the UART channel. We connected the TM4C to an I/O device and used the UART to communicate with the human. In this section, we will build on those ideas and introduce the concepts of networks by investigating a couple of simple networks. In particular, we will use the UART channel to connect multiple microcontrollers together, creating a network. A communication network includes both the physical channel (hardware) and the logical procedures (software) that allow users or software processes to communicate with each other. The network provides the transfer of information as well as the mechanisms for process synchronization. When faced with a complex problem, one could develop a solution on one powerful and **centralized** computer system. Alternatively a **distributed** solution could be employed using multiple computers connected by a network. The processing elements in Figure 11.11 may be a powerful computer, a microcontroller, an application-specific integrated circuit (ASIC), or a smart sensor/actuator.

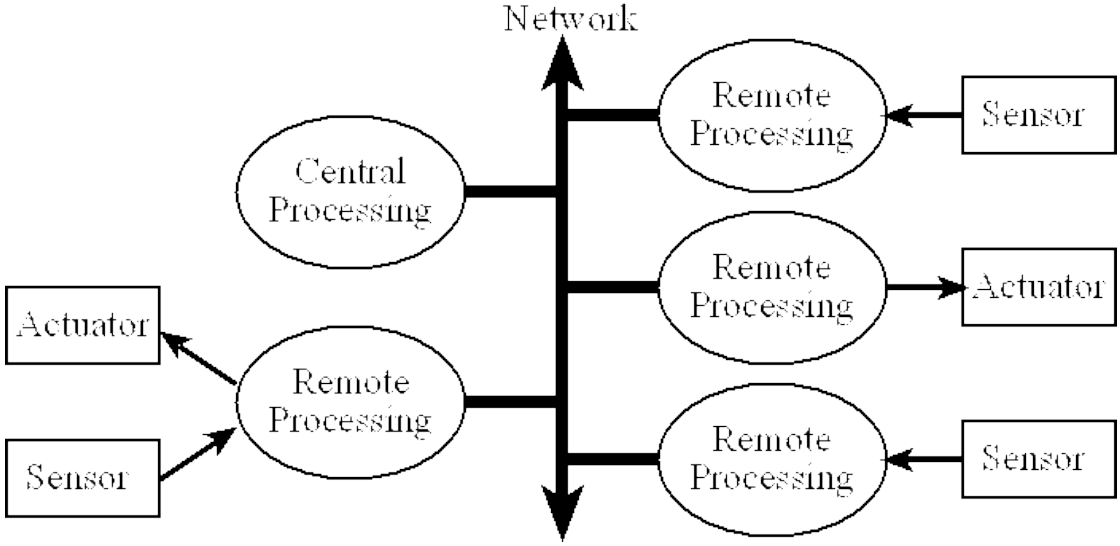


Figure 11.11. Distributed processing places input, output and processing at multiple locations connected together with a network.

There are many reasons to consider a distributed solution (network) over a centralized solution. Often multiple simple microcontrollers can provide a higher performance at lower cost compared to one

computer powerful enough to run the entire system. Some embedded applications require input/output activities that are physically distributed. For real-time operation there may not be enough time to allow communication a remote sensor and a central computer. Another advantage of distributed system is improved debugging. For example, we could use one node in a network to monitor and debug the others. Often, we do not know the level of complexity of our problem at design time. Similarly, over time the complexity may increase or decrease. A distributed system can often be deployed that can be scaled. For example, as the complexity increases more nodes can be added, and if the complexity were to decrease nodes could be removed.

Example 11.2. Develop a communication network between two LaunchPads. The switches are inputs, LED is output, and the UART is used to communicate. There will be five questions and three responses. The information is encoded as colors on the LED. The five questions are

Red: Are you there in your office?
Yellow: Are you happy?
Green: Are you hungry, want to have lunch?
Blue: Are you thirsty, want to meet for a beverage?
LightBlue: Shall I come to your office to talk?

The three answers are

White: Yes
Purple: Maybe
Dark: No

Solution: The operator selects the message to send by pushing the SW1/PF4 switch. While selecting the message the LED displays the message to be sent. Each time the operator pushes the SW1/PF4 switch, the system will cycle through the 8 possible colors on the LED. When the operator pushes the SW2/PF0 the message is sent. The message content is encoded as an ASCII character '0' to '7' (0x30 to 0x37) and send via the UART. The driver from Program 11.8 is used. When a UART frame is received, the data is encoded and the'0' to '7' data is displayed as the corresponding color on the LED. Figure 11.12 shows the hardware, which involves a 3-wire cable connecting the two LaunchPads.

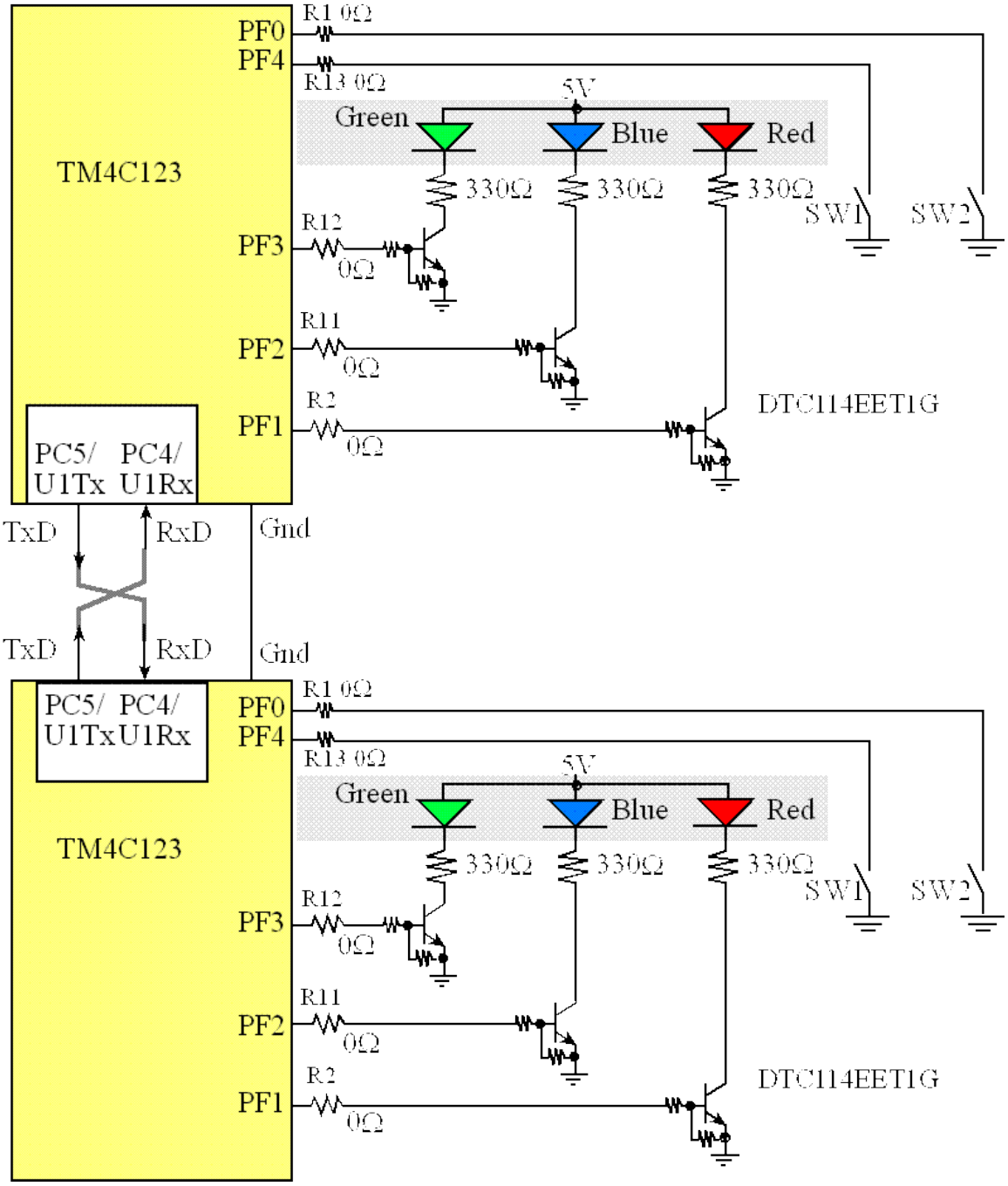


Figure 11.12. Distributed using two LaunchPads connected together by the UARTs.

```
//  red, yellow, green, light blue, blue, purple,  white,  dark
const long ColorWheel[8] = {0x02,0x0A,0x08,0x0C,0x04,0x06,0x0E,0x00};
int main(void){ unsigned long SW1,SW2;
  long prevSW1 = 0;         // previous value of SW1
  long prevSW2 = 0;         // previous value of SW2
  unsigned char inColor;    // color value from other microcontroller
  unsigned char color = 0; // this microcontroller's color value
  PLL_Init();               // set system clock to 80 MHz
  SysTick_Init();           // initialize SysTick
  UART_Init();              // initialize UART
  PortF_Init();             // initialize buttons and LEDs on Port F
  while(1){
    SW1 = GPIO_PORTF_DATA_R&0x10; // Read SW1
    if((SW1 == 0) && prevSW1){    // falling of SW1?
      color = (color+1)&0x07;     // step to next color
```

```
    }
    prevSW1 = SW1; // current value of SW1
    SW2 = GPIO_PORTF_DATA_R&0x01; // Read SW2
    if((SW2 == 0) && prevSW2){    // falling of SW2?
      UART_OutChar(color+0x30);   // send color as '0' - '7'
    }
    prevSW2 = SW2; // current value of SW2
    inColor = UART_InCharNonBlocking();
    if(inColor){ // new data have come in from the UART??
      color = inColor&0x07;     // update this computer's color
    }
    GPIO_PORTF_DATA_R = ColorWheel[color];  // update LEDs
    SysTick_Wait10ms(2);        // debounce switch
  }
}
```

Program 11.8. High-level communication network (C11_Network).

11.5. Interfacing the Nokia 5110 Using a Synchronous Serial Port

Microcontrollers employ multiple approaches to communicate synchronously with peripheral devices and other microcontrollers. The synchronous serial interface (SSI) system can operate as a master or as a slave. The channel can have one master and one slave, or it can have one master and multiple slaves. With multiple slaves, the configuration can be a star (centralized master connected to each slave), or a ring (each node has one receiver and one transmitter, where the nodes are connected in a circle.) The master initiates all data communication. The Nokia5110 is an optional LCD display as shown in Figure 11.13. The interface uses one of the synchronous serial ports on the TM4C123.

Figure 11.13. Optional Nokia 5110 LCD. Notice the PCB gives the signal names. Use the signal names not the numbers when connecting.

The TM4C123 microcontrollers has 4 **Synchronous Serial Interface** or **SSI** modules. Another name for this protocol is Serial Peripheral Interface or SPI. The fundamental difference between a UART, which implements an asynchronous protocol, and a SSI, which implements a synchronous protocol, is the manner in which the clock is implemented. Two devices communicating with asynchronous serial interfaces (UART) operate at the same frequency (baud rate) but have two separate clocks. With a UART protocol, the clock signal is not included in the interface cable between devices. Two UART devices can communicate with each other as long as the two clocks have frequencies within ±5% of

each other. Two devices communicating with synchronous serial interfaces (SSI) operate from the same clock (synchronized). With a SSI protocol, the clock signal is included in the interface cable between devices. Typically, the master device creates the clock, and the slave device(s) uses the clock to latch the data (in or out.) The SSI protocol includes four I/O lines. The slave select SSI0Fss is a negative logic control signal from master to slave signal signifying the channel is active. The second line, SCK, is a 50% duty cycle clock generated by the master. The SSI0Tx (master out slave in, MOSI) is a data line driven by the master and received by the slave. The SSI0Rx (master in slave out, MISO) is a data line driven by the slave and received by the master. In order to work properly, the transmitting device uses one edge of the clock to change its output, and the receiving device uses the other edge to accept the data. SSI allows data to flow both directions, but the Nokia5110 interface only transmits data from the TM4C123. Notice the pin PA4 is not used, which would have allowed for receiving data from the device. The Nokia5110 interface does not use PA4.

Program 11.9 shows the I/O port connections and the Nokia display. Be careful, there are multiple displays for sale on the market with the same LCD but different pin locations for the signals. Please look on your actual display for the pin name and not the pin number. Program 11.9 also lists some of the prototypes for public functions available in the software stater project. If you have ordered and received the Nokia5110 display, open the C11_Nokia5110 starter project, connect the display to PortA. Be careful when connecting the backlight, at 3.3V, the back light draws 80 mA. If you want a dimmer back light connect 3.3V to a 100 ohm resistor, and the other end of the resistor to the **BL** pin.

```
// Blue Nokia 5110
// ---------------
// Signal         (Nokia 5110) LaunchPad pin
// Reset          (RST, pin 1) connected to PA7
// SSI0Fss        (CE,  pin 2) connected to PA3
// Data/Command   (DC,  pin 3) connected to PA6
// SSI0Tx         (Din, pin 4) connected to PA5
// SSI0Clk        (Clk, pin 5) connected to PA2
// 3.3V           (Vcc, pin 6) power
// back light     (BL,  pin 7) not connected, consists of 4 white LEDs which draw
~80mA total
// Ground         (Gnd, pin 8) ground
// Red SparkFun Nokia 5110 (LCD-10168)
// -----------------------------------
// Signal         (Nokia 5110) LaunchPad pin
// 3.3V           (VCC, pin 1) power
// Ground         (GND, pin 2) ground
// SSI0Fss        (SCE, pin 3) connected to PA3
// Reset          (RST, pin 4) connected to PA7
// Data/Command   (D/C, pin 5) connected to PA6
// SSI0Tx         (DN,  pin 6) connected to PA5
// SSI0Clk        (SCLK, pin 7) connected to PA2
// back light     (LED, pin 8) not connected, consists of 4 white LEDs which draw
~80mA total//********Nokia5110_Init*****************
// Initialize Nokia 5110 48x84 LCD by sending the proper
// commands to the PCD8544 driver.
// inputs: none
// outputs: none
// assumes: system clock rate of 50 MHz or less
void Nokia5110_Init(void);
//********Nokia5110_OutChar*****************
// Print a character to the Nokia 5110 48x84 LCD.  The
// character will be printed at the current cursor position,
// the cursor will automatically be updated, and it will
// wrap to the next row or back to the top if necessary.
// One blank column of pixels will be printed on either side
// of the character for readability.  Since characters are 8
```

```
// pixels tall and 5 pixels wide, 12 characters fit per row,
// and there are six rows.
// inputs: data  character to print
// outputs: none
// assumes: LCD is in default horizontal addressing mode (V = 0)
void Nokia5110_OutChar(unsigned char data);
//********Nokia5110_OutString****************
// Print a string of characters to the Nokia 5110 48x84 LCD.
// The string will automatically wrap, so padding spaces may
// be needed to make the output look optimal.
// inputs: ptr  pointer to NULL-terminated ASCII string
// outputs: none
// assumes: LCD is in default horizontal addressing mode (V = 0)
void Nokia5110_OutString(char *ptr);
//********Nokia5110_OutUDec****************
// Output a 16-bit number in unsigned decimal format with a
// fixed size of five right-justified digits of output.
// Inputs: n  16-bit unsigned number
// Outputs: none
// assumes: LCD is in default horizontal addressing mode (V = 0)
void Nokia5110_OutUDec(unsigned short n);
//********Nokia5110_SetCursor****************
// Move the cursor to the desired X- and Y-position.  The
// next character will be printed here.  X=0 is the leftmost
// column.  Y=0 is the top row.
// inputs: newX  new X-position of the cursor (0<=newX<=11)
//         newY  new Y-position of the cursor (0<=newY<=5)
// outputs: none
void Nokia5110_SetCursor(unsigned char newX, unsigned char newY);
//********Nokia5110_Clear****************
// Clear the LCD by writing zeros to the entire screen and
// reset the cursor to (0,0) (top left corner of screen).
// inputs: none
// outputs: none
void Nokia5110_Clear(void);
```

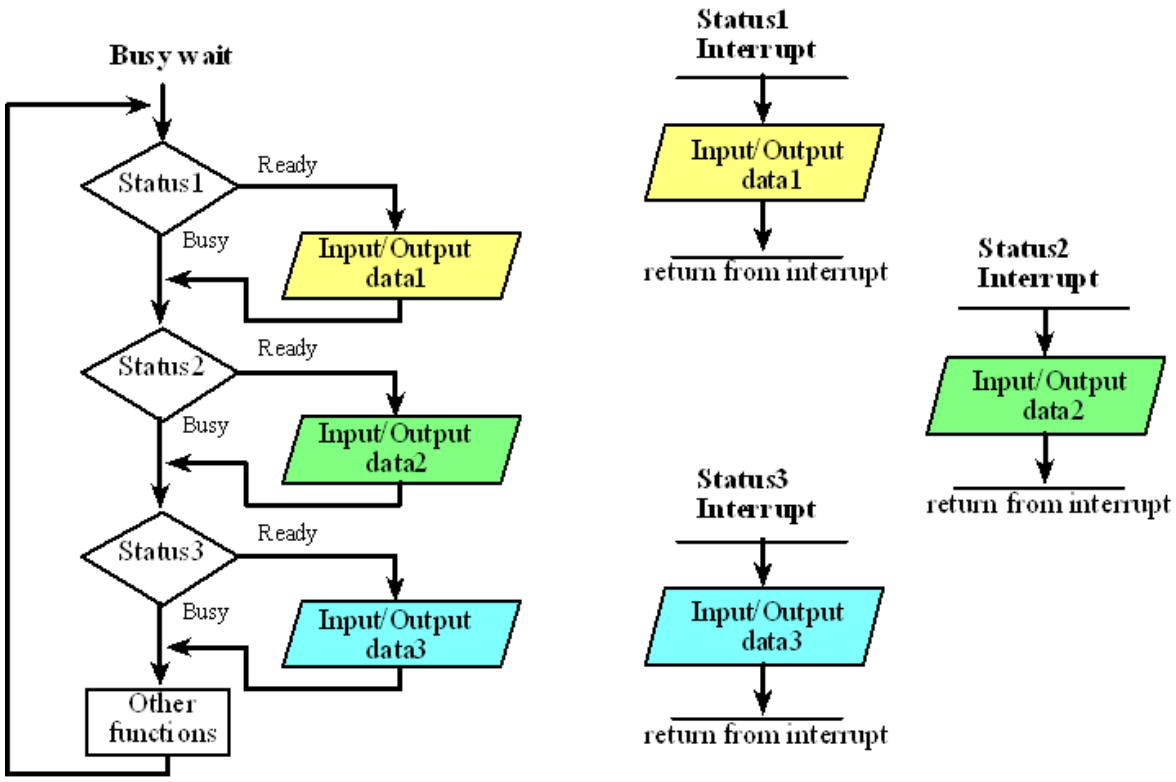
Program 11.9. Wiring connections and high-level connection between the LaunchPad and the Nokia5110 LCD display (C11_Nokia).

Chapter 12: Interrupts
Embedded Systems - Shape The World
Jonathan Valvano and Ramesh Yerraballi

An embedded system uses its input/output devices to interact with the external world. Input devices allow the computer to gather information, and output devices can display information. Output devices also allow the computer to manipulate its environment. The tight-coupling between the computer and external world distinguishes an embedded system from a regular computer system. The challenge is under most situations the software executes much faster than the hardware. E.g., it might take the software only 1 us to ask the hardware to clear the LCD, but the hardware might take 1 ms to complete the command. During this time, the software could execute tens of thousands of instructions. Therefore, the synchronization between the executing software and its external environment is critical for the success of an embedded system. This chapter presents general concepts about interrupts, and specific details for the Cortex™-M microcontroller. We will then use periodic interrupts to cause a software task to be executed on a periodic basis. If a GPIO pin is configured as an input, it can also be armed to invoke an interrupt on falling edges, rising edges or both falling and rising edges. Using interrupts allows the software to respond quickly to changes in the external environment.

Learning Objectives:

- Appreciate the need to perform multiple tasks concurrently.
- Understand perform measures of a real-time system such as bandwidth and latency
- Learn how interrupts can be used to minimize latency.
- Study the basics of interrupt programming: arm, enable, trigger, vector, priority, acknowledge.
- Understand how to use SysTick to create periodic interrupts
- Use SysTick to create sounds and spin motors.



12.1. Interrupt Concepts

An **interrupt** is the automatic transfer of software execution in response to a hardware event that is asynchronous with the current software execution. This hardware event is called a **trigger**. The

hardware event can either be a busy to ready transition in an external I/O device (like the UART input/output) or an internal event (like bus fault, memory fault, or a periodic timer). When the hardware needs service, signified by a busy to ready state transition, it will request an interrupt by setting its trigger flag. A **thread** is defined as the path of action of software as it executes. The execution of the interrupt service routine is called a background thread. This thread is created by the hardware interrupt request and is killed when the interrupt service routine returns from interrupt (e.g., by executing a **`BX LR`**). A new thread is created for each interrupt request. It is important to consider each individual request as a separate thread because local variables and registers used in the interrupt service routine are unique and separate from one interrupt event to the next interrupt. In a **multi-threaded** system, we consider the threads as cooperating to perform an overall task. Consequently we will develop ways for the threads to communicate (e.g., FIFO) and to synchronize with each other. Most embedded systems have a single common overall goal. On the other hand, general-purpose computers can have multiple unrelated functions to perform. A **process** is also defined as the action of software as it executes. Processes do not necessarily cooperate towards a common shared goal. Threads share access to I/O devices, system resources, and global variables, while processes have separate global variables and system resources. Processes do not share I/O devices.

There are no standard definitions for the terms mask, enable, and arm in the professional, Computer Science, or Computer Engineering communities. Nevertheless, in this class we will adhere to the following specific meanings. To **arm** a device means to allow the hardware trigger to interrupt. Conversely, to **disarm** a device means to shut off or disconnect the hardware trigger from the interrupts. Each potential interrupting trigger has a separate arm bit. One arms a trigger if one is interested in interrupts from this source. Conversely, one disarms a trigger if one is not interested in interrupts from this source. To **enable** means to allow interrupts at this time. Conversely, to **disable** means to postpone interrupts until a later time. On the ARM Cortex-M processor there is one interrupt enable bit for the entire interrupt system. We disable interrupts if it is currently not convenient to accept interrupts. In particular, to disable interrupts we set the I bit in **PRIMASK**. In C, we enable and disable interrupts by calling the functions **`EnableInterrupts()`** and **`DisableInterrupts()`** respectively.

The software has dynamic control over some aspects of the interrupt request sequence. First, each potential interrupt trigger has a separate **arm** bit that the software can activate or deactivate. The software will set the arm bits for those devices from which it wishes to accept interrupts, and will deactivate the arm bits within those devices from which interrupts are not to be allowed. In other words it uses the arm bits to individually select which devices will and which devices will not request interrupts. For most devices there is a enable bit in the NVIC that must be set (periodic SysTick interrupts are an exception, having no NVIC enable). The third aspect that the software controls is the interrupt enable bit. Specifically, bit 0 of the special register **PRIMASK** is the interrupt mask bit, **I**. If this bit is 1 most interrupts and exceptions are not allowed, which we will define as **disabled**. If the bit is 0, then interrupts are allowed, which we will define as **enabled**. The fourth aspect is priority. The **BASEPRI** register prevents interrupts with lower priority interrupts, but allows higher priority interrupts. For example if the software sets the **BASEPRI** to 3, then requests with level 0, 1, and 2 can interrupt, while requests at levels 3 and higher will be postponed. The software can also specify the priority level of each interrupt request. If **BASEPRI** is zero, then the priority feature is disabled and all interrupts are allowed. The fifth aspect is the external hardware trigger. One example of a hardware trigger is the **Count** flag in the **NVIC_ST_CTRL_R** register which is set periodically by SysTick. Another example of hardware triggers are bits in the **GPIO_PORTF_RIS_R** register that are set on rising or falling edges of digital input pins. Five conditions must be true for an interrupt to be generated:

1) device arm,
2) NVIC enable,
3) global enable,
4) interrupt priority level must be higher than current level executing, and

5) hardware event trigger.

For an interrupt to occur, these five conditions must be simultaneously true but can occur in any order. An interrupt causes the following sequence of five events. First, the current instruction is finished. Second, the execution of the currently running program is suspended, pushing eight registers on the stack (**R0**, **R1**, **R2**, **R3**, **R12**, **LR**, **PC**, and **PSR** with the **R0** on top). If the floating point unit on the TM4C123 is active, an additional 18 words will be pushed on the stack representing the floating point state, making a total of 26 words. Third, the **LR** is set to a specific value signifying an interrupt service routine (ISR) is being run (bits [31:4] to 0xFFFFFFF, and bits [3:0] specify the type of interrupt return to perform). In our examples we will see LR is set to 0xFFFFFFF9. If the floating point registers were pushed, the LR will be 0xFFFFFFE9. Fourth, the **IPSR** is set to the interrupt number being processed. Lastly, the **PC** is loaded with the address of the ISR (vector).

1) Current instruction is finished,
2) Eight registers are pushed on the stack,
3) LR is set to 0xFFFFFFF9,
4) IPSR is set to the interrupt number,
5) PC is loaded with the interrupt vector

These five steps, called a **context switch**, occur automatically in hardware as the context is switched from a foreground thread to a background thread. We can also have a context switch from a lower priority ISR to a higher priority ISR. Next, the software executes the ISR.

If a trigger flag is set, but the interrupts are disabled (I=1), the interrupt level is not high enough, or the flag is disarmed, the request is not dismissed. Rather the request is held **pending**, postponed until a later time, when the system deems it convenient to handle the requests. In other words, once the trigger flag is set, under most cases it remains set until the software clears it. The five necessary events (device arm, NVIC enable, global enable, level, and trigger) can occur in any order. For example, the software can set the I bit to prevent interrupts, run some code that needs to run to completion, and then clear the I bit. A trigger occurring while running with I=1 is postponed until the time the I bit is cleared again.

Clearing a trigger flag is called **acknowledgement**, which occurs only by specific software action. Each trigger flag has a specific action software must perform to clear that flag. We will pay special attention to these enable/disable software actions. The SysTick periodic interrupt will be the only example of an automatic acknowledgement. For SysTick, the periodic timer requests an interrupt, but the trigger flag will be automatically cleared when the ISR runs. For all the other trigger flags, the ISR must explicitly execute code that clears the flag.

The **interrupt service routine** (ISR) is the software module that is executed when the hardware requests an interrupt. There may be one large ISR that handles all requests (polled interrupts), or many small ISRs specific for each potential source of interrupt (vectored interrupts). The design of the interrupt service routine requires careful consideration of many factors. Except for the SysTick interrupt, the ISR software must explicitly clear the trigger flag that caused the interrupt (acknowledge). After the ISR provides the necessary service, it will execute **`BX LR`**. Because LR contains a special value (e.g., 0xFFFFFFF9), this instruction pops the 8 registers from the stack, which returns control to the main program. If the LR is 0xFFFFFFE9, then 26 registers (R0-R3,R12,LR,PC,PSW, and 18 floating point registers) will be popped by **`BX LR`**. There are two stack pointers: PSP and MSP. The software in this class will exclusively use the MSP. It is imperative that the ISR software balance the stack before exiting. Execution of the previous thread will then continue with the exact stack and register values that existed before the interrupt. Although interrupt handlers can create and use local variables, parameter passing between threads must be implemented using shared global memory variables. A private global variable can be used if an interrupt thread wishes to pass information to itself, e.g., from one interrupt

instance to another. The execution of the main program is called the foreground thread, and the executions of the various interrupt service routines are called background threads.

An axiom with interrupt synchronization is that the ISR should execute as fast as possible. The interrupt should occur when it is time to perform a needed function, and the interrupt service routine should perform that function, and return right away. Placing backward branches (busy-wait loops, iterations) in the interrupt software should be avoided if possible. The percentage of time spent executing interrupt software should be small when compared to the time between interrupt triggers.

Performance measures: latency and bandwidth. For an input device, the **interface latency** is the time between when new input is available, and the time when the software reads the input data. We can also define **device latency** as the response time of the external I/O device. For example, if we request that a certain sector be read from a disk, then the device latency is the time it take to find the correct track and spin the disk (seek) so the proper sector is positioned under the read head. For an output device, the interface latency is the time between when the output device is idle, and the time when the software writes new data. A **real-time** system is one that can guarantee worst case interface latency. **Bandwidth** is defined as the amount of data/sec being processed.

Many factors should be considered when deciding the most appropriate mechanism to synchronize hardware and software. One should not always use busy wait because one is too lazy to implement the complexities of interrupts. On the other hand, one should not always use interrupts because they are fun and exciting. Busy-wait synchronization is appropriate when the I/O timing is predictable and when the I/O structure is simple and fixed. Busy wait should be used for dedicated single thread systems where there is nothing else to do while the I/O is busy. Interrupt synchronization is appropriate when the I/O timing is variable, and when the I/O structure is complex. In particular, interrupts are efficient when there are I/O devices with different speeds. Interrupts allow for quick response times to important events. In particular, using interrupts is one mechanism to design real-time systems, where the interface latency must be short and bounded. **Bounded** means it is always less than a specified value. **Short** means the specified value is acceptable to our consumers.

Interrupts can also be used for infrequent but critical events like power failure, memory faults, and machine errors. Periodic interrupts will be useful for real-time clocks, data acquisition systems, and control systems. For extremely high bandwidth and low latency interfaces, direct memory access (DMA) should be used. DMA is described in Embedded Systems: Real-Time Operating Systems for ARM® Cortex™-M Microcontrollers . This example uses a timer and DMA to input data from an input port and store it in RAM *http://users.ece.utexas.edu/~valvano/arm/DMATimer_4F120.zip* and is beyond the scope of this introductory class.

An **atomic** operation is a sequence that once started will always finish, and cannot be interrupted. All instructions on the ARM® Cortex™-M processor are atomic except store and load multiple, **`STM LDM PUSH POP`**. If we wish to make a section of code atomic, we can run that code with I=1. In this way, interrupts will not be able to break apart the sequence. Again, requested interrupts that are triggered while I=1 are not dismissed, but simply postponed until I=0. In particular, to implement an atomic operation we will 1) save the current value of the **PRIMASK**, 2) disable interrupts, 3) execute the operation that needs to run atomically, and 4) restore the **PRIMASK** back to its previous value.

Checkpoint 12.1: What five conditions must be true for an interrupt to occur?
Checkpoint 12.2: How do you enable interrupts?
Checkpoint 12.3: What are the steps that occur when an interrupt is processed?

As you develop experience using interrupts, you will come to notice a few common aspects that most computers share. The following paragraphs outline three essential mechanisms that are needed to utilize

interrupts. Although every computer that uses interrupts includes all three mechanisms, how the mechanisms operate will vary from one computer to another.

All interrupting systems must have the **ability for the hardware to request action from computer**. In general, the interrupt requests can be generated using a separate connection to the processor for each device. The TM4C microcontrollers use separate connections to request interrupts.

All interrupting systems must have the **ability for the computer to determine the source**. A vectored interrupt system employs separate connections for each device so that the computer can give automatic resolution. You can recognize a vectored system because each device has a separate interrupt vector address. With a polled interrupt system, the interrupt software must poll each device, looking for the device that requested the interrupt. Most interrupts on the TM4C microcontrollers are vectored, but there are some triggers that share the same vector. For these interrupts the ISR must poll to see which trigger caused the interrupt. For example, all input pins on one GPIO port can trigger an interrupt, but the trigger flags share the same vector. So if multiple pins on one GPIO port are armed, the shared ISR must poll to determine which one(s) requested service.

The third necessary component of the interface is the **ability for the computer to acknowledge the interrupt.** Normally there is a trigger flag in the interface that is set on the busy to ready state transition. In essence, this trigger flag is the cause of the interrupt. Acknowledging the interrupt involves clearing this flag. It is important to shut off the request, so that the computer will not mistakenly request a second (and inappropriate) interrupt service for the same condition. Except for periodic SysTick interrupts, TM4C microcontrollers use software acknowledge. So when designing an interrupting interface, it will be important to know exactly what hardware condition sets the trigger flag (and request an interrupt) and how the software will clear it (acknowledge) in the ISR.

Common Error: The system will crash if the interrupt service routine doesn't either acknowledge or disarm the device requesting the interrupt.
Common Error: The ISR software should not disable interrupts at the beginning nor should it reenable interrupts at the end. Which interrupts are allowed to run is automatically controlled by the priority set in the NVIC.

12.2. Interthread Communication and Synchronization

For regular function calls we use the registers and stack to pass parameters, but interrupt threads have logically separate registers and stack. More specifically, registers are automatically saved by the processor as it switches from main program (foreground thread) to interrupt service routine (background thread). Exiting an ISR will restore the registers back to their previous values. Thus, all parameter passing must occur through global memory. One cannot pass data from the main program to the interrupt service routine using registers or the stack.

In this chapter, multi-threading means one main program (foreground thread) and multiple ISRs (background threads). An operating system allows multiple foreground threads. For more information on operating systems see Embedded Systems: Real-Time Operating Systems for ARM® Cortex™-M Microcontrollers. Synchronizing threads is a critical task affecting efficiency and effectiveness of systems using interrupts. In this section, we will present in general form three constructs to synchronize threads: binary semaphore, mailbox, and FIFO queue.

A **binary semaphore** is simply a shared flag, as described in Figure 12.0. There are two operations one can perform on a semaphore. **Signal** is the action that sets the flag. **Wait** is the action that checks the flag, and if the flag is set, the flag is cleared and important stuff is performed. This flag must exist as a private global variable with restricted access to only these two code pieces. In C, we add the qualifier **`static`** to a global variable to restrict access to software within the same file. In order to reduce

complexity of the system, it will be important to limit the access to this flag to as few modules as possible.

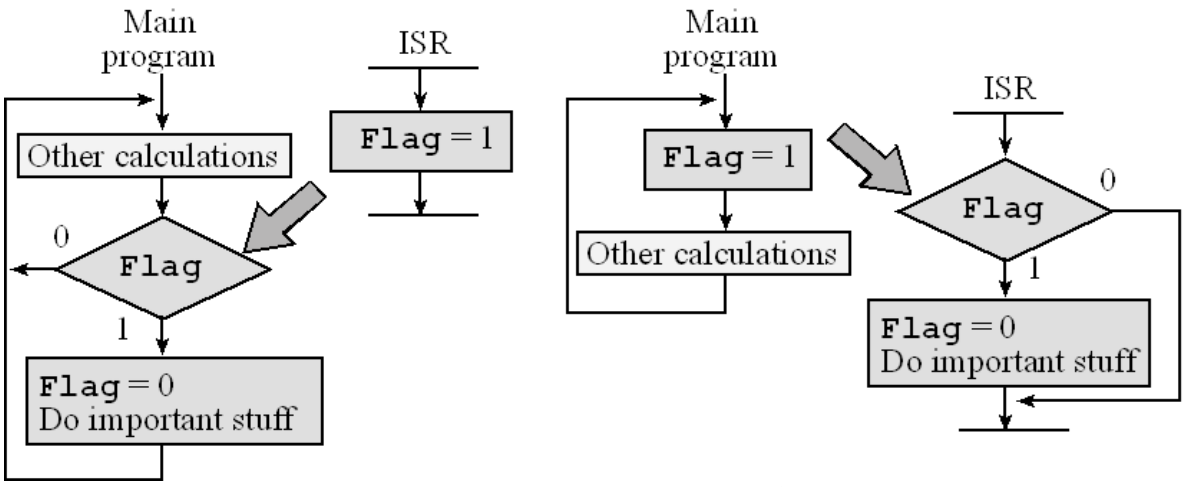


Figure 12.0. A semaphore can be used to synchronize threads.

A flag of course has two states: 0 and 1. However, it is good design to assign a meaning to this flag. For example, 0 might mean the switch has not been pressed, and 1 might mean the switch has been pressed. Figure 12.0 shows two examples of the binary semaphore. The big arrows in this figure signify synchronization links between the threads. In the example on the left, the ISR signals the semaphore and the main program waits on the semaphore. Notice the "important stuff" is run in the foreground once per execution of the ISR. In the example on the right, the main program signals the semaphore and the ISR waits. It is good design to have NO backwards jumps in an ISR. In this particular application, if the ISR is running and the semaphore is 0, the action is just skipped and the computer returns from the interrupt.
The second inter-thread synchronization scheme is the **mailbox**. The mailbox is a binary semaphore with associated data variable. Interactive Tool 12.1 illustrates an input device interfaced using interrupt synchronization and uses a mailbox to send data from ISR to the main program. The mailbox structure is implemented with two shared global variables. **`Mail`** contains data, and **`Status`** is a semaphore flag specifying whether the mailbox is full or empty.

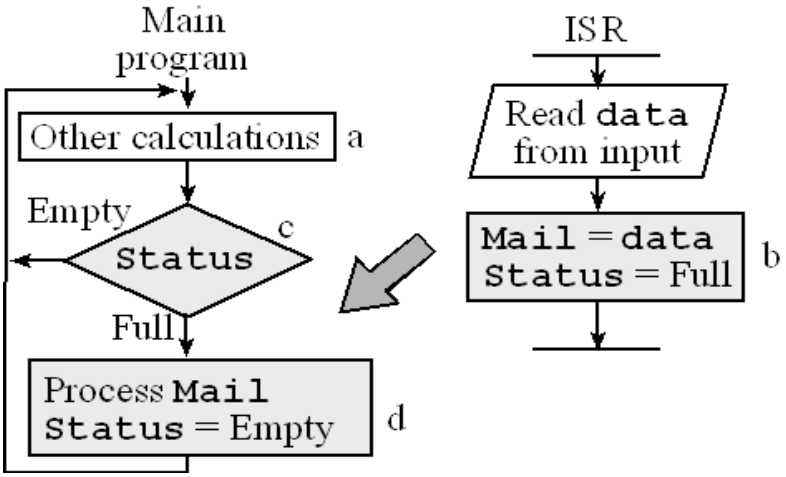


Figure 12.1. A mailbox can be used to pass data between threads.

Code Blocks, A, B, C and D refer to the blocks labelled (lower-case) in Figure 12.1. The interrupt is requested when its trigger flag is set, signifying new data are ready from the input device. The ISR will read the data from the input device and store it in the shared global variable **`Mail`**, then update its **`Status`** to full. The main program will perform other calculations, while occasionally checking the

status of the mailbox. When the mailbox has data, the main program will process it. This approach is adequate for situations where the input bandwidth is slow compared to the software processing speed. One way to visualize the interrupt synchronization is to draw a state versus time plot of the activities of the hardware, the mailbox, and the two software threads .

Using the tool demonstrates that during execution of block A, the mailbox is empty, the input device is idle and the main program is performing other tasks, because mailbox is empty. When new input data are ready, the trigger flag will be set, and an interrupt will be requested. In block B the ISR reads data from input device and saves it in **`Mail`**, and then it sets **`Status`** to full. The main program recognizes **`Status`** is full in Block C. In Block D, the main program processes data from **`Mail`**, sets **`Status`** to empty. Notice that even though there are two threads, only one is active at a time. The interrupt hardware switches the processor from the main program to the ISR, and the return from interrupt switches the processor back.

The third synchronization technique is the **FIFO queue**. The use of a FIFO is similar to the mailbox, but allows buffering, which is storing data in a first come first served manner. For an input device, an interrupt occurs when new input data are available, the ISR reads the data from the input device, and puts the data in the FIFO. Whenever the main program is idle, it will attempt to get data from the FIFO. If data were to exist, that data will be processed. The big arrows in Figures 12.4 and 12.5 signify the communication and synchronization link between the background and foreground.

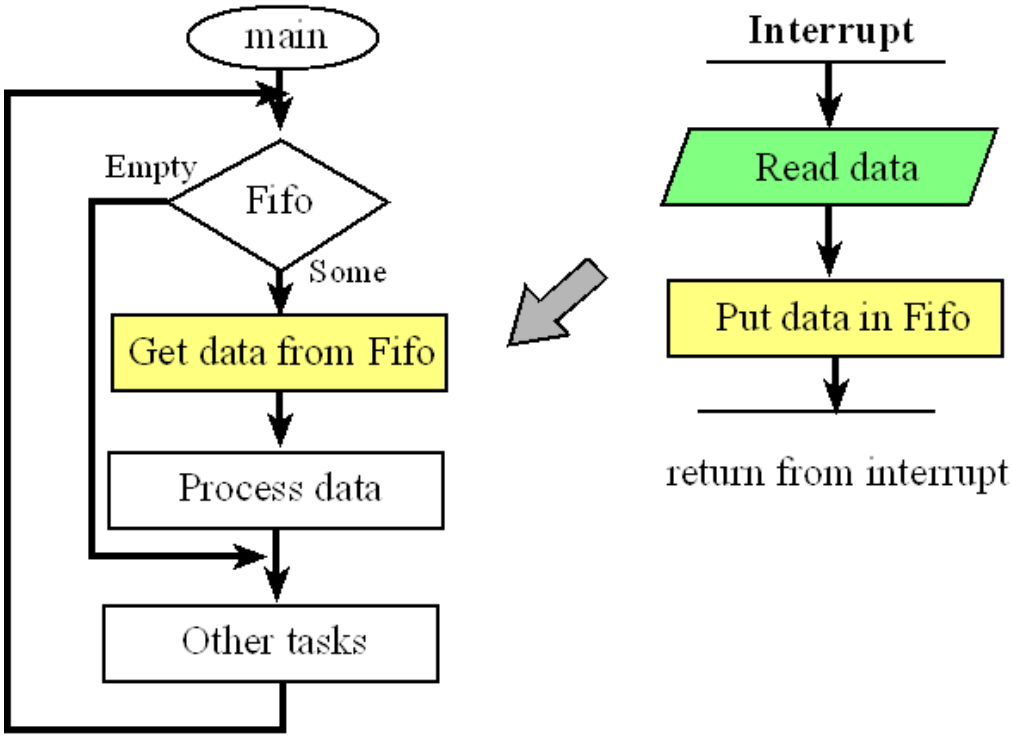


Figure 12.2. For an input device we can use a FIFO to pass data from the ISR to the main program.

For an output device, the main program puts data into the FIFO whenever it wishes to perform output. This data is buffered, and if the system is properly configured, the FIFO never becomes full and the main program never actually waits for the output to occur. An interrupt occurs when the output device is idle, the ISR gets from the FIFO and write the data to the output device. Whenever the ISR sees the FIFO is empty, it could cause the output device to become idle. The direction of the big arrows in Figures 12.2 and 12.3 signify the direction of data flow in these buffered I/O examples.

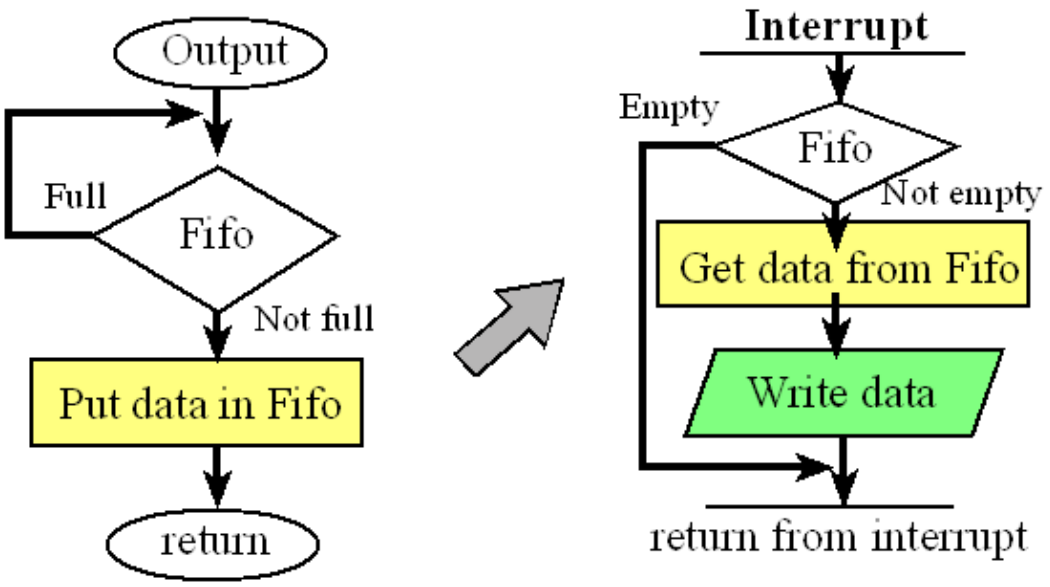


Figure 12.3. For an output device we can use a FIFO to pass data from the main program to the ISR.

There are other types of interrupt that are not an input or output. For example we will configure the computer to request an interrupt on a periodic basis. This means an interrupt handler will be executed at fixed time intervals. This periodic interrupt will be essential for the implementation of real-time data acquisition and real-time control systems. For example if we are implementing a digital controller that executes a control algorithm 100 times a second, then we will set up the internal timer hardware to request an interrupt every 10 ms. The interrupt service routine will execute the digital control algorithm and then return to the main thread. In a similar fashion, we will use periodic interrupts to perform analog input and/or analog output. For example if we wish to sample the ADC 100 times a second, then we will set up the internal timer hardware to request an interrupt every 10 ms. The interrupt service routine will sample the ADC, process (or save) the data, and then return to the main thread.

Performance Tip: It is poor design to employ backward jumps in an ISR, because they may affect the latency of other interrupt requests. Whenever you are thinking about using a backward jump, consider redesigning the system with more or different triggers to reduce the number of backward jumps.

12.3. NVIC on the ARM Cortex-M Processor

On the ARM Cortex-M processor, **exceptions** include resets, software interrupts and hardware interrupts. Interrupts on the Cortex-M are controlled by the Nested Vectored Interrupt Controller (NVIC). Each exception has an associated 32-bit vector that points to the memory location where the ISR that handles the exception is located. Vectors are stored in ROM at the beginning of memory. Program 12.1 shows the first few vectors as defined in the **Startup.s** file. **`DCD`** is an assembler pseudo-op that defines a 32-bit constant. ROM location 0x0000.0000 has the initial stack pointer, and location 0x0000.0004 contains the initial program counter, which is called the reset vector. It points to a function called the reset handler, which is the first thing executed following reset. There are up to 240 possible interrupt sources and their 32-bit vectors are listed in order starting with location 0x0000.0008. From a programming perspective, we can attach ISRs to interrupts by writing the ISRs as regular assembly subroutines or C functions with no input or output parameters and editing the **Startup.s** file to specify those functions for the appropriate interrupt. For example, if we wrote a Port F interrupt service routine named **`PortFISR`**, then we would replace **`GPIOPortF_Handler`** with **`PortFISR`**. In this class, we will write our ISRs using standard function names so that the **Startup.s** file need not be edited. I.e., we will simply name the ISR for edge-triggered interrupts on Port F as **`GPIOPortF_Handler`**. The ISR for this interrupt is a 32-bit pointer located at ROM address 0x0000.00B8. Because the vectors

are in ROM, this linkage is defined at compile time and not at run time. For more details see the **Startup.s** files within the interrupt examples posted on the book web site.

Checkpoint 12.4: Where is the vector for SysTick? What is the standard name for this ISR?

```
        EXPORT  __Vectors
__Vectors                               ; address    ISR
        DCD     StackMem + Stack        ; 0x00000000 Top of Stack
        DCD     Reset_Handler           ; 0x00000004 Reset Handler
        DCD     NMI_Handler             ; 0x00000008 NMI Handler
        DCD     HardFault_Handler       ; 0x0000000C Hard Fault Handler
        DCD     MemManage_Handler       ; 0x00000010 MPU Fault Handler
        DCD     BusFault_Handler        ; 0x00000014 Bus Fault Handler
        DCD     UsageFault_Handler      ; 0x00000018 Usage Fault Handler
        DCD     0                       ; 0x0000001C Reserved
        DCD     0                       ; 0x00000020 Reserved
        DCD     0                       ; 0x00000024 Reserved
        DCD     0                       ; 0x00000028 Reserved
        DCD     SVC_Handler             ; 0x0000002C SVCall Handler
        DCD     DebugMon_Handler        ; 0x00000030 Debug Monitor Handler
        DCD     0                       ; 0x00000034 Reserved
        DCD     PendSV_Handler          ; 0x00000038 PendSV Handler
        DCD     SysTick_Handler         ; 0x0000003C SysTick Handler
        DCD     GPIOPortA_Handler       ; 0x00000040 GPIO Port A
        DCD     GPIOPortB_Handler       ; 0x00000044 GPIO Port B
        DCD     GPIOPortC_Handler       ; 0x00000048 GPIO Port C
        DCD     GPIOPortD_Handler       ; 0x0000004C GPIO Port D
        DCD     GPIOPortE_Handler       ; 0x00000050 GPIO Port E
        DCD     UART0_Handler           ; 0x00000054 UART0
        DCD     UART1_Handler           ; 0x00000058 UART1
        DCD     SSI0_Handler            ; 0x0000005C SSI
        DCD     I2C0_Handler            ; 0x00000060 I2C
        DCD     PWM0Fault_Handler       ; 0x00000064 PWM Fault
        DCD     PWM0Generator0_Handler  ; 0x00000068 PWM 0 Generator 0
        DCD     PWM0Generator1_Handler  ; 0x0000006C PWM 0 Generator 1
        DCD     PWM0Generator2_Handler  ; 0x00000070 PWM 0 Generator 2
        DCD     Quadrature0_Handler     ; 0x00000074 Quadrature Encoder 0
        DCD     ADC0Seq0_Handler        ; 0x00000078 ADC0 Sequence 0
        DCD     ADC0Seq1_Handler        ; 0x0000007C ADC0 Sequence 1
        DCD     ADC0Seq2_Handler        ; 0x00000080 ADC0 Sequence 2
        DCD     ADC0Seq3_Handler        ; 0x00000084 ADC0 Sequence 3
        DCD     WDT_Handler             ; 0x00000088 Watchdog
        DCD     Timer0A_Handler         ; 0x0000008C Timer 0 subtimer A
        DCD     Timer0B_Handler         ; 0x00000090 Timer 0 subtimer B
        DCD     Timer1A_Handler         ; 0x00000094 Timer 1 subtimer A
        DCD     Timer1B_Handler         ; 0x00000098 Timer 1 subtimer B
        DCD     Timer2A_Handler         ; 0x0000009C Timer 2 subtimer A
        DCD     Timer2B_Handler         ; 0x000000A0 Timer 2 subtimer B
        DCD     Comp0_Handler           ; 0x000000A4 Analog Comp 0
        DCD     Comp1_Handler           ; 0x000000A8 Analog Comp 1
        DCD     Comp2_Handler           ; 0x000000AC Analog Comp 2
        DCD     SysCtl_Handler          ; 0x000000B0 System Control
        DCD     FlashCtl_Handler        ; 0x000000B4 Flash Control
        DCD     GPIOPortF_Handler       ; 0x000000B8 GPIO Port F
```

Program 12.1. Software syntax to set the interrupt vectors for the TM4C (only some vectors are shown, see the startup.s file for a complete list).

Program 12.2 shows that the syntax for an ISR looks like a function with no parameters. Notice that each ISR (except for SysTick) must acknowledge the interrupt in software by clearing the flag that caused the interrupt. In Program 12.2, we assume the interrupt was caused by an edge on PF4, so writing to the ICR register will clear trigger flag 4.

```
void GPIOPortF_Handler(void){
  GPIO_PORTF_ICR_R = 0x10; // ack, clear interrupt flag4
  // stuff
}
```
Program 12.2. Typical interrupt service routine.

Vector address	Number	IRQ	ISR name in **Startup.s**	NVIC	Priority bits
0x00000038	14	-2	**PendSV_Handler**	**NVIC_SYS_PRI3_R**	23 – 21
0x0000003C	15	-1	**SysTick_Handler**	**NVIC_SYS_PRI3_R**	31 – 29
0x00000040	16	0	**GPIOPortA_Handler**	**NVIC_PRI0_R**	7 – 5
0x00000044	17	1	**GPIOPortB_Handler**	**NVIC_PRI0_R**	15 – 13
0x00000048	18	2	**GPIOPortC_Handler**	**NVIC_PRI0_R**	23 – 21
0x0000004C	19	3	**GPIOPortD_Handler**	**NVIC_PRI0_R**	31 – 29
0x00000050	20	4	**GPIOPortE_Handler**	**NVIC_PRI1_R**	7 – 5
0x00000054	21	5	**UART0_Handler**	**NVIC_PRI1_R**	15 – 13
0x00000058	22	6	**UART1_Handler**	**NVIC_PRI1_R**	23 – 21
0x0000005C	23	7	**SSI0_Handler**	**NVIC_PRI1_R**	31 – 29
0x00000060	24	8	**I2C0_Handler**	**NVIC_PRI2_R**	7 – 5
0x00000064	25	9	**PWM0Fault_Handler**	**NVIC_PRI2_R**	15 – 13
0x00000068	26	10	**PWM0_Handler**	**NVIC_PRI2_R**	23 – 21
0x0000006C	27	11	**PWM1_Handler**	**NVIC_PRI2_R**	31 – 29
0x00000070	28	12	**PWM2_Handler**	**NVIC_PRI3_R**	7 – 5
0x00000074	29	13	**Quadrature0_Handler**	**NVIC_PRI3_R**	15 – 13
0x00000078	30	14	**ADC0_Handler**	**NVIC_PRI3_R**	23 – 21
0x0000007C	31	15	**ADC1_Handler**	**NVIC_PRI3_R**	31 – 29
0x00000080	32	16	**ADC2_Handler**	**NVIC_PRI4_R**	7 – 5
0x00000084	33	17	**ADC3_Handler**	**NVIC_PRI4_R**	15 – 13
0x00000088	34	18	**WDT_Handler**	**NVIC_PRI4_R**	23 – 21
0x0000008C	35	19	**Timer0A_Handler**	**NVIC_PRI4_R**	31 – 29
0x00000090	36	20	**Timer0B_Handler**	**NVIC_PRI5_R**	7 – 5
0x00000094	37	21	**Timer1A_Handler**	**NVIC_PRI5_R**	15 – 13
0x00000098	38	22	**Timer1B_Handler**	**NVIC_PRI5_R**	23 – 21
0x0000009C	39	23	**Timer2A_Handler**	**NVIC_PRI5_R**	31 – 29
0x000000A0	40	24	**Timer2B_Handler**	**NVIC_PRI6_R**	7 – 5
0x000000A4	41	25	**Comp0_Handler**	**NVIC_PRI6_R**	15 – 13
0x000000A8	42	26	**Comp1_Handler**	**NVIC_PRI6_R**	23 – 21
0x000000AC	43	27	**Comp2_Handler**	**NVIC_PRI6_R**	31 – 29
0x000000B0	44	28	**SysCtl_Handler**	**NVIC_PRI7_R**	7 – 5
0x000000B4	45	29	**FlashCtl_Handler**	**NVIC_PRI7_R**	15 – 13
0x000000B8	46	30	**GPIOPortF_Handler**	**NVIC_PRI7_R**	23 – 21
0x000000BC	47	31	**GPIOPortG_Handler**	**NVIC_PRI7_R**	31 – 29
0x000000C0	48	32	**GPIOPortH_Handler**	**NVIC_PRI8_R**	7 – 5
0x000000C4	49	33	**UART2_Handler**	**NVIC_PRI8_R**	15 – 13
0x000000C8	50	34	**SSI1_Handler**	**NVIC_PRI8_R**	23 – 21
0x000000CC	51	35	**Timer3A_Handler**	**NVIC_PRI8_R**	31 – 29
0x000000D0	52	36	**Timer3B_Handler**	**NVIC_PRI9_R**	7 – 5
0x000000D4	53	37	**I2C1_Handler**	**NVIC_PRI9_R**	15 – 13
0x000000D8	54	38	**Quadrature1_Handler**	**NVIC_PRI9_R**	23 – 21
0x000000DC	55	39	**CAN0_Handler**	**NVIC_PRI9_R**	31 – 29
0x000000E0	56	40	**CAN1_Handler**	**NVIC_PRI10_R**	7 – 5
0x000000E4	57	41	**CAN2_Handler**	**NVIC_PRI10_R**	15 – 13
0x000000E8	58	42	**Ethernet_Handler**	**NVIC_PRI10_R**	23 – 21
0x000000EC	59	43	**Hibernate_Handler**	**NVIC_PRI10_R**	31 – 29
0x000000F0	60	44	**USB0_Handler**	**NVIC_PRI11_R**	7 – 5
0x000000F4	61	45	**PWM3_Handler**	**NVIC_PRI11_R**	15 – 13
0x000000F8	62	46	**uDMA_Handler**	**NVIC_PRI11_R**	23 – 21
0x000000FC	63	47	**uDMA_Error**	**NVIC_PRI11_R**	31 – 29

Table 12.1. Some of the interrupt vectors for the TM4C. The TM4C123 has over 100 possible interrupt sources.

To activate an interrupt source we need to set its priority and enable that source in the NVIC. This activation is in addition to the arm and enable steps. Table 12.1 lists some of the interrupt sources available on the TM4C family of microcontrollers. Interrupt numbers 0 to 15 contain the faults, software interrupt and SysTick; these interrupts will be handled differently from interrupts 16 and up.

Table 12.2 shows some of the priority registers on the NVIC. Each register contains an 8-bit priority field for four devices. On the TM4C microcontrollers, only the top three bits of the 8-bit field are used. This allows us to specify the interrupt priority level for each device from 0 to 7, with 0 being the highest priority. The interrupt number (number column in Table 12.1) is loaded into the **IPSR** register. The servicing of interrupts does not set the I bit in the **PRIMASK**, so a higher priority interrupt can suspend the execution of a lower priority ISR. If a request of equal or lower priority is generated while an ISR is being executed, that request is postponed until the ISR is completed. In particular, those devices that need prompt service should be given high priority.

Address	31 – 29	23 – 21	15 – 13	7 – 5	Name
0xE000E400	GPIO Port D	GPIO Port C	GPIO Port B	GPIO Port A	NVIC_PRI0_R
0xE000E404	SSI0, Rx Tx	UART1, Rx Tx	UART0, Rx Tx	GPIO Port E	NVIC_PRI1_R
0xE000E408	PWM Gen 1	PWM Gen 0	PWM Fault	I2C0	NVIC_PRI2_R
0xE000E40C	ADC Seq 1	ADC Seq 0	Quad Encoder	PWM Gen 2	NVIC_PRI3_R
0xE000E410	Timer 0A	Watchdog	ADC Seq 3	ADC Seq 2	NVIC_PRI4_R
0xE000E414	Timer 2A	Timer 1B	Timer 1A	Timer 0B	NVIC_PRI5_R
0xE000E418	Comp 2	Comp 1	Comp 0	Timer 2B	NVIC_PRI6_R
0xE000E41C	GPIO Port G	GPIO Port F	Flash Control	System Control	NVIC_PRI7_R
0xE000E420	Timer 3A	SSI1, Rx Tx	UART2, Rx Tx	GPIO Port H	NVIC_PRI8_R
0xE000E424	CAN0	Quad Encoder 1	I2C1	Timer 3B	NVIC_PRI9_R
0xE000E428	Hibernate	Ethernet	CAN2	CAN1	NVIC_PRI10_R
0xE000E42C	uDMA Error	uDMA Soft Tfr	PWM Gen 3	USB0	NVIC_PRI11_R
0xE000ED20	SysTick	PendSV	--	Debug	NVIC_SYS_PRI3_R

Table 12.2. The LM3S/TM4C NVIC registers. Each register is 32 bits wide. Bits not shown are zero.

There are five enable registers **`NVIC_EN0_R`** through **`NVIC_EN4_R`**. The 32 bits in register **`NVIC_EN0_R`** control the IRQ numbers 0 to 31 (interrupt numbers 16 – 47). In Table 12.1 we see UART0 is IRQ=5. To enable UART0 interrupts we set bit 5 in **`NVIC_EN0_R`**, see Table 12.3. The 32 bits in **`NVIC_EN1_R`** control the IRQ numbers 32 to 63 (interrupt numbers 48 – 79). In Table 12.1 we see UART2 is IRQ=33. To enable UART interrupts we set bit 1 (33-32=1) in **`NVIC_EN1_R`**, see Table 12.3. Not every interrupt source is available on every TM4C microcontroller, so you will need to refer to the data sheet for your microcontroller when designing I/O interfaces. Writing zeros to the **`NVIC_EN0_R`** through **`NVIC_EN4_R`** registers has no effect. To disable interrupts we write ones to the corresponding bit in the **`NVIC_DIS0_R`** through **`NVIC_DIS4_R`** register.

Address	31	30	29-7	6	5	4	3	2	1	0	Name
0xE000E100	G	F	…	UART1	UART0	E	D	C	B	A	NVIC_EN0_R
0xE000E104			…						UART2	H	NVIC_EN1_R

Table 12.3. Some of the TM4C NVIC interrupt enable registers. There are five such registers defining 139 interrupt enable bits.

Interactive tool 12.2 shows the context switch from executing in the foreground to running a periodic SysTick ISR. Before the interrupt occurs, the I bit in the PRIMASK is 0 signifying interrupts are enabled, and the interrupt number (ISRNUM) in the **IPSR** register is 0, meaning we are running in **Thread mode** (i.e., the main program, and not an ISR). **Handler mode** is signified by a nonzero value

in **IPSR**. When **BASEPRI** register is zero, all interrupts are allowed and the **BASEPRI** register is not active.

When the SysTick counter goes from 1 to 0, the **Count** flag in the **NVIC_ST_CTRL_R** register is set, triggering an interrupt. The current instruction is finished. (a) Eight registers are pushed on the stack with **R0** on top. These registers are pushed onto the stack . (b) The vector address is loaded into the **PC** ("Vector address" column in Table 12.1). (c) The **IPSR** register is set to 15 ("Number" column in Table 12.1) (d) The top 24 bits of **LR** are set to 0xFFFFFF, signifying the processor is executing an ISR. The bottom eight bits specify how to return from interrupt.

0xE1 Return to Handler mode MSP (using floating point state on TM4C)
0xE9 Return to Thread mode MSP (using floating point state on TM4C)
0xED Return to Thread mode PSP (using floating point state on TM4C)
0xF1 Return to Handler mode MSP
0xF9 Return to Thread mode MSP ← in this class we will always be using this one
0xFD Return to Thread mode PSP

After pushing the registers, the processor always uses the main stack pointer (**MSP**) during the execution of the ISR. Events 2, 3, and 4 can occur simultaneously

To **return from an interrupt**, the ISR executes the typical function return **`BX LR`**. However, since the top 24 bits of **LR** are 0xFFFFFF, it knows to return from interrupt by popping the eight registers off the stack. Since the bottom eight bits of **LR** in this case are 0b11111001, it returns to thread mode using the **MSP** as its stack pointer. Since the **IPSR** is part of the **PSR** that is popped, it is automatically reset its previous state.

A **nested interrupt** occurs when a higher priority interrupt suspends an ISR. The lower priority interrupt will finish after the higher priority ISR completes. When one interrupt preempts another, the **LR** is set to 0xFFFFFFF1, so it knows to return to handler mode. **Tail chaining** occurs when one ISR executes immediately after another. Optimization occurs because the eight registers need not be popped only to be pushed once again. If an interrupt is triggered and is in the process of stacking registers when a higher priority interrupt is requested, this **late arrival interrupt** will be executed first.

Priority determines the order of service when two or more requests are made simultaneously. Priority also allows a higher priority request to suspend a lower priority request currently being processed. Usually, if two requests have the same priority, we do not allow them to interrupt each other. NVIC assigns a priority level to each interrupt trigger. This mechanism allows a higher priority trigger to interrupt the ISR of a lower priority request. Conversely, if a lower priority request occurs while running an ISR of a higher priority trigger, it will be postponed until the higher priority service is complete.
Observation: There are many interrupt sources, but an effective system will use only a few.

Program 12.3 gives the definitions in **startup.s** that allow the software to enable and disable interrupts. These functions are callable from either assembly or C code. The wait for interrupt can be used to place the processor in low-power sleep mode while it waits for an interrupt.

```
;*********** DisableInterrupts ***************
; disable interrupts
; inputs:  none
; outputs: none
DisableInterrupts  CPSID  I     ;set I=1
                   BX     LR

;*********** EnableInterrupts ***************
```

```
; enable interrupts
; inputs:  none
; outputs: none
EnableInterrupts   CPSIE  I    ;set I=0
                   BX     LR
;*********** WaitForInterrupt ************************
; go to low power mode while waiting for the next interrupt
; inputs:  none
; outputs: none
WaitForInterrupt
        WFI
        BX     LR
```

Program 12.3. Assembly functions needed for interrupt enabling and disabling.

12.4. Edge-triggered Interrupts

Synchronizing software to hardware events requires the software to recognize when the hardware changes states from busy to done. Many times the busy to done state transition is signified by a rising (or falling) edge on a status signal in the hardware. For these situations, we connect this status signal to an input of the microcontroller, and we use edge-triggered interfacing to configure the interface to set a flag on the rising (or falling) edge of the input. Using edge-triggered interfacing allows the software to respond quickly to changes in the external world. If we are using busy-wait synchronization, the software waits for the flag. If we are using interrupt synchronization, we configure the flag to request an interrupt when set. Each of the digital I/O pins on the TM4C family can be configured for edge triggering. Table 12.4 shows the registers needed to set up edge triggering for Port A. The differences between members of the TM4C family include the number of ports (e.g., the TM4C123 has ports A – F) and the number of pins in each port (e.g., the TM4C123 only has pins 4 – 0 in Port F). For more details, refer to the datasheet for your specific microcontroller. Any or all of digital I/O pins can be configured as an edge-triggered input. When writing C code using these registers, include the header file for your particular microcontroller (e.g., **tm4c123ge6pm.h**). To use any of the features for a digital I/O port, we first enable its clock in the Run Mode Clock Gating Control Register 2 (RCGC2). For each bit we wish to use we must set the corresponding **DEN** (Digital Enable) bit. To use edge triggered interrupts we will clear the corresponding bits in the **PCTL** register, and we will clear bits in the **AFSEL** (Alternate Function Select) register. We clear **DIR** (Direction) bits to make them input. On the TM4C123, only pins PD7 and PF0 need to be unlocked. We clear bits in the **AMSEL** register to disable analog function.

Address	7	6	5	4	3	2	1	0	Name
$4000.43FC	DATA	DATA	DATA	DATA	DATA	DATA	DATA	DATA	GPIO_PORTA_DATA_R
$4000.4400	DIR	DIR	DIR	DIR	DIR	DIR	DIR	DIR	GPIO_PORTA_DIR_R
$4000.4404	IS	IS	IS	IS	IS	IS	IS	IS	GPIO_PORTA_IS_R
$4000.4408	IBE	IBE	IBE	IBE	IBE	IBE	IBE	IBE	GPIO_PORTA_IBE_R
$4000.440C	IEV	IEV	IEV	IEV	IEV	IEV	IEV	IEV	GPIO_PORTA_IEV_R
$4000.4410	IME	IME	IME	IME	IME	IME	IME	IME	GPIO_PORTA_IM_R
$4000.4414	RIS	RIS	RIS	RIS	RIS	RIS	RIS	RIS	GPIO_PORTA_RIS_R
$4000.4418	MIS	MIS	MIS	MIS	MIS	MIS	MIS	MIS	GPIO_PORTA_MIS_R
$4000.441C	ICR	ICR	ICR	ICR	ICR	ICR	ICR	ICR	GPIO_PORTA_ICR_R
$4000.4420	SEL	SEL	SEL	SEL	SEL	SEL	SEL	SEL	GPIO_PORTA_AFSEL_R
$4000.4500	DRV2	DRV2	DRV2	DRV2	DRV2	DRV2	DRV2	DRV2	GPIO_PORTA_DR2R_R
$4000.4504	DRV4	DRV4	DRV4	DRV4	DRV4	DRV4	DRV4	DRV4	GPIO_PORTA_DR4R_R
$4000.4508	DRV8	DRV8	DRV8	DRV8	DRV8	DRV8	DRV8	DRV8	GPIO_PORTA_DR8R_R
$4000.450C	ODE	ODE	ODE	ODE	ODE	ODE	ODE	ODE	GPIO_PORTA_ODR_R
$4000.4510	PUE	PUE	PUE	PUE	PUE	PUE	PUE	PUE	GPIO_PORTA_PUR_R
$4000.4514	PDE	PDE	PDE	PDE	PDE	PDE	PDE	PDE	GPIO_PORTA_PDR_R
$4000.4518	SLR	SLR	SLR	SLR	SLR	SLR	SLR	SLR	GPIO_PORTA_SLR_R
$4000.451C	DEN	DEN	DEN	DEN	DEN	DEN	DEN	DEN	GPIO_PORTA_DEN_R
$4000.4524	CR	CR	CR	CR	CR	CR	CR	CR	GPIO_PORTA_CR_R
$4000.4528	AMSEL	AMSEL	AMSEL	AMSEL	AMSEL	AMSEL	AMSEL	AMSEL	GPIO_PORTA_AMSEL_R

	31-28	27-24	23-20	19-16	15-12	11-8	7-4	3-0	
$4000.452C	PMC7	PMC6	PMC5	PMC4	PMC3	PMC2	PMC1	PMC0	GPIO_PORTA_PCTL_R
$4000.4520	LOCK (32 bits)								GPIO_PORTA_LOCK_R

Table 12.4. Some TM4C port A registers. We will clear PMC bits to used edge triggered interrupts.

To configure an edge-triggered pin, we first enable the clock on the port and configure the pin as a regular digital input. Clearing the **IS** (Interrupt Sense) bit configures the bit for edge triggering. If the **IS** bit were to be set, the trigger occurs on the level of the pin. Since most busy to done conditions are signified by edges, we typically trigger on edges rather than levels. Next we write to the **IBE** (Interrupt Both Edges) and **IEV** (Interrupt Event) bits to define the active edge. We can trigger on the rising, falling, or both edges, as listed in Table 12.5.

The hardware sets an **RIS** (Raw Interrupt Status) bit (called the trigger) and the software clears it (called the acknowledgement). The triggering event listed in Table 12.5 will set the corresponding **RIS** bit in the **`GPIO_PORTA_RIS_R`** register regardless of whether or not that bit is allowed to request an interrupt. In other words, clearing an **IM** bit disables the corresponding pin's interrupt, but it will still set the corresponding **RIS** bit when the interrupt would have occurred. The software can acknowledge the event by writing ones to the corresponding **IC** (Interrupt Clear) bit in the **`GPIO_PORTA_IC_R`** register. The **RIS** bits are read only, meaning if the software were to write to this register, it would have no effect. For example, to clear bits 2, 1, and 0 in the **`GPIO_PORTA_RIS_R`** register, we write a 0x07 to the **`GPIO_PORTA_IC_R`** register. Writing zeros into **IC** bits will not affect the **RIS** bits.

DIR	AFSEL	PMC	IS	IBE	IEV	IME	Port mode
0	0	0000	0	0	0	0	Input, falling edge trigger, busy wait
0	0	0000	0	0	1	0	Input, rising edge trigger, busy wait
0	0	0000	0	1	-	0	Input, both edges trigger, busy wait
0	0	0000	0	0	0	1	Input, falling edge trigger, interrupt
0	0	0000	0	0	1	1	Input, rising edge trigger, interrupt
0	0	0000	0	1	-	1	Input, both edges trigger, interrupt

Table 12.5. Edge-triggered modes.

For input signals we have the option of adding either a pull-up resistor or a pull-down resistor. If we set the corresponding **PUE** (Pull-Up Enable) bit on an input pin, the equivalent of a 13 kΩ to 30 kΩ resistor to +3.3 V power is internally connected to the pin. Similarly, if we set the corresponding **PDE** (Pull-Down Enable) bit on an input pin, the equivalent of a 13 kΩ to 35 kΩ resistor to ground is internally connected to the pin. We cannot have both pull-up and a pull-down resistor, so setting a bit in one register automatically clears the corresponding bit in the other register.

A typical application of pull-up and pull-down mode is the interface of simple switches. Using these modes eliminates the need for an external resistor when interfacing a switch. Compare the interfaces on Port A to the interfaces on Port B illustrated in Figure 12.4. The PA2 and PA3 interfaces will use software-configured internal resistors, while the PB2 and PB3 interfaces use actual resistors. The PA2 and PB2 interfaces in Figure 12.4a) implement negative logic switch inputs, and the PA3 and PB3 interfaces in Figure 12.4b) implement positive logic switch inputs.

Checkpoint 12.5: What do negative logic and positive logic mean in this context?

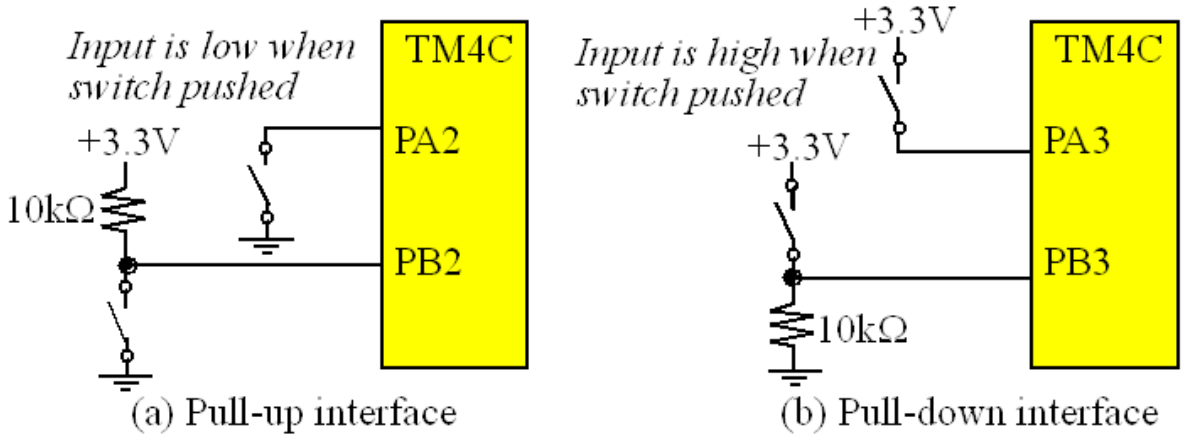


Figure 12.4. Edge-triggered interfaces can generate interrupts on a switch touch.

Using edge triggering to synchronize software to hardware centers around the operation of the trigger flags, **RIS**. A busy-wait interface will read the appropriate **RIS** bit over and over, until it is set. When the **RIS** bit is set, the software will clear the **RIS** bit (by writing a one to the corresponding **IC** bit) and perform the desired function. With interrupt synchronization, the initialization phase will arm the trigger flag by setting the corresponding **IM** bit. In this way, the active edge of the pin will set the **RIS** and request an interrupt. The interrupt will suspend the main program and run a special interrupt service routine (ISR). This ISR will clear the **RIS** bit and perform the desired function. At the end of the ISR it will return, causing the main program to resume. In particular, five conditions must be simultaneously true for an edge-triggered interrupt to be requested:

- The trigger flag bit is set (RIS)
- The arm bit is set (IME)
- The level of the edge-triggered interrupt must be less than BASEPRI
- The edge-triggered interrupt must be enabled in the NVIC_EN0_R
- The I bit, bit 0 of the special register PRIMASK, is 0

Checkpoint 12.6: What values to you write into DIR, AFSEL, PUE, and PDE to configure the switch interfaces of PA2 and PA3 in Figure 12.4?

Table 12.4 listed the registers for Port A. The other ports have similar registers. We will begin with a simple example that counts the number of rising edges on Port F bit 4 (Program 12.4). The initialization requires many steps. (a) The clock for the port must be enabled. (b) The global variables should be initialized. (c) The appropriate pins must be enabled as inputs. (d) We must specify whether to trigger on the rise, the fall, or both edges. In this case we will trigger on the rise of PF4. (e) It is good design to clear the trigger flag during initialization so that the first interrupt occurs due to the first rising edge after the initialization has been run. We do not wish to count a rising edge that might have occurred during the power up phase of the system. (f) We arm the edge-trigger by setting the corresponding bits in the **IM** register. (g) We establish the priority of Port F by setting bits 23 – 21 in the **`NVIC_PRI7_R`** register as listed in Table 9.2. We activate Port F interrupts in the NVIC by setting bit 30 in the **`NVIC_EN0_R`** register, Table 12.3. There is no need to unlock PF4.

```
volatile unsigned long FallingEdges = 0;
void EdgeCounter_Init(void){
  SYSCTL_RCGC2_R |= 0x00000020; // (a) activate clock for port F
  FallingEdges = 0;             // (b) initialize count and wait for clock
  GPIO_PORTF_DIR_R &= ~0x10;    // (c) make PF4 in (built-in button)
  GPIO_PORTF_AFSEL_R &= ~0x10;  //     disable alt funct on PF4
  GPIO_PORTF_DEN_R |= 0x10;     //     enable digital I/O on PF4
  GPIO_PORTF_PCTL_R &= ~0x000F0000; //  configure PF4 as GPIO
```

```
  GPIO_PORTF_AMSEL_R& = ~0x10;  //    disable analog functionality on PF4
  GPIO_PORTF_PUR_R |= 0x10;     //     enable weak pull-up on PF4
  GPIO_PORTF_IS_R &= ~0x10;     // (d) PF4 is edge-sensitive
  GPIO_PORTF_IBE_R &= ~0x10;    //     PF4 is not both edges
  GPIO_PORTF_IEV_R &= ~0x10;    //     PF4 falling edge event
  GPIO_PORTF_ICR_R = 0x10;      // (e) clear flag4
  GPIO_PORTF_IM_R |= 0x10;      // (f) arm interrupt on PF4
  NVIC_PRI7_R = (NVIC_PRI7_R&0xFF00FFFF)|0x00A00000; // (g) priority 5
  NVIC_EN0_R = 0x40000000;      // (h) enable interrupt 30 in NVIC
  EnableInterrupts();           // (i) Enable global Interrupt flag (I)
}
void GPIOPortF_Handler(void){
  GPIO_PORTF_ICR_R = 0x10;      // acknowledge flag4
  FallingEdges = FallingEdges + 1;
}
int main(void){
   EdgeCounter_Init(); // initialize GPIO Port F interrupt
   while(1){
      WaitForInterrupt();
   }
}
```

Program 12.4. Interrupt-driven edge-triggered input that counts rising edges of PF4 (C12_EdgeInterrupt).

This initialization is shown to enable interrupts in step (i). However, in most systems we would not enable interrupts in the device initialization. Rather, it is good design to initialize all devices in the system, then enable interrupts. All ISRs must acknowledge the interrupt by clearing the trigger flag that requested the interrupt. For edge-triggered PF4, the trigger flag is bit 4 of the **GPIO_PORTF_RIS_R** register. This flag can be cleared by writing a 0x10 to **GPIO_PORTF_ICR_R**.

If two or more triggers share the same vector, these requests are called **polled interrupts**, and the ISR must determine which trigger generated the interrupt. If the requests have separate vectors, then these requests are called **vectored interrupts** and the ISR knows which trigger caused the interrupt.

One of the problems with switches is called **switch bounce**. Many inexpensive switches will mechanically oscillate for up to a few milliseconds when touched or released. It behaves like an underdamped oscillator. These mechanical oscillations cause electrical oscillations such that a port pin will oscillate high/low during the bounce. In some cases this bounce should be removed. To remove switch bounce we can ignore changes in a switch that occur within 10 ms of each other. In other words, recognize a switch transition, disarm interrupts for 10ms, and then rearm after 10 ms. Alternatively, we could record the time of the switch transition. If the time between this transition and the previous transition is less than 10ms, ignore it. If the time is more than 10 ms, then accept and process the input as a real event.

12.5. SysTick Periodic Interrupts

One application of periodic interrupts is called “intermittent polling” or “periodic polling”. Figure 12.5 shows busy wait side by side with periodic polling. In busy-wait synchronization, the main program polls the I/O devices continuously. With periodic polling, the I/O devices are polled on a regular basis (established by the periodic interrupt.) If no device needs service, then the interrupt simply returns.

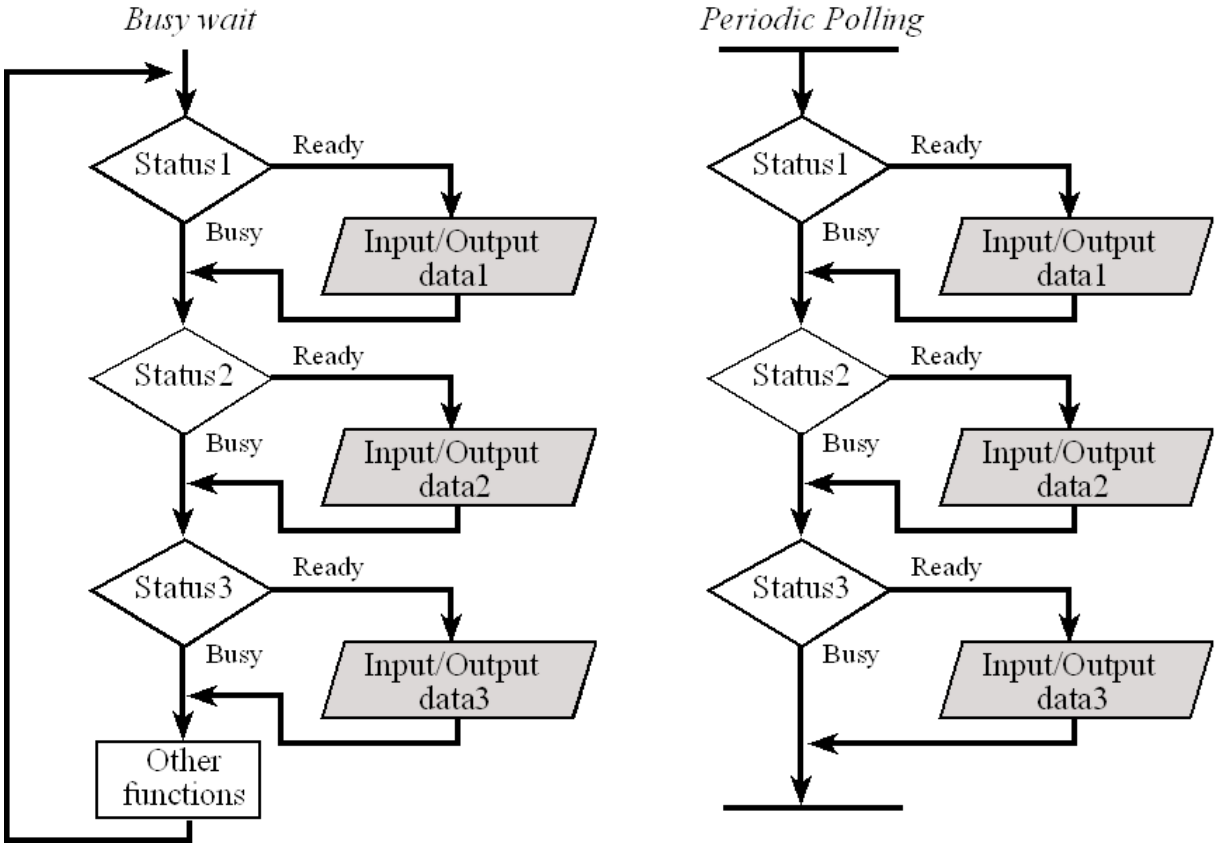


Figure 12.5. On the left is busy-wait, and on the right is periodic polling.

If the polling period is Δt, then on average the interface latency will be ½Δt, and the worst case latency will be Δt. Periodic polling is appropriate for low bandwidth devices where real-time response is not necessary. This method frees the main program to perform other functions. We use periodic polling if the following two conditions apply:

1. The I/O hardware cannot generate interrupts directly
2. We wish to perform the I/O functions in the background

For a data acquisition system, it is important to establish an accurate sampling rate. The time in between ADC samples must be equal and known in order for the digital signal processing to function properly. Similarly for microcontroller-based control systems, it is important to maintain a periodic rate for reading data from the sensors and outputing commands to the actuators.

The SysTick timer is a simple way to create periodic interrupts. A periodic interrupt is one that is requested on a fixed time basis. This interfacing technique is required for data acquisition and control systems, because software servicing must be performed at accurate time intervals.

Table 12.6 shows the SysTick registers used to create a periodic interrupt. SysTick has a 24-bit counter that decrements at the bus clock frequency. Let f_{BUS} be the frequency of the bus clock, and let n be the value of the **RELOAD** register. The frequency of the periodic interrupt will be $f_{BUS}/(n+1)$. First, we clear the **ENABLE** bit to turn off SysTick during initialization. Second, we set the **RELOAD** register. Third, we write any value to **`NVIC_ST_CURRENT_R`** to clear the counter. Lastly, we write the desired mode to the control register, **`NVIC_ST_CTRL_R`**. We must set **CLK_SRC**=1, because **CLK_SRC**=0 external clock mode is not implemented on the LM3S/TM4C family. We set **INTEN** to enable interrupts. We establish the priority of the SysTick interrupts using the TICK field in the **`NVIC_SYS_PRI3_R`** register. We need to set the **ENABLE** bit so the counter will run. When the **CURRENT** value counts down from 1 to 0, the **COUNT** flag is set. On the next clock, the **CURRENT**

is loaded with the **RELOAD** value. In this way, the SysTick counter (**CURRENT**) is continuously decrementing. If the **RELOAD** value is *n*, then the SysTick counter operates at modulo *n*+1 (…*n*, *n*-1, *n*-2 … 1, 0, *n*, *n*-1, …). In other words, it rolls over every *n*+1 counts. Thus, the **COUNT** flag will be set every *n*+1 counts. Program 12.5 shows a simple example of SysTick. SysTick is the only interrupt on the TM4C that has an automatic acknowledge. Notice there is no explicit software step in the ISR to clear the **COUNT** flag.

Address	31-24	23-17	16	15-3	2	1	0	Name
$E000E010	0	0	COUNT	0	CLK_SRC	INTEN	ENABLE	NVIC_ST_CTRL_R
$E000E014	0	24-bit RELOAD value						NVIC_ST_RELOAD_R
$E000E018	0	24-bit CURRENT value of SysTick counter						NVIC_ST_CURRENT_R

Address	31-29	28-24	23-21	20-8	7-5	4-0	Name
$E000ED20	TICK	0	PENDSV	0	DEBUG	0	NVIC_SYS_PRI3_R

Table 12.6. SysTick registers.

```
volatile unsigned long Counts=0;
void SysTick_Init(unsigned long period){
  NVIC_ST_CTRL_R = 0;          // disable SysTick during setup
  NVIC_ST_RELOAD_R = period-1;// reload value
  NVIC_ST_CURRENT_R = 0;       // any write to current clears it
  NVIC_SYS_PRI3_R = (NVIC_SYS_PRI3_R&0x00FFFFFF)|0x40000000; // priority
2
  NVIC_ST_CTRL_R = 0x07; // enable SysTick with core clock and interrupts
  // enable interrupts after all initialization is finished
}
void SysTick_Handler(void){
  GPIO_PORTF_DATA_R ^= 0x04;       // toggle PF2
  Counts = Counts + 1;
}
int main(void){ // running at 16 MHz
  SYSCTL_RCGC2_R |= 0x00000020; // activate port F
  Counts = 0;
  GPIO_PORTF_DIR_R |= 0x04;   // make PF2 output (PF2 built-in LED)
  GPIO_PORTF_AFSEL_R &= ~0x04;// disable alt funct on PF2
  GPIO_PORTF_DEN_R |= 0x04;   // enable digital I/O on PF2
  GPIO_PORTF_PCTL_R = (GPIO_PORTF_PCTL_R&0xFFFFF0FF)+0x00000000;
  GPIO_PORTF_AMSEL_R = 0;     // disable analog functionality on PF
  SysTick_Init(16000);        // initialize SysTick timer, every 1ms
  EnableInterrupts();      // enable after everything initialized
  while(1){                   // interrupts every 1ms, 500 Hz flash
    WaitForInterrupt();
  }
}
```

Program 12.5 Implementation of a periodic interrupt using SysTick (C12_PeriodicSysTickInts).

Example 12.1. Design an interface 32 Ω speaker and use it to generate a soft 1 kHz sound.

Solution: To make sound we need to create an oscillating wave. In this example, the wave will be a simple square wave. At 3.3V, a 32 Ω speaker will require a current of about 100 mA. The maximum the TM4C123 can produce on an output pin is 8 mA. If we place a resistor in series with the head phones, then the current will only be 3.3V/(1500+32Ω) = 2.2mA. To generate the 1 kHz sound we need a 1 kHz square wave. There are many good methods to generate square waves. In this example we will implement one of the simplest methods: period interrupt and toggle an output pin in the ISR. To generate a 1 kHz wave we will toggle the PA5 pin every 500 µs. We will assume the PLL is active and

the system is running at 80 MHz. We wish to initialize the SysTick to interrupt with a period of 500 µs. The correct value for reload is 39999 ((500µs/12.5ns)-1). If the bus frequency were to be 16 MHz, we would set the reload value to be 7999 ((500µs/62.5ns)-1). Since this sound wave output is a real time signal, we set its priority to highest level, which is 0. See Program 12.6.

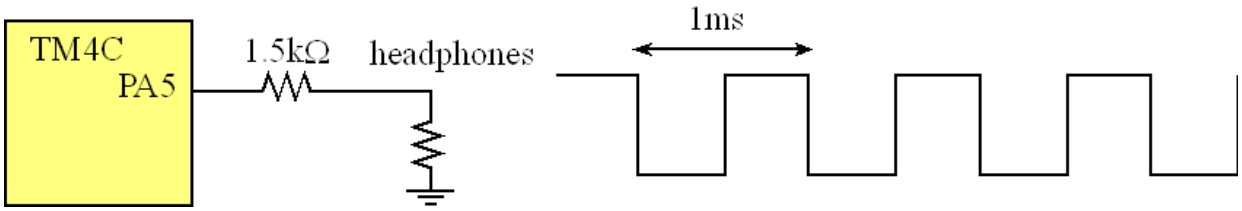


Figure 12.6. A squarewave output connected to a speaker will generate sound.

```
void Sound_Init(void){ unsigned long volatile delay;
  SYSCTL_RCGC2_R |= 0x00000001; // activate port A
  delay = SYSCTL_RCGC2_R;
  GPIO_PORTA_AMSEL_R &= ~0x20;      // no analog
  GPIO_PORTA_PCTL_R &= ~0x00F00000; // regular function
  GPIO_PORTA_DIR_R |= 0x20;     // make PA5 out
  GPIO_PORTA_DR8R_R |= 0x20;    // can drive up to 8mA out
  GPIO_PORTA_AFSEL_R &= ~0x20;  // disable alt funct on PA5
  GPIO_PORTA_DEN_R |= 0x20;     // enable digital I/O on PA5
  NVIC_ST_CTRL_R = 0;           // disable SysTick during setup
   NVIC_ST_RELOAD_R  =  39999;      //  reload  value  for  500us  (assuming
80MHz)
  NVIC_ST_CURRENT_R = 0;        // any write to current clears it
     NVIC_SYS_PRI3_R    =    NVIC_SYS_PRI3_R&0x00FFFFFF;    //    priority
0
  NVIC_ST_CTRL_R = 0x00000007;  // enable with core clock and interrupts
  EnableInterrupts();
}
void SysTick_Handler(void){
  GPIO_PORTA_DATA_R ^= 0x20;     // toggle PA5
}
```

Program 12.6. Sound output using a periodic interrupt (C12_SoftSound).

Observation: To make a quieter sound, we could use a larger resistor between the PA5 output and the speaker.

12.6. DC Motor Interface with PWM

The DC motor has a **frame** that remains motionless (called the **stator**), and an **armature** that moves (called the **rotor**). A **brushed DC motor** has an electromagnetic coil as well, located on the rotor, and the rotor is positioned inside the stator. In Figure 12.7, North and South refer to a permanent magnet, generating a constant *B* field from left to right. In this case, the rotor moves in a circular manner. When current flows through the coil, a magnetic force is created causing a rotation of the shaft. A brushed DC motor uses commutators to flip the direction of the current in the coil. In this way, the coil on the right always has an up force, and the one on the left always has a down force. Hence, a constant current generates a continuous rotation of the shaft. When the current is removed, the magnetic force stops, and the shaft is free to rotate. In a pulse-width modulated DC motor, the computer activates the coil with a current of fixed magnitude but varies the duty cycle in order to adjust the power delivered to the motor.

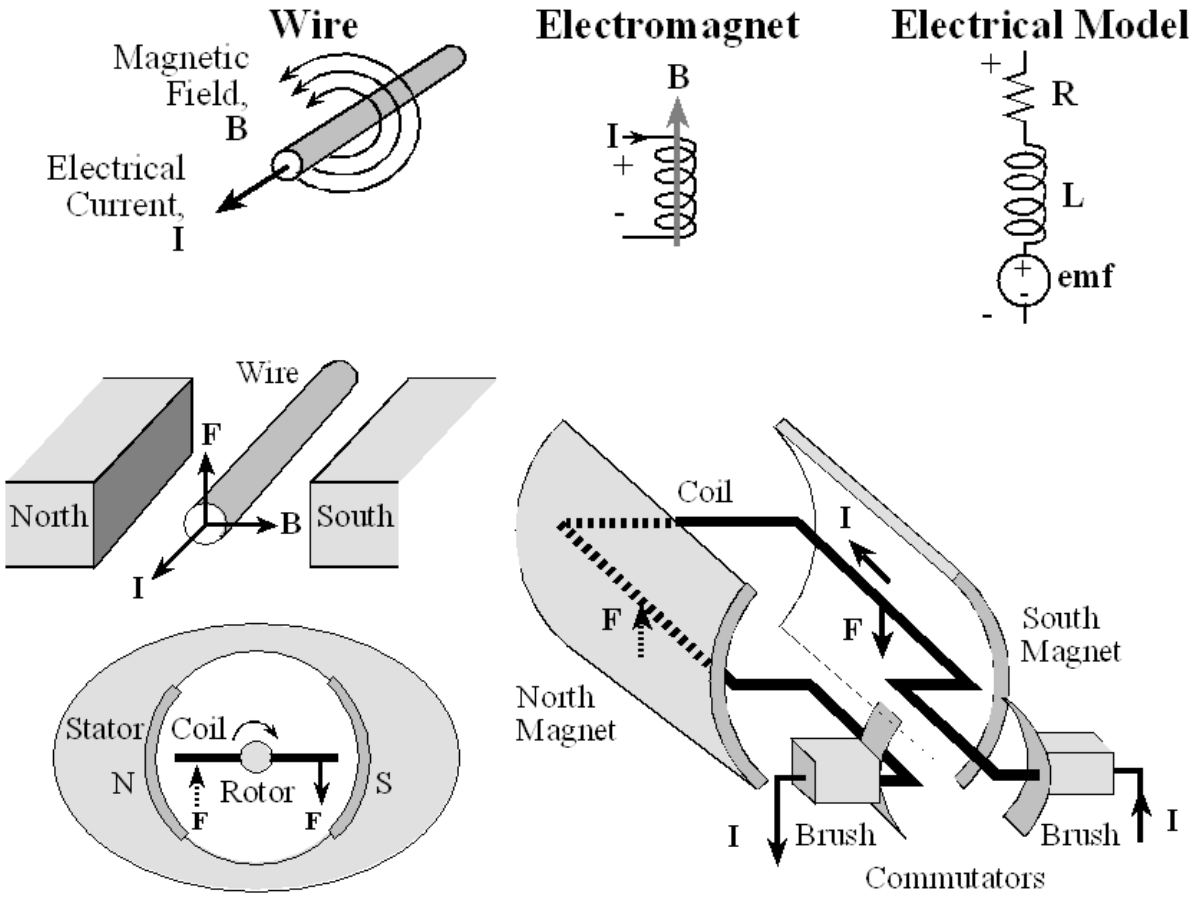


Figure 12.7. A brushed DC motor uses a commutator to flip the coil current.

In the LED interface the microcontroller was able to control electrical power to the LED in a binary fashion: either all on or all off. Sometimes it is desirable for the microcontroller to be able to vary the delivered power in a variable manner. One effective way to do this is to use pulse width modulation (PWM). The basic idea of PWM is to create a digital output wave of fixed frequency, but allow the microcontroller to vary its duty cycle. The system is designed in such a way that **High+Low** is constant (meaning the frequency is fixed). The **duty cycle** is defined as the fraction of time the signal is high:

$$duty\ cycle = \frac{High}{High + Low}$$

Hence, duty cycle varies from 0 to 1. We interface this digital output wave to an external actuator (like a DC motor), such that power is applied to the motor when the signal is high, and no power is applied when the signal is low. We purposely select a frequency high enough so the DC motor does not start/stop with each individual pulse, but rather responds to the overall average value of the wave. The average value of a PWM signal is linearly related to its duty cycle and is independent of its frequency. Let *P* (*P=V*I*) be the power to the DC motor, shown in Figure 12.8, when the PA5 signal is high. Under conditions of constant speed and constant load, the delivered power to the motor is linearly related to duty cycle.

$$Delivered\ Power = duty\ cycle * P = \frac{High}{High + Low} * P$$

Unfortunately, as speed and torque vary, the developed emf will affect delivered power. Nevertheless, PWM is a very effective mechanism, allowing the microcontroller to adjust delivered power.

The resistance in the coil (R) comes from the long wire that goes from the + terminal to the – terminal of the motor, see Figure 12.8. The inductance in the coil (L) arises from the fact that the wire is wound into coils to create the electromagnetics. The coil itself can generate its own voltage (emf) because of the interaction between the electric and magnetic fields. If the coil is a DC motor, then the emf is a function of both the speed of the motor and the developed torque (which in turn is a function of the applied load on the motor.) Because of the internal emf of the coil, the current will depend on the mechanical load. For example, a DC motor running with no load might draw 100 mA, but under load (friction) the current may jump to 1 A.

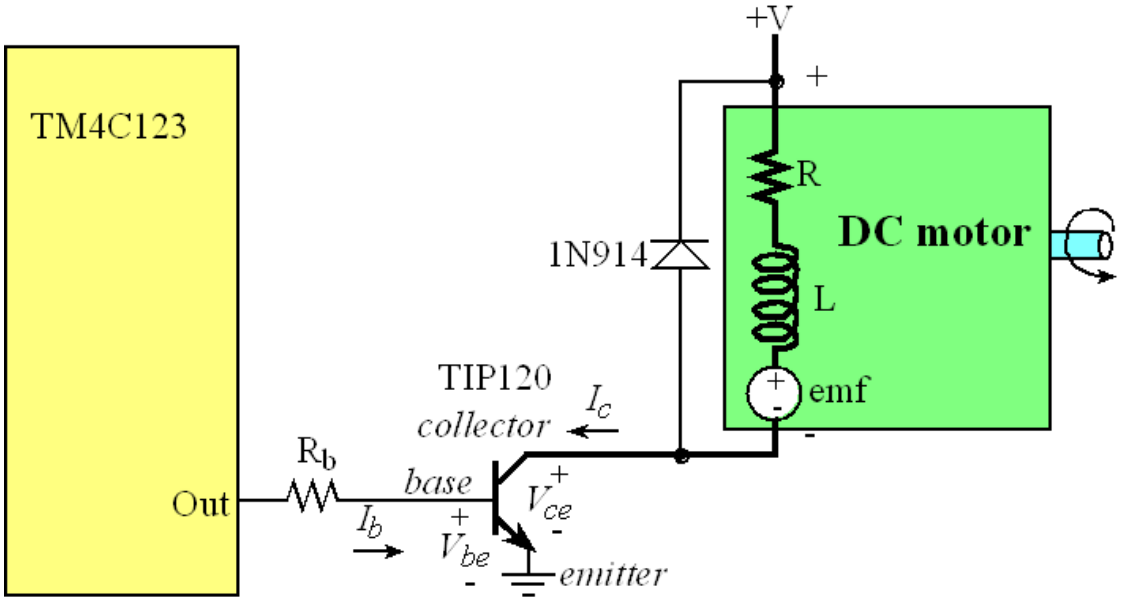


Figure 12.8. The system uses a transistor to turn the motor on and off.

There are lots of motor driver chips, but we will use an NPN Darlington transistor, e.g., TIP120, to allow the software to turn a motor on and off. If the port output is low, no current can flow into the base, so the TIP120 transistor is off, and the collector current, I_C, will be zero. If the port output is high, current does flow into the base and V_{BE} goes above V_{BEsat} turning on the TIP120 transistor. The transistor is in the linear range if $V_{BE} \leq V_{BEsat}$ and $I_c = h_{fe} \cdot I_b$. The transistor is in the saturated mode if $V_{BE} \geq V_{BEsat}$, V_{CE} = 0.7V and $I_c < h_{fe} \cdot I_b$. We select the value of the R_b resistor to operate the transistor in its saturated mode. We start with the desired motor current, which will be the collector current I_c. The voltage across the coil will be the $V - V_{CE}$. Next, we calculate the needed base current (I_b) given the current gain of the NPN

$$I_b = I_c / h_{fe}$$

knowing the current gain of the NPN (h_{fe}), see Table 12.7. Finally, given the output high voltage of the microcontroller (V_{OH} is about 3.3 V) and base-emitter voltage of the NPN (V_{BEsat}) needed to activate the transistor, we can calculate the desired interface resistor.

$$R_b \leq (V_{OH} - V_{BEsat}) / I_b = h_{fe} * (V_{OH} - V_{BEsat}) / I_c$$

The inequality means we can choose a smaller resistor, creating a larger I_b. Because the h_{fe} of the transistors can vary a lot, it is a good design practice to make the R_b resistor 0.1 to 0.5 times the value shown in the above equation. Since the transistor is saturated, the increased base current produces the same V_{CE} and thus the same coil current.

Parameter	TIP120 (at I_{CE}=200mA)
h_{fe}	900
V_{BEsat}	1.3 V
V_{CE}	0.7 V

I_{CE}	Up to 5A

Table 12.7. Design parameters for the TIP120.

Example 12.2. Design an interface for an 8-V 200-mA geared DC motor. SW1 will make the motor spin faster and SW2 will make it spin slower.

Solution: We will use the TIP120 circuit as shown in Figure 12.8 because the TIP120 can sink many times the current needed for this motor. We select a +8.4V battery supply and connect it to the positive side of the motor. Just like the stepper motor, when designing with motors we will not use the 3.3V or +5V from the LaunchPad. Although the current is only 200 mA when spinning unloaded, it will increase to 1 A if the motor experiences mechanical friction. The needed base current is

$$I_b = I_{coil} / h_{fe} = 200\text{mA}/900 = 0.22\text{mA}$$

The desired interface resistor.

$$R_b \le (V_{OH} - V_{be})/ I_b = (3.3\text{-}1.3)/0.22\text{mA} = 9\ \text{k}\Omega$$

To cover the variability in h_{fe}, we will use a 1 kΩ resistor instead of the 9 kΩ. The actual voltage across the motor when active will be +8.4-0.7 = 7.7V. Motors and transistors vary a lot, so it is appropriate to experimentally verify the design by measuring the voltages and currents.

Program 12.7 is used to control the motor. Interrupts are triggered on the falling edges of PF4 and PF0. The period of the PWM output is chosen to be about 10 times shorter than the time constant of the motor. The electronic driver will turn on and off every 1ms, but the motor only responds to the average level. The software sets the duty cycle of the PWM to adjust the delivered power. When active, the interface will drive +7.65 V across the motor. The actual current will be a function of the friction applied to the shaft. The software maintains **`H+L`** to be 80,000 (Figure 12.9); this will set the period on PA5 to be a fixed value of 1ms. The duty cycle, **`L/(H+L)`**, will specify the amount of electrical power delivered to the motor.

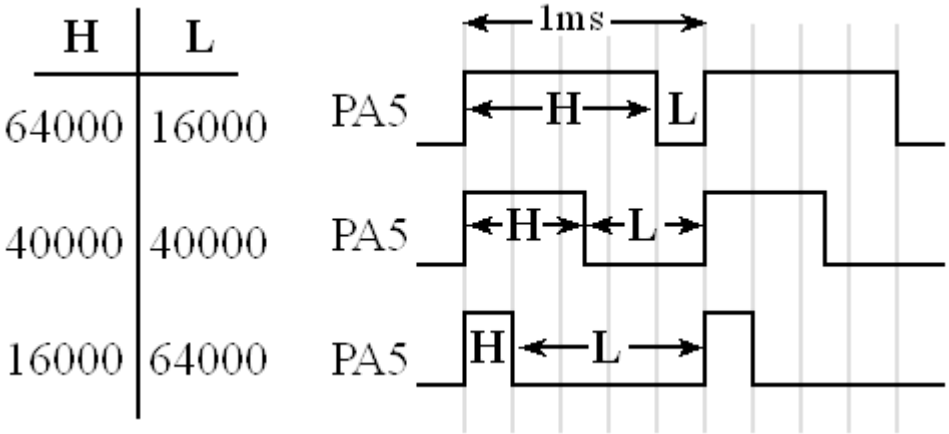


Figure 12.9. The duty cycle of the PWM output sets the power to the motor.

The interface for one motor is shown in Figure 12.10. If the robot has two motors then a second copy of the TIP120 circuit will be needed. The Port F edge-triggered interrupt will change the duty cycle by ±10% depending on whether the operator touches SW1 or SW2. Since the two ISRs pass data (**`H`**,**`L`** passed from **`GPIOPortF_Handler`** to **`SysTick_Handler`**), we will set the interrupt priorities to be equal. With equal priorities neither ISR will interrupt the other. This way, the global variables **`H`**,**`L`** will always be in a consistent state.

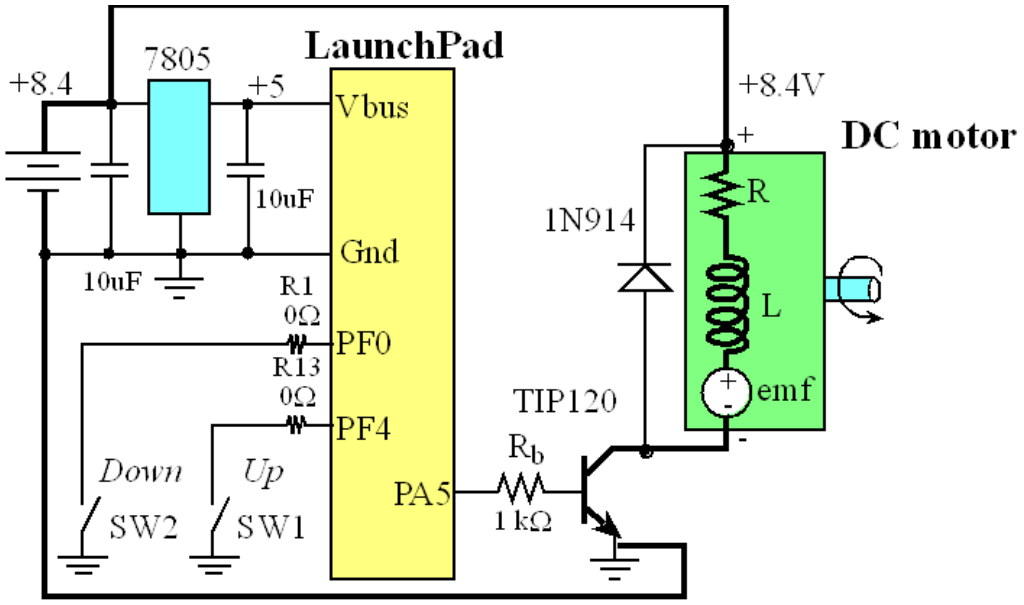


Figure 12.10. DC motor interface (bold lines show path of the large current).

```
unsigned long H,L;
void Motor_Init(void){
  SYSCTL_RCGC2_R |= 0x00000001; // activate clock for port A
  H = L = 8000;                 // 10%
  GPIO_PORTA_AMSEL_R &= ~0x20;      // disable analog functionality on PA5
  GPIO_PORTA_PCTL_R &= ~0x00F00000; // configure PA5 as GPIO
  GPIO_PORTA_DIR_R |= 0x20;     // make PA5 out
  GPIO_PORTA_DR8R_R |= 0x20;    // enable 8 mA drive on PA5
  GPIO_PORTA_AFSEL_R &= ~0x20;  // disable alt funct on PA5
  GPIO_PORTA_DEN_R |= 0x20;     // enable digital I/O on PA5
  GPIO_PORTA_DATA_R &= ~0x20;   // make PA5 low
  NVIC_ST_CTRL_R = 0;           // disable SysTick during setup
  NVIC_ST_RELOAD_R = L-1;       // reload value for 500us
  NVIC_ST_CURRENT_R = 0;        // any write to current clears it
  NVIC_SYS_PRI3_R = (NVIC_SYS_PRI3_R&0x00FFFFFF)|0x40000000; // priority 2
  NVIC_ST_CTRL_R = 0x00000007;  // enable with core clock and interrupts
}
void SysTick_Handler(void){
  if(GPIO_PORTA_DATA_R&0x20){   // toggle PA5
    GPIO_PORTA_DATA_R &= ~0x20; // make PA5 low
    NVIC_ST_RELOAD_R = L-1;     // reload value for low phase
  } else{
    GPIO_PORTA_DATA_R |= 0x20;  // make PA5 high
    NVIC_ST_RELOAD_R = H-1;     // reload value for high phase
  }
}
void Switch_Init(void){  unsigned long volatile delay;
  SYSCTL_RCGC2_R |= 0x00000020; // (a) activate clock for port F
  delay = SYSCTL_RCGC2_R;
  GPIO_PORTF_LOCK_R = 0x4C4F434B; // unlock GPIO Port F
  GPIO_PORTF_CR_R = 0x11;           // allow changes to PF4,0
  GPIO_PORTF_DIR_R &= ~0x11;    // (c) make PF4,0 in (built-in button)
  GPIO_PORTF_AFSEL_R &= ~0x11;  //     disable alt funct on PF4,0
  GPIO_PORTF_DEN_R |= 0x11;     //     enable digital I/O on PF4,0
  GPIO_PORTF_PCTL_R &= ~0x000F000F; //  configure PF4,0 as GPIO
  GPIO_PORTF_AMSEL_R& = ~0x11;  //     disable analog functionality on PF4,0
  GPIO_PORTF_PUR_R |= 0x11;     //     enable weak pull-up on PF4,0
  GPIO_PORTF_IS_R &= ~0x11;     // (d) PF4,PF0 is edge-sensitive
  GPIO_PORTF_IBE_R &= ~0x11;    //     PF4,PF0 is not both edges
  GPIO_PORTF_IEV_R &= ~0x11;    //     PF4,PF0 falling edge event
```

```
  GPIO_PORTF_ICR_R = 0x11;      // (e) clear flags 4,0
  GPIO_PORTF_IM_R |= 0x11;      // (f) arm interrupt on PF4,PF0
  NVIC_PRI7_R = (NVIC_PRI7_R&0xFF00FFFF)|0x00400000; // (g) priority 2
  NVIC_EN0_R = 0x40000000;      // (h) enable interrupt 30 in NVIC
}
// L range: 8000,16000,24000,32000,40000,48000,56000,64000,72000
// power:   10%    20%   30%   40%   50%   60%   70%   80%   90%
void GPIOPortF_Handler(void){ // called on touch of either SW1 or SW2
  if(GPIO_PORTF_RIS_R&0x01){  // SW2 touch
    GPIO_PORTF_ICR_R = 0x01;  // acknowledge flag0
    if(L>8000) L = L-8000;    // slow down
  }
  if(GPIO_PORTF_RIS_R&0x10){  // SW1 touch
    GPIO_PORTF_ICR_R = 0x10;  // acknowledge flag4
    if(L<72000) L = L+8000;   // speed up
  }
  H = 80000-L; // constant period of 1ms, variable duty cycle
}
int main(void){
  DisableInterrupts();  // disable interrupts while initializing
  PLL_Init();           // bus clock at 80 MHz
  Motor_Init();         // output from PA5, SysTick interrupts
  Switch_Init();        // arm PF4, PF0 for falling edge interrupts
  EnableInterrupts();   // enable after all initialization are done
  while(1){ // main program is free to perform other tasks
    WaitForInterrupt(); // low power mode
  }
}
```

Program 12.7. Motor output using a periodic interrupt. Switches are used to adjust the power (C12_DCMotor).

Chapter 13: Digital to Analog Conversion and Sound
Embedded Systems - Shape the World
Jonathan Valvano and Ramesh Yerraballi

As we have seen throughout this class, an embedded system uses its input/output devices to interact with the external world. Input devices allow the system to gather information about the world, and output devices can affect visual, mechanical, chemical, auditory, and biologic processes in the world. In this chapter we will literally "Shape The World". We present a technique for the system to generate an analog output using a digital to analog converter (DAC). Together with periodic interrupts the system can generate waveforms, which are analog voltages that vary in time and in amplitude. We will then connect the waveform to a speaker and generate sound.

Learning Objectives:

- Develop a means for a digital computer to interact with the analog world.
- Study digitization: Quantization, range, precision and resolution.
- Introduce sampling and the Nyquist Theorem.
- Study the basics of sound: electromagnets, speakers, AC vs. DC power, perception of sound.
- Understand how to create sound: loudness, pitch, envelope, and shape
- Use SysTick to create sounds by programming variable frequencies.

13.1. Approximating continuous signals in the digital domain

An **analog signal** is one that is continuous in both amplitude and time. Neglecting quantum physics, most signals in the world exist as continuous functions of time in an analog fashion (e.g., voltage, current, position, angle, speed, force, pressure, temperature, and flow etc.) In other words, the signal has amplitude that can vary over time, but the value cannot instantaneously change. To represent a signal in the digital domain we must approximate it in two ways: amplitude quantizing and time quantizing (or Sampling). From an amplitude perspective, we will first place limits on the signal restricting it to exist between a minimum and maximum value (e.g., 0 to +3V), and second, we will divide this amplitude range into a finite set of discrete values. The **range** of the system is the maximum minus the minimum value. The **precision** of the system defines the number of values from which the amplitude of the digital signal is selected. Precision can be given in number of alternatives, binary bits, or decimal digits. The **resolution** is the smallest change in value that is significant. Furthermore, with respect to time one considers analog signals to exist from time equals minus infinity to plus infinity. Because memory is finite, when representing signals on a digital computer, we will restrict signal to a **finite time**, or we could have a finite set of data that are repeated over and over.

Figure 13.1 shows a temperature waveform (solid line), with a corresponding digital representation sampled at 1 Hz and stored as a 5-bit integer number with a range of 0 to 31 ^{o}C. Because it is digitized in both amplitude and time, the digital samples (individual dots) in Figure 13.1 must exist at an intersection of grey lines. Because it is a time-varying signal (mathematically, this is called a function), we have one amplitude for each time, but it is possible for there to be 0, 1, or more times for each amplitude.

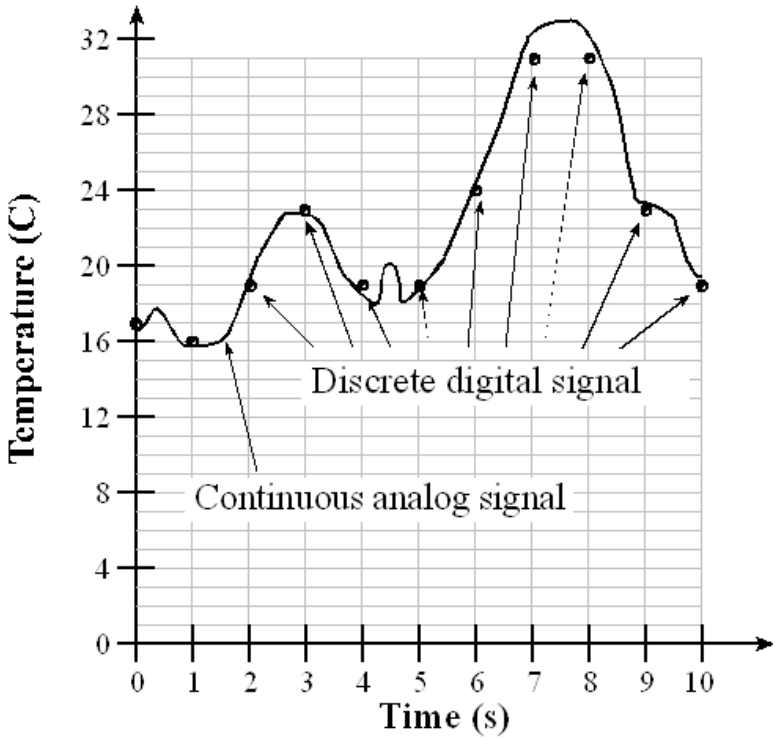


Figure 13.1. An analog signal is represented in the digital domain as discrete samples

The second approximation occurs in the time domain. Time quantizing is caused by the finite sampling interval. For example, the data are sampled every 1 second in Figure 13.1. In practice we will use a periodic timer to trigger an analog to digital converter (ADC) to digitize information, converting from the analog to the digital domain. Similarly, if we are converting from the digital to the analog domain, we use the periodic timer to output new data to a digital to analog converter (DAC). The **Nyquist Theorem** states that if the signal is sampled with a frequency of f_s, then the digital samples only contain frequency components from 0 to ½ f_s. Conversely, if the analog signal does contain frequency components larger than ½ f_s, then there will be an **aliasing** error during the sampling process (performed with a frequency of f_s). Aliasing is when the digital signal appears to have a different frequency than the original analog signal. Also note, the digital data has 11 values at times 0 to 10, but no information before time=0, and no information after time=10.

Checkpoint 13.1: Why can't the digital samples represent the little wiggles in the analog signal?
Checkpoint 13.2: Why can't the digital samples represent temperatures above 31 °C?
Checkpoint 13.3: What range of frequencies is represented in the digital samples when the ADC is sampled once every millisecond?
Checkpoint 13.4: If I wanted to create an analog output wave with frequency components from 0 to 11 kHz, what is the slowest rate at which I could output to the DAC?

Proving the Nyquist Theorem mathematically is beyond the scope of this course, but we can present a couple examples to support the basic idea of the Nyquist Theorem: we must sample at a rate faster than twice the rate at which signal itself is oscillating.

Example 1) There is a long distance race with runners circling around an oval track and it is your job to count the number of times a particular runner circles the track. Assume the fastest time a runner can make a lap is 60 seconds. You want to read a book and occasionally look up to see where your runner is. How often to you have to look at your runner to guarantee you properly count the laps? If you look at a period faster than every 30 seconds you will see the runner at least twice per lap and properly count the laps. If you look at a period slower than every 30 seconds, you may only see the runner once per lap and

not know if the runner is going very fast or very slow. In this case, the runner oscillates at most 1 lap per minute and thus you must observe the runner at a rate faster than twice per minute.

Example 2) You live on an island and want to take the boat back to the mainland as soon as possible. There is a boat that arrives at the island once a day, waits at the dock for 12 hours and then it sets sail to the mainland. Because of weather conditions, the exact time of arrival is unknown, but the boat will always wait at the dock for 12 hours before it leaves. How often do you need to walk down to the dock to see if the boat is there? If the boat is at the dock, you get on the boat and take the next trip back to the mainland. If you walk down to the dock every 13 hours, it is possible to miss the boat. However, if you walk down to the dock every 12 hours or less, you'll never miss the boat. In this case, the boat frequency is once/day and you must sample it (go to the dock) two times/day.

13.2. Digital to Analog Conversion

A DAC converts digital signals into analog form as illustrated in Figure 13.2. Although one can interface a DAC to a regular output port, most DACs are interfaced using high-speed synchronous protocols, like the SSI. For more information about SSI, see Embedded Systems: Real-Time Interfacing to ARM® Cortex™-M Microcontrollers, 2014, ISBN: 978-1463590154. The DAC output can be current or voltage. Additional analog processing may be required to filter, amplify, or modulate the signal. We can also use DACs to design variable gain or variable offset analog circuits.

The DAC **precision** is the number of distinguishable DAC outputs (e.g., 16 alternatives, 4 bits). The DAC **range** is the maximum and minimum DAC output (volts, amps). The DAC **resolution** is the smallest distinguishable change in output. The units of resolution are in volts or amps depending on whether the output is voltage or current. The **resolution** is the change in output that occurs when the digital input changes by 1. For most DACs there is a simple relationship between range precision and resolution.

Range(volts) = Precision(alternatives) • Resolution(volts)

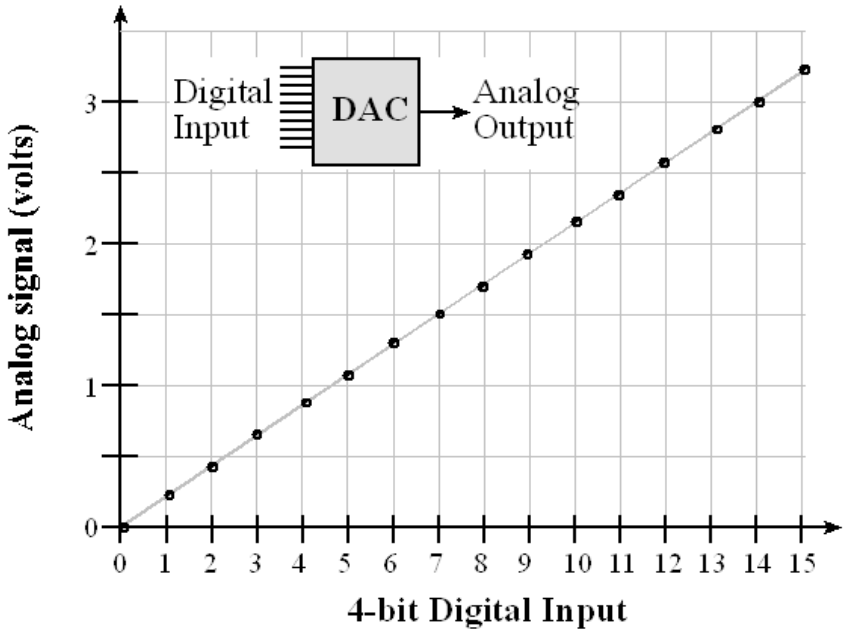


Figure 13.2. A 4-bit DAC provides analog output.

The DAC **accuracy** is (Actual - Ideal) / Ideal where Ideal is referred to the National Institute of Standards and Technology (NIST). One can choose the full scale **range** of the DAC to simplify the use of fixed-point math. For example, if an 8-bit DAC had a full scale range of 0 to 2.55 volts, then the

resolution would be exactly 10 mV. This means that if the DAC digital input were 123, then the DAC output voltage would be 1.23 volts.

Checkpoint 13.5: A 10-bit DAC has a range of 0 to 2.5V, what is the approximate resolution?
Checkpoint 13.6: You need a DAC with a range of 0 to 2V, and a resolution of 1 mV. What is the smallest number of bits could you use for the DAC?

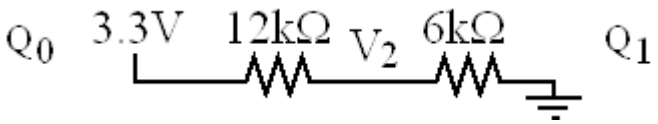
Example 13.1. Design a 2-bit binary-weighted DAC with a range of 0 to +3.3V using resistors.

Solution: We begin by specifying the desired input/output relationship of the 2-bit DAC. The design specifications are shown in Table 13.1.

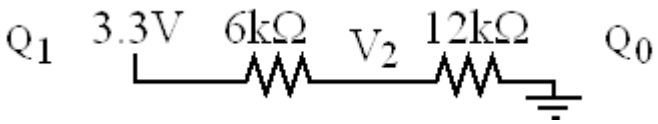
N	Q_1	Q_0	V_{out} (V)
0	0	0	0.0
1	0	3.3	1.1
2	3.3	0	2.2
3	3.3	3.3	3.3

Table 13.1. Specifications of the 2-bit binary-weighted DAC.

Assume the output high voltage (V_{OH}) of the microcontroller is 3.3 V, and its output low voltage (V_{OL}) is 0. With a binary-weighted DAC, we choose the resistor ratio to be 2/1 so Q_1 bit is twice as significant as the Q_0 bit, as shown in Figure 13.3. Considering the circuit on the left (no headphones), if both Q_1 and Q_0 are 0, the output V_{out} is zero. If Q_1 is 0 and Q_0 is +3.3V, the output V_{out} is determined by the resistor divider network



Note the total impedance from 3.3V to ground in the above circuit is 18kΩ. Using Ohm's Law, with the voltage divider equation, we can calculate V_{out} to be 3.3V*6kΩ/18kΩ, which is 1.1V. If Q_1 is +3.3V and Q_0 is 0, the output V_{out} is determined by the network



Again notice the total impedance from 3.3V to ground in this second circuit is 18kΩ. But this time we calculate V_{out} to be 3.3V*12kΩ/18kΩ, which is 2.2V. If both Q_1 and Q_0 are +3.3V, the output V_{out} is +3.3V. The output impedance of this DAC is approximately 12 kΩ, which means it cannot source or sink much current.

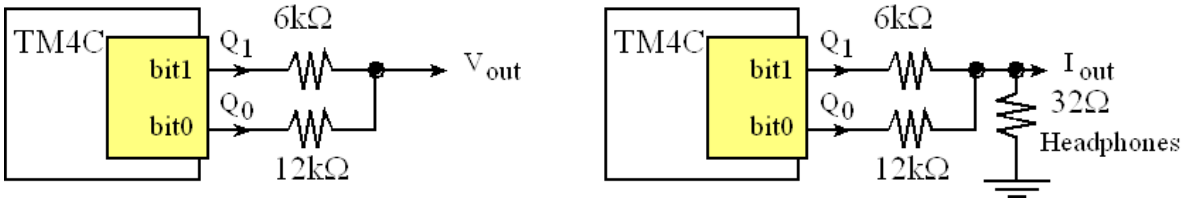


Figure 13.3. A 2-bit binary-weighted DAC.

If we connect headphones to this DAC, as shown in the right side of Figure 13.3, we could hear sounds generated by software writing a sequence of data to the DAC. However, since the impedance of the

headphones is much smaller than the impedance of the DAC, the output voltages will be very small, but we could calculate the currents into the headphones. Considering the circuit on the right (with headphones), if both Q_1 and Q_0 are 0, the output current is zero. If Q_1 is 0 and Q_0 is +3.3V, the output current, I_{out}, is 3.3V divided by 12.032kΩ which is 0.275mA. If Q_0 is 0 and Q_1 is +3.3V, the output current, I_{out}, is 3.3V divided by 6.032kΩ which is 0.550mA. And finally, if both Q_1 and Q_0 are 3,3V, I_{out} is the sum of 0.275+0.550 (Kirkhoff's Current Law), which is about 0.825mA. Notice the current into the headphones is linearly related to the digital value. In other words, digital values of 0,1,2,3 map to currents of 0,0.275,0.550,0.825mA.

You can realistically build a 6-bit DAC using the binary-weighted method.

Checkpoint 13.7: How do you build a 3-bit binary-weighted DAC using this method?

13.3. Sound Generated by Speakers

Before we generate sound, let's look at how a typical speaker works. Sound exists as varying pressure waves that are created when a physical object moves, vibrating the air next to it. These air pressure waves travel through the air in all directions at about 343 m/sec. Sound can also be generated in water, where it travels at 1484 m/sec. Our ears can sense sounds from 20 Hz to about 20 kHz. In other words, the pressure wave must be oscillating faster than 20 Hz and slower than 20 kHz for us to hear it. See Figure 13.4.

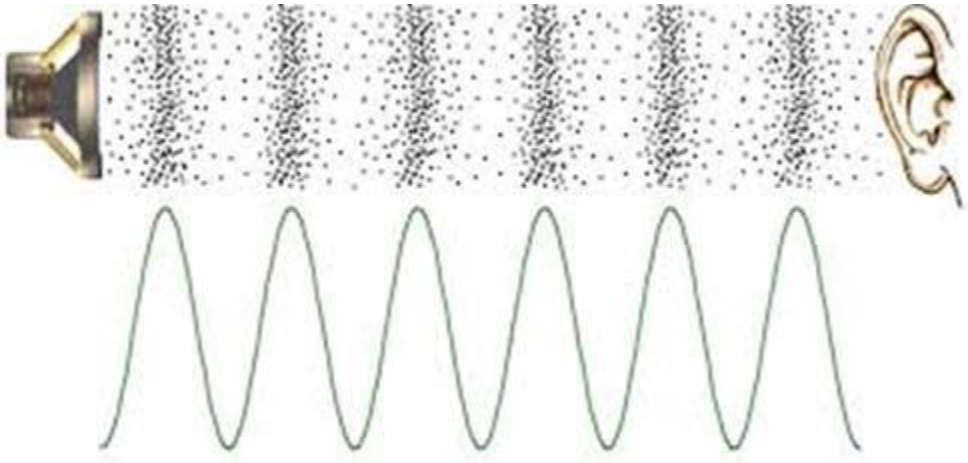


Figure 13.4. Sound waves exist as pressure waves in media such as air, water, and non-porous solids.

There are two magnets in a speaker. There is a large permanent magnetic that creates a static magnetic field oriented in the direction the speaker is facing, and there is an electromagnet created by a spiral-wound coil oriented in the same direction. Figure 13.5 shows a magnetic field generated by a cylindrically-wound coil.

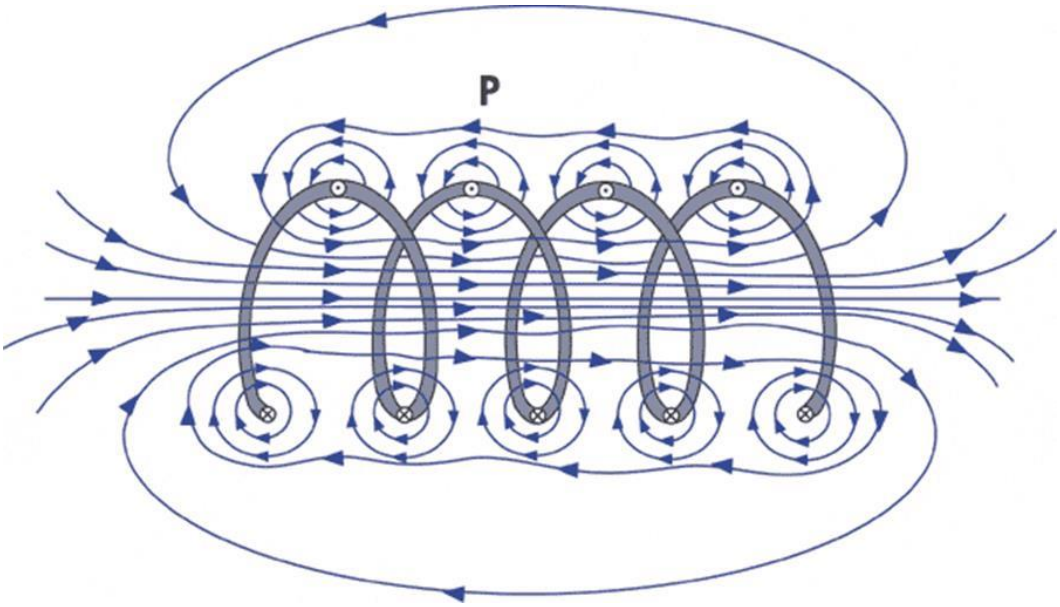


Figure 13.5. The magnetic field produced by a coil is very strong oriented in the direction of the cylinder.

The strength and direction of the magnetic field are related to the strength and direction of the electrical current conducted through the coiled wire. Figure 13.6 shows a cut away of a typical speaker. The alternating magnetic field generated by the coil interacts with the constant magnetic field produced by the permanent magnet. To generate sound we will create an oscillating current through the coil, this will create an oscillating magnetic field, and will vibrate the voice coil. The diaphragm and spider hold the voice coil to the suspension allowing it to vibrate up and down. When the voice coil vibrates up and down it creates sound waves. The frequency and amplitude of the sound is directly related to the frequency and amplitude of the current passing through the coil. The resistance of the coil in a typical headphone is 32 Ω.

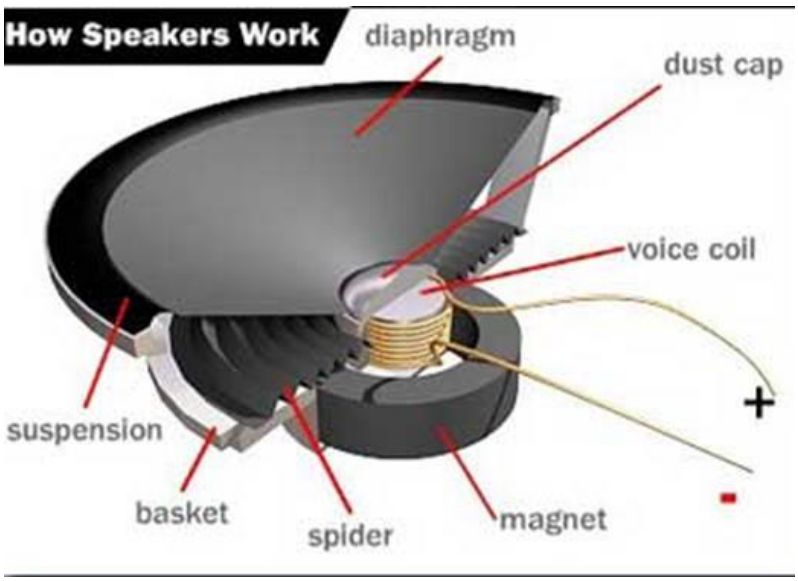


Figure 13.6. A speaker can generate sound by vibrating the voice coil using an electromagnet.

Checkpoint 13.8: Some speakers are heavier than others. These heavy speakers have larger permanent magnets. Why would we want a permanent magnet that can create a larger magnetic field?

13.4. Music Generation

Most digital music devices rely on high-speed DACs to create the analog waveforms required to produce high-quality sound. In this section, we will discuss a very simple sound generation system that

illustrates this application of the DAC. The hardware consists of a DAC and a speaker interface. You can drive headphones directly from a DAC output, but to drive a regular speaker, you will need to add an audio amplifier, as illustrated in Figure 13.7.

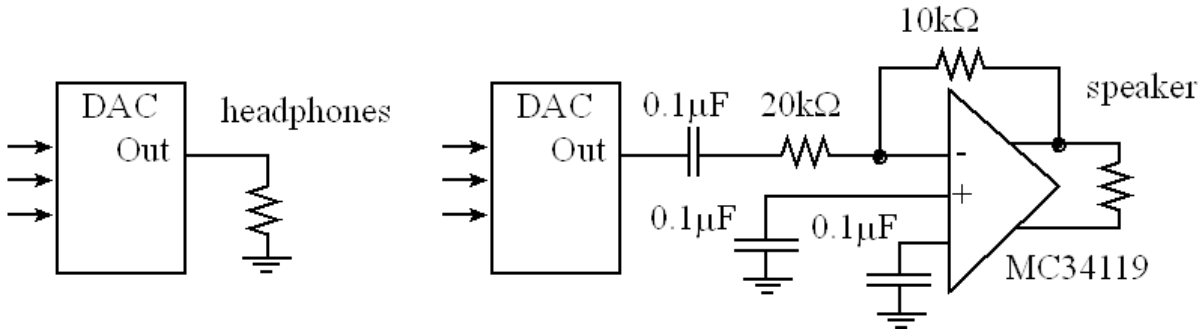


Figure 13.7. DAC allows the software to create music.

The quality of the music will depend on both hardware and software factors. The precision of the DAC, external noise, and the dynamic range of the speaker are some of the hardware factors. Software factors include the DAC output rate and the complexity of the stored sound data. If you output a sequence of numbers to the DAC that form a sine wave, then you will hear a continuous tone on the speaker, as shown in Figure 13.8. The **loudness** of the tone is determined by the amplitude of the wave. The **pitch** is defined as the frequency of the wave. Table 13.2 contains frequency values for the notes in one octave. The frequency of the wave, f_{sin}, will be determined by the frequency of the interrupt, f_{int}, divided by the size of the table n. The size of the table in Program 13.1 is n=16.

$$f_{sin} = f_{int} / n$$

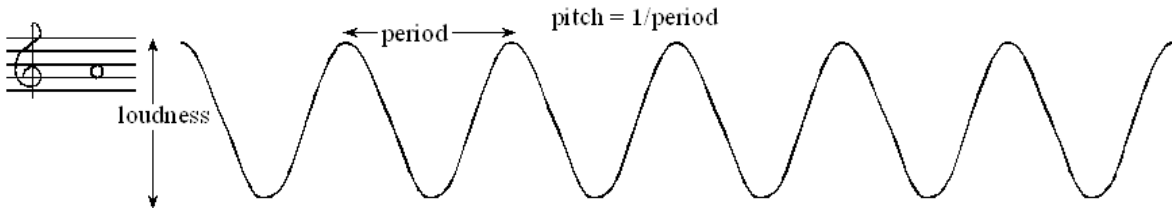


Figure 13.8. The loudness and pitch are controlled by the amplitude and frequency.

The frequency of each musical note can be calculated by multiplying the previous frequency by $\sqrt[12]{2}$. You can use this method to determine the frequencies of additional notes above and below the ones in Table 13.2. There are twelve notes in an octave, therefore moving up one octave doubles the frequency.

Note	frequency
C	523 Hz
B	494 Hz
B^b	466 Hz
A	*440 Hz*
A^b	415 Hz
G	392 Hz
G^b	370 Hz
F	349 Hz
E	330 Hz
E^b	311 Hz
D	294 Hz
D^b	277 Hz
C	262 Hz

Table 13.2. Fundamental frequencies of standard musical notes. The frequency for 'A' is exact.

Figure 13.9 illustrates the concept of **instrument**. You can define the type of sound by the shape of the voltage versus time waveform. Brass instruments have a very large first harmonic frequency.

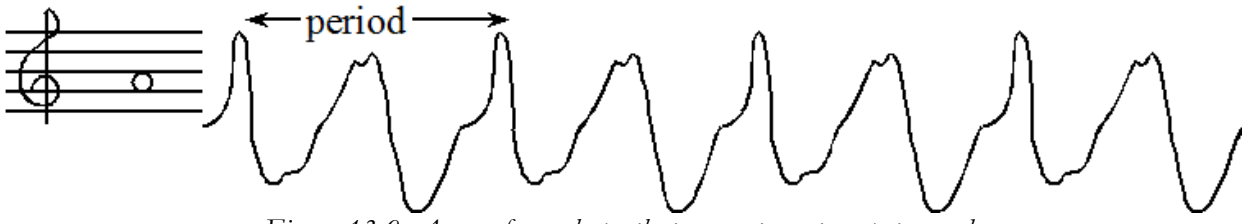


Figure 13.9. A waveform shape that generates a trumpet sound.

The **tempo** of the music defines the speed of the song. In 2/4 3/4 or 4/4 music, a **beat** is defined as a quarter note. A moderate tempo is 120 beats/min, which means a quarter note has a duration of ½ second. A sequence of notes can be separated by pauses (silences) so that each note is heard separately. The **envelope** of the note defines the amplitude versus time relationship. A very simple envelope is illustrated in Figure 13.10. The Cortex™-M processor has plenty of processing power to create these types of waves.

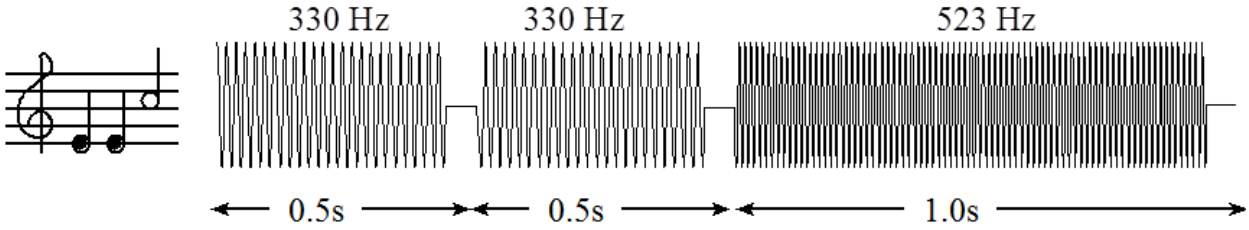


Figure 13.10. You can control the amplitude, frequency and duration of each note (not drawn to scale).

The smooth-shaped envelope, as illustrated in Figure 13.11, causes a less staccato and more melodic sound. This type of sound generation is possible to produce in real time on the Cortex™-M microcontroller.

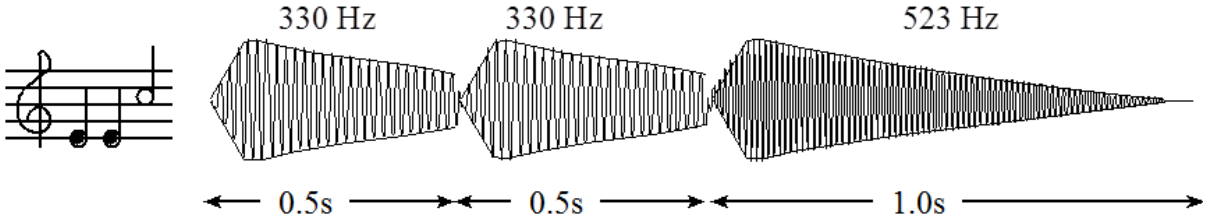


Figure 13.11. The amplitude of a plucked string drops exponentially in time.

A **chord** is created by playing multiple notes simultaneously. When two piano keys are struck simultaneously both notes are created, and the sounds are mixed arithmetically. You can create the same effect by adding two waves together in software, before sending the wave to the DAC. Figure 13.12 plots the mathematical addition of a 262 Hz (low C) and a 392 Hz sine wave (G), creating a simple chord.

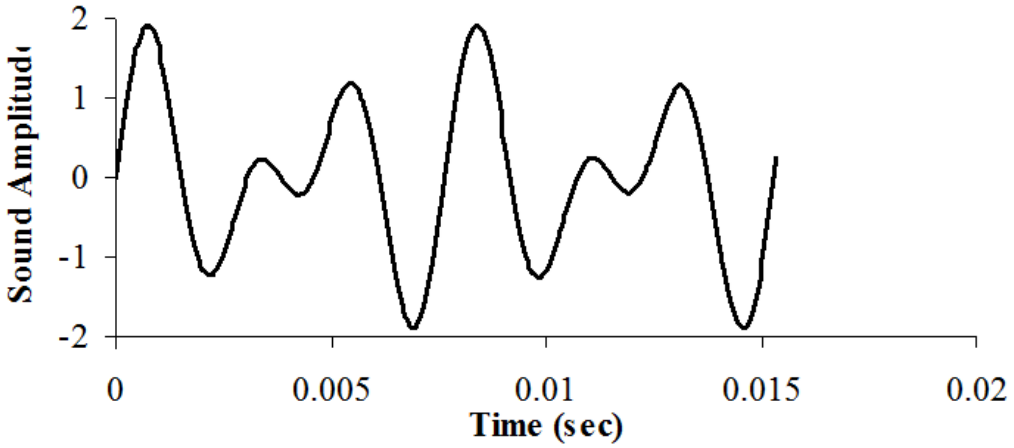


Figure 13.12. A simple chord mixing the notes C and G.

Example 13.2. Design a 3-bit R-2R DAC and use it to create a 100 Hz sine wave.
Solution: We begin by specifying the desired input/output relationship of the 3-bit DAC. Table 13.3 shows the design specification. It will be able to generate 8 different current outputs with a resolution of about 12.5 μA. The circuit is presented in Figure 13.13.

N	Q_2	Q_1	Q_0	I_{out} (μA)
0	0	0	0	0.0
1	0	0	3.3	12.5
2	0	3.3	0	25.0
3	0	3.3	3.3	37.5
4	3.3	0	0	50.0
5	3.3	0	3.3	62.5
6	3.3	3.3	0	75.0
7	3.3	3.3	3.3	87.5

Table 13.3. Specifications of the 3-bit R-2R DAC.

We assume the output high voltage (V_{OH}) of the microcontroller is 3.3 V, and its output low voltage (V_{OL}) is 0. A 3-bit R-2R DAC will use two R resistors (11kΩ) and five 2R resistors (22kΩ), as shown in Figure 13.13. The maximum current output will be (1/2 + 1/4 + 1/8)*3.3V/33kΩ = 87.5 μA.

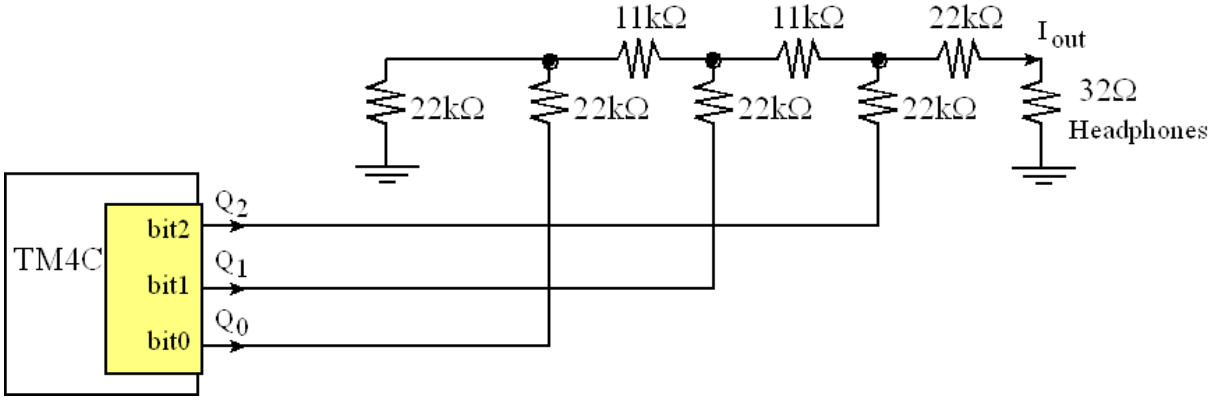


Figure 13.13. A 3-bit R-2R DAC.

The way to analyze an R-2R DAC is to employ the Law of Superposition. We will consider the three basis elements (1, 2, and 4). If these three cases are demonstrated, then the Law of Superposition guarantees the other five will work. When one of the digital inputs is true then V_{ref} (3.3V) is connected

to the R-2R ladder, and when the digital input is false, then the connection is grounded. See Figure 13.14. Since 22,000 is much bigger than 32, we can neglect the 32-Ω resistance when calculating current.

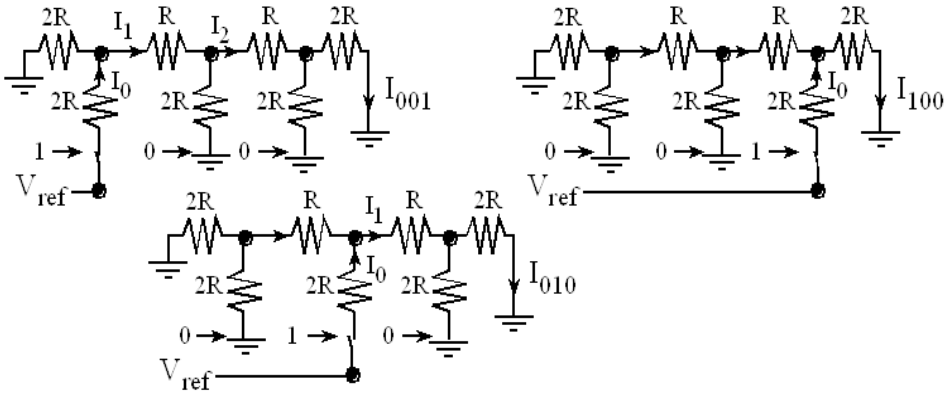


Figure 13.14. Analysis of the three basis elements {001, 010, 100} of the 3-bit unsigned R-2R DAC.

In each of the three test cases, the current across the active switch is $I_0=V_{ref}$ / (3*R*). This current is divided by 2 at each branch point. I.e., $I_1 = I_0/2$, and $I_2 = I_1/2$. Current injected from the lower bits will be divided more times. Since each stage divides by two, the exponential behavior is produced. An actual DAC is implemented with a current switch rather than a voltage switch. Nevertheless, this simple circuit illustrates the operation of the R-2R ladder function. When the input is 001, V_{ref} is presented to the left. The effective impedance to ground is 3*R*, so the current injected into the R-2R ladder is $I_0=V_{ref}$ / (3*R*). The current is divided in half three times, and $I_{001}=V_{ref}$ / (24*R*).

When the input is 010, V_{ref} is presented in the middle. The effective impedance to ground is still 3*R*, so the current injected into the R-2R ladder is $I_0=V_{ref}$ / (3*R*). The current is divided in half twice, and $I_{010}=V_{ref}$ / (12*R*).

When the input is 100, V_{ref} is presented on the right. The effective impedance to ground is once again 3*R*, so the current injected into the R-2R ladder is $I_0=V_{ref}$ / (3*R*). The current is divided in half once, and $I_{100}=V_{ref}$ / (6*R*).

Using the Law of Superposition, the output voltage is a linear combination of the three digital inputs, $I_{out}=(4b_2 + 2b_1 + b_0)V_{ref}$ / (24*R*). A current to voltage circuit is used to create a voltage output. To increase the precision one simply adds more stages to the R-2R ladder.

To generate sound we need a table of data and a periodic interrupt. Program 13.1 shows C code that defines a 16-element 3-bit sine wave. The frequency of the sound will be the interrupt frequency divided by 16 (size of the table). So, to create a 100 Hz wave we need SysTick to interrupt at 16*100 Hz, 1600 Hz. If the bus clock is 80 MHz, then the initialization should be called with an input parameter of value 50000. The **`const`** modifier will place the data in ROM. The interrupt software will output one value to the DAC. See Figure 13.15. In this example, the 3-bit DAC is interfaced to output pins PB2-0. To output to this DAC we simply write to Port B. In order to create the sound, it is necessary to output just one number to the DAC upon each interrupt. The DAC range is 0 to 87.5 μA.

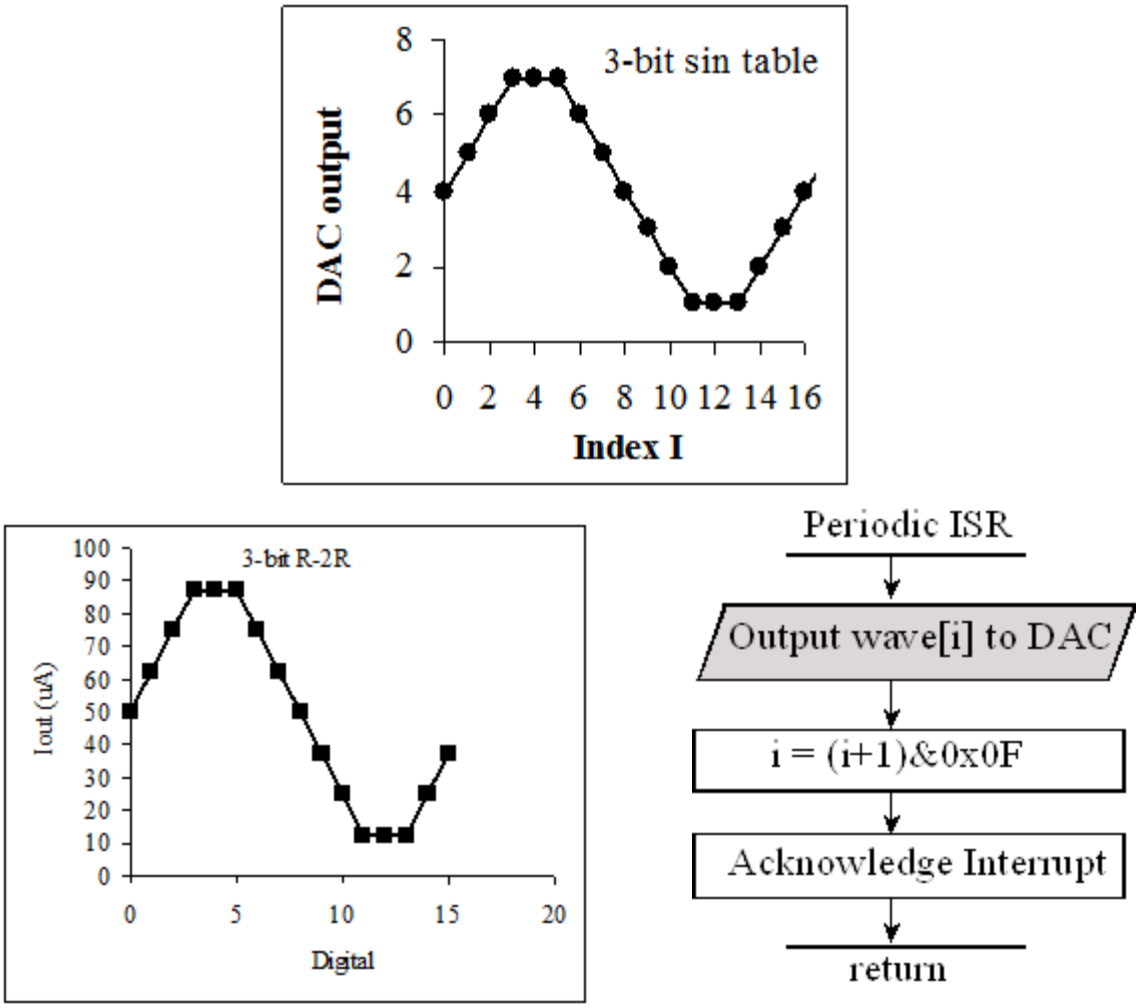


Figure 13.15. A DAC and a periodic interrupt are used to create sound. Output is discrete in time and voltage.

```
const unsigned char SineWave[16] = {4,5,6,7,7,7,6,5,4,3,2,1,1,1,2,3};
unsigned char Index=0;           // Index varies from 0 to 15
// **************DAC_Init**********************
// Initialize 3-bit DAC
// Input: none
// Output: none
void DAC_Init(void){unsigned long volatile delay;
  SYSCTL_RCGC2_R |= SYSCTL_RCGC2_GPIOB; // activate port B
  delay = SYSCTL_RCGC2_R;    // allow time to finish activating
  GPIO_PORTB_AMSEL_R &= ~0x07;      // no analog
  GPIO_PORTB_PCTL_R &= ~0x00000FFF; // regular GPIO function
  GPIO_PORTB_DIR_R |= 0x07;      // make PB2-0 out
  GPIO_PORTB_AFSEL_R &= ~0x07;   // disable alt funct on PB2-0
  GPIO_PORTB_DEN_R |= 0x07;      // enable digital I/O on PB2-0
}

// **************Sound_Init**********************
// Initialize Systick periodic interrupts
// Input: interrupt period
//        Units of period are 12.5ns
//        Maximum is 2^24-1
//        Minimum is determined by length of ISR
// Output: none
void Sound_Init(unsigned long period){
  DAC_Init();          // Port B is DAC
  Index = 0;
  NVIC_ST_CTRL_R = 0;         // disable SysTick during setup
```

```
  NVIC_ST_RELOAD_R = period-1;// reload value
  NVIC_ST_CURRENT_R = 0;      // any write to current clears it
  NVIC_SYS_PRI3_R = (NVIC_SYS_PRI3_R&0x00FFFFFF)|0x20000000; // priority 1
  NVIC_ST_CTRL_R = 0x0007; // enable SysTick with core clock and interrupts
}
// **************DAC_Out*********************
// output to DAC
// Input: 3-bit data, 0 to 7
// Output: none
void DAC_Out(unsigned long data){
  GPIO_PORTB_DATA_R = data;
}
// the sound frequency will be (interrupt frequency)/(size of the table)
void SysTick_Handler(void){
  Index = (Index+1)&0x0F;      // 4,5,6,7,7,7,6,5,4,3,2,1,1,1,2,3,...
  DAC_Out(SineWave[Index]);    // output one value each interrupt
}
void main(void){
  PLL_Init();          // bus clock at 80 MHz
  Switch_Init();       // Port F is onboard switches, LEDs, profiling
  Sound_Init(50000);   // initialize SysTick timer, 1.6kHz
  while(1){
  }
}
```

Program 13.1. The periodic interrupt outputs one value to the DAC.

Throughout this course we have seen that an embedded system uses its input/output devices to interact with the external world. In this chapter we will focus on input devices that we use to gather information about the world. More specifically, we present a technique for the system to measure analog inputs using an analog to digital converter (ADC). We will use periodic interrupts to sample the ADC at a fixed rate. We will then combine sensors, the ADC, software, PWM output and motor interfaces to implement intelligent control on our robot car.

Learning Objectives:

- Develop a means for a digital computer to sense its analog world.
- Review digitization: Quantization, range, precision and resolution.
- Extend the Nyquist Theorem to cases the ADC is used to sense information.
- Study the basics of transducers: conversion of physical to electrical.
- Use an optical sensor to measure distance to an object.

14.1. Analog to Digital Conversion

An analog to digital converter (ADC) converts an analog signal into digital form, shown in Figure 14.1. An embedded system uses the ADC to collect information about the external world (data acquisition system.) The input signal is usually an analog voltage, and the output is a binary number. The ADC precision is the number of distinguishable ADC inputs (e.g., 4096 alternatives, 12 bits). The ADC **range** is the maximum and minimum ADC input (e.g., 0 to +3.3V). The ADC **resolution** is the smallest distinguishable change in input (e.g., 3.3V/4096, which is about 0.81 mV). The resolution is the change in input that causes the digital output to change by 1.

Range(volts) = Precision(alternatives) • Resolution(volts)

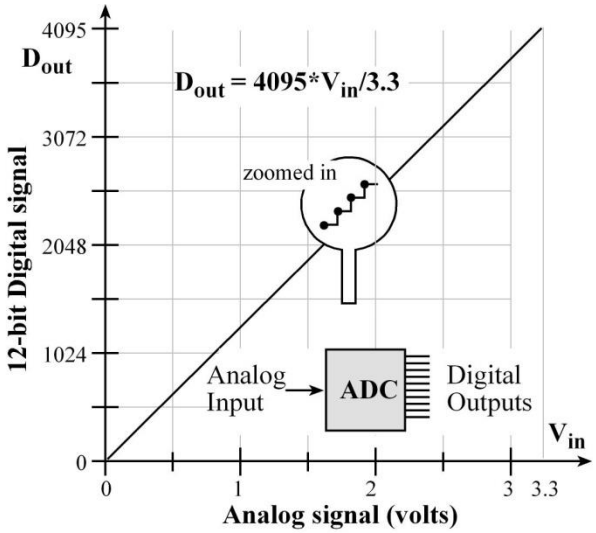


Figure 14.1. A 12-bit ADC converts 0 to 3.3V on its input into a digital number from 0 to 4095.

The most pervasive method for ADC conversion is the **successive approximation** technique, as illustrated in Figure 14.2. A 12-bit successive approximation ADC is clocked 12 times. At each clock another bit is determined, starting with the most significant bit. For each clock, the successive approximation hardware issues a new "guess" on V_{dac} by setting the bit under test to a "1". If V_{dac} is now higher than the unknown input, V_{in}, then the bit under test is cleared. If V_{dac} is less than V_{in}, then the bit under test is remains 1. In this description, *bit* is an unsigned integer that specifies the bit under test. For

a 12-bit ADC, *bit* goes 2048, 1024, 512, 256,...,1. D_{out} is the ADC digital output, and *Z* is the binary input that is true if V_{dac} is greater than V_{in}.

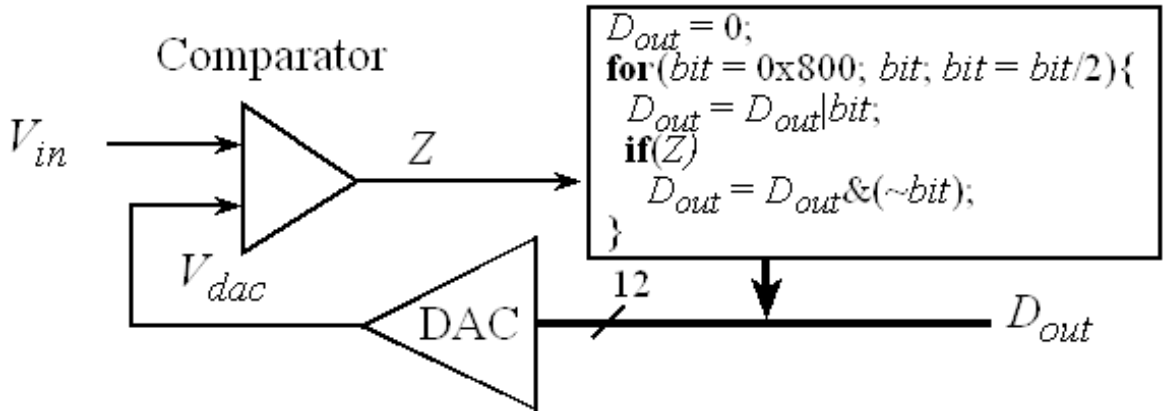


Figure 14.2. A 12-bit successive approximation ADC.

Observation: The speed of a successive approximation ADC relates linearly with its precision in bits.

Normally we don't specify accuracy for just the ADC, but rather we give the accuracy of the entire system (including transducer, analog circuit, ADC and software). An ADC is **monotonic** if it has no missing codes as the analog input slowly rises. This means if the analog signal is a slowly rising voltage, then the digital output will hit all values one at a time, always going up, never going down. The **figure of merit** of an ADC involves three factors: precision (number of bits), speed (how fast can we sample), and power (how much energy does it take to operate). How fast we can sample involves both the ADC conversion time (how long it takes to convert), and the bandwidth (what frequency components can be recognized by the ADC). The ADC cost is a function of the number and quality of internal components. Two 12-bit ADCs are built into the TM4C123/LM4F120 microcontroller. You will use ADC0 to collect data and we will use ADC1 and the PD3 pin to implement a voltmeter and oscilloscope.

14.2. ADC on the TM4C123/LM4F120

Table 14.1 shows the ADC0 register bits required to perform sampling on a single channel. There are two ADCs; you will use ADC0 and the grader uses ADC1. For more complex configurations refer to the specific data sheet. Bits 8 and 9 of the **SYSCTL_RCGC0_R** specify the maximum sampling rate, see Table 14.2. The TM4C123 can sample up to 1 million samples per second. Bits 8 and 9 of the **SYSCTL_RCGC0_R** specify how fast it COULD sample; the actual sampling rate is determined by the rate at which we trigger the ADC. In this chapter we will use software trigger mode, so the actual sampling rate is determined by the SysTick periodic interrupt rate; the SysTick ISR will take one ADC sample. On the TM4C123, we will need to set bits in the **AMSEL** register to activate the analog interface.

Address	31-17	16	15-10	9	8	7-0	Name
0x400F.E100		ADC		MAXADCSPD			SYSCTL_RCGC0_R

	31-14	13-12	11-10	9-8	7-6	5-4	3-2	1-0	
0x4003.8020		SS3		SS2		SS1		SS0	ADC0_SSPRI_R

	31-16	15-12	11-8	7-4	3-0	
0x4003.8014		EM3	EM2	EM1	EM0	ADC0_EMUX_R

	31-4	3	2	1	0	
0x4003.8000		ASEN3	ASEN2	ASEN1	ASEN0	ADC0_ACTSS_R
0x4003.80A0		MUX0				ADC0_SSMUX3_R
0x4003.80A4		TS0	IE0	END0	D0	ADC0_SSCTL3_R
0x4003.8028		SS3	SS2	SS1	SS0	ADC0_PSSI_R
0x4003.8004		INR3	INR2	INR1	INR0	ADC0_RIS_R
0x4003.800C		IN3	IN2	IN1	IN0	ADC0_ISC_R

	31-12	11-0	
0x4003.80A8		DATA	ADC0_SSFIFO3

Table 14.1. The TM4C ADC0 registers. Each register is 32 bits wide. You will use ADC0 and we will use ADC1 for the grader and to implement the oscilloscope feature.

Value	*Description*
0x3	1M samples/second
0x2	500K samples/second
0x1	250K samples/second
0x0	125K samples/second

Table 14.2. The ADC MAXADCSPD bits in the SYSCTL_RCGC0_R register.

Table 14.3 shows which I/O pins on the TM4C123 can be used for ADC analog input channels.

IO	Ain	0	1	2	3	4	5	6	7	8	9	14
PB4	Ain10	Port		SSI2Clk		M0PWM2			T1CCP0	CAN0Rx		
PB5	Ain11	Port		SSI2Fss		M0PWM3			T1CCP1	CAN0Tx		
PD0	Ain7	Port	SSI3Clk	SSI1Clk	I_2C3SCL	M0PWM6	M1PWM0		WT2CCP0			
PD1	Ain6	Port	SSI3Fss	SSI1Fss	I_2C3SDA	M0PWM7	M1PWM1		WT2CCP1			
PD2	Ain5	Port	SSI3Rx	SSI1Rx		M0Fault0			WT3CCP0	USB0epen		
PD3	Ain4	Port	SSI3Tx	SSI1Tx				IDX0	WT3CCP1	USB0pflt		
PE0	Ain3	Port	U7Rx									
PE1	Ain2	Port	U7Tx									
PE2	Ain1	Port										
PE3	Ain0	Port										
PE4	Ain9	Port	U5Rx		I_2C2SCL	M0PWM4	M1PWM2			CAN0Rx		
PE5	Ain8	Port	U5Tx		I_2C2SDA	M0PWM5	M1PWM3			CAN0Tx		

Table 14.3. Twelve different pins on the LM4F/TM4C can be used to sample analog inputs. You will use ADC0 and PE2 to sample analog input. If your PE2 pin is broken, you will have the option to perform Lab 14 with PE3 or PE5. We use ADC1 and PD3 to implement the oscilloscope feature.

The ADC has four sequencers, but you will use only sequencer 3 in Labs 14 and 15. We set the **ADC0_SSPRI_R** register to 0x0123 to make sequencer 3 the highest priority. Because we are using just one sequencer, we just need to make sure each sequencer has a unique priority. We set bits 15–12 (**EM3**) in the **ADC0_EMUX_R** register to specify how the ADC will be triggered. Table 14.4 shows the various ways to trigger an ADC conversion. More advanced ADC triggering techniques are presented in the book Embedded Systems: Real-Time Interfacing to ARM® Cortex™-M Microcontrollers. However in this course, we use software start (**EM3**=0x0). The software writes an 8 (**SS3**) to the **ADC0_PSSI_R** to initiate a conversion on sequencer 3. We can enable and disable the sequencers using the **ADC0_ACTSS_R** register. There are twelve ADC channels on the

LM4F120/TM4C123. Which channel we sample is configured by writing to the **ADC0_SSMUX3_R** register. The mapping between channel number and the port pin is shown in Table 14.3. For example channel 9 is connected to the pin PE4. The **ADC0_SSCTL3_R** register specifies the mode of the ADC sample. We set **TS0** to measure temperature and clear it to measure the analog voltage on the ADC input pin. We set **IE0** so that the **INR3** bit is set when the ADC conversion is complete, and clear it when no flags are needed. When using sequencer 3, there is only one sample, so **END0** will always be set, signifying this sample is the end of the sequence. In this class, the *sequence* will be just one ADC conversion. We set the **D0** bit to activate differential sampling, such as measuring the analog difference between two ADC pins. In our example, we clear **D0** to sample a single-ended analog input. Because we set the **IE0** bit, the **INR3** flag in the **ADC0_RIS_R** register will be set when the ADC conversion is complete, We clear the **INR3** bit by writing an 8 to the 8 to the **ADC0_ISC_R** register.

Value	*Event*
0x0	Software start
0x1	Analog Comparator 0
0x2	Analog Comparator 1
0x3	Analog Comparator 2
0x4	External (GPIO PB4)
0x5	Timer
0x6	PWM0
0x7	PWM1
0x8	PWM2
0x9	PWM3
0xF	Always (continuously sample)

Table 14.4. The ADC EM3, EM2, EM1, and EM0 bits in the ADC_EMUX_R register.

We perform the following steps to configure the ADC for software start on one channel. Program 14.1 shows a specific details for sampling PE4, which is channel 9. The function **ADC0_InSeq3** will sample PE4 using software start and use busy-wait synchronization to wait for completion.

Step 1. We enable the port clock for the pin that we will be using for the ADC input.
Step 2. Make that pin an input by writing zero to the **DIR** register.
Step 3. Enable the alternative function on that pin by writing one to the **AFSEL** register.
Step 4. Disable the digital function on that pin by writing zero to the **DEN** register.
Step 5. Enable the analog function on that pin by writing one to the **AMSEL** register.
Step 6. We enable the ADC clock by setting bit 16 of the **SYSCTL_RCGC0_R** register.
Step 7. Bits 8 and 9 of the **SYSCTL_RCGC0_R** register specify the maximum sampling rate of the ADC. In this example, we will sample slower than 125 kHz, so the maximum sampling rate is set at 125 kHz. This will require less power and produce a longer sampling time, creating a more accurate conversion.
Step 8. We will set the priority of each of the four sequencers. In this case, we are using just one sequencer, so the priorities are irrelevant, except for the fact that no two sequencers should have the same priority.
Step 9. Before configuring the sequencer, we need to disable it. To disable sequencer 3, we write a 0 to bit 3 (**ASEN3**) in the **ADC_ACTSS_R** register. Disabling the sequencer during programming prevents erroneous execution if a trigger event were to occur during the configuration process.
Step 10. We configure the trigger event for the sample sequencer in the **ADC_EMUX_R** register. For this example, we write a 0000 to bits 15–12 (**EM3**) specifying software start mode for sequencer 3.
Step 11. Configure the corresponding input source in the **ADCSSMUXn** register. In this example, we write the channel number to bits 3–0 in the **ADC_SSMUX3_R** register. In this example, we sample channel 9, which is PE4.

Step 12. Configure the sample control bits in the corresponding nibble in the **ADC0SSCTLn** register. When programming the last nibble, ensure that the **END** bit is set. Failure to set the **END** bit causes unpredictable behavior. Sequencer 3 has only one sample, so we write a 0110 to the **ADC_SSCTL3_R** register. Bit 3 is the **TS0** bit, which we clear because we are not measuring temperature. Bit 2 is the **IE0** bit, which we set because we want to the **RIS** bit to be set when the sample is complete. Bit 1 is the **END0** bit, which is set because this is the last (and only) sample in the sequence. Bit 0 is the **D0** bit, which we clear because we do not wish to use differential mode.

Step 13. We enable the sample sequencer logic by writing a 1 to the corresponding **ASENn**. To enable sequencer 3, we write a 1 to bit 3 (**ASEN3**) in the **ADC_ACTSS_R** register.

```
void ADC0_InitSWTriggerSeq3_Ch9(void){ volatile unsigned long delay;
  SYSCTL_RCGC2_R |= 0x00000010;   // 1) activate clock for Port E
  delay = SYSCTL_RCGC2_R;         //    allow time for clock to stabilize
  GPIO_PORTE_DIR_R& = ~0x04;      // 2) make PE4 input
  GPIO_PORTE_AFSEL_R |= 0x04;     // 3) enable alternate function on PE2
  GPIO_PORTE_DEN_R &= ~0x04;      // 4) disable digital I/O on PE2
  GPIO_PORTE_AMSEL_R |= 0x04;     // 5) enable analog function on PE2
  SYSCTL_RCGC0_R |= 0x00010000;   // 6) activate ADC0
  delay = SYSCTL_RCGC2_R;
  SYSCTL_RCGC0_R &= ~0x00000300;  // 7) configure for 125K
  ADC0_SSPRI_R = 0x0123;          // 8) Sequencer 3 is highest priority
  ADC0_ACTSS_R &= ~0x0008;        // 9) disable sample sequencer 3
  ADC0_EMUX_R &= ~0xF000;         // 10) seq3 is software trigger
  ADC0_SSMUX3_R &= ~0x000F;       // 11) clear SS3 field
  ADC0_SSMUX3_R += 9;             //    set channel Ain9 (PE4)
  ADC0_SSCTL3_R = 0x0006;         // 12) no TS0 D0, yes IE0 END0
  ADC0_ACTSS_R |= 0x0008;         // 13) enable sample sequencer 3
}
```

Program 14.1. Initialization of the ADC using software start and busy-wait (C14_ADCSWTrigger).

Program 14.2 gives a function that performs an ADC conversion. There are four steps required to perform a software-start conversion. The range is 0 to 3.3V. If the analog input is 0, the digital output will be 0, and if the analog input is 3.3V, the digital output will be 4095.

Digital Sample = (Analog Input (volts) • 4095) / 3.3V(volts)

Step 1. The ADC is started using the software trigger. The channel to sample was specified earlier in the initialization.

Step 2. The function waits for the ADC to complete by polling the RIS register bit 3.

Step 3. The 12-bit digital sample is read out of sequencer 3.

Step 4. The RIS bit is cleared by writing to the ISC register.

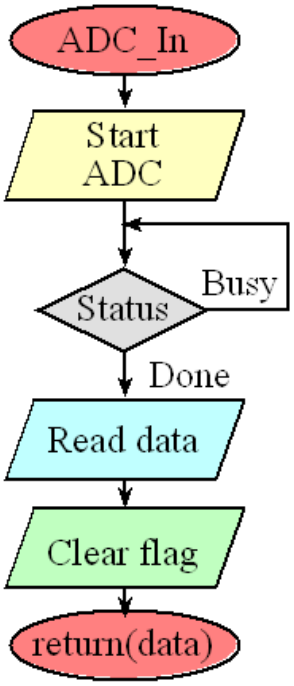


Figure 14.3. The four steps of analog to digital conversion: 1) initiate conversion, 2) wait for the ADC to finish, 3) read the digital result, and 4) clear the completion flag.

```
//------------ADC_InSeq3------------
// Busy-wait analog to digital conversion
// Input: none
// Output: 12-bit result of ADC conversion
unsigned long ADC0_InSeq3(void){  unsigned long result;
  ADC0_PSSI_R = 0x0008;            // 1) initiate SS3
  while((ADC0_RIS_R&0x08)==0){};   // 2) wait for conversion done
  result = ADC0_SSFIFO3_R&0xFFF;   // 3) read result
  ADC0_ISC_R = 0x0008;             // 4) acknowledge completion
  return result;
}
```

Program 14.2. ADC sampling using software start and busy-wait (C14_ADCSWTrigger).

There is software in the book Embedded Systems: Real-Time Interfacing to ARM® Cortex™-M Microcontrollers showing you how to configure the ADC to sample a single channel at a periodic rate using a timer trigger. The most accurate sampling method is timer-triggered sampling (**EM3**=0x5).

Checkpoint 14.1: If the input voltage is 1.5V, what value will the TM4C 12-bit ADC return?
Checkpoint 14.2: If the input voltage is 0.5V, what value will the TM4C 12-bit ADC return?

14.3. Nyquist Theorem

To collect information from the external world into the computer we must convert it from analog into digital form. This conversion process is called sampling and because the output of the conversion is one digital number at one point in time, there must be a finite time in between conversions, Δt. If we use SysTick periodic interrupts, then this Δt is the time between SysTick interrupts. We define the sampling rate as

$$f_s = 1/\Delta t$$

If this information oscillates at frequency f, then according to the **Nyquist Theorem**, we must sample that signal at

$$f_s > 2f$$

Furthermore, the **Nyquist Theorem** states that if the signal is sampled with a frequency of f_s, then the digital samples only contain frequency components from 0 to ½ f_s. Conversely, if the analog signal does contain frequency components larger than ½ f_s, then there will be an aliasing error during the sampling process (performed with a frequency of f_s). **Aliasing** is when the digital signal appears to have a different frequency than the original analog signal.

Figure 14.4 shows what happens when the Nyquist Theorem is violated. In both cases a signal was sampled at 2000 Hz (every 0.5 ms). In the first figure the 200 Hz signal is properly sampled, which means the digital samples accurately describe the analog signal. However, in the second figure, the 2200 Hz signal is not sampled properly, which means the digital samples do not accurately describe the analog signal. This error is called aliasing. Aliasing occurs when the input signal oscillates faster than the sampling rate and it characterized by the digital samples "looking like" it is oscillating at a different rate than the original analog signal. For these two sets of sampled data, notice the digital data are exactly the same.

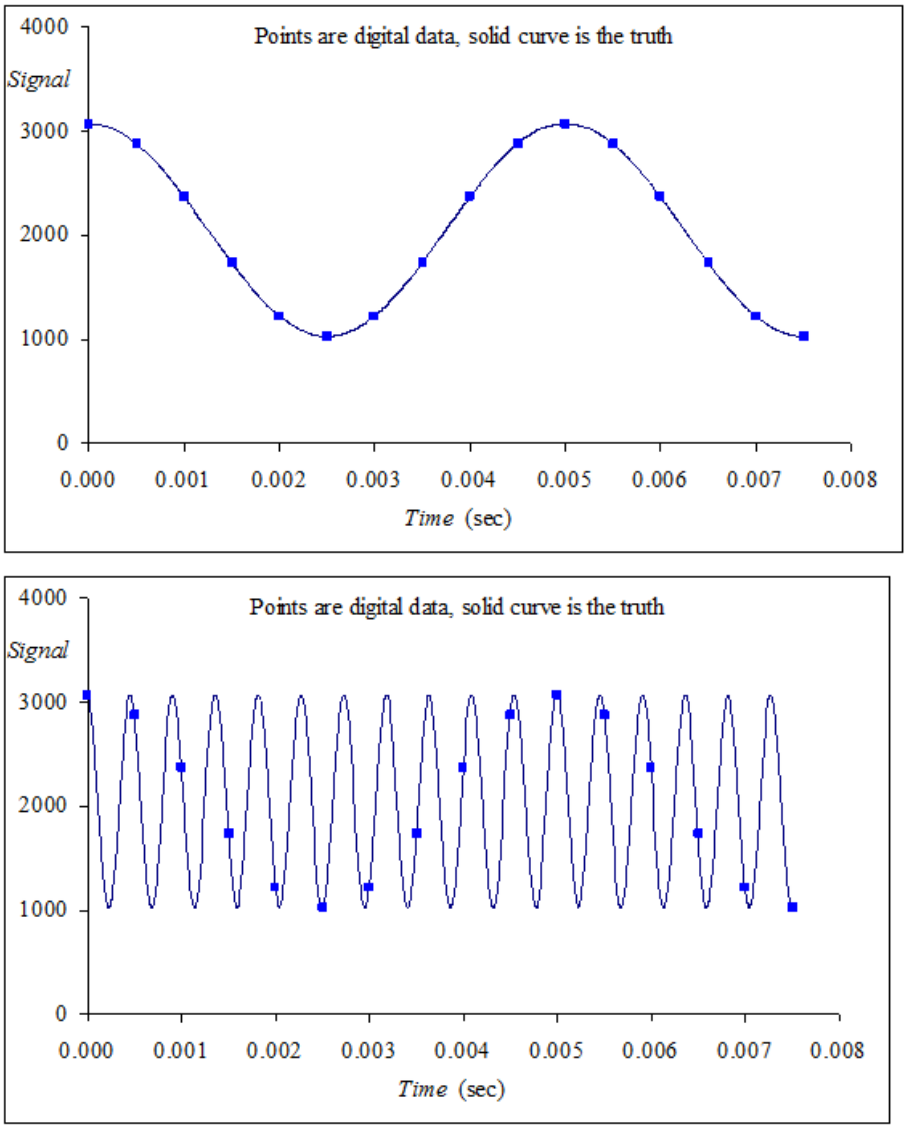


Figure 14.4. Aliasing occurs when the input analog signal oscillates faster than the rate of the ADC sampling.

Valvano Postulate: If f_{max} is the largest frequency component of the analog signal, then you must sample more than ten times f_{max} in order for the reconstructed digital samples to look like the original signal when plotted on a voltage versus time graph.

14.4. Data Acquisition and Control Systems

The **measurand** is a real world signal of interest like sound, distance, temperature, force, mass, pressure, flow, light and acceleration. Figure 14.5 shows the data flow graph for a data acquisition system or control system. The **control system** uses an actuator to drive a measurand in the real world to a desired value while the **data acquisition system** has no actuator because it simply measures the measurand in a nonintrusive manner.

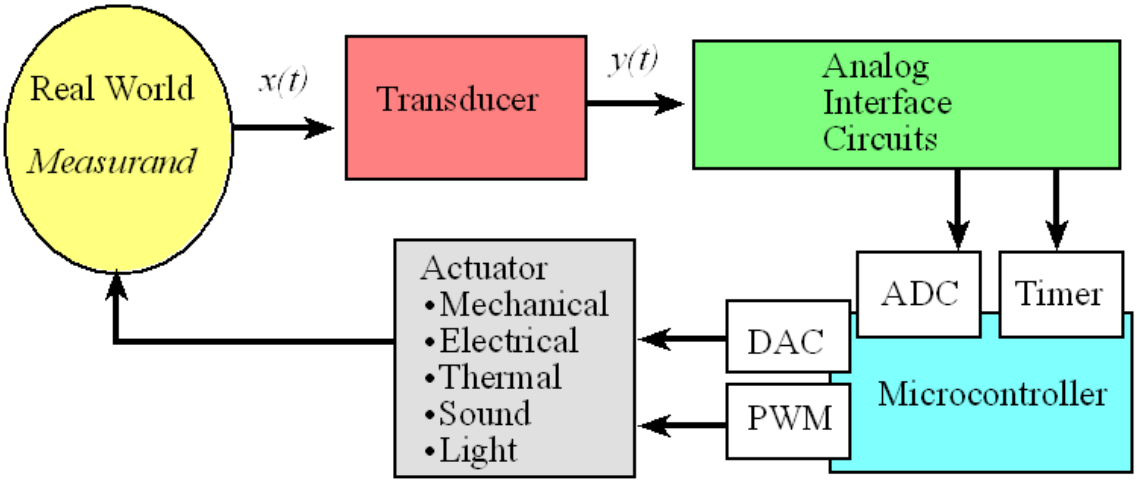


Figure 14.5. Signal paths a data acquisition system.

The input or measurand is *x*. The output is *y*. A **transducer** converts *x* into *y*. Examples include

- Sound — Microphone
- Pressure, mass, force — Strain gauge, force sensitive resistor
- Temperature — Thermistor, thermocouple, integrated circuits
- Distance — Ultrasound, lasers, infrared light
- Flow — Doppler ultrasound, flow probe
- Acceleration — Accelerometer
- Light — Camera
- Biopotentials — Silver-Silver Chloride electrode

A **nonmonotonic** transducer is an input/output function that does not have a mathematical inverse. For example, if two or more input values yield the same output value, then the transducer is nonmonotonic. Software will have a difficult time correcting a nonmonotonic transducer. For example, the Sharp GP2Y0A21YK IR distance sensor has a transfer function as shown in Figure 14.6. If you read a transducer voltage of 2 V, you cannot tell if the object is 3 cm away or 12 cm away. However, if we assume the distance is always greater than 10cm, then this transducer can be used.

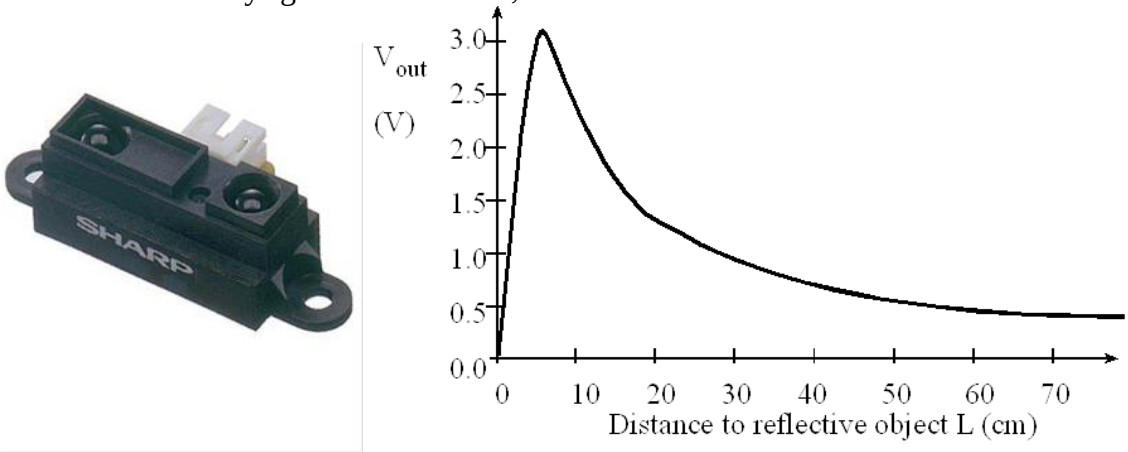


Figure 14.6. The Sharp IR distance sensor exhibits nonmonotonic behavior.

Details about transducers and actuators can be found in Embedded Systems: Real-Time Interfacing to ARM® Cortex™-M Microcontrollers, 2014, ISBN: 978-1463590154.

14.5. Robot Car Controller

The goal is to drive a robot car autonomously down a road. Autonomous driving is a difficult problem, and we have greatly simplified it and will use this simple problem to illustrate the components of a control system. Every control system has real-world parameters that it wishes to control. These parameters are called **state variables**. In our system we wish to drive down the middle of the road, so our state variables will be the distance to the left side of the road and the distance to the right side of the road as illustrated in Figure 14.7. When we are in the middle of the road these two distances will be equal. So, let's define *Error* as:

$$Error = D_{left} - D_{right}$$

If *Error* is zero we are in the middle of the road, so the controller will attempt to drive the *Error* parameter to zero.

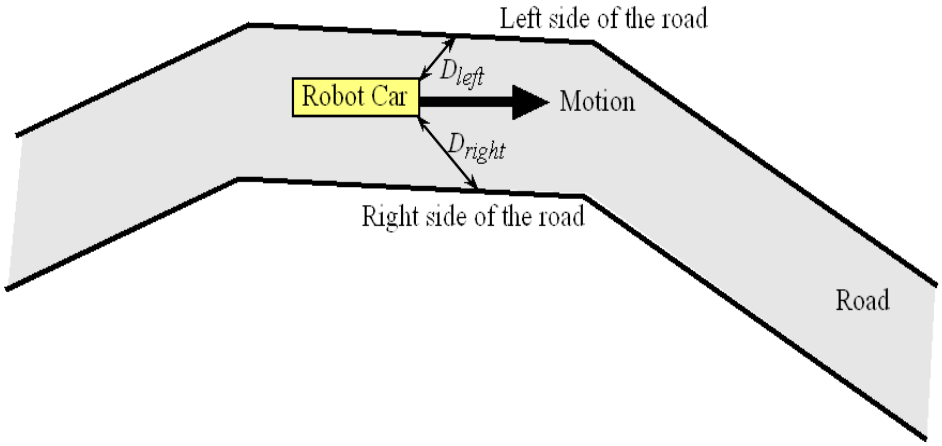


Figure 14.7. Physical layout of the autonomous robot as is drives down the road.

We will need sensors and a data acquisition system to measure D_{left} and D_{right}. In order to simplify the problem we will place pieces of wood to create walls along both sides of the road, and make the road the same width at all places along the track. The Sharp GP2Y0A21YK0F infrared object detector can measure distance from the robot to the wood. This sensor creates a continuous analog voltage between 0 and +3V that depends inversely on distance to object, see Figure 14.6. We will avoid the 0 to 10 cm range where the sensor has the nonmonotonic behavior. We will use two ADC channels (PE4 and PE5) to convert the two analog voltages to digital numbers. Let **`Left`** and **`Right`** be the ADC digital samples measured from the two sensors. We can assume distance is linearly related to 1/voltage, we can implement software functions to calculate distance in mm as a function of the ADC sample (0 to 4095). The 241814 constant was found empirically, which means we collected data comparing actual distance to measured ADC values.

`Dleft` = 241814/**`Left`**
`Dright` = 241814/**`Right`**

Figure 14.8 shows the accuracy of this data acquisition system, where the estimated distance, using the above equation, is plotted versus the true distance.

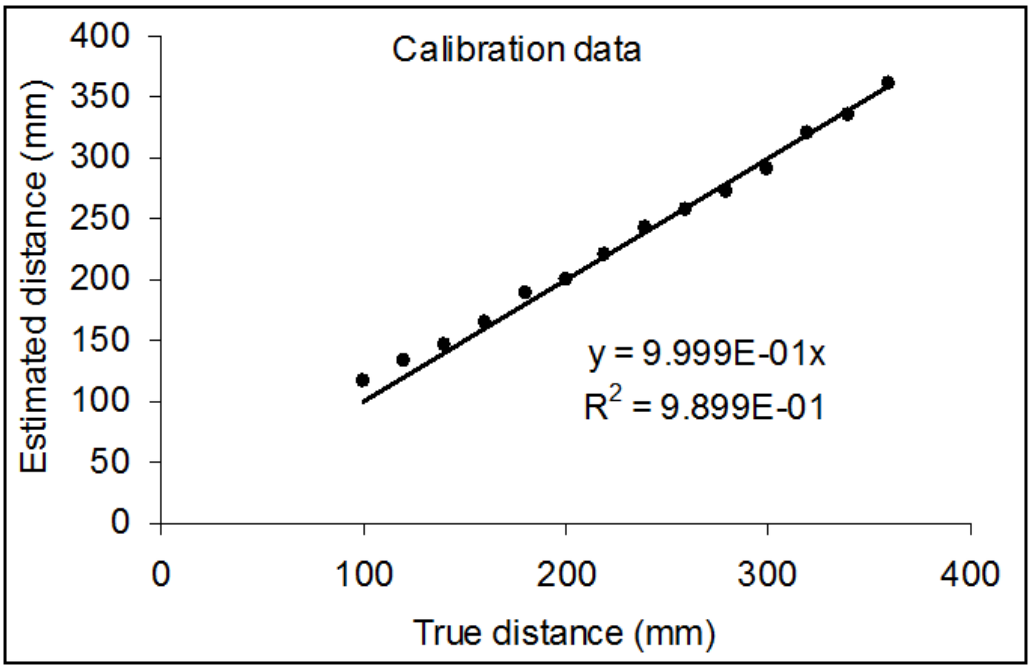


Figure 14.8. Measurement accuracy of the Sharp GP2Y0A21YK0F distance sensor used to measure distance to wall.

Next we need to extend the robot built in Example 12.2. First we build two motor drivers and connect one to each wheel, as shown in Figure 14.9. There will be two PWM outputs: PA6 controls the right motor attached to the right wheel, and PA5 controls the left motor attached to the left wheel. The motors are classified as actuators because they exert force on the world. Similar to Example 12.2 we will write software to create two PWM outputs so we can independently adjust power to each motor. If the friction is constant, the resistance of the motor, *R*, will be fixed and the power is

$$Power = (8.4^2/R)*H/(H+L)$$

When creating PWM, the period (H+L) is fixed and the duty cycle is varied by changing H. So we see the robot controller changes H, it has a linear effect on delivered power to the motor.

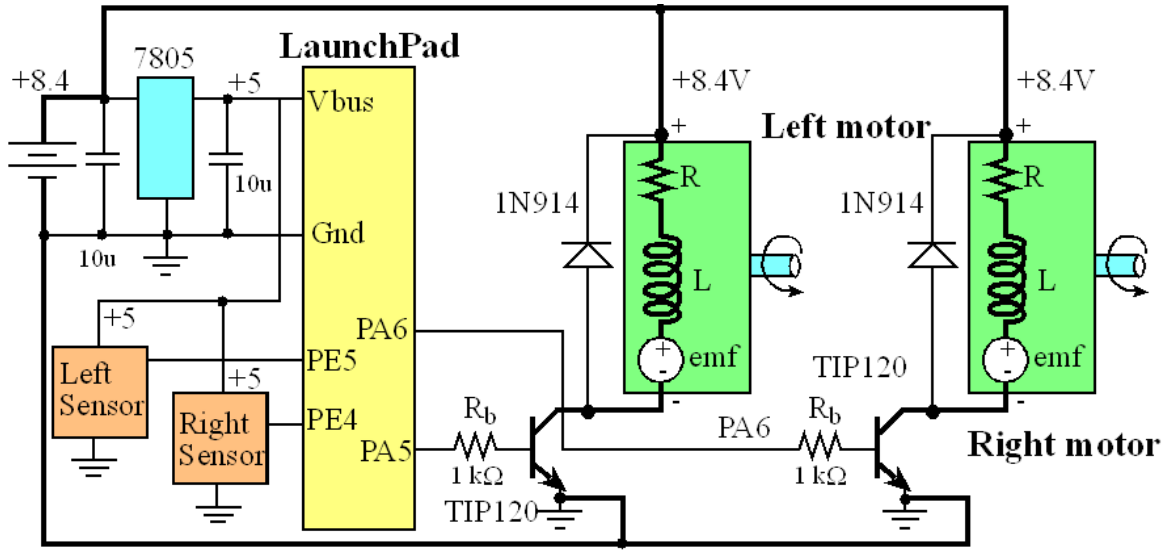


Figure 14.9. Circuit diagram of the robot car. One motor is wire reversed from the other, because to move forward one motor must spin clockwise while the other spins counterclockwise.

The currents can range from 500mA to 1 A, so TIP120 Darlington transistors are used, because they can sink up to 3 A. Notice the dark black lines in Figure 14.9; these lines signify the paths of these large

currents. Notice also the currents do not pass into or out of the LaunchPad. Figure 14.10 shows the robot car. The two IR sensors are positioned in the front at about 45 degrees.

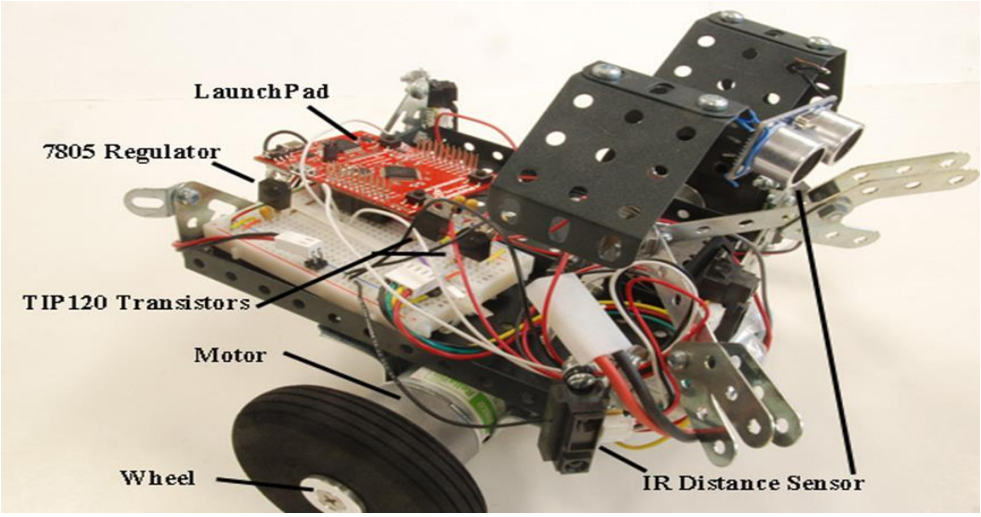


Figure 14.10. Photo of the robot car.

Figure 14.11 illustrates the feedback loop of the control system. The state variables are Dleft and Dright.

The two sensors create voltages that depend on these two state variables. The ADC samples these two voltages, and software calculates the estimates **`Dleft`** and **`Dright`**. **`Error`** is the difference between **`Dleft`** and **`Dright`**. The right motor is powered with a constant duty cycle of 40%, while the duty cycle of the left motor is adjusted in an attempt to drive down the middle of the road. We will constrain the duty cycle of the left motor to between 30% and 50%, so it doesn't over compensate and spin in circles. If the robot is closer to the left wall (**`Dleft`** < **`Dright`**) the error will be negative and more power will be applied to the left motor, turning it right. Conversely, if the robot is closer to the right wall (**`Dleft`**> **`Dright`**) the error will be positive and less power will be applied to the left motor, turning it left. Once the robot is in the middle of the road, error will be zero, and power will not be changed. This control algorithm can be written as a set of simple equations. The number "200" is the controller gain and is found by trial and error once the robot is placed on the road. If it is slow to react, then we increase gain. If it reacts too quickly, we decrease the gain.

```
Error = Dleft - Dright
LeftH = LeftH – 200*Error;
if(LeftH < 30*800) LeftH=30*800;  // 30% min
if(LeftH > 50*800) LeftH=50*800;  // 50% max
LeftL = 80000 - LeftH;            // constant period
```

Observation: In the field of control systems, a popular approach is called PID control, which stands for proportional integral derivative. The above simple algorithm actually implements the integral term of a PID controller. Furthermore, the two **if** statements in the control software implement a feature called **anti-reset windup**.

These controller equations are executed in the SysTick ISR so the controller runs at a periodic rate.

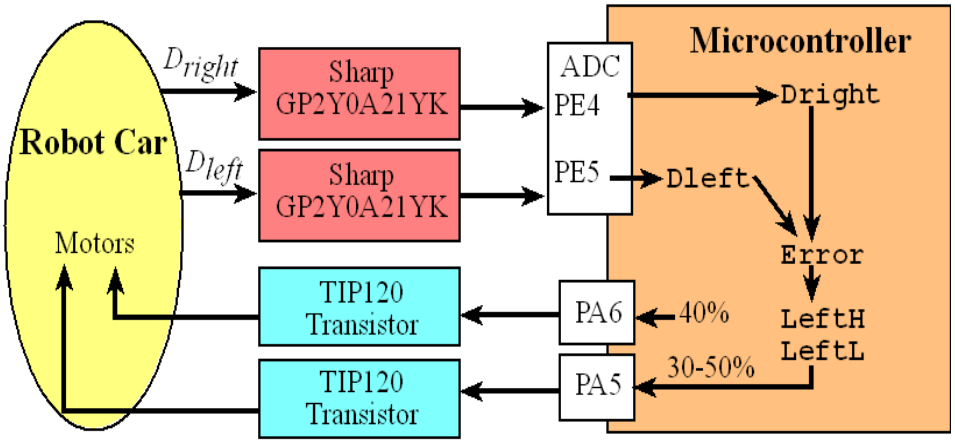


Figure 14.11. Block diagram of the closed loop used in the robot car.

Details about microcontroller-based control systems can be found in Chapter 10 of Embedded Systems: Real-Time Operating Systems for ARM® Cortex-M Microcontrollers.

Bill of Materials

1) Two DC geared motors, HN-GH12-1640Y, GH35GMB-R, Jameco Part no. 164786
- 0.23in or 6 mm shaft (get hubs to match)

2) Metal or wood for base,

3) Hardware for mounting
- 2 motor mounts 1-1/4 in. PVC Conduit Clamps Model # E977GC-CTN Store SKU # 178931
- some way to attach the LaunchPad (I used an Erector set, but you could use rubber bands)

4) Two wheels and two hubs to match the diameter of the motor shaft
- Shepherd 1-1/4 in. Caster Rubber Wheel Model # 9487
- 2 6mm hubs Dave's Hubs - 6mm Hub Set of Two Part# 0-DWH6MM
- 2 3-Inch Diameter Treaded Lite Flite Wheels 2pk Part# 0-DAV5730

5) Two GP2Y0A21YK IR range sensors

6) Battery
- 8.4V NiMH or 11.1V LiIon.

7) Electronic components
- two TIP120 Darlington NPN transistors
-2 1N914 diodes
-2 10uF tantalum caps
- 7805 regular
-2 10k resistors

Chapter 15: Complete Embedded System
Embedded Systems - Shape the World
Jonathan Valvano and Ramesh Yerraballi

So far in this course have presented embedded systems from an interfacing or component level. This chapter will introduce systems level design. The chapter begins with a discussion of requirements document and modular design. Next, we will describe data structures used to represent graphics images. We will conclude this course with a project of building a hand-held game. We will call it a project rather than a lab because we have no automatic grader capable of evaluating a game. However, we will have a mechanism to share games between students.

Learning Objectives:

- Review course contents
- Integrate components into a complete embedded system
- Use structures to organize data
- Introduce graphics
- Build a hand-held game

15.1. Requirements Document

Back in Chapter 7 we presented an outline of a **Requirements Document**. A requirements document states what the system will do. It does not state how the system will do it. The main purpose of a requirements document is to serve as an agreement between you and your clients describing what the system will do. This agreement can become a legally binding contract. We should write the document so that it is easy to read and understand by others. It should be unambiguous, complete, verifiable, and modifiable. In this chapter we will use the framework of the requirements document to describe the hand-held game project.

1. Overview

1.1. Objectives: Why are we doing this project? What is the purpose? The overall objective of this project is to integrate the individual components taught in this class into a single system. More specifically, the objectives of this project are: 1) design, test, and debug a large C program; 2) to review I/O interfacing techniques used in this class; and 3) to design a system that performs a useful task. In particular we will design an 80's-style shoot-em up game like **Space Invaders**.

1.2. Process: How will the project be developed? Similar to the labs, this project will have a starter project, Lab15_SpaceInvaders, which will include some art and sounds to get you started.

1.3. Roles and Responsibilities: Who will do what? Who are the clients? Students may develop their games individually or in teams. An effective team size for this project ranges from 1 to 3 members. There is no upper limit to team size, but above 3 members will present difficulty in communication and decision making. The clients for this project will be other classmates and your professors of the UT.6.01x course.

1.4. Interactions with Existing Systems: How will it fit in? The game must be developed in C on the Keil IDE and run on a Stellaris/Tiva LaunchPad. We expect you to combine your solutions to Lab 8 (switches, LED), Lab 12 (interrupts), Lab 13 (DAC and sounds), and Lab 14 (slide pot and ADC) into one system. We expect everyone to use the slide pot, two switches, two LEDs, one 4-bit DAC, and the Nokia5110 LCD screen. If you do not own a Nokia5110 there will be a mechanism to pass the Nokia graphic commands to the PC and the application TExaSdisplay will show the images in real time as your game software runs on your real LaunchPad.

1.5. Terminology: Define terms used in the document. **BMP** is a simple file format to store graphical images. A **sprite** is a virtual entity that is created, moves around the screen, and might disappear. A **public** function is one that can be called by another module. For example if the main program calls **`Sound_Play`**, then **`Sound_Play`** is a public function.

1.6. Security: How will intellectual property be managed? Since this project does not contribute to the final grade in UT.6.01x you do not need to upload your solution. If you do upload your source code, then other students who have uploaded will be able to see and download your source code. To reduce the chance of spreading viruses, we will restrict the upload to a single text-formatted source code file. More specifically, the upload must be a `SpaceInvaders.c` file, and this file must compile within a project like the `Lab15_SpaceInvaders` starter project within the 32k-limit of the free version of the Keil IDE.

2. Function Description

2.1. Functionality: What will the system do precisely? You will design, implement and debug an 80's or 90's-style video game. You are free to simplify the rules but your game should be recognizable as one of these six simple games: Space Invaders, Asteroids, Missile Command, Centipede, Snood, or Defender. Buttons, and the slide pot are inputs. The LCD, LEDs, and sound (Lab 13) are the outputs. The slide pot is a simple yet effective means to move your ship. Interrupts must be appropriately used control the input/output, and will make a profound impact on how the user interacts with the game. You could use an edge-triggered interrupt to execute software whenever a button is pressed. You could create two periodic interrupts. Use one fixed-frequency periodic interrupt to output sounds with the DAC. You could decide to move a sprite using a second periodic interrupt, although the actual LCD output should always be performed in the main program.

2.2. Scope: List the phases and what will be delivered in each phase. The first phase is forming a team and defining the exact rules of game play. You next will specify the modules: e.g., the main game engine, a module to input from switches, a module to output to LEDs, a module to draw images on the Nokia, and a module that inputs from the slide pot. Next you will design the prototypes for the public functions. At this phase of the project, individual team members can develop and test modules concurrently. The last phase of the project is to combine the modules to create the overall system.

2.3. Prototypes: How will intermediate progress be demonstrated? In a system such as this each module must be individually tested. Your system will have four or more modules. Each module has a separate header and code file. For each module create a header file, a code file and a separate main program to test that particular module. After the game is completed, you will create a single C file that compiles within the starter project to be uploaded for others to play.

2.4. Performance: Define the measures and describe how they will be determined. The game should be easy to learn, and fun to play. You will have the option to share your game with other students in the class.

2.5. Usability: Describe the interfaces. Be quantitative if possible. In order to allow other students to download, compile and run your game you must follow strict requirements for the pins used for input and output. These requirements are detailed at the top of the `SpaceInvaders.c` file within the starter project.

2.6. Safety: Explain any safety requirements and how they will be measured. To reduce the chance of spreading viruses we will only allow text files with C code to be uploaded. The usual rules about respect and tolerance as defined for the forums apply as well to the output of the video games.

3. Deliverables

3.1. Reports: How will the system be described? Add comments to the top of your C file to explain the purpose and functionality of your game.

3.2. Audits: How will the clients evaluate progress? There will be a discussion forum that will allow you to evaluate the performance (easy to learn, fun to play) of the other games.

3.3. Outcomes: What are the deliverables? How do we know when it is done? You will upload a single C file. We will disable uploads and downloads on the last class day.

15.2. Modular Design

The design process involves the conversion of a problem statement into hardware and software components. Successive refinement is the transformation from the general to the specific. In this section, we introduce the concept of modular programming and demonstrate that it is an effective way to organize our software projects. There are four reasons for forming modules. First, functional abstraction allows us to reuse a software module from multiple locations. Second, complexity abstraction allows us to divide a highly complex system into smaller less complicated components. The third reason is portability. If we create modules for the I/O devices, then we can isolate the rest of the system from the hardware details. This approach is sometimes called a hardware abstraction layer. Since all the software components that access an I/O port are grouped together, it will be easier to redesign the embedded system on a machine with different I/O ports. Finally, another reason for forming modules is security. Modular systems by design hide the inner workings from other modules and provide a strict set of mechanisms to access data and I/O ports. Hiding details and restricting access generates a more secure system.

Software must deal with **complexity**. Most real systems have many components, which interact in a complex manner. The size and interactions will make it difficult to conceptualize, abstract, visualize, and document. In this chapter we will present data flow graphs and call graphs as tools to describe interactions between components. Software must deal with **conformity**. All design, including software design, must interface with existing systems and with systems yet to be designed. Interfacing with existing systems creates an additional complexity. Software must deal with **changeability**. Most of the design effort involves change. Creating systems that are easy to change will help manage the rapid growth occurring in the computer industry.

The key to completing any complex task is to break it down into manageable subtasks. Modular programming is a style of software development that divides the software problem into distinct well-defined modules. The parts are as small as possible, yet relatively independent. Complex systems designed in a modular fashion are easier to debug because each module can be tested separately. Industry experts estimate that 50 to 90% of software development cost is spent in maintenance. All five aspects of software maintenance

- Correcting mistakes,
- Adding new features,
- Optimizing for execution speed or program size,
- Porting to new computers or operating systems, and
- Reconfiguring the software to solve a similar related program

are simplified by organizing the software system into modules. The approach is particularly useful when a task is large enough to require several programmers.

A **program module** is a self-contained software task with clear entry and exit points. There is a distinct difference between a module and a C language function. A module is usually a collection of functions that in its entirety performs a well-defined set of tasks. A collection of 32-bit trigonometry functions is an example of a module. A device driver is a software module that facilitates the use of I/O. In particular it is collection of software functions for a particular I/O device. Modular programming involves both the specification of the individual modules and the connection scheme whereby the modules are interfaced together to form the software system. While the module may be called from many locations throughout the software, there should be well-defined **entry points**. In C, the entry point of a module is defined in the header file and is specified by a list of function prototypes for the public functions.

Common Error: In many situations the input parameters have a restricted range. It would be inefficient for the module and the calling routine to both check for valid input. On the other hand, an error may occur if neither checks for valid input.

An **exit point** is the ending point of a program module. The exit point of a function is used to return to the calling routine. We need to be careful about exit points. Similarly, if the function returns parameters, then all exit points should return parameters in an acceptable format. If the main program has an exit point it either stops the program or returns to the debugger. In most embedded systems, the main program does not exit.

In this section, an object refers to either a function or a data element. A **public** object is one that is shared by multiple modules. This means a public object can be accessed by other modules. Typically, we make the most general functions of a module public, so the functions can be called from other modules. For a module performing I/O, typical public functions include initialization, input, and output. A **private** object is one that is not shared. I.e., a private object can be accessed by only one module. Typically, we make the internal workings of a module private, so we hide how a private function works from user of the module. In an object-oriented language like C++ or Java, the programmer clearly defines a function or data object as public or private. The software in this course uses the naming convention of using the module name followed by an underline to identify the public functions of a module. For example if the module is ADC, then **`ADC_Init`** and **`ADC_Input`** are public functions. Functions without the underline in its name are private. In this manner we can easily identify whether a function or data object as public or private.

At a first glance, I/O devices seem to be public. For example, Port D resides permanently at the fixed address of 0x400073FC, and the programmer of every module knows that. In other words, from a syntactic viewpoint, any module has access to any I/O device. However, in order to reduce the complexity of the system, we will restrict the number of modules that actually do access the I/O device. From a "what do we actually do" perspective, however, we will write software that considers I/O devices as private, meaning an *I/O device should be accessed by only one module*. In general, it will be important to clarify which modules have access to I/O devices and when they are allowed to access them. When more than one module accesses an I/O device, then it is important to develop ways to arbitrate or synchronize. If two or more want to access the device simultaneously arbitration determines which module goes first. Sometimes the order of access matters, so we use synchronization to force a second module to wait until the first module is finished. Most microcontrollers do not have architectural features that restrict access to I/O ports, because it is assumed that all software burned into its ROM was designed for a common goal, meaning from a security standpoint one can assume there are no malicious components. However, as embedded systems become connected to the Internet, providing the power and flexibility, security will become important issue.

Checkpoint 15.1: Multiple modules may use Port F, where each module has an initialization. What conflict could arise around the initialization of a port?

Information hiding is similar to minimizing coupling. It is better to separate the mechanisms of software from its policies. We should separate "what the function does" from "how the function works". What a function does is defined by the relationship between its inputs and outputs. It is good to hide certain inner workings of a module and simply interface with the other modules through the well-defined input/output parameters. For example we could implement a variable size buffer by maintaining the current byte count in a global variable, **`Count`**. A good module will hide how **`Count`** is implemented from its users. If the user wants to know how many bytes are in the buffer, it calls a function that returns the count. A badly written module will not hide **`Count`** from its users. The user simply accesses the global variable **`Count`**. If we update the buffer routines, making them faster or

better, we might have to update all the programs that access **`Count`** too. Allowing all software to access **`Count`** creates a security risk, making the system vulnerable to malicious or incompetent software. The object-oriented programming environments provide well-defined mechanisms to support information hiding. This separation of policies from mechanisms is discussed further in the section on layered software.

Maintenance Tip: It is good practice to make all permanently-allocated data and all I/O devices private. Information is transferred from one module to another through well-defined function calls.

The **Keep It Simple Stupid** approach tries to generalize the problem so that the solution uses an abstract model. Unfortunately, the person who defines the software specifications may not understand the implications and alternatives. As a software developer, we always ask ourselves these questions:

"How important is this feature?"
"What if it worked this different way?"

Sometimes we can restate the problem to allow for a simpler and possibly more powerful solution. We begin the design of the game by listing possible modules for our system.

ADC	The interface to the joystick
Switch	User interaction with LEDs and switches
Sound	Sound output using the DAC
Nokia5110	Images displayed on the LCD
Game engine	The central controller that implements the game

Figure 15.1 shows a possible **call graph** for the game. An arrow in a call graph means software in one module can call functions in another module. This is a very simple organization with one master module and four slave modules. Notice the slave modules do not call each other. This configuration is an example of good modularization because there are 5 modules but only 4 arrows.

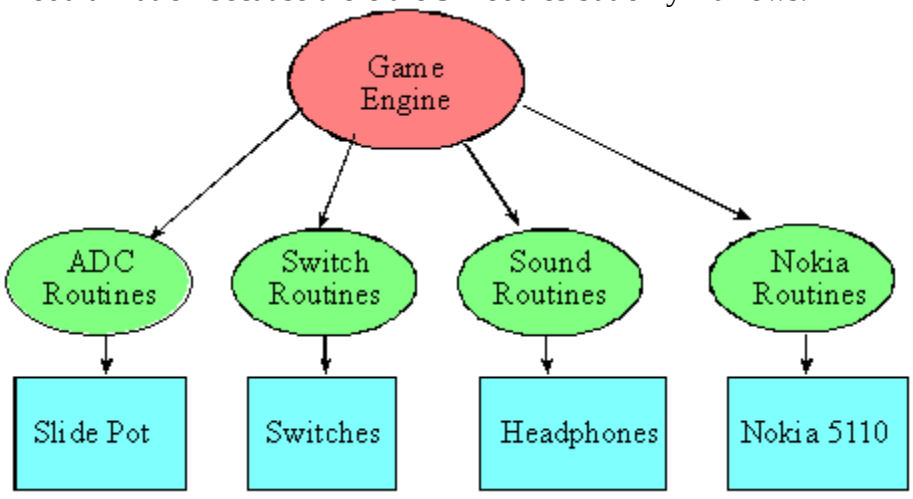


Figure 15.1. Possible call graph for the game.

Figure 15.2 shows on possible data flow graph for the game. Recall that arrows in a data flow graph represent data passing from one module to another. Notice the high bandwidth communication occurs between the sound module and its hardware, and between the Nokia5110 module and its hardware. We will design the system such that software modules do not need to pass a lot of data to other software modules.

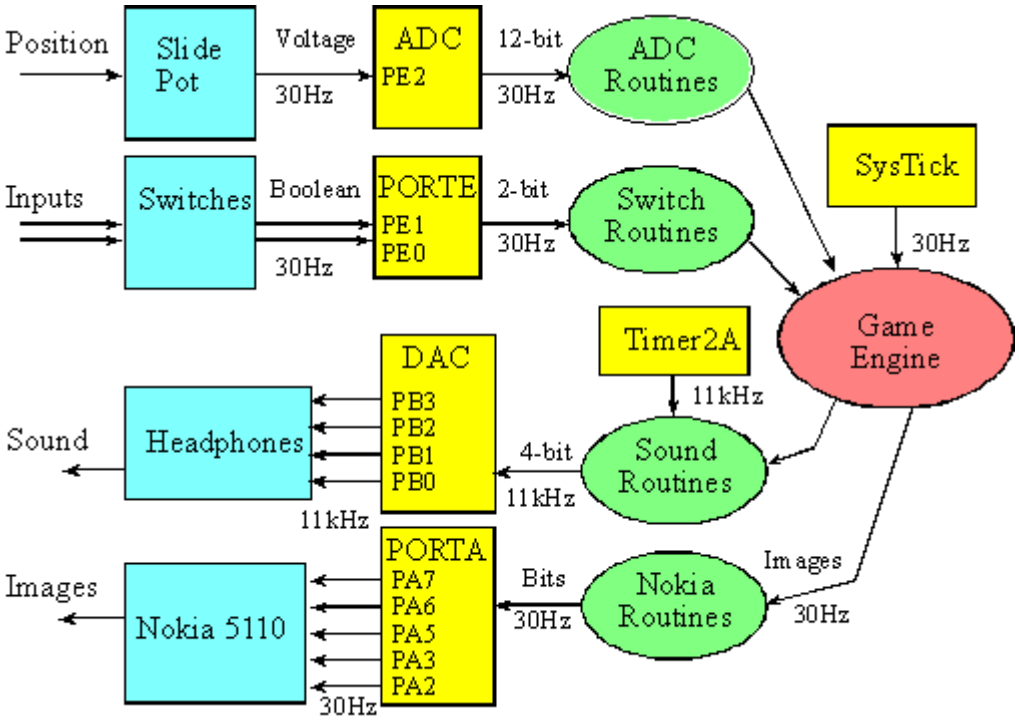


Figure 15.2. Possible data flow graph for the game. If you wish to add LEDs place them on PB4 and PB5.

The Timer2A ISR will output a sequence of numbers to the DAC to create sound. Let **`explosion`** be an array of 2000 4-bit numbers, representing a sound sampled at 11 kHz. If the game engine wishes to make the explosion sound, it calls **`Sound_Play(explosion,2000);`** This function call simply passes a pointer to the **`explosion`** sound array into the sound module. The Timer2A ISR will output one 4-bit number to the DAC for the next 2000 interrupts. Notice the data flow from the game engine to the sound module is only two parameters (pointer and count), causing 2000 4-bit numbers to flow from the sound module to the DAC.

The Nokia5110 module needs to send 504 bytes to the LCD to create a new image on the screen. Since the screen is updated 30 times per second, 15120 bytes/sec will flow from the Nokia5110 module to its hardware. Let **`SmallEnemy30PointA`** be an array of numbers, representing a 16 by 10 pixel image of a small enemy. If the game engine wishes to place this enemy in the center of the screen, it calls **`Nokia5110_PrintBMP(24,48,SmallEnemy30PointA,0);`** This function call simply passes four parameters, one of which is a pointer to the image array into the Nokia5110 module. If the enemy is moving, then 15120 bytes/sec are flowing from the Nokia5110 module to the LCD, but the data flow from the game engine to the Nokia5110 module is only the four parameters (location, pointer and threshold) 30 times per second, which is 16 bytes*30/sec = 480 bytes/sec. The data flow from the game engine to the Nokia5110 module will increase linearly with the number of objects moving on the screen, but remain much smaller than the data into the LCD hardware.

Figure 15.3 shows on possible flow chart for the game engine. It is important to perform the actual LCD output in the foreground. In this design there are three threads: the main program and two interrupts. Multithreading allows the processor to execute multiple tasks. The main loop performs the game engine and updates the image on the screen. At 30 Hz, which is fast enough to look continuous, the SysTick ISR will sample the ADC and switch inputs. Based on user input and the game function, the ISR will decide what actions to take and signal the main program. To play a sound, we send the Sound module an array of data and arm Timer2A. Each Timer2A interrupt outputs one value to the DAC. When the sound is over we disarm Timer2A.

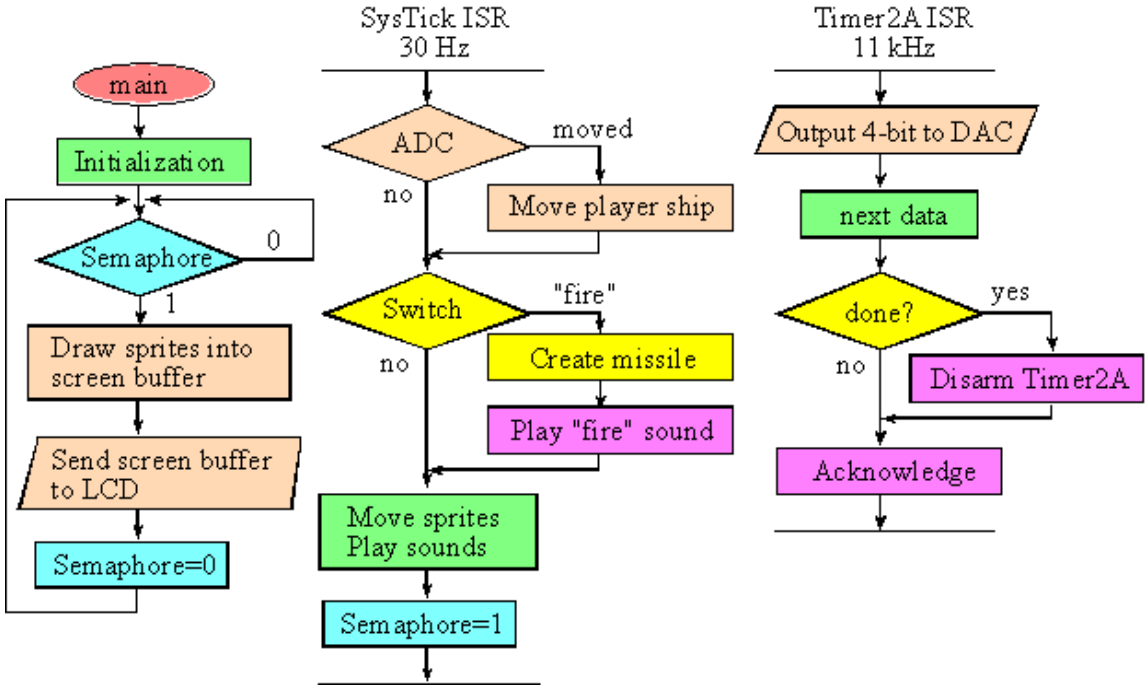


Figure 15.3. Possible flowchart for the game.

For example, if the ADC notices a motion to the left, the SysTick ISR can tell the main program to move the player ship to the left. Similarly, if the SysTick ISR notices the fire button has been pushed, it can create a missile object, and for the next 100 or so interrupts the SysTick ISR will move the missile until it goes off screen or hits something. In this way the missile moves a pixel or two every 33.3ms, causing its motion to look continuous. In summary, the ISR responds to input and time, but the main loop performs the actual output to the LCD.

Checkpoint 15.2: Notice the algorithm ins Figure 15.3 samples the ADC and the fire button at 30 Hz. How times/sec can we fire a missile or wiggle the slide pot? Hint: think Nyquist Theorem.
Checkpoint 15.3: Similarly, in Figure 15.3, what frequency components are in the sound output?

15.3. Introduction to Graphics

A **matrix** is a two-dimensional data structure accessed by row and column. Each element of a matrix is the same type and precision. In C, we create matrices using two sets of brackets. Figure 15.4 shows this byte matrix with six 8-bit elements. The figure also shows two possible ways to map the two-dimensional data structure into the linear address space of memory.

```
unsigned char M[2][3]; // byte matrix with 2 rows and 3 columns
```

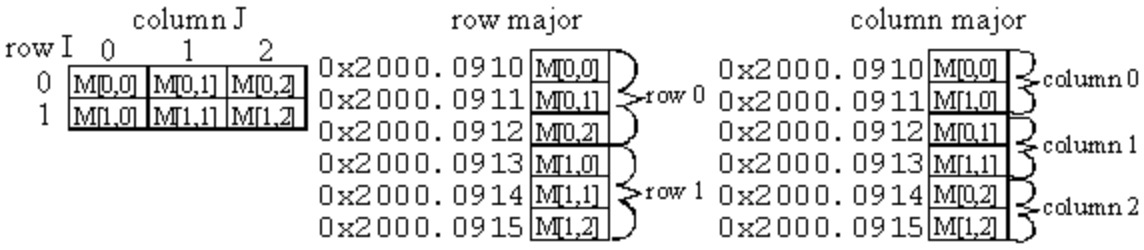


Figure 15.4. A byte matrix with 2 rows and 3 columns.

With row-major allocation, the elements of each row are stored together. Let **`i`** be the row index, **`j`** be the column index, **`n`** be the number of bytes in each row (equal to the number of columns), and **`Base`** is the base address of the byte matrix, then the address of the element at **`i,j`** is

```
Base+n*i+j
```

With a halfword matrix, each element requires two bytes of storage. Let **i** be the row index, **j** be the column index, **n** be the number of halfwords in each row (equal to the number of columns), and **Base** is the base address of the word matrix, then the address of the element at **i,j** is

```
Base+2*(n*i+j)
```

With a word matrix, each element requires four bytes of storage. Let **i** be the row index, **j** be the column index, **n** be the number of words in each row (equal to the number of columns), and **Base** is the base address of the word matrix, then the address of the element at **i,j** is

```
Base+4*(n*i+j)
```

As an example of a matrix, we will develop a set of driver functions to manipulate a 48 by 84 by 1-bit graphics display, see Figure 15.5.

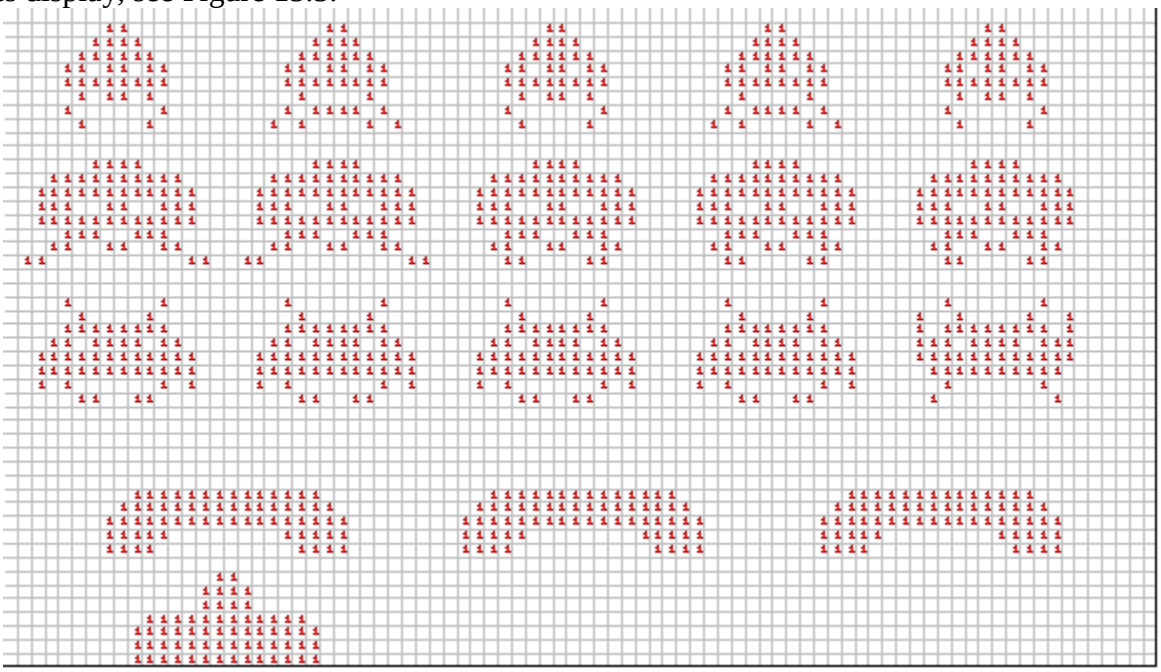


Figure 15.5. A 1-bit matrix with 48 rows and 84 columns, each pixel is 1 bit.

Placing a 0 into a pixel location will display that pixel in a color ranging from off (0) and a 1 is fully on. In this display, the first bit is the top left corner of the display, and the last bit is the bottom right corner. The graphical image on this 48 by 84 display will be stored in the 1-bit array called **Screen**. Since there are a total of 4032 pixels, and each byte can store 8 pixels, we need 504 bytes to store the entire image. In C, we define the following in global RAM,

```
char Screen[504]; // stores the next image to be printed on the screen
//********Nokia5110_Init*****************
// Initialize Nokia 5110 48x84 LCD by sending the proper
// commands to the PCD8544 driver.
// inputs: none
// outputs: none
// assumes: system clock rate of 50 MHz or less
void Nokia5110_Init(void);

//********Nokia5110_OutChar*****************
// Print a character to the Nokia 5110 48x84 LCD.  The
// character will be printed at the current cursor position,
// the cursor will automatically be updated, and it will
// wrap to the next row or back to the top if necessary.
// One blank column of pixels will be printed on either side
```

```
// of the character for readability.  Since characters are 8
// pixels tall and 5 pixels wide, 12 characters fit per row,
// and there are six rows.
// inputs: data  character to print
// outputs: none
// assumes: LCD is in default horizontal addressing mode (V = 0)
void Nokia5110_OutChar(unsigned char data);

//********Nokia5110_OutString*****************
// Print a string of characters to the Nokia 5110 48x84 LCD.
// The string will automatically wrap, so padding spaces may
// be needed to make the output look optimal.
// inputs: ptr  pointer to NULL-terminated ASCII string
// outputs: none
// assumes: LCD is in default horizontal addressing mode (V = 0)
void Nokia5110_OutString(char *ptr);

//********Nokia5110_OutUDec*****************
// Output a 16-bit number in unsigned decimal format with a
// fixed size of five right-justified digits of output.
// Inputs: n  16-bit unsigned number
// Outputs: none
// assumes: LCD is in default horizontal addressing mode (V = 0)
void Nokia5110_OutUDec(unsigned short n);

//********Nokia5110_SetCursor*****************
// Move the cursor to the desired X- and Y-position.  The
// next character will be printed here.  X=0 is the leftmost
// column.  Y=0 is the top row.
// inputs: newX  new X-position of the cursor (0<=newX<=11)
//         newY  new Y-position of the cursor (0<=newY<=5)
// outputs: none
void Nokia5110_SetCursor(unsigned char newX, unsigned char newY);

//********Nokia5110_Clear*****************
// Clear the LCD by writing zeros to the entire screen and
// reset the cursor to (0,0) (top left corner of screen).
// inputs: none
// outputs: none
void Nokia5110_Clear(void);

//********Nokia5110_DrawFullImage*****************
// Fill the whole screen by drawing a 48x84 bitmap image.
// inputs: ptr  pointer to 504 byte bitmap
// outputs: none
// assumes: LCD is in default horizontal addressing mode (V = 0)
void Nokia5110_DrawFullImage(const char *ptr);

//********Nokia5110_PrintBMP*****************
// Bitmaps contain their header data and may contain padding
// to preserve 4-byte alignment.  This function takes a
// bitmap in the previously described format and puts its
// image data in the proper location in the buffer so the
// image will appear on the screen after the next call to
//   Nokia5110_DisplayBuffer();
// inputs: xpos      horizontal position of bottom left corner of image,
//                     columns from the left edge
//                     must be less than 84
//                     0 is on the left; 82 is near the right
//         ypos      vertical position of bottom left corner of image,
//                     rows from the top edge
//                     must be less than 48
//                     2 is near the top; 47 is at the bottom
```

```
//         ptr       pointer to a 16 color BMP image
//         threshold grayscale colors above this number make pixel 'on'
//                     0 to 14
//                 0 is fine for ships, explosions, projectiles, and bunkers
// outputs: none
void Nokia5110_PrintBMP(unsigned char xpos, unsigned char ypos, const unsigned char
*ptr, unsigned char threshold);

// There is a buffer in RAM that holds one screen
// This routine clears this buffer
void Nokia5110_ClearBuffer(void);

//********Nokia5110_DisplayBuffer*****************
// Fill the whole screen by drawing a 48x84 screen image.
// inputs: none
// outputs: none
// assumes: LCD is in default horizontal addressing mode (V = 0)
void Nokia5110_DisplayBuffer(void);
```

Program 15.1. Functions that display images on the LCD.

In the game industry an entity that moves around the screen is called a *sprite*. You will find lots of sprites in the Lab15Files directory of the starter project. You can create additional sprites as needed using a drawing program like Paint. Most students will be able to complete the project using only the existing sprites in the starter package. Because of the way pixels are packed onto the screen, we will limit the placing of sprites to even addresses along the x-axis. Sprites can be placed at any position along the y-axis. Having a 2-pixel black border on the left and right of the image will simplify moving the sprite 2 pixels to the left and right without needing to erase it. Similarly having a 1-pixel black border on the top and bottom of the image will simplify moving the sprite 1 pixel up or down without needing to erase it. You can create your own sprites using Paint by saving the images as 16-color BMP images. Figure 15.6 is an example BMP image. Because of the black border, this image can be moved left/right 2 pixels, or up/down 1 pixel. Use the **BmpConvert.exe** program to convert the BMP image into a two-dimensional array that can be displayed on the LCD using the function **`Nokia5110_PrintBMP()`**. To build an interactive game, you will need to write programs for drawing and animating your sprites.

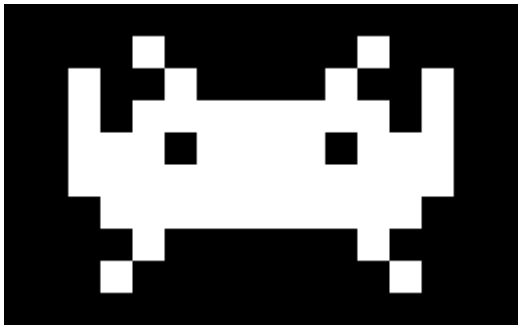


Figure 15.6. Example BMP file. Each is 16-color, 16 pixels wide by 10 pixels high.

Program 15.2 shows an example BMP file in C program format. There are 0x76 bytes of header data. At locations 0x12-0x15 is the width in little endian format. In this case (shown in blue) the width for this sprite is 16 pixels. At locations 0x16-0x19 is the height also in little endian format.

```
const unsigned char Enemy10Point1[] = {
0x42,0x4D,0xC6,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x76,0x00,0x00,0x00,0x28,0x00,
0x00,0x00,
0x10,0x00,0x00,0x00,  // width is 16 pixels
0x0A,0x00,0x00,0x00,  // height is 16 pixels
0x01,0x00,0x04,0x00,0x00,0x00,
0x00,0x00,0x50,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x80,0x00,0x00,0x80,
```

```
0x00,0x00,0x00,0x80,0x80,0x00,0x80,0x00,0x00,0x00,0x80,0x00,0x80,0x00,0x80,0x80,
0x00,0x00,0x80,0x80,0x80,0x00,0xC0,0xC0,0xC0,0x00,0x00,0x00,0xFF,0x00,0x00,0xFF,
0x00,0x00,0x00,0xFF,0xFF,0x00,0xFF,0x00,0x00,0x00,0xFF,0x00,0xFF,0x00,0xFF,0xFF,
0x00,0x00,0xFF,0xFF,0xFF,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,  // bottom row
0x00,0x0F,0x00,0x00,0x00,0x00,0xF0,0x00,
0x00,0x00,0xF0,0x00,0x00,0x0F,0x00,0x00,
0x00,0x0F,0xFF,0xFF,0xFF,0xFF,0xF0,0x00,
0x00,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0x00,
0x00,0xFF,0xF0,0xFF,0xFF,0x0F,0xFF,0x00,
0x00,0xF0,0xFF,0xFF,0xFF,0xFF,0x0F,0x00,
0x00,0xF0,0x0F,0x00,0x00,0xF0,0x0F,0x00,
0x00,0x00,0xF0,0x00,0x00,0x0F,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,  // top row
0xFF};
```

Program 15.2. Example BMP file written as a C constant allocated in ROM.

In this case (shown in red) the height for this sprite is 10 pixels. The **`Nokia5110_PrintBMP()`** function restricts the width to an even number. This function also assumes the entire image will fit onto the screen, and not stick off the side, the top, or the bottom. There is other information in the header, but it is ignored. As mentioned earlier, you must save the BMP as a 16-color image. These sixteen colors will map to the on/off format of the LCD pixels. The 4-bit color 0xF is on and the color 0x0 is off or black. The function takes a threshold parameter to decide whether the colors 1 to 14 will be on or off. Starting in position 0x76, the image data is stored as 4-bit color pixels, with two pixels packed into each byte. Program 15.1 shows the purple data with 16 pixels per line in row-major. If the width of your image is not a multiple of 16 pixels, the BMP format will pad extra bytes into each row so the number of bytes per row is always divisible by 8. In this case, no padding is needed. The **`Nokia5110_PrintBMP()`** function will automatically ignore the padding.

Figure 15.7 shows the data portion of the BMP file as one digit hex with 0's replaced with dots. The 2-D image is stored in row-major format. Notice in Program 15.2 the image is stored up-side down. When plotting it on the screen the **`Nokia5110_PrintBMP()`** function will reverse it so it is seen right-side up.

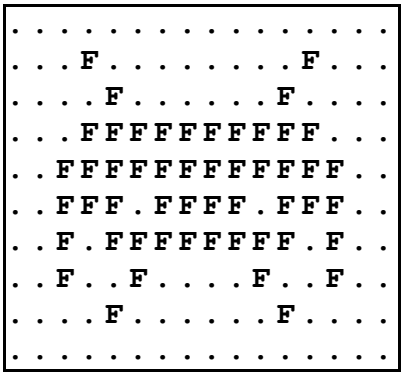


Figure 15.7. The raw data from BMP file to illustrate how the image is stored (0s replaced with dots).

Custom Audio

Say you want to convert a wav file **(blah.wav**) you want to use in your game. Here is a Matlab (or a free alternative to Matlab from GNU called Octave) script file you can use to convert the wav file into a C array declaration that can be used in your code. Run the script by passing it the file as input: **WavConv('blah')**. That's it, you should have a file (called **blah.txt**) with a declaration you can cut and paste in your code. Note that the samples are 4-bit samples to be played at 11.025kHz.

15.4. Using Structures to Organizing Data

When defining the variables used to store the state of the game, we collect the attributes of a virtual object and store/group them together. In C, the **`struct`** allows us to build new data types. In the following example we define a new data type called **`STyp`** that we will use to define sprites.

```
struct State {
  unsigned long x;      // x coordinate
  unsigned long y;      // y coordinate
  const unsigned char *image; // ptr->image
  long life;            // 0=dead, 1=alive
};
typedef struct State STyp;
STyp Enemy[4];
void Init(void){ int i;
  for(i=0;i<4;i++){
    Enemy[i].x = 20*i;
    Enemy[i].y = 10;
    Enemy[i].image = SmallEnemy30PointA;
    Enemy[i].life = 1;
   }
}
void Move(void){ int i;
  for(i=0;i<4;i++){
    if(Enemy[i].x < 72){
      Enemy[i].x += 2;
    }else{
      Enemy[i].life = 0;
    }
  }
}
void Draw(void){ int i;
  Nokia5110_ClearBuffer();
  for(i=0;i<4;i++){
    if(Enemy[i].life > 0){
     Nokia5110_PrintBMP(Enemy[i].x, Enemy[i].y, Enemy[i].image, 0);
    }
  }
  Nokia5110_DisplayBuffer();      // draw buffer
}
int main(void){
  PLL_Init();                   // set system clock to 80 MHz
  Nokia5110_Init();
  Nokia5110_ClearBuffer();
  Init();
  Draw();
  while(1){
    Move();
    Draw();
    Delay100ms(2);
  }
}
```

Program 15.3. Example use of structures (see the file sprite.c).

15.5. Periodic Interrupt using Timer 2A

The TM4C123 has six timers and each timer has two modules, as shown in Figure 15.8. In periodic timer mode the timer is configured as a 32-bit down-counter. When the timer counts from 1 to 0 it sets the trigger flag. On the next count, the timer is reloaded with the value in **`TIMER2_TAILR_R`**. We select periodic timer mode by setting the 2-bit TAMR field of the **`TIMER2_TAMR_R`** to 0x02. In periodic mode the timer runs continuously. The timers can be used to create pulse width modulated outputs and measure pulse width, period, or frequency. For more information on the timers see Chapter

6 of Volume 2 (Embedded Systems: Real-Time Interfacing to ARM® Cortex™-M Microcontrollers, 2014.)

In this section we will use Timer2A to trigger a periodic interrupt. The precision is 32 bits and the resolution will be the bus cycle time of 12.5 ns. This means we could trigger an interrupt as slow as every 2^{32}*12.5ns, which is 53 seconds. The interrupt period will be

(**TIMER2_TAILR_R** +1)*12.5ns

Each periodic timer module has

A clock enable bit, bit 2 in **SYSCTL_RCGCTIMER_R**
A control register, **TIMER2_CTL_R** (set to 0 to disable, 1 to enable)
A configuration register, **TIMER2_CFG_R** (set to 0 for 32-bit mode)
A mode register, **TIMER2_TAMR_R** (set to 2 for periodic mode)
A 32-bit reload register, **TIMER2_TAILR_R**
A resolution register, **TIMER2_TAPR_R** (set to 0 for 12.5ns)
An interrupt clear register, **TIMER2_ICR_R** (bit 0)
An interrupt arm bit, TATOIM, **TIMER2_IM_R** (bit 0)
A flag bit, TATORIS, **TIMER2_RIS_R** (bit 0)

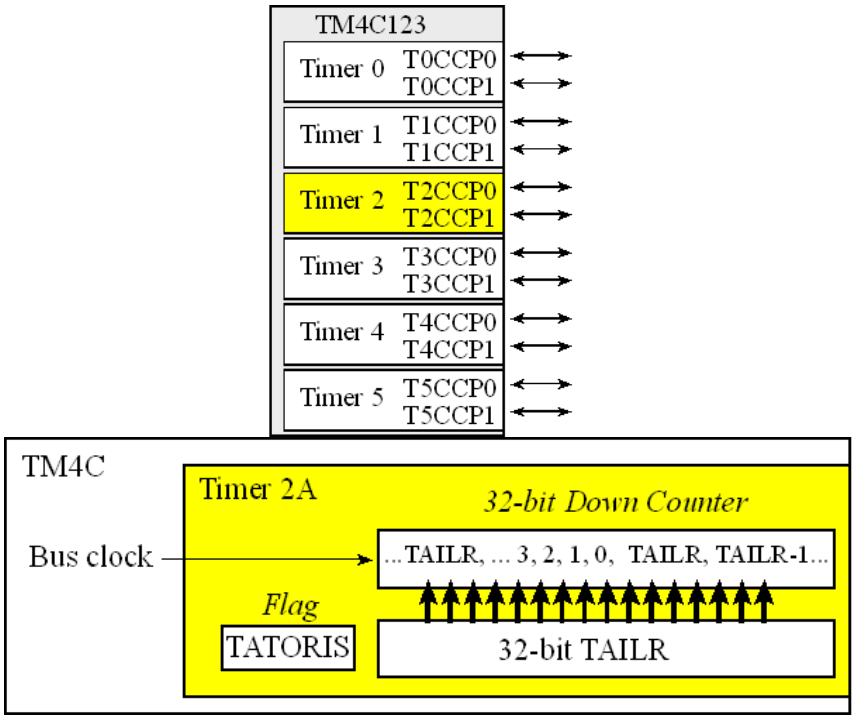


Figure 15.8. Periodic timers on the TM4C123.

```
unsigned long TimerCount;
void Timer2_Init(unsigned long period){
  unsigned long volatile delay;
  SYSCTL_RCGCTIMER_R |= 0x04;   // 0) activate timer2
  delay = SYSCTL_RCGCTIMER_R;
  TimerCount = 0;
  TIMER2_CTL_R = 0x00000000;    // 1) disable timer2A
  TIMER2_CFG_R = 0x00000000;    // 2) 32-bit mode
  TIMER2_TAMR_R = 0x00000002;   // 3) periodic mode
  TIMER2_TAILR_R = period-1;    // 4) reload value
  TIMER2_TAPR_R = 0;            // 5) clock resolution
  TIMER2_ICR_R = 0x00000001;    // 6) clear timeout flag
  TIMER2_IMR_R = 0x00000001;    // 7) arm timeout
```

```
  NVIC_PRI5_R = (NVIC_PRI5_R&0x00FFFFFF)|0x80000000;
// 8) priority 4
  NVIC_EN0_R = 1<<23;          // 9) enable IRQ 23 in
  TIMER2_CTL_R = 0x00000001;   // 10) enable timer2A
}
// trigger is Timer2A Time-Out Interrupt
// set periodically TATORIS set on rollover
void Timer2A_Handler(void){
  TIMER2_ICR_R = 0x00000001;  // acknowledge
  TimerCount++;
// run some background stuff here
}
void Timer2A_Stop(void){
  TIMER2_CTL_R &= ~0x00000001; // disable
}
void Timer2A_Start(void){
  TIMER2_CTL_R |= 0x00000001;   // enable
}
```

Program 15.4. Periodic interrupts using Timer2A (included in Lab15 starter project).

15.6. Random Number Generator

The starter project includes a random number generator. To learn more about this simple method for creating random numbers, do a web search for **linear congruential multiplier**. The random number generator in the starter file seeds the number with a constant; this means you get exactly the same random numbers each time you run the program. To make your game more random, you could seed the random number sequence using the SysTick counter that exists at the time the user first pushes a button (copy the value from NVIC_ST_CURRENT_R into the private variable M). The problem with LCG functions is the least significant bits go through very short cycles. For example

bit 0 has a cycle length of 2, repeating the pattern 0,1,....
bit 1 has a cycle length of 4, repeating the pattern 0,0,1,1,....
bit 2 has a cycle length of 8, repeating the pattern 0,1,0,0,1,0,1,1,....

Therefore using the lower order bits is not recommended. For example

```
n = Random()&0x03;   // has the short repeating pattern 1 0 3 2
m = Random()&0x07;   // has the short repeating pattern 0 7 2 1 4 3 6 5
```

You will need to extend this random number module to provide random numbers as needed for your game. For example, if you wish to generate a random number between 1 and 5, you could define this function

```
unsigned long Random5(void){
  return ((Random()>>24)%5)+1;  // returns 1, 2, 3, 4, or 5
}
```

Using bits 31-24 of the number will produce a random number sequence with a cycle length of 2^{24}. Seeding it with 1 will create the exact same sequence each execution. If you wish different results each time, seed it once after a button has been pressed for the first time, assuming SysTick is running

```
Seed(NVIC_ST_CURRENT_R);
```

15.7. Summary and Best Practices

As we bring this class to a close, we thought we'd review some of the important topics and end with a list of best practices. Most important topics, of course, became labs. So, let's review what we learned.

Embedded Systems encapsulate physical, electrical and software components to create a device with a dedicated purpose. In this class, we assumed the device was controlled by a single chip computer hidden inside. A single chip computer includes a processor, memory, and I/O and is called a **microcontroller**. The TM4C123 was our microcontroller, which is based on the ARM Cortex M4 processor.

Systems are constructed by **components**, connected together with **interfaces**. Therefore all engineering design involves either a component or an interface. The focus of this class has been the interface, which includes hardware and software so information can flow into or out of the computer. A second focus of this class has been **time**. In embedded system it was not only important to get the right answer, but important to get it at the correct time. Consequently, we saw a rich set of features to measure time and control the time events occurred.

We learned the tasks performed by a **computer**: collect inputs, perform calculations, make decisions, store data, and affect outputs. The microcontroller used **ROM** to store programs and constants, and **RAM** to store data. ROM is **nonvolatile**, so it retains its information when power is removed and then restored. RAM is **volatile**, meaning its data is lost when power is removed.

We wrote our software in **C**, which is a structured language meaning there are just a few simple building blocks with which we create software: sequence, if-then and while-loop. First, we organized software into functions, and then we collected functions and organized them in modules. Although programming itself was not the focus of this class, you were asked to write and debug a lot of software. We saw four mechanisms to represent data in the computer. A **variable** was a simple construct to hold one number. We grouped multiple data of the same type into an **array**. We stored variable-length ASCII characters in a **string**, which had a null-termination. During the FSM (Lab 10) and again in the game (Lab 15) we used **structs** to group multiple elements of different types into one data object. In this chapter, we introduced **two-dimensional arrays** as a means to represent graphical images.

The focus of this class was on the **input/output** performed by the microcontroller. We learned that parallel ports allowed multiple bits to be input or output at the same time. Digital input signals came from sensors like switches and keyboards. The software performed input by reading from input registers, allowing the software to sense conditions occurring outside of the computer. For example, the software could detect whether or not a switch is pressed. Digital outputs went to lights and motors. We could toggle the outputs to flash LEDs, make sound or control motors. When performing port input/output the software reads from and writes to I/O registers. In addition to the registers used to input/output most ports have multiple registers that we use to configure the port. For example, we used **direction registers** to specify whether a pin was an input or output.

We saw two types of **serial input/output**, UART and SSI. Serial I/O means we transmit and receive one bit at a time. There are two reasons serial communication is important. First, serial communication has fewer wires so it is less expensive and occupies less space than parallel communication. Second, it turns out, if distance is involved, serial communication is faster and more reliable. Parallel communication protocols are all but extinct: parallel printer, SCSI, IEEE488, and parallel ATA are examples of obsolete parallel protocols, where 8 to 32 bits are transmitted at the same time. However, two examples of parallel communication persist: memory to processor interfaces, and the PCI graphics card interface. In this class, we used the **UART** to communicate between computers. The UART protocol is classified as **asynchronous** because the cable did not include the clock. We used the **SSI** to communicate between the microcontroller and the Nokia display. The SSI protocol is classified as **synchronous** because the clock was included in the cable. Although this course touched on two of the simplest protocols, serial communication is ubiquitous in the computer field, including Ethernet, CAN, SATA, FireWire, Thunderbolt, HDMI, and wireless.

While we are listing I/O types, let's include two more: analog and time. The essence of **sampling** is to represent continuous signals in the computer as discrete digital numbers sampled at finite time intervals. The **Nyquist Theorem** states that if we sample data at frequency f_s, then the data can faithfully represent information with frequency components 0 to ½ f_s. We built and used the **DAC** to convert digital numbers into analog voltages. By outputting a sequence of values to the DAC we created waveform outputs. When we connected the DAC output to headphones, the system was able to create **sounds**.

Parameters of the DAC included **precision**, **resolution**, **range** and **speed**. We used the ADC to convert analog signals into digital form. Just like the DAC, we used the Nyquist Theorem to choose the ADC sampling rate. If we were interested in processing a signal that could oscillate up to f times per second, then we must choose a sampling rate greater than $2f$. Parameters of the ADC also included **precision**, **resolution**, **range** and **speed**.

One of the factors that make embedded systems so pervasive is their ability to measure, control and manipulate **time**. Our TM4C123 had a timer called SysTick. We used SysTick three ways in this class. First, we used SysTick to measure elapsed time by reading the counter before and after a task. Second, we used SysTick to control how often software was executed. In Lab 10, we used it to create accurate time delays, and then in Labs 12-15, we used SysTick to create **periodic interrupts**. Interrupts allowed software tasks could be executed at a regular rate. Lastly, we used SysTick to create **pulse width modulated (PWM)** signals. The PWM outputs gave our software the ability to adjust power delivered to the DC motors.

In general, **interrupts** allow the software to operate on multiple tasks concurrently. For example, in your game you could use one periodic interrupt to move the sprites, a second periodic interrupt to play sounds, and edge-triggered interrupts to respond to the buttons. A fourth task is the main program, which outputs graphics to the LCD display.

One of the pervasive themes of this class was how the software interacted with the hardware. In particular, we developed three ways to **synchronize** quickly executing software with slowly reacting hardware device. The first technique was called blind. With **blind synchronization** the software executed a task, blindly waited a fixed amount of time, and then executed another tasks. The SOS output in Lab 7 was an example of blind synchronization. The second technique was called busy wait. With **busy-wait** synchronization, there was a status bit in the hardware that the software could poll. In this way the software could perform an operation and wait for the hardware to complete. The UART I/O in Labs 5 and 11, and the ADC input in Lab 14 were examples of busy-wait synchronization. The third method was interrupts. With **interrupt synchronization**, there is a hardware status flag, but we arm the flag to cause an interrupt. In this way, the interrupt is triggered whenever the software has a task to perform. In Labs 12, 13, and 14 we used SysTick interrupts to execute a software task at a regular rate. In Chapter 12, we saw that interrupts could be triggered on rising or falling edges of digital inputs. In this chapter we added more periodic interrupts using the timers. Embedded systems must respond to external events. **Latency** is defined as the elapsed time from a request to its service. A **real-time system**, one using interrupts, guarantees the latency to be small and bounded. By the way, there is a fourth synchronization technique not discussed in this class called **direct memory access** (DMA). With DMA synchronization, data flows directly from an input device into memory or from memory to an output device without having to wait on or trigger software.

When synchronizing one software task with another software tasks we used semaphores, mailboxes, and FIFO queues. Global memory was required to pass data or status between interrupt service routines and the main program. A **semaphore** is a global flag that is set by one software task and read by another. When we added a data variable to the flag, it became a **mailbox**. The **FIFO queue** is an order-preserving data structure used to stream data in a continuous fashion from one software task to another. You should have noticed that most of the I/O devices on the microcontroller also use FIFO queues to stream data: the UART, SSI and ADC also employ hardware FIFO queues in the data stream.

Another pervasive theme of this class was **debugging** or testing. The entire objective of Lab 9 was for you to learn debugging techniques. However, each of the labs had a debugging component. A benefit of you interacting with the automatic graders in the class was that it allowed us to demonstrate to you how we would test lab assignments. For example, the Lab 10 grader would complain if you moved a light from green to red without first moving through yellow. QUESTION: How does the Lab 10 grader work?

ANSWER: It first sets the input parameter, then it dumps your I/O data into a buffer just like Lab 9, and then looks to see if your I/O data makes sense. The Lab 13 grader tested both the shape and frequency of your sound outputs. Lab 14 tested the **accuracy** and **linearity** of your distance measurement system.

Furthermore, you had the opportunity to use test equipment such as a **voltmeter** (PD3+TExaS), **logic analyzer** (Keil simulation), and **oscilloscope** (PD3+TExaSdisplay). Other debugging tools you used included **heartbeats**, **dumps**, **breakpoints**, and **single stepping**. **Intrusiveness** is the level at which the debugging itself modifies the system you are testing. One of the most powerful debugging skills you have learned is to connect unused output pins to a scope or logic analyzer so that you could profile your real-time system. A **profile** describes when and where our software is executing. Debugging is not a process we perform after a system is built; rather it is a way of thinking we consider at all phases of a design. Debugging is like solving a mystery, where you have to ask the right questions and interpret the responses. Remember the two keys to good debugging: **control and observability**.

Although this was just an introductory class, we hope you gained some insight into the **design process**. The **requirements document** defines the scope, purpose, and expected outcomes. Even though Lab 15 is not graded, we hope you practice the skills you learned in this class to design a fun game to share with friends and classmates.

Our parting thoughts about best practices (in no particular order of importance):
Here are thoughts about things to remember when designing or building embedded systems, in no particular order of importance:

- Consider debugging when defining, designing, implementing, building and deploying.
- Careful thought during design can save lots of time during implementation and debugging.
- Choose good variable names so the software is easier to understand.
- Divide large projects into modules and test each module separately.
- Separate hardware from software bugs by first testing the software on a simulator.
- When designing modules start with the interfaces, e.g., the header files.
- The second step when designing modules is pseudo code typed in as comments.
- Make the time to service an interrupt short compared to the time between interrupts.
- When developing a modular system, try not to change the header files.
- Use a consistent coding style so all your software is easy to read, change, and debug.
- Most of your time is spent changing or fixing existing code called **maintenance**.
- So, when designing code plan for testing and make it easy to change.
- Writing friendly code makes it easier to combine components into systems.
- Use quality connectors, because faulty connectors can be a difficult flaw to detect.
- It is your responsibility to debug your hardware and software.
- It is also your responsibility to debug other hardware/software you put into your system.
- A simple solution is often more powerful than a complex solution.
- Listen carefully to your customer so you can understand their needs.
- Draw wiring diagrams of electrical circuits before building.
- Double-check all the wiring before turning on the power.
- Double-check all signals in cables, don’t assume red is power and black is ground.
- Be courageous enough to show your work to others.
- Be humble enough to allow others to show you how your system could be better.