Chapter 1

All the world's a bit-pattern

1.1 Introduction

Digital computers manipulate patterns of **b**inary dig**its**, usually called **bits**. One bit is a single binary digit, just about the smallest piece of information that you can think of. It has exactly two states called variously true and false, high and low, on and off, one and zero, etc. We can use single bits to represent very simple pieces of information—a switch is closed, a light must turn on—or we can put them together to make multi-bit patterns. All that a computer does is to manipulate those patterns.

When we use a computer *we* decide what the patterns mean. A single bit pattern, such as 01001011, could mean all sorts of things. For example, depending on the context in which it is used, 01001011 could mean

- the number seventy five
- the letter K
- one tiny sample of a sound
- how much green there should be at a particular spot in a picture
- which of a set of 8 switches is turned on and which turned off
- the instruction to subtract 1 from the value of the accumulator in a microcomputer

or many other things. All the computer knows is that it is a pattern of bits. It knows how to store those bits for future use and it knows how to perform all kinds of operations on bits. It can interpret bit patterns as instructions to do things; it can treat them as numbers and perform arithmetic on them; it can send them to a display and show them as text or as pictures. Which of these things happens is entirely up to the programmer. If the programmer tells the computer to send a collection of bits to the display then they will show up as an image, even if they were actually meant to be the text of a laundry list. If the computer sees the pattern when it is processing instructions then it will execute the appropriate instruction even if that bit pattern started life as the number of pencils in the third drawer from the

right. All a computer can manipulate is bit patterns. How it manipulates them is up to you.

1.2 Patterns of bits

The most compact representation for a single bit is a 0 or a 1. Depending on context you might treat these single-bit value as high and low or true and false (or even as false and true!) but the shortest way to write one bit is as a 0 or 1. Then we can write patterns of bits as strings of 0's or 1's like this.

01 110100 0010111010

Bit patterns can obviously consist of any number of bits. Digital computers, however, are designed to think of bits as coming in fixed length groups. The fundamental number of bits in a group varies from computer to computer. There have been computers built using at least the following numbers of bits in their basic group, which is usually called a **word**

4, 8, 12, 16, 24, 36, 39, 40, 48, 64

Despite the variety of word lengths, a standard has emerged that groups bits into bunches that are multiples of 8 bits and then gives the groups different names thus

- 8 bits = 1 **byte**
- 16 bits = 2 bytes = 1 **short** (often called 1 **word**)
- 32 bits = 4 bytes = 1 **long**

Memorize this table!

Bin	Hex
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Table 1.1

Most current computers come in either 8-bit, 16-bit, or 32-bit lengths. The major exceptions are some very tiny computers used in, for example, thermostats or toasters, which only have a 4-bit word. Such a small bit pattern, only half of a byte, is often called a **nibble**.

When dealing with a byte-organized computer it is usual to write all bit patterns in whole bytes even though some of the bits might not be used in a particular application. For example, it only takes 6 bits to represent the states of 6 switches but it would be common to write them as a full group of 8 bits and just ignore the 2 extras. So, the bit pattern showing all 6 switches turned on might well be displayed as 00111111.

Obviously, every written out bit pattern is also a set of characters that could be a number. For example, the bit pattern 10 could easily be confused with the number ten. When there is any possibility of confusion we mark a binary pattern in some way. Here are three common ways of marking the same bit pattern.

1101B 0b1101 1101_2 %1101

In this text I will usually use the 0b1101 form whenever there is a likelihood of confusion.

1.2.1 Hex notation

Long bit patterns, such as the 32-bit and 64-bit patterns natural to a modern computer are very time consuming to write out. A shorthand method is universally used to get round this. A complete bit pattern is first broken up into 4-bit sections, starting at the *right hand* end. Then each 4-bit group is replaced by a unique single character representation called a **hexadecimal** or **hex** digit. Table 1.1 shows the 16 possible 4-bit patterns and their 1-character hex representations.

We use hex notation all the time as a shorthand way to represent binary numbers.

Thus the binary bit pattern `0b001111001110` would be written in Hex as `3CE`.

Again, we have to be careful with hex patterns since many of them are also legal decimal numbers and some are English words. If there is any doubt then we add a tag as we did with the binary patterns. Here are some common ways to write the hex pattern corresponding to the bit pattern `0b001000000011`.

203H 0x203 203_{16} $203

For our purposes hex is interchangable with binary. More precisely, hex is a shorthand way to represent binary. It is trivial to turn a number given in hex into one in binary and vice versa.

In this text, I will use the 0x203 or the $203 notation if there is any possibility of confusion. A hex pattern like 3CE might well not get a $ or 0x though, as it really does not look like anything except a hex pattern.

It is very easy to convert a binary number into its hex representation or a hex number into its binary representation. The rules are given below.

Binary-to-Hex Conversion
1) If the number of bits is not a multiple of 4, then pad the number on the left end with 0 bits until it is.

2) Starting at the right hand end split the number up into 4-bit groups
3) Replace each 4 bit group by its corresponding hex digit

Hex-to-Binary conversion
Replace each hex digit by its corresponding 4-bit binary pattern, keeping the order of the bits correct.

Let's try a few examples.

Example 1.2.1

Convert 0b11001010010 to hex.

There are 11 bits in the pattern. That is less than a multiple of 4 so add an extra zero on the left to get

0b011001010010

Next split the number into groups starting at the right end.

0b0110 0101 0010

Finally, replace each 4 bit group using the table.

$0110 \rightarrow 6, 0101 \rightarrow 5, 0010 \rightarrow 2$ so we have 6 5 2

To be tidy, we put the whole thing back together again and attach a 0x to show that it is a hex pattern.

0b011001010010 = 0x652

Example 1.2.2

Convert 0b01011111011101011110101110101 to hex
Step 1: 0b01011111011101011110101110101 has 29 bits.
Pad to 32 bits getting
0b00001011111011101011110101110101
Step 2: split into 4-bit blocks to get 0000 1011 1110 1110 1011 1101 0111 0101
Step 3: replace each 4 bit block with its hex representation to get 0 B E E B D 7 5
Step 4: compact and stick a 0x on the front to get 0x0BBEEBD75

Example 1.2.3

Convert 0xFE03 to binary
Step 1: From Table 1-1 above we see that the digit F corresponds to 0b1111 so those are the first four bits.
Step 2: Similarly E corresponds to 0b1110 so now we have 0b1111 0b1110
Step 3: Hex 0 is obviously 0b0000 and 3 is just 0b0011 so altogether we have
0b1111 0b1110 0b0000 0b0011
Step 4: Finally we remove the spaces and extra hex markers to get the 16-bit number 0b1111111000000011.

1.3 Binary Numbers

Significance
Since, in Arabic notation, each digit corresponds to larger power of the base than does its neighbor to the right, we say that a digit is *more significant* than the digits to its right. Thus we call the leftmost digit the *most significant digit* (abbreviated *msd*) and the rightmost digit the *least significant digit* (*lsd*).
When we are working with bit patterns we use the terms *most significant bit* (*msb*) and *least significant bit* (*lsb*) for the leftmost and rightmost bits respectively.

One of the commonest uses for a bit-pattern is to represent a number. We are used to writing numbers in base 10 Arabic notation. In this scheme, we use ten different unique symbols, 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, and use the position of the digit in the number to give it extra magnitude meaning. For example, in the number 252 we know that the 2 in the leading (leftmost) position means two hundred while the trailing (rightmost) 2 just means two. Each digit, working from right to left, is worth one more power of ten than the digit to its right.

With only two states for each digit we can only write numbers in base 2 or binary. This time we have the two unique symbols, 0 and 1, and each position corresponds to one more power of two than the one to its right. Thus we can work out what a bit pattern like 0b0110 means as a number. The right most 0 means that there are no 1's in the number. The next 1 to the right means that there is 1 two in the number. The left most 1 means that there is 1 four in the number while the leftmost 0 means there are no eights in the number. That gives us a total number of

$0 + 2 + 4 = 6.$

It is obviously no coincidence that 6 is the hex representation for the bit pattern 0b0110!

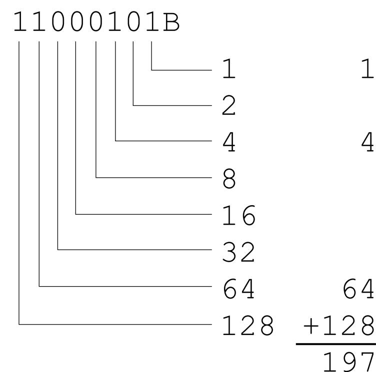

Figure 1.1

Figure 1.1 shows the meanings of the first few bit positions and a longer example of how to turn a bit pattern into a number.

The principle extends easily and so we can convert any bit pattern to its corresponding decimal number. Table 1.2 is a table of powers of 2 to help.

Obviously we sometimes need to go the other way. It is a little trickier to convert a decimal number to binary. Basically we ask if the number is

Table 1.2: Powers of Two

2^0	1	2^7	128	2^{14}	16,384	2^{21}	2,097,152	2^{28}	268,435,456
2^1	2	2^8	256	2^{15}	32,768	2^{22}	4,194,304	2^{29}	536,870,912
2^2	4	2^9	512	2^{16}	65,536	2^{23}	8,388,608	2^{30}	1,073,741,824
2^3	8	2^{10}	1,024128	2^{17}	131,072	2^{24}	16,777,216	2^{31}	2,147,483,648
2^4	16	2^{11}	2,048	2^{18}	262,144	2^{25}	33,554,432	2^{32}	4,294,967,296
2^5	32	2^{12}	4,096	2^{19}	524,288	2^{26}	67,108,864	2^{33}	8,589,934,592
2^6	64	2^{13}	8,192	2^{20}	1,048,576	2^{27}	134,217,728	2^{34}	17,179,869,184

divisible by each power of two in turn. The best way to do this is to divide the number by two for as long as we can, keeping track of the remainders. Those remainders, taken in reverse order, form the binary representation of the number. If we do the divisions down the page, as shown below, then we make the binary number by reading the remainders from bottom to top. For example, to find the bit pattern that corresponds to the number two hundred and forty-three, whose decimal representation is 243, we perform the calculation shown in Figure 1.2.

As the first stage, we divide 243 by 2 to get 121 remainder 1. We write the 121 below and the remainder to right. We repeat this process until there is nothing left. In the last stage we divide 1 by 2 to get 0 remainder 1. Lastly, we read the remainders off from bottom to top to get the answer and put a 0b on the front to show that we know it is binary.

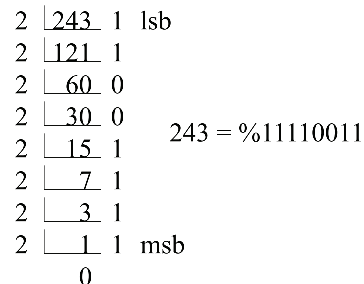

Figure 1.2

1.3.1 Counting in binary

It is often useful to know how to count in binary. The process follows the same pattern as counting in decimal except that it is simpler. In decimal we increase the rightmost digit until we run out of digits at 9. Then we add one to the next place to the left and reset this digit to 0, as below.

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,$$

In binary we do the same but we run out of digits a lot more often. It is easy to add 1 to 0 but then we have to think. 1 is the highest value digit so we have to add one to the next digit to the left and put a zero at the end. That gives us

$$0, 1, 10$$

Next we add 1 to the rightmost digit making it a 1. There is no overflow so the next number is 0b11

When we add 1 to the rightmost 1, we have to play our overflow trick. We put a zero in the rightmost place and add 1 to the digit to its left. The digit to the left is a 1 so the overflow happens again. We put a 0 in this place and add 1 to the digit to the left. Since nothing is written there, that digit

was a 0 and when we add one we have a 1 and we are done. That makes the fourth non-zero bit pattern 100

Dec.	Bin.	Hex
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	10000	10
17	10001	11
18	10010	12
19	10011	13

Table 1.3: Binary, Decimal, and Hex

We can continue this forever. Table 1-3 contains the first twenty numbers in decimal, binary, 8-bit binary, and hex.

One important idea comes out of this. There are 2n unique patterns of n bits. For example, there are four 2-bit patterns, 0b00, 0b 01, 0b10, 0b11, and so we can represent anything that comes in only 4 or fewer states using a 2-bit pattern. Here are some particularly useful numbers

8 bits = 256 unique patterns
10 bits = 1024 unique patterns
16 bits = 32,768 unique patterns

Because 210 is so close to one thousand, and because all things in computers come in powers of two, we regularly misuse the terms kilo and mega for binary numbers. When referring to binary objects, 1k is 1024 and 1M is 1024x1024 = 1048,576. So for example, 1 kilobyte of memory contains 1024 memory cells not 1000 memory cells.

1.3.2 Gray Codes

A counting sequence like this is not the only useful sequence of bit patterns, though it is by far the most common. Other sequences are used for special purposes. This section describes one such sequence.

Position measurements are fundamental to many different control systems. For example, a wheeled robot builds its map of the world in terms of how far its wheels have to turn to get from point to point. Computer Automated Manufacturing (CAM) systems can machine extremely complex shapes because they can move a lump of metal around a cutting tool under computer control. They can produce the correct shapes because they can measure the position of a work-piece very precisely. A powerful way of converting a position into a binary is with an optical encoder (Figure 1.3). The optical encoder is a glass or plastic plate with a pattern of dark areas and clear spaces. Light shines through the plate and falls on a row of light sensors, which output a 1 if light falls on them and a 0 if light is blocked.

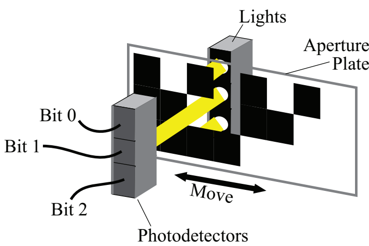

Figure 1.3: Optical Encoder

As the plate moves, either round and round or back and forth, the pattern of spaces moves past the sensor and the pattern of bits coming from the sensor changes. Life is all very well when the sensor is exactly lined up with one column of the pattern, as shown in Figure 1.3. However, when the pattern moves, there are places where the sensor is illuminated partly by one row and partly by the next. If you use an ordinary counting pattern on the encoder plate, then there are places where many bits are changing at once and a tiny position error can cause a large change in the output number. The cure for this is to use a Gray Code. It is designed so that only one bit

changes each time we change number.

Example 1.3.1

When we go from seven, 0111, to eight, 1000, every bit in the number has to change at once. When the encoder plate is part way between seven and eight, it could output any number as different bits change at slightly different times.

A Gray Code is a modified counting sequence in which the numbers no longer have their sensible place notation. Instead, the sequence has the properties

1. Each possible n bit pattern occurs exactly once
2. Exactly one bit changes when you go from one number to the next
3. The patterns form a circular set so that only one bit changes when you go from the last pattern to the first.

Gray codes are not unique but here is a common 3-bit version.

$$000, 001, 011, 010, 110, 111, 101, 100, 000, 001, 011, etc.$$

As you see, all 8 possible patterns are present but they occur in an unusual order. Because only one bit changes when you go from one pattern to the next, when the encoder is part way between two patterns only that one bit will wobble in value and so the output will only wander between these two values. This is much better than the counting sequence where there are places where an error in one bit can make a huge difference in the value of the pattern.

Example 1.3.2

When the gray code goes from 3 to 4 the output value is always either a 3 or a 4. When a standard binary code goes from 3 to 4 then the following values could be produced if one of the changing bits were in error

011 $\rightarrow$ 100 error in bit 0 produces 010 or 101

011 $\rightarrow$ 100 error in bit 1 produces 001 or 110

011 $\rightarrow$ 100 error in bit 2 produces 111 or 000

Thus, during the transition from a 3 to a 4 the sensor could produce any of the values
0, 1, 2, 3, 4, 5, 6, or 7

1.4 Arithmetic with binary numbers

We have seen that binary patterns naturally represent numbers. That means that we can do arithmetic on binary numbers just as we can do arithmetic on decimal numbers. This section explores the arithmetic of binary numbers.

1.4.1 Addition

We add two binary numbers in exactly the same way that we add two decimal numbers, one digit at a time. The only difference is that the number of 1-bit

addition facts is a lot smaller than the number for 1-digit addition. Here are the 1-bit addition facts.

$$0+0=0 \quad 1+0=1$$
$$0+1=1 \quad 1+1=10$$
$$1+1+1=11$$

The last fact is not strictly necessary but it is useful when there has been a carry from one bit to the next. Let's look at an example. The binary equivalent of 7 is 0b111 and the binary equivalent of 3 is 0b11. We know that 7 + 3 = 10 in decimal so let's see what happens in binary.

```
  1 1 1
+   1 1
    carry: 1
-------
      0
```

We start at the least significant bit. 0b1 + 0b1 = 0b10 so write down the 0 and carry the 1, just as we would in decimal addition.

```
  1 1 1
+   1 1
  carries: 1 1
-------
    1 0
```

Next, we move one place to the left, to the 2's column. Here we have 0b1 + 0b1 + 0b1 = 0b11 because of the carry from the units position. We write down the 1 and carry 1.

In the last column, we have only two digits; 0b1 + 0b1 = 0b10. Since there are no more bits to the left, we write both bits down and we have our answer.

```
  1 1 1
+   1 1
  carries: 1 1
-------
1 0 1 0
```

It works! 0b1010 is the binary representation for ten. We get the same answer whether we work in decimal or in binary.

There are two things you should notice in this example. The first is the pivotal role that the carry plays in passing information from bit to bit. The second is the number of bits in the answer. We started with two numbers that could each be represented with only three bits (3 = 0b011) but we finished with an answer that occupies four bits. On paper this is not a problem, but in a computer we have a fixed number of bits in each pattern. That means that it is possible to add together two perfectly legal numbers and get an answer that will not fit in the space available. This problem is called overflow. Overflow occurs when the answer will not fit in the number of bits provided, i.e. there is a carry from the leftmost bit. Overflow presents a problem. When it occurs, the answer is wrong. For example, if we had done our example in a true 3-bit representation then the answer would have been 0b010 with a carry of 1. That would mean that 0b111 + 0b011 = 0b010 or 7 + 3 = 2, which is clearly not what we want.

There is no cure for overflow. Any representation that uses a fixed number of digits, whether binary or decimal, can only represent a finite range of numbers. When a result exceeds that number of digits, you start getting wrong answers. The only thing you can do to avoid this is to make sure that you always use a representation that is big enough to hold the largest number you will encounter. For example, the 8-bit values that most computers work with can hold positive integers up to 255. You can safely use them to count

the lines on a page but will have to do better if you want to count the words in a book.

It is worthwhile to note here the ranges of the common widths of number. For each width, I give two ranges. The first is for unsigned numbers, numbers that are always positive. The second is for signed numbers, numbers that can be either positive or negative.

Type	Bits	Unsig. Range.	Signed Range
char	8	0 to 255	-128 to +127
short	16	0 to 65535	-32768 to +32767
int	32	0 to 4,294,967,295	-2,147,483,648 to 2,147,483,647

Table 1.4: Number Ranges

1.4.2 Negative Numbers

We can now add binary numbers of any magnitude but before we can look at subtraction we need to know how to represent negative numbers. As humans, we deal with the problem of negative numbers by going outside the scheme of the numbers. We use a separate symbol, the minus sign, to mark negative numbers. Computers can't do this. Remember, everything is a bit pattern. The only way a computer can mark a negative number is with a bit. On paper we can just stick an extra bit in front of every binary number and say e.g., that if the bit is 0 then the number is positive and if it is 1 then the number is negative. Inside the computer, we are limited by the fixed size of the bit patterns and we have to steal a bit from the number to use for a sign. Thus, while an 8-bit pattern can hold positive numbers up to 255, once you have stolen one bit to use for the sign a byte can only hold signed numbers up to 127.

There are several different ways that we can represent signed numbers with a sign bit and some value bits. All modern computer use 2's complement representation in which the top bit tells you the sign of the number, 0 for positive and 1 for negative. The rest of the bits contain the size of the number. Positive numbers have their usual values, e.g. 19 = 0x13, 115 = 0x73, etc. Negative numbers are more peculiar. The easiest way to understand them is to see how they are formed. Here is the algorithm.

Algorithm
An algorithm is a set of rules for performing an operation. One common example would be a recipe and another would be the assembly instructions that come with so many toys and pieces of furniture.
It is a set of rules with a clear order. For it to work properly you have to not only perform the correct steps, you have to do them in the correct order.
We shall see later that algorithms are the basis of computer programs.

Algorithm to convert binary from negative to positive or vice versa

1. Write number in binary.
2. Pad to width with leading zeros.
3. Invert every bit (replace 1's with 0's and 0's with 1's).
4. Add 1 to the number.

The result is the two's complement of the original number.

Example 1.4.1

Find the 2-'s complement 8-bit binary form of -19.

From Table 1-3 we know 19 = 0x13 = 0b00010011 in 8-bit binary. To find -19 we first invert all of the bits of 0x13

0x13 = 0b00010011 -> 0b11101100 = 0xEC

and then add 1.

0xEC + 1 = 0b11101100 +1= 0b11101101=0xED.

As expected, the top bit is 1 so the number is negative.

The result of this process is that an 8-bit signed number can take on 128 different positive values, $0 \rightarrow 127$, and 128 negative values, $-1 \rightarrow -128$. There is an apparent discrepancy caused by the largest positive being +127 while the largest negative number is -128. This is because zero is positive and uses up one positive number.

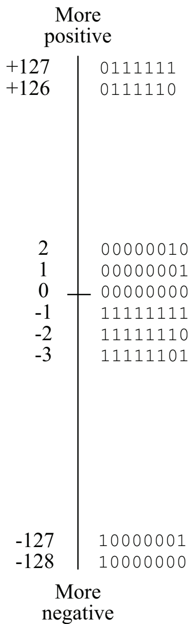

Figure 1.4: 8-bit 2's complement number line

Figure 1.4 shows a number line for the 8-bit 2's complement numbers. If you examine the line carefully then you will see that the 8-bit representation of -n is the same as the 8-bit unsigned number 256-n. Thus -1 has the same bit pattern as 255, -2 the same pattern as 254, and so on. Neither you nor the computer can tell whether a number is signed or not simply by looking at it! Only the way that the number is used allows you to tell whether we mean a number to be signed or unsigned. So when a computer sees the bit pattern 0b11101101 it can use that pattern as either the positive number 237 or the negative number -19. Or, it could use it as something else entirely. A bit pattern only represents a number when the computer programmer writes a program that uses the pattern as a number.

One wonderful thing about 2's complement numbers is that you can make a negative number positive in exactly the same way that you make a positive number negative. The negate operation is unique; invert all the bits and add 1. So when we apply the same operation twice we get back to where we started.

Example 1.4.2

Let's take a number and negate it twice.

90 = 0b01011010

-90 = 0b10100101 + 1 = 0b1010011

-(-90) = 0b01011001 + 1 = 0b01011010 = +90 as wanted.

The other wonderful thing about 2's complement numbers is that they can be added just like ordinary binary numbers. So, for example, we can add -3 to 17 and get 14, so long as we are careful (Figure 1-5).

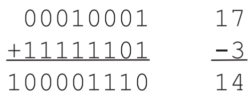

Figure 1.5

We have to be careful about the answer. 2's complement is a fixed width representation so we only keep the bottom 8 bits of the answer, 0b00001110 = 14. The operation gives the right answer and it results in a carry bit of 1.

f you subtract a larger unsigned number from a smaller then the answer is automatically the correct signed number. This is even true if one or both of the original positive numbers is too big to fit in a signed value. For example

$$231 - -245 = 0xE7 - 0xF5 = 0xF2$$

which is the correct signed 8-bit representation of –14.

One problem arises when we move from unsigned numbers to signed, 2's complement, numbers; we can no longer use the carry flag to signal overflow. Overflow occurs when the result will not fit in the current representation. In 8-bit 2's complement notation the largest positive number we can represent is 127. Thus if we add 100 and 46 we get a number that will fit in an 8-bit unsigned value but not in a signed one. Similarly, if we subtract 43 from -100, we get a value that overflows the representation. The signs that overflow has occurred are a bit more complex than they were with unsigned numbers but a little logic can easily determine when it has happened. This information can be saved in a flag in the same way that the carry is saved. Such a flag is called an overflow flag and it is available on a lot of computers. If you are worried about the validity of your arithmetic then you can check the state of the overflow flag and take some action if an error has occurred.

1.4.3 Multiplication

Long multiplication in binary is both simpler than it is in decimal and much more long winded. There are no multiplication tables to learn since you only ever have to multiply by one (copy) or by zero (omit). However, since we only know how to add binary numbers two at a time, the addition of all the partial sums can get fiddly.

It is easiest to understand with an example and to keep the example reasonably short we will multiply two 4 bit numbers,

$$13 \times 11 = 0b1101 \times 0b1011$$

The bottom number in a multiplication is called the **multiplier**. The top number is called the **multiplicand** and the answer is the **product**.

We will start with the leftmost bit of the multiplier. Since there are three digits to the right of this bit, we start by writing down three zeros on the right-hand end of the answer line. Next we multiply the top number by one, working from right to left, and write the digits down. Because multiplication by 1 is trivial, we simply copy the top number down in the answer.

```
   1101
×  1011
1011000
```

Once we are done with the most significant digit, we move down to the next digit. This one is a zero. Anything we multiply by it will also be zero so we shall skip this digit.

```
   1101
×  1011
1011000
  11010
```

The next digit along is another 1. We start a new line in the answer and again we write down a zero for every place to the right, one in this case. Then we do the multiplication by copying the top line into place.

At this point we have two partial results and we have to add them together in the usual fashion. Note that this has produced an answer with 8 bits in it. This is normal; multiplication increases the number of bits needed to represent the answer. Indeed if we multiply two n bit numbers together the answer will in general need 2n bits to hold it.

```
     1101
×    1011
---------
  1011000
 1 1 111010
---------
 10000010
     10 11
       1
---------
 10001101
```

The intermediate sum replaces the two lines that went to make it up. We have one last digit, the least significant, in the multiplier. It is again a 1 and so we copy down the multiplicand for the last time and perform one last addition to get the final answer; 10b0001101 = 0x8F.

So the final answer to our long multiplication is that $13 \times 11 = 0xD \times 0xA = 0x8F = 143$. So the method works.

As we have just seen, if you multiply two n-bit numbers together then you get a 2n-bit answer. Since computers are fundamentally tied to fixed-length bit patterns this produces a problem. Real machines usually handle it by saving a double-length result in two storage locations, one after another. However, there may be very little that you can do with such a double-sized number other than truncate it back to the original length and take note if you throw anything away by doing this!

It is worth noting that almost all computers used to perform multiplication one bit at a time in just the way that we have seen. This made multiplication take much longer than addition or subtraction. It is possible to design a circuit that can multiply two numbers in about the same time as it takes to add them, but it is extremely complex. Such parallel multipliers used to appear only on computers that need to perform vast numbers of multiplications in a very short time; the classic example is digital signal processing computers. However, the number of transistors that we can put on a chip has now grown so large that desktop computers, and even some embedded systems, routinely incorporate multiplication hardware that is practically as fast as addition.

Because the top bit of a 2's complement number contains the sign information, this shifting trick only works for signed numbers if you are careful to treat the top bit separately. Computers usually provide a special pair of left and right shift operations called arithmetic shifts (as opposed to logical shifts) that take care of this for you.

In systems where multiplication is very slow there is a trick that is worth remembering. You can multiply a binary number by 2 simply by shifting it one place to the left. This is like multiplying a decimal number by 10 by shifting it one place to the left. This can be extended. A shift of two places to the left is equivalent to multiplying by 4, a 3 place shift to multiplying by 8, etc. Similarly, you can divide a binary number by a power of two by shifting to the right. A shift of 1 place to the right is equivalent to dividing by 2 and so on.

1.4.4 Division

Long division is one of the nastiest pieces of arithmetic that most of us have ever had to learn. It is not significantly nicer in binary. The principle is the same. Start from the most significant end of the dividend and keep trying

to subtract the divisor. There is one slight simplification in binary. There is no question of how many times you can subtract the divisor; the choice is either 1 or 0.

Just as multiplication increases the width of numbers, so division reduces it. A computer that provides an 8-bit $\times$ 8-bit$\rightarrow$16-bit multiplication often also provides a 16-bit/8-bit$\rightarrow$8-bit divide. The problem with division is that it is rarely exact. When you multiply one integer by another you always get an integer answer. When you divide one integer by another you usually don't get an integer answer. Computers deal with this by returning the remainder as well as the quotient.

Example 1.4.3

We'll divide the 143 that we got from our multiplication example by 12.

0b1100 into 0b1 can't be done so we write down a zero for the first bit of the answer.

We keep working along to the right one bit at a time finding that 0b1100 will not divide the dividend until we have 5 bits. 0b1100 into 0b10001 can be done so we have a 1 in the quotient and we subtract 0b1100 from 0b10001.

Because we only know how to do subtraction by adding the 2's complement of the number we are subtracting it is best to do the subtractions off to one side or on a separate piece of paper. When we do that we find that 0x10001-0x1100 = 0x101.

Now, just as we would in decimal, we use 0b101 as the first bits of the new quotient and bring bits down to join it. The first bit to come down is a 1, making 0x1011. 0x1100 does not divide 0x1011 so the next digit in the answer is another 0.

We keep going in the same way. The next bit to come down is also a 1 so our new dividend is 0x10111. 0x1100 does divide into 0x10111 so the next bit of the answer is 1 and we have another subtraction; 0x10111-0x1100=0x1011.

Again, we start bringing down bits to join the 0b1011. The only bit left is a 1 and 0x1100 does divide 0x10111 so the last bit in the quotient is another 1 and we have

Our division is complete. We find that 0b10001111/0b1100 = 0b1011 remainder 0b1011. That is, 143/12 = 11 remainder 11. So it works.

```
        00001011
1100 | 10001111
       -1100
         10111
        -1100
          10111
         -1100
           1011
```

Notice that, unlike multiplication, division can overflow. Because the dividend is twice as long as the divisor, which could be as small as 1, it is possible for there to be more significant digits in the quotient than will fit. If this happens, the operation has overflowed and all we can do is set a flag and hope the program notices. Worse still, division can fail altogether. We know that we can't divide a number by 0, but it is possible for a program to try (in error, we hope). When that happens we have to abort the operation and warn the user somehow. We shall see how some computers deal with problems like this later on. For now, try to avoid dividing by zero!

1.4.5 Multiple Precision Arithmetic

Most computers provide instructions that support addition of 8-bit numbers and often also 16-bit numbers. More powerful computers support 32-bit numbers and even 64-bit numbers. However, all computers give up at some point. What do you do if you need a number that is larger than the largest your computer supports? The answer is to spread the information over two or more numbers.

For example, if your computer can read, write, and add 8-bit bytes, then you can use 16-bit quantities by spreading them across two bytes. You put the rightmost 8 bits, the least significant bits, in one byte, and put the leftmost 8 bits, the most significant bits, in the other byte.

Addition

Although there may be no instructions to add 16-bit quantities, you can use the fact that addition takes place one bit at a time, from right to left, to do the addition one byte at a time. First, you add together the least significant bytes from the two numbers, keeping track of any carry that occurs. The resulting byte is the bottom 8 bits of the answer. Next you add together the most significant bytes and add in the carry from the previous stage. The result is the top 8 bits of the answer. If there is a carry from this stage then you have overflowed the capacity of even 16-bits. If you need to use even bigger numbers then you just spread them over more bytes and do the arithmetic the same way, one byte at a time. The only trick is that you have to remember to include the carry from the previous byte when adding the more significant bytes.

Note that in order to implement multi-byte addition like this we need to save the carry from each operation so that we can use it in the next most significant operation. We need to have a Carry bit separate from the normal data bits. Computers often provide two different addition operations. The usual operation adds two bytes and sets the flag while the second one includes the carry in the addition, just the way we did in the example.

Example 1.4.4

That all sounds more complicated than it is. Let's try it with some numbers. We'll add 1,475,399 to 13,236,432 and hope that we get 14,711,831. Those are mighty big numbers, much larger than the 32,767 maximum for 16-bit numbers so we'll use 32-bit numbers spread over 4 bytes each. Note that I am writing all the bit patterns in hex rather than binary because the numbers are so long. I encourage you to check the working as we go, using a calculator that speaks hex.

- 1,475, 399 = 0x168347 so we store this as the 4 bytes 0x00 0x16 0x83 0x47
- 13,236,432 = 0xC9F8D0 which we store as the 4 bytes 0x00 0xC9 0xF8 0xD0
- We start addition by adding together the least significant bytes
- 0x47 + 0xD0 = 0x117 so the least significant byte of the result is 0x17
  and we carry the 1
- 0x83 + 0xF8 + Carry = 0x17C giving us 0x7C carry 1.
- 0x16 + 0xC9 + Carry = 0xE0, which gives us 0xE0 with no carry.
- 0x00 + 0x00 + 0 = 0x00 and our most significant byte is 0x00.
  There is no carry from this byte so the answer does not overflow.
  Note that we still have to include the carry from the previous byte, but this time the carry was 0.
- So the answer is 0x00 0xE0 0x7C 0x17 with no carry.
  That is our representation of the hex number 0xE07C17 = 14,711,831. It worked.

Subtraction

Just as we can extend addition to multi-byte numbers using the carry, so we can extend subtraction. We use the same flag for both operations and

sometimes extend the flag's name to Carry/$\overline{\text{Borrow}}$. We subtract two multi-byte numbers one byte at a time, working from least significant to most significant byte. For example, we subtract two 2-byte numbers like this:

1. Subtract the bottom byte of number B from the bottom byte of number A. Save the carry/borrow.
2. Store the result as the bottom byte of the answer.
3. Subtract the top byte of B from the top byte of A taking the carry/borrow into account.
4. Save the result as the top byte of the answer.

There must be a subtract instruction that takes the Carry/Borrow into account in just the same way that there is a separate add instruction that uses the Carry.

Example 1.4.5

Let's subtract 0x64D0 = 25,808 from 0x4E6B = 20,075.

1. Subtract 0xD0 from 0x6B.
    - First we find the 2's complement of 0xD0
    - - 0xD0 = 0x2F + 1 = 0x30
    - Add 0x6B and 0x30 to get 0x9B with Carry = 0, so that Borrow = 1.
2. Store 0x9B as the bottom byte of the answer.
3. Subtract 0x64 from 0x4E with borrow.
    - First we find the 2's complement of 0x64 - 0x64 = 0x9B + 1 = 0x9C.
    - Add 0x4E and 0x9C to get 0xEA and subtract the borrow to get 0xE9.
4. Store 0xE9 as top byte of answer.
    5. Answer is 0xE99B = -5733. It works!

1.5 Bit Patterns as Text

While we spend a lot of our time using bit patterns to represent numbers we also often need to represent other things. The most common other use is to represent text. This use of bit patterns predates computers. Some of the first binary codes, Morse Code (using dot and dash as its two symbols) and the Baudot telegraph code, were designed specifically to represent text. They adopted the obvious scheme of assigning a bit pattern to each letter of the Roman alphabet and to each number digit and punctuation symbol that they thought useful. Neither of these codes is well suited for modern computer use and over time a number of other schemes were adopted. Once there were enough computers in the world for anyone to care whether you could move information from one to another, it became obvious that some standard code was needed. The one that has come to into nearly universal use is called the American Standard Code for Information Interchange (ASCII).

Unicode
Because the ASCII code is limited, even in its extended forms, to 8 bits, it can only handle 256 characters. That is far more than English needs but it is not sufficient to handle non-English languages. As computers have spread into all parts of the world people have come to need a way to represent any mixture of languages. This has lead to the development of a 16-bit encoding called Unicode. Unicode already includes characters and ideographs for all the major languages of the world, including Chinese, and work is underway to make it cover all the rest as well. In its basic form Unicode can represent more than 65,000 different characters and the extensions push this into the millions. For maximum compatibility with existing systems Unicode defines an 8-bit subset that includes the standard ASCII code. For more information you can visit the Unicode web site at www.unicode.org.

1.5.1 ASCII Character Code

The ASCII code is basically a 7-bit code so that every character appears as a positive integer in an 8-bit representation. ASCII includes the decimal digits, the lower and upper case English letters, and almost all the common punctuation and special symbols used in English. It also includes an extensive set of non-printable characters that were intended to provide fine control over the process of sending data from one computer to another. Most of

those control characters have fallen into disuse but a few, such as carriage return, are still very important. Table 1.5 is a complete table of the ASCII character codes.

		Second Hex Digit							
		0	1	2	3	4	5	6	7
F	0	NULL	DLE	SP	0	@	P	`	p
i	1	SOH	DC1	!	1	A	Q	a	q
r	2	STX	DC2	"	2	B	R	b	r
s	3	ETX	DC3	#	3	C	S	c	s
t	4	EOQ	DC4	$	4	D	T	d	t
	5	ENQ	NAK	%	5	E	U	e	u
H	6	SOH	SYN	&	6	F	V	f	v
e	7	BELL	ETB	’	7	G	W	g	w
x	8	BS	CAN	(	8	H	X	h	x
	9	HT	EM	)	9	I	Y	i	y
D	A	LF	SUB		:	J	Z	j	z
i	B	VT	ESC	+	;	K	[	k	{
g	C	FF	FS	,	<	L		a	|
i	D	CR	GS	-	=	M	]	m	}
t	E	SO	RS	.	>	N	ˆ	n	~
.	F	SI	US	/	?	O	_	o	DEL

Table 1.5: ASCII Character Code

The table is organized according to the hex representation of the bit pattern for each character. Each character is associated with a unique 7-bit pattern shown as a hex number between 0x00 and 0x7F. The table is laid out in columns under the first digit of the hex number. All the characters in the first column have code between 0x00 and 0x0F, all those in the second column codes between 0x10 and 0x1F, and so on. Within each column the characters are ordered by increasing second digit.

Because the ASCII code only uses the first 128 of the possible 8-bit numbers, it has been extended to 256 codes by various different computer makers and program designers. Unfortunately, there is nothing standard about these extensions. For example, both the Windows world and the Macintosh world define full sets of 256 codes. However, the Windows world uses most of the extra characters for special symbols used to draw boxes on text-only screens (not very useful these days!), while the Macintosh uses most of them for accented letters and other letters not found in the English alphabet.

To find the ASCII code for any character

1. Find the character in the table and then
2. Get first hex digit from the number at the top of the column.
3. Get second hex digit from the number at the left of the row.

Example 1.5.1

The letter ‘H’ is the column headed 4 and in the row labeled 8. Thus its ASCII code is 0x48.

Similarly, you can find the character corresponding to any ASCII code by using the first hex digit to select a column and the second to select a row. The character whose code you have used lies at the intersection of row and column.

Example 1.5.2

The code 0x6E takes us to the last column but one, where the first half of the lower case alphabet lives, and to the last row but one. The character in that place is 'n'. Thus, 'n' is the character with ASCII code 0x6E.

Several nice things about the ASCII codes are worth remembering.

- First all the numerals are in order and have codes that begin 0x3 so we can turn a single digit number into its ASCII character by adding 0x30 or turn the character into a number by subtracting 0x30.
- Second all the alphabetic characters are in order in two groups. The capitals all begin with hex 0x4 and the lower case all begin with 0x60. This means that you can change the case just by adding 0x20 to or subtracting 0x20 from the code for a number.
- Lastly, the characters with codes less than 0x20 are reserved for non-printing characters designed to alter the flow of text in some way. Many are now obsolete but a few are still current, including:
  - 0x00 emphNULL used to end strings
  - 0x07 emphBEL sounds an audible warning on some systems
  - 0x08 emphBS **B**ack**s**pace, a favorite of poor typists like me!
  - 0x09 emphHT the tab character (**H**orizontal **T**ab)
  - 0x0A emphLF **L**ine **F**eed, moves the pointer down one row
  - 0x0B emphVT **V**ertical **T**ab, rare vertical version of tab
  - 0x0C emphFF **F**orm **F**eed, starts a new page
  - 0x0D emphCR **C**arriage **R**eturn, returns pointer to start of line
  - 0x1B emphESC **Esc**ape, used for various special purposes
  - 0x7F *DEL* **Del**ete, deletes the next character

1.6 Summary

Everything in a computer is stored as patterns of bits, binary digits. Bits are usually grouped into 8-bit bytes and their multiples. We represent these bit patterns either as strings of 1's and 0's or as hexadecimal numbers.

We can convert a positive decimal number into a binary number by repeatedly dividing by 2. The remainders form the bits of the binary number. The first remainder is the Least Significant (rightmost) bit and the last remainder the Most Significant (leftmost) bit.

We convert a number from binary to decimal by adding up the powers of two represented by the 1 bits in the number. The left most bit corresponds

Dec	Bin	8-Bit Bin	Hex	Dec	Bin	8-Bit Bin	Hex
0	0	00000000	0	10	1010	00001010	A
1	0	00000001	1	11	1011	00001011	B
2	0	00000010	2	12	1100	00001100	C
3	0	00000011	3	13	1101	00001101	D
4	0	00000100	4	14	1110	00001110	E
5	0	00000101	5	15	1111	00001111	F
6	0	00000110	6	16	10000	00010000	10
7	0	00000111	7	17	10001	00010001	11
8	0	00001000	8	18	10010	00010010	12
9	0	00001001	9	19	10011	00010011	13

to 20 = 1, the next bit to the number of 2's, the next to the number of 4's and so. Thus the number 0b11000101 = 197.

Negative numbers are represented in two's complement notation. In this notation the number of bits in the representation is fixed and the top bit of the number determines the sign, 1 for negative and 0 for positive.

You convert a binary number from negative to positive or vice versa by

1. Write the number out in binary with all bits present, including leading zeros.
2. Invert every bit (that is, replace all 1's with 0's and all 0's with 1's).
3. Add 1 to the number. The result is the two's complement of the original number.

When we use the counting sequence encoding for numbers we can do arithmetic on binary numbers in much the same way that we work with decimal numbers.

English text can be represented by bit patterns using the ASCII code.

Exercises

NOTE all of these exercises are to be worked by hand without using a hexadecimal calculator.

1. Convert 115 to an 8-bit binary number.

2. Convert 23,496 to a 16-bit binary number and to a 4-digit hex number.

3. Convert -17 to an 8-bit binary number.

4. Add binary 115 to binary -17 giving the answer as an 8-bit binary number.

5. Multiply 0b01101010 times 0b01011101.

6. Divide, using unsigned arithmetic, 0b1101 into 0b10110110 giving answer as an 8-bit quotient and 8-bit remainder.

Chapter 2

Introduction to Microcomputers

2.1 Introduction

All computers, from the largest mainframes to the smallest embedded devices, share a common structure and scheme of operation. We shall explore those common ideas in this chapter and illustrate them with examples from the TM4C computer family discussed in more detail later in this text.

We start with an exploration of the hardware structure of a computer and a look at how computers connect to the real world. Then we take a very brief look at a simple program for an Texas Instruments (TI) TM4C123G computer using the Energia programming system. The form of the language that we will use is essentially that used by the Arduino family of computers that has become extremely popular in the last few years. Actual Arduinos range from small systems such as Uno to highly capable ones like the Due based on the same ARM architecture that we shall be using.

2.1.1 The building blocks.

Every computer can be viewed as being built from a set of common building blocks (Figure 2.1). At the core of every computer is the **C**entral **P**rocessor **U**nit, or **CPU**–the brain of the computer. By itself it is a blind, deaf, and amnesiac. It needs one or more **Memories** to store both the information that describes the state of a program and the information that describes the program itself. It also needs one or more Interfaces to the world. All this then provides the framework upon which the Program runs. The program is the set of instructions that tell the computer what sequence of operations to perform. It is this ability for a single set of hardware to perform many different functions depending on the program that is responsible for the

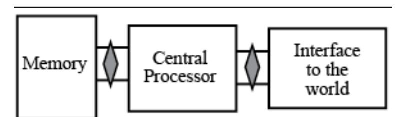

Figure 2.1:
A Generic Computer

success of the computer. A generic piece of hardware can now be turned to almost any function simply by altering the program.

In operation, the CPU reads program instructions from the memory and performs (**executes**) the operations those instructions describe; reading and modifying the memory as required. As a result of the program, the central processor communicates with the outside world over various interfaces. Those interfaces can be as a single switch and a single light or they can be complex devices to show pictures, store and retrieve large amounts of information, understand speech, generate sound output, or accept handwritten input.

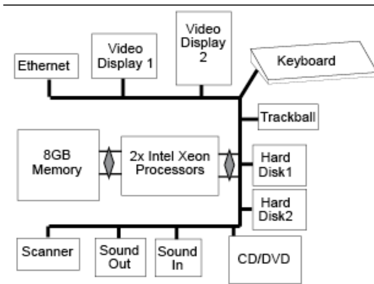

Figure 2.2:
A Desktop Computer

My previous desktop computer (Figure 2.2) is fairly typical It had a fast 64-bit central processor, a large amount of memory, and a lot of complex interface equipment, including two graphical displays, a keyboard, trackball, several disk and DVD subsystems, a sound system, and support for networks, printers, and a scanner. A desktop computer runs many different user programs, word processors, spreadsheets, drawing programs, games, etc., under the control of a massive and sophisticated program called the **operating system**. These programs typically require gigabytes of memory and may cost hundreds of dollars. During normal operation the computer may change which program is in use many times, switching between an email program, a word processor, or a web browser and so forth.

By contrast, many embedded systems have only a few bytes of working memory and few hundred bytes of program memory. Such a system has a small slow processor and fairly simple interface equipment; a few lines of digital input and output, a serial port, and maybe some analogue input. A common example would be an electronic thermostat (Figure 2.3). This is the sort of task for which an Arduino-style system would be appropriate.

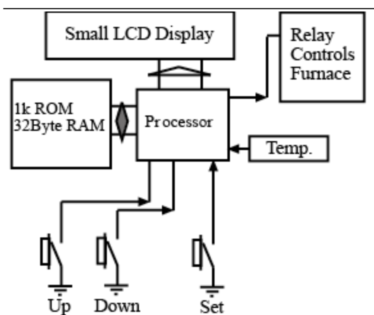

Figure 2.3:
Electronic Thermostat

This is not to say that all embedded systems look like this. At the other end of the range there are machines such as laser printers and automobile engine controllers that have embedded computers as powerful as many desktop machines. A laser printer controller might use a processor as fast and power-hungry as a desktop system, have many megabytes of memory, and even have a local hard disk to store font information. It is still an embedded device because it looks like a printer not a computer.

Somewhere between these two extremes there are the mobile devices that are becoming so much a part of our lives. The cellphone has grown far from its roots as a device for making telephone calls and has grown into the smartphone. Today's smartphones have similar speed, storage, as display capabilities to desktop computers of a less than a decade ago. But instead of posing as the universal computing devices that they really are, they hide behind elegant interfaces that give us easy access to a limited range of functions including communication, media, games, and simple office functions such as calendar and note taking. Looking only a little into the future we can see smartphones extending their functions beyond their cases as more and more devices join the "internet of things". Already fitness watches and

some home automation systems talk to smartphones and the trend will only continue as embedded systems find their way into a wider and wider range of niche markets. Most of these systems are based on various versions of the ARM processor that is at the heart of our TM4C123G computer.

Block diagrams like Figure 2-2 and Figure 2-3 give us a broad idea of the organization of a computer but they do not show the software: the programs that the computer executes and that control everything that it does. Like the interface to the world, the software varies enormously depending on the task that the computer must perform.

An embedded computer may have a program that is only tens or hundreds of bytes long or may have a program as large as a desktop machine. The difference is that the program is built into the computer so that it cannot be changed, or not easily. This program will probably have been written specially for its task rather than bought.

2.2 The Central Processor Unit

The CPU, is the real brain of any computer. It reads the instructions from the memory and performs the operations that they describe. Those operations are very simple ones such as adding two numbers together, comparing two numbers, and deciding which instruction to perform next. Later in this book we shall look inside the CPU of our computers and investigate the individual instructions that it understands. For the moment, however, we will be content to see it as a little black box that can read our programs and, one step at a time, do exactly what they say.

Because individual CPU instructions are very simple they can do very little and it takes a very large number of them to do even simple tasks. The first computers had to programmed by literally flipping switches and pushing buttons to load long patterns of bits into the memories and then telling the CPU where to start working. Very soon people developed programs that could do the tedious parts of this. They created programs called **loaders** that could read data, in the form of holes punched into pieces of cardboard or strips of paper tape, into the memory and run them. They then created programs called **assemblers** that read strings of words and numbers representing a program and translated them into the actual patterns of instruction bits. Finally, they designed new languages, new ways to express programs that were easy for people to understand, and programs called **compilers** that did the much more complicated translation from this new **high-level language** to the machine language.

Some terminology.
Microprocessor: a complete central processor on a single chip. Such a chip forms the heart of a computer but needs external support chips to build a complete system. Microprocessors are found as the key components in laptops, desktops, and workstations, where they are supported by external memory and interface systems. Examples include Intel's Core processors and Freescale's PowerPC chips.
Microcomputer: a self-contained computer on a chip. A microcomputer has not only the CPU but also enough on-chip memory and interface systems to build a complete computer with no other chips. Microcomputers are found in embedded systems from toasters, through cell-phones and MP3 players, to palm-top computers. Examples include NXP's 9S08 family, the venerable Intel 8051 and its descendants, and the currently popular ARM family to which our TM4C123G belongs.

The first successful high-level language, FORTRAN, could calculate mathematical expressions, make simple decisions, and jump from one line in the program to any other line if needed. Decades of research by mathematicians and physicists and, once they invented themselves, computer scientists have

led to the invention of dozens of different high-level languages, most targeted at some particular niche such as scientific programming, systems programming, web scripting, etc. Probably the most successful single language is C, developed in the late 1970s at Bell Labs by Brian Kernighan and Dennis Ritchie. It and its descendants, C++, C#, Java, and D, have proved useful in almost all fields of computing. In particular, the basic C language is particularly well suited for writing programs that work very closely with the hardware, the kinds of programs that run embedded computers. In particular, Arduino-compatible systems are supported by a very nice C++ development system so we shall start our exploration of programming with C and its big cousin, C++.

John Backus and his team at IBM created FORTRAN (FORmula TRANslator) in 1954 and released it commercially in 1957. Since then the language has undergone many changes and re-designs but versions of FORTRAN are still in use on some of the largest, fastest computers in todays world. It remains an important language for scientific programming.

2.3 The Memory

Memory lies at the heart of every device that we could call a computer. The memory stores the information that the computer works with, whether that information represents words on a page, images on a screen, temperatures in a house, or the pressure in a nuclear reactor vessel. The memory also stores the information that represents the program that the computer follows, the instructions that it performs. At the lowest level the memory is made up from individual flip-flop circuits each storing 1 bit of information. These individual bits are grouped into multi-bit cells or **bytes** and those cells in turn grouped into larger blocks.

How Big is a Byte ? 8 bits!
Over the years, computers have been built with memory cell sizes ranging from 1 bit up to at least 36 bits. The larger the cell, the larger the number of different bit patterns that it can hold and so the more flexible it is. At the same time, the larger it is the more costly it is and the more chance that we shall be wasting some of the bits. Since the 1970's most computers have settled on a basic size of 1 byte = 8 bits, as a good compromise size, especially since it is a power of 2 and so fits very nicely into an addressing scheme based on binary number.

At its simplest, we can think of a computer memory as a being like a set of mailboxes. Each memory location has a unique number, its address, and room to hold one item of information, its value. The amount of room in a memory location, usually called one byte, is fixed in size by the designer of the computer but both the value and the meaning of the contents are entirely determined by the program that the computer is running and thus by the person who wrote the program.

Addr 000	Addr 001	Addr 002	Addr 003	Addr 004	Addr 005	Addr 006	Addr 007
Val 02	Val 4d	Val a4	Val 31	Val 2f	Val 73	Val a9	Val c6
Addr 008	Addr 009	Addr 010	Addr 011	Addr 012	Addr 013	Addr 014	Addr 015
Val 1d	Val 1e	Val 21	Val 3f	Val 46	Val f2	Val ca	Val 00

Figure 2.4: Memory Organization

Each memory location holds a binary bit pattern of fixed size, almost always 8 bits which can represent all sorts of things, e.g. a number, a letter, a set of bits describing various binary conditions, or a machine instruction.

If a piece of information is too large to fit in one memory location then we have to group several cells together and spread one piece of information

across several cells. For example, the largest signed number that will fit in one 8-bit byte is 127. If we want to store a larger number then we have to use several cells. If we use two cells then we can store a 16-bit number with its upper half in one cell and its lower half in another and we can represent numbers up to 32,767. In principle, those two cells could be located anywhere in the memory but in practice computers use adjacent cells to hold the several parts of a larger object.

High level languages like C, Pascal, and Java hide the simple pile-of-boxes view of the memory and impose a structure that is easier for humans to work with. In this view the memory consists of named locations called **variables** that represent quantities and concepts in the program. Depending on the need a single variable might hold a single number, a string of letters, or a whole matrix of numbers. When the high-level language works its magic it maps these high-level concepts onto the simple array of bytes and takes care of the mapping so that we never have to worry about what information is is in which memory cell.

Memory units
We count the number of cells in a memory in units of bytes, kilobytes, megabytes, and gigabytes, etc. For this special use the prefixes have slightly different meanings. One kilometer is 1000 meters, 10^3 meters, but one kilobyte is 1024 bytes, 2^{10} bytes. Since memories are always addressed using binary numbers, it makes much more sense to use a power of 2 as the counting unit. Since 2^{10} is so close to 10^3 the choice of power is easy. Here then are the most common binary prefixes and their multipliers

1. 1 kilobyte = 1 KB = 1024 bytes
2. 1 megabyte = 1 MB = 1024 KB = 1,048,576 bytes
3. 1 gigabyte = 1 GB = 1024 MB = 1,073,741,824 bytes
4. 1 terabyte = 1 TB = 1024 GB = 1,099,511,627,776 bytes

2.4 The Interface to the World

All computers have memories organized in similar ways and all have CPUs that perform the same sort of tasks, even if the programming model and the details of the language differ from one processor to the next, but there is enormous variation in the range of interfaces to the world. Different kinds of computers have such different uses that there is no similarity between the interface to the computer in a toaster and that to a desktop computer.

Desktop and laptop computers, machines designed to process information in close cooperation with humans, have a fairly uniform set of interfaces that make up most of our picture of a computer. Such a computer has a keyboard

for the user to type on and a video display on which the computer shows information. It has a hard disk subsystem so that the machine can store large amounts of information even when the power is turned off and the main memory is not working. It has a wired or wireless network connection to share information with other computers, a mouse or its equivalent to communicate visually, a DVD or USB port to load or save large amounts of information, and a sound system capable of some sophisticated effects. All of these interfaces, or **peripherals** (so called because they are attached to edges, or periphery, of the computer) are there to store information and communicate with the human users of that information.

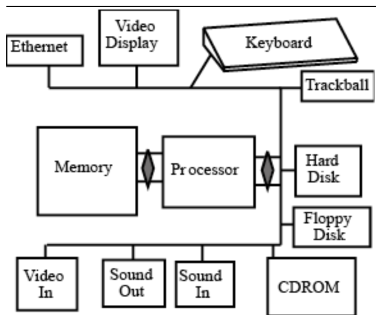

Figure 2.5: My Original Desktop

Figure 2.5 is a block diagram of the desktop computer on which I began writing this book. Since then I have upgraded several times and now have two video displays and haven't seen a floppy disk in years, but the rest is pretty much the same. Notice that each of the peripheral blocks is a complex piece of electronics. Indeed most of these peripherals have circuitry nearly as complex as the computers they serve and many of them have their own embedded computers!

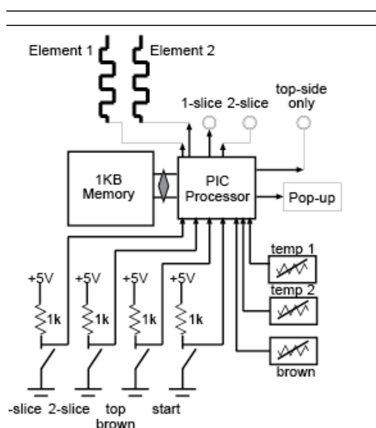

Figure 2.6: A Toaster

By contrast, the computer in a toaster (Figure 2.6) has a minimal interface. For output, it has a few power switches to turn the elements on an off and control the pop-up mechanism and maybe a light or two to tell the user what is going on. For input, it has a few buttons and a sliding or rotary control to set how brown you like your toast. The buttons allow you to make such epic choices as one slice or two, brown on both sides or only one, and of course there is the ultimate button, the one you push down to start the thing toasting. If it is a fancy toaster, then it may also have a sensor or two to monitor the temperatures of the cooking areas. That way the toaster can make the second and third slices exactly as brown as the first, even though the toaster is already warmed up and the cooking time is shorter. The toaster actually has more peripheral devices than the desktop computer but now they are extremely simple; a switch, a light, or a sensor.

2.4.1 Simple interfaces

At its simplest, every interface between a computer and the world takes the form of one or more electrical connections between the CPU and some circuit in the outside world. These connections are brought out to pins on the chip or board that constitutes the CPU. Such connections come in classes, inputs and outputs, and two basic forms, digital connections and analog connections. Any such pin that carries information between the computer and the outside world can be described as an *I/O pin*, meaning an *Input/Output pin.*

Connections are usually described from the point of view of the CPU. Thus a connection is an Output if it carries a signal from inside the CPU out to the world. A connection is an input if carries a signal from the outside

world into the CPU. It is not uncommon to find connections that can be used either way, sometimes as an input and sometimes as an output, but at any point in time they must be one or the other.

Digital connections carry only single binary values between the computer and the world. A binary output is a bin that the computer can set to one of two voltages, high or low (usually a small positive voltage such as 5V or 3.3V for high and 0V for low). A binary input is one that tell the computer whether something in the outside is high or low. It acts like a comparator, giving a value of 1 if the external signal is above some test level and 0 if it lies below. Single binary inputs can, for example, tell if a button is pushed or not. Single binary outputs might control whether a light is on or off at any instant in time. The TM4C123G chip, on which we focus, has 43 such digital connections, of which 35 are connected to the connectors on the LaunchPad board that we use.

Analog connections are more powerful. They can exchange multi-bit numbers with the world. An analog input converts a voltage on a pin to a number proportional to the voltage using an analog-to-digital converter. For example, ten of the TM4C123G pins can be converted, by software, from digital inputs to analog inputs that can convert a voltage in the range 0-3.3V into a number from 1 to 4096. An analog output does the reverse. It converts a number inside the CPU into a voltage in the outside world using a digital-to-analog converter. While analog inputs are a standard feature of almost all embedded processors, analog outputs are still quite rare. It is far more common to want to measure something in the outside than to need to output a controlled voltage. The only common use for analog output is to generate sound and then the task of conversion is often left to external DACs that connect to the computer through digital connections.

2.4.2 The computer's view of the interface

No matter what it looks like to the outside world, inside the computer every device must appear to the CPU, and thus to the programmer, as set of memory locations. The program can write bit patterns to these locations in the usual way but then special hardware can interpret those bit patterns to do something else. Similarly, the program can read from the locations and get information about the state of the hardware rather than just getting back the last value that it wrote there. Such special locations, called Special Registers in the TI documentation, act a bit like the night safes that see on some banks or the book drops in a library. One side opens to the world and allows people to put things into the slot and the other side opens onto either the bank or the library and allows staff to pick up the items.

The simplest kind of interface is the digital **port** or textbfGeneral textbfPurpose textbfInput textbfOutput (GPIO) port. This is is basically a memory location connected to 8-wires, one wire per bit. Each wire has a well defined

The ARM family, like most modern microcomputers, use what is called **Memory-Mapped I/O**. This means that the special memory locations that form the interface to the outside world are mixed in with the regular memory locations. This means that ordinary memory read and write operations are all that it takes to perform I/O.
Older designs featured separate I/O and memory spaces. They had separate wires to connect to I/O memory locations and to regular memory locations. This meant that the I/O space didn't steal from memory, which was important when memory space was scarce

direction either from the computer to the outside world, textbfoutput, or from the outside world to the computer, textbfinput. To the program it is a single memory cell. To the hardware designer it is a set of 8 pins on the computer chip. All of the I/O pins on the TM4C123G chips that we shall use are grouped into such GPIO ports.

2.4.3 Digital ports on the TM4C123G

The TM4C123G chips have six such ports available. They are given simple alphabetic names so we have Port A, Port B, Port C, Port D, Port E, and port F. Internally, each consists of 8 individual connections that can be set to operate as either inputs or outputs on a pin-by-pin basis. Externally, not all of the connections have actual pins which is why there are only 43 connections in the six ports and then some of the pins are used internally on the LaunchPad board so that only 35 pins are available for us to use. Only Port B is complete. Ports E and F are incomplete on the chip and ports A, C, and D have some pins used for special purposes on the LaunchPad.

The Missing Pins
Port A pins 0 and 1 are used to allow programs to talk to the PC in a specially easy way. Port C pins 0-3 implement the debug interface that we use to get our programs into the chip. Port D pins 4 & 5 are brought to a USB connection that we almost never use. I have altered the boards to make these available.

We name the individual I/O pins by combining a port name with a bit number. Thus the lowest order bit of port A is PA_0 and the highest order pin is PA_7 but only PA_2 to PA_7 are available for use. Port B has all eight pins PB0 to PB7 available. Port C has pins PC_0 to PC_3 used for the debug interface while PC_4, PC_5, PC_6, and PC_7 are available. Port D is missing PD_4 and PD_5. Port E has PE0 to PE5 and Port F has only PF_0 to PF_4.

2.5 A very brief introduction to C

Why can't we just program in English?
English turns out to be far too complex and subtle for a computer to translate into an executable program. Part of the problem is that the language has a lot of ambiguity that allows us to communicate subtle shades of meaning while making even simple sentences hard to understand when taken out of their context. The problem of getting a computer to understand natural language is a very hot research topic but we are still a long way from a robust solution. So far it is just a lot easier to teach the people a simplified language to talk to the computer than it is to teach the computer a real language to talk to us.

High-level languages like C++ combine mathematical expressions with simple one-word commands taken from English to produce programs that are accessible to humans and yet can be translated into something that a computer can understand. C++ is a member of a family of languages known as procedural languages. A program in a procedural language is rather like a recipe, a set of step-by-step instructions that, if followed in sequence, take you through a process. At each step the program tells the computer what to do. In fact there are a number of similarities between a cookbook and a C++ program.

After some preliminary material, the main bulk of a cookbook consists of one or more recipes, each a more-or-less independent unit. Similarly, after some preliminary material, a C program consists of a one or more **procedures**. An ordinary C++ program has one extra rule. For it to be a complete program there must be exactly one procedure called **main** in the file. Once the program has been compiled the computer starts running it at the start of

the main procedure. That is why it is called main. The simplest C program consists only of a main procedure.

Each recipe in the cookbook has a name by which you can refer to one recipe in another. For example, my favorite English cookbook has a recipe for a basic kind of cake called a Victoria Sandwich. It also has a number of more involved recipes that use a Victoria Sandwich at some point. These other recipes don't have to repeat all the information in the basic recipe, they just tell you to make a Victoria Sandwich and tell you how much flour to use. Similarly, we could write a simple C procedure to, e.g., write a number on an LCD and then, every time we wanted to write a number on the screen just tell the computer to use the number program and tell it which number to write.

Procedures, subroutines, & functions.

A **procedure** is any self-contained collection of code that can be run as a single unit.

If the procedure returns one or more values to the caller then is called a **function**, by analogy with a function in mathematics. If it does not return anything then it is a **subroutine**.

Example 2.5.1 For example, a piece of code that computes the trigonometric sine of a number would have to be a function to return the value of sine. A piece of code that writes a number to a screen can be a subroutine since it does not need to return anything.

Either kind of procedure can receive information from a calling program in the form of **arguments**. For example our sine function would need to be passed an angle as its argument and would return the sine of that angle as its **result**.

In the same way that a recipe begins with a list of ingredients, a C procedure begins with a list of the variables that the program uses. This analogy goes a little further. An ingredients list tells you how much of each ingredient to use. The C list tells you how much memory each variable needs and also what kind of items you can put into it.

```
/*
* A simple Energia
* program to blink an
* LED connected to PF1
* of a TM4C123G.
* Brian Collett
*/
void setup() {
pinMode(PF_1,OUTPUT);
}
void loop() {
digitalWrite(PF_1,HIGH);
delay(200);
digitalWrite(PF_1,LOW);
delay(200);
}
```

Figure 2.7: Our First Program

2.5.1 Anatomy of a simple Arduino/Energia program

The actual Arduino family of computers use a slightly modified version of the C++ language along with a custom library to create an easy-to-use environment for writing Arduino programs that are called **sketches**. The Energia system that we are using copies this system very closely.

Probably the best way to get started is by examining a simple program and trying to understand the pieces and their relationships. Figure 2-7 shows our first little program. This piece of text would be stored in a file with a

".ino" extension and translated by the Energia system into the machine code that the CPU understands. We'll work our way through it one piece at a time.

```
/*
* A simple Energia program to blink an LED connected
* to PF1 of a TM4C123G.
* Brian Collett
*/
```

These first few lines, beginning with /* and going as far as */ form a **comment**. This is not actually part of the program and it does not get sent to the computer. As soon as the compiler sees the character pair /* it starts throwing everything away until it sees a */. That ends the comment and the compiler goes back to its usual state. Note that the *'s that begin the middle lines are pure decoration. I like to put them into a comment block like this because it makes it easier to see that these lines are part of a multi-line comment. However, they are not required. Note also that while this comment runs over several complete lines, a comment can be as short as you like. Indeed /**/ is a perfectly legal, if rather useless, comment.

A comment is a way for the programmer who wrote the program to leave useful information for anyone who reads it. Every program should have at least one comment, placed at the top of the file like this one. That comment should tell a reader briefly what the program is for. The more complicated the program, the longer and more detailed the comment should be. The idea is that a reader should be able to understand *what* the program does simply by reading the comment, though they would have to read the program to know *how* it does it. It is also a very good idea to include the name of the author in this header comment and a date can also be useful.

The next line is a blank line that does nothing. It is just put there to separate the main routine from the preliminary stuff. You can put blank lines in anywhere to improve the formatting of the code and its readability.

```
void setup() {
```

This line must be present in an Arduino/Energia program. It marks the start of the required setup procedure and it has a lot of parts, all of which are important.

`void` is a **type declaration**. Every object in C has a type associated with it; a description of the amount of memory needed to hold the value and of the kind of object that it holds, such as whether or not it is signed. A `void` object has no value and takes no memory at all. In this case it tells us the `setup` procedure does not return a value. That is, `setup` is a *subroutine*.

`setup()` tells us two things. The pair of parentheses (round brackets) tell us that this defines a procedure. The word in front of them, `setup`, is then defined to be the **name** of the procedure. As we have seen, a complete

Energia program must contain a `setup` procedure and the computer will execute this procedure as soon as the program starts. The parentheses contain declarations of the arguments that the procedure expects. In this case there are none.

The left curly bracket,{, is the start of the **body** of the procedure. The body is a collection of valid lines of C code that ends with a closing curly bracket, }. This is the set of lines of code that get run when you invoke the procedure.

```
pinMode(PF_1,OUTPUT);
```

This is the first line that actually does something! In C terms this "*calls the function named pinMode*" and "*passes it the arguments PF_1 and OUTPUT*". `pinMode` is a *subroutine* that the Arduino library provides to extend the basic C/C++ language.

In this case the `pinMode` subroutine allows us to choose whether an I/O pin shall behave as in input or as an output. This one tells the computer to make port F pin 1 into an output pin.

The semicolon on the end of the line is very important. Every line of C code must end in either a semicolon or a closing curly brace. The semicolon tells the compiler that this is the end of a single complete piece of code, a statement.

Then next line of code is the ending curly bracket for our loop subroutine. The computer will execute all of the code between the opening and closing brackets as a single unit.

```
}
```

We recognize the next two lines

```
void loop()
{
```

as the start of another subroutine. This is the second subroutine required by the Arduino/Energia system. It makes up the main body of our code. As soon as the computer has finished executing the `setup()` code it starts in on `loop()`. It executes the code between the curly brackets (the body of loop) and then goes back and does it again, and again, and again. It keeps doing it until either the power is removed or another program is loaded into the computer. This is known as an infinite loop.

digitalWrite(PF_1,HIGH);

This is another subroutine call. This time we call the subroutine named `digitalWrite`. According to the documentation, it knows how to change the state of a digital output pin. In this case we set port F pin 1 to the HIGH state. If there is suitably connected LED on the pin then this will light the LED.

```
delay(200);
```

This time we call the subroutine named `delay`, passing it the argument 200. `delay` is a routine that simlpy does nothing for some number of milliseconds. It causes the program to pause for 200mS, one fifth of a second.

Why the delay?
Computers live on a very different time-scale than people. The TM4C123G has a clock that ticks 80 million times a second. It takes less than 1 microsecond to turn an LED on and about the same to turn it off. Without the delay this program would turn the LED on and off about 500,000 times per second. That is far too fast for us to see the blink. By adding a delay we can bring the program down to our timescale.

```
digitalWrite(PF_1,LOW);
delay(200);
```

These are now easy. The first sets port F pin 1to the LOW state, turning off the LED, and then the second waits a further 200mS.

Altogether, these last four lines makeup the **body** of the loop. They will be executed repeatedly for ever so the LED connected to Port F pin 1 will turn on for 200mS, then off for 200mS, turn on for 200mS, and so on for as long as we leave the power on.

```
}
```

The closing brace, }, marks the end of the piece of code that began with the most recent {, the body of the loop subroutine.

I am still fighting TeX to force it to put the indentations into code fragments in sidebars!

Note that I normally indent all the lines between the opening { and the this closing }. This is not a part of the language, but it makes the program a lot more readable. Every time I use an opening brace I increase the indentation by one more tab and I decrease the indentation at every closing brace. This way you can look at the shape of the program and see from the indentation which lines are in which block of code.

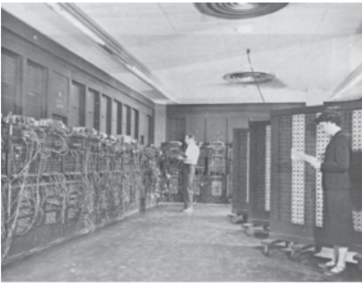

Figure 2.8: ENIAC, one of the world's first computers.

2.6 A Short History of Computers

We owe our modern idea of a computer to the successful marriage between wartime machines to compute artillery firing tables and electromechanical calculators for business accounting. The first fruits of this marriage were vacuum tube behemoths that required the continuous attendance of a team of technicians just keep replacing the components that failed every day; machines that used so much power that their waste heat could have warmed a building.

ENIAC was built at the University of Pennsylvania over a 2 year period starting in 1944. It was first used to solve a problem for the Manhattan project and then dismantled and moved to the Army's Aberdeen Proving Ground in Maryland where it was used to compute flight tables for artillery shells. The machine operated for a total of 80,223 hours over the period 1948-1955. Some portions are still operable and reside in a museum back at the University of Pennsylvania. A huge amount of information on the system can be found at http://ftp.arl.army.mil/~mike/comphist/

The first digital computers built, machines such as Eckert and Mauchly's ENIAC and the code-breaking COLOSSUS, were programmed by rewiring them. The sequence of operations was determined by the way the processor units were connected together. In Figure 2.8 you can see the maze of programming cords connecting the individual units around the left and back walls. These were unplugged and plugged back in in different patterns to alter the flow of operation and thus to program the computer.

In 1945, John von Neumann at Princeton, along with Eckert and Mauchly, proposed that the program could itself be represented by codes stored in the computer memory and so the machine could be reprogrammed merely by altering the contents of the memory. Cambridge University's EDSAC machine, built in 1949, was probably the first machine to operate as a stored program machine, though ENIAC was similarly modified soon after.

Since EDVAC, almost all computers have had a "von Neumann architecture" with a single memory used to store both the program and the data. The alternative is a design pioneered at Harvard that uses two separate memories, one for the program and one for the data. The enormous cost of memory in the early days meant that the von Neumann architecture became the standard design for computers. More recently, the Harvard architecture, with its two separate memories, has been revived for some processors, in particular very small single-chip computers and the ultra-fast specialized digital signal processors.

With the advent of the transistor in the late 1940's, their size and power consumption shrank and the reliability increased so that, by the late 1950's, computer models ceased to be made in ones and twos and were made in tens or hundreds. These are the machines of the science fiction movie, with their air-conditioned rooms, rows of flashing lights, and racks of merrily spinning tape drives.

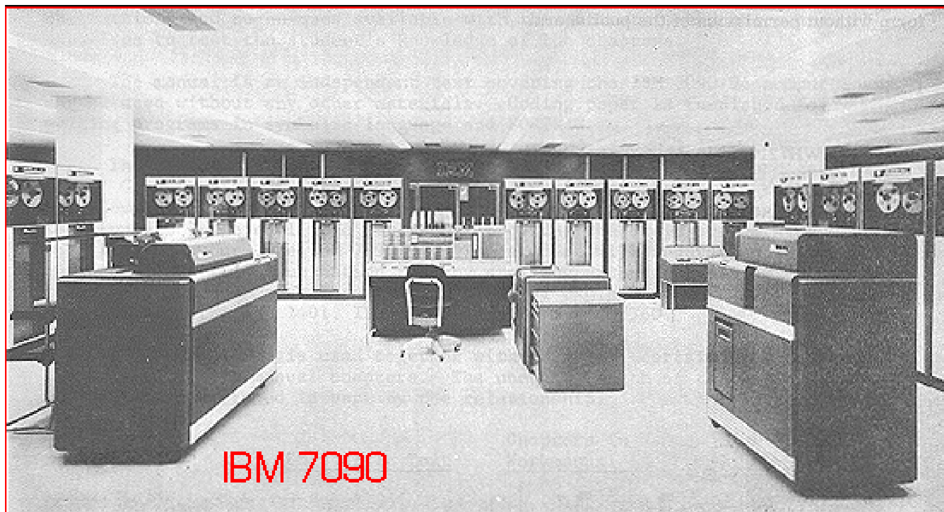

Figure 2.9: IBM7090, The First Commercially Successful Computer

The IBM 7090 was a simple rebuilding of the vacuum tube model 709 using transistors. The resulting computer occupied a fraction of the space and ran 5 times as fast. Following its introduction in 1959 it became the first computer to sell in significant numbers, more than 300 over a 6 year period. They were used by businesses, by the military, and by NASA, during the Mercury and Gemini programs. The last 7090s were retired by the Air Force in the 1980s! Lots more information can be found at http://www.frobenius.com/7090.htm.

The 1970's saw a split between these room sized mainframes, with their staffs of attendant high priests, and the much smaller minicomputers that filled only one or two relay racks and were operated by their users. This signaled a vast change from the massive mainframes, locked away in special sanctuaries tended by a high-priesthood of operators who interceded between the actual machines and their users. Suddenly, instead of a computer being something huge and mysterious that lived behind windows and counters, the computer emerged into the office and, especially, the laboratory and the user became the operator.

Figure 2.10: DEC PDP-1

Figure 2.10 shows a view of of the DEC PDP-1, one of the first minicomputers to enter the market. It has a single small panel of switches and lights (just above the chair) and has a modified electric typewriter for input and output (on the table to the left).

DEC and the PDP Computers
The Digital Electronics Corporation, usually known as DEC, introduced its PDP-1 in 1960. It began a line of PDP computes that would dominate mid-scale computing for more than 2 decades. Here is a quote from the original manual.

> The Programmed Data Processor (PDP-1) is a high speed, solid state digital computer designed to operate with many types of input-output devices with no internal machine changes. It is a single address, single instruction, stored program computer with powerful program features. Five-megacycle circuits, a magnetic core memory and fully parallel processing make possible a computation rate of 100,000 additions per second. The PDP-1 is unusually versatile. It is easy to install, operate and maintain. Conventional 110-volt power is used, neither air conditioning nor floor reinforcement is necessary, and preventive maintenance is provided for by built-in marginal checking circuits.

More information, including the manual, can be found at http://www.dbit.com/~greeng3/pdp1/.

DEC prospered as a company throughout the 1980s but faltered as desktop computers grew in power and the niche for the mini-computer grew smaller and smaller. The company was bought by Compaq in 1998 and then merged into Hewlatt Packard in 2002.

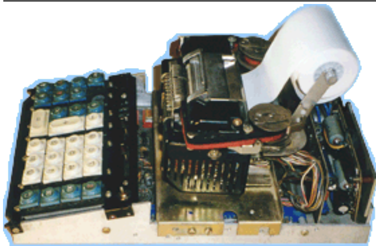

Figure 2.11: First 4004 chips in original prototype calculator

At about the same time the newly formed Intel corporation was working on a design for components to build an electronic calculator that would lead, in 1971, to the first microprocessor, the Intel 4004. This was a set of four chips that together made up a complete, but very limited, computer.

The new idea of a microprocessor caught on quickly. Intel followed the 4004 with an 8-bit version, the 8008 and then an improved, single-chip 8-bit version called the 8080. The 8080 was such a success that it really spawned the microcomputer age. A number of engineers left Intel and founded Zilog to make the Z-80, an improved 8080 and small firms started to make whole computers based on these new chips. One such computer was the MITS

ALTAIR, for which an unknown Harvard student called Bill Gates wrote his first software. The other big semiconductor firms joined in as well with Motorola developing the 6800, the direct ancestor of the chips we use in this course, and RCA developing the first low power microprocessor, the 1802.

Round such microprocessors grew up a new type of computer, the desktop computer. This was first made successful by Apple, using the 6502 derivative of Motorola's 6800. Then around 1980 IBM moved in with the IBM PC, based on Intel's new 8088 microprocessor, a 16-bit enlargement of the venerable 8080. These small machines grew in speed and power and fell in size and cost until today we have laptop computers with more computing power than the mainframes of the 1970's.

At the same time that general-purpose computers were growing smaller, faster, and far more numerous, a much less visible class of computers was emerging. Unlike their better-known cousins, these hidden computers did not run accounting systems or word processors, keep track of appointments, or draw plans for new buildings. Instead of being flexible, human oriented tools with a very distinct personality of their own, these computers were just components in some other piece of equipment. Jobs that had been performed by simple mechanical or electromechanical controllers, such as thermostats, timers, engine control systems, and telephone switching systems, came more and more to be performed by stripped down computers. These hidden computers became called *embedded systems* or *embedded computers* because the computer is embedded inside some other mechanism and does not expose its nature to the world.

At first, these embedded computers were extraordinarily expensive masterpieces of miniaturization each designed for a special high-tech task such as missile navigation or space-flight control. The advent of the microprocessor changed that. Within a few years of the first microprocessor, manufacturers such as Intel, Motorola, and Texas Instruments were producing devices tailored not to making desktop computers but to doing small control tasks; the embedded microprocessor was born. It has been fantastically successful. Today almost all the computers in the world are embedded in something else. Despite the vast growth in the numbers of general purpose computers, they comprise only a small fraction of all the computers being sold. The rest are hiding inside TV's and VCR's, inside remote controls and programmable thermostats, inside laser printers and cellular phones, inside toasters and car engines. They range from unbelievably tiny and powerless devices, fitting in an 8 pin package and boasting only 32 bytes of RAM, to systems that rival desktops with clock speeds in the GHz range and vast memories.

2.7 Summary

A computer consists of a Central Processor Unit that does the work, a Memory that stores the instructions and data for the CPU, and an Interface to the outside world.

The interface consists of I/O pins that can be Inputs from the world to the CPU or Outputs from the CPU to the world.

A digital I/O pin can only be in one of two states, HIGH or LOW. An output pin goes to a high voltage (5V or 3.3V) if set HIGH and to ground if set LOW. An input appears to the computer as 0 (LOW) if the input voltage is less than about 1V and as 1 (HIGH) if the input voltage is greater.

An analog I/O pin can take a large number of states.

A *comment* in C is a string that is completely ignored by the compiler. It is there to make the program easier for humans to read. C treats all text enclosed in /* and */ delimiters as a comment. Similarly, // starts a comment that runs to the end of the current line.

`pinMode(<pin>,<state>)` tells the computer to make <pin> into an input or output depending on the value of <state>, which must be INPUT or OUTPUT.

`digitalWrite(<pin>,<value>)` sets the state of <pin> to either HIGH or LOW depending on the <value>.

`delay(<number>)` causes the program to pause for <number> milliseconds.

All Arduino programs or Sketches must contain two subroutines

```
void setup() {
// Code here will be executed once at the start
}
```

and

```
void loop()
{
// Main code goes here and we be run repeatedly
}
```

The code in `setup` will be executed once at the start of the program and then the code within `loop` will be executed repeatedly until the power is turned off.

Chapter 3

Programming the TM4C123 with Energia

3.1 Introduction

Texas Instruments, the company that produces the TM4C microcomputers featured in this book, produces a suite of programming tools for them called CodeComposer. Later in the semester we will meet CodeComposer and see some of the fancy things that it can do for us. To begin with, however, we will use a much simpler development environment called Energia, that is a recreation of the popular Arduino programming environment adapted to the ARM computers that we use.

Like Arduino, Energia is a sort of portmanteau term, as Humpty Dumpty would say. It refers both to the programming environment that we shall be using and to the set of libraries that it uses to abstract away many of the details of the particular hardware that we shall be using. I shall be talking about both meanings. There are large amounts of additional information available on both the Energia website, *www.energia.nu* and on the original Arduino website, *www.arduino.org*.

Portmanteau
A portmanteau is really a suitcase. The idea is that it is a case that contains a large number of individual clothes. Here we use it to mean a word that contains several different meanings.
The reference is to Humpty Dumpty in Lewis Carrol's *Alice Through the Looking Glass* in the section where Humpty is explaining the poem *Jaberwocky*.

This document introduces Energia and provides a short tutorial to get you started writing programs for the TM4C. I will present an example based on the LaunchPad system used for much of this book.

This document is meant to serve as a tutorial introduction to Energia. I have also included some more detailed information but I have set it off in boxes that you can skim over at a first reading but should come back and re-read once you have built and tried out your first one or two programs.

Before we start the tutorial I want to introduce a few terms.

Host
The desktop computer that runs the development software to create and

debug a program. It transfers the final program to the actual TM4C chip and communicates with it to allow you to debug the program, controlling the progress through the program and viewing the internal state of the chip as the program runs. This is a full-fledged desktop computer with thousands of times the resources of the TM4C chip that we are actually talking to. Its job is to make our life easier!

Target
The actual TM4C computer chip. It talks to the host over a debugging interface built into the LauchPad board. This chip does all the actual work of running a program.

Source code
The human readable form of the instructions that make up a program. This same term is used for assembler source and for programs written in a high-level language. We shall be working with source code in the Arduino version of the C++ language. A complete Arduino/Energia program is called a **Sketch** and is stored in files with a **.ino** extension.

Object code/ Machine code
The machine readable form of the instructions and data that make up a program. This is a long list of binary (often displayed as hex) numbers that make up the actual instructions that the CPU reads form memory as it runs the program. machine code is stored in files you rarely see whose names end in **.o**.

Executable
The complete machine code for an entire program. This is the thing that gets sent to the chip and which runs on the target. It may be the result of gluing together object code from several different source files. This is actually stored in two different files. The raw code for the chip is in a file that ends **.cpp.bin**. It is a vaguely human-readable string of hex numbers. There is also a **.cpp.elf** file that contains the code plus a lot of extra information that the debugging system uses to find things in the running code. You may see these names show up in the output pane of Energia.

Assembler
Either the human readable language used to represent a program or the PC application that translates assembler source into object code. This is a straightforward 1-1 translation. One line of assembler results in one or fewer machine instructions. Some lines are comments for human eyes and others are commands to the assembler.

Compiler
The PC program that translates human readable C++ code into object code (or sometimes into assembler code that is then fed to the assembler and thus turned into object code. Since this process is transparent to the user we don't usually care which happens.). C++ is a general purpose language, not tied to the actual instructions of the particular CPU. Thus one line of C++

usually results in more than one machine instruction. Some lines of C++ may, indeed, require hundreds of machine instructions to implement. This is why we call C++ a high-level language.

Linker
The PC program that takes a set of object code files and glues them all together into a single executable. The linker allows us to split complex programs across several source files. This is particularly useful once we have written a number of programs because it makes it easy to re-use code from one program in another.

IDE
An **I**ntegrated **D**evelopment **E**nvironment is whole package of applications to write and debug programs all gathered together under a single umbrella application, the IDE. This is the thing that we start by double-clicking the Energia icon. Within the IDE, Energia includes an editor for preparing the code, the compiler itself, an assembler, various utilities, and the all important uploader that transfers our code to the target processor ans sets it running.

3.2 Getting Started

The best way to start to learn a complex system like Energia is to do something with it. So we will start by developing a simple program to blink an LED on and off. The LaunchPad board that we use has a rather nice (and very bright) three colour LED included on the board. We will try to make one of the LED blink steadily.

3.2.1 Meet Energia

You can start Energia either by double-clicking the shortcut icon on the Desktop. First a splash screen appears, followed shortly by a window something like Figure 3.1, on the next page. This is a rather simple window. There are two main panes, distinguished by their background colors. The upper pane, with the white background, is the editor. Here you see, and can alter, the text version of your program; its *source code*. Above the editor pane are a tab-bar with a light red background, a toolbar with a dark red background, and the menu bar with five menus, File, Edit, Sketch, Tools, and Help. Below the editor, separated from it by an empty tab-bar with a light red background, is the message pane, which has a black background. You can't type in this pane. Instead, Energia will post messages about your code and the process of sending to the target board.

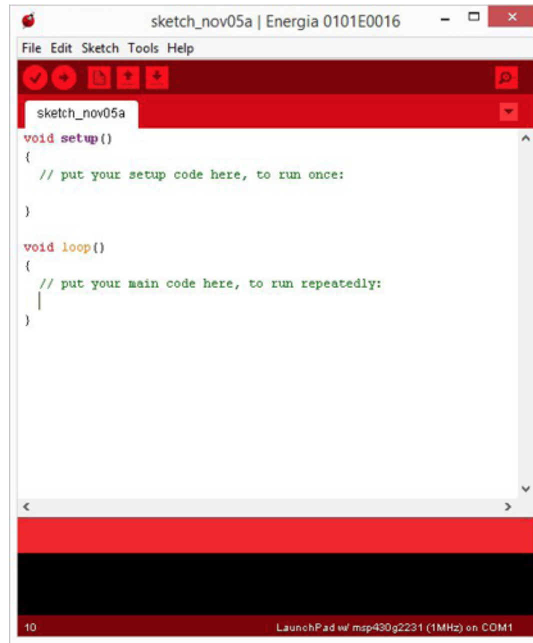

Figure 3.1: Energia starts up

3.2.2 Making the code our own

This template does nothing at all. It is a perfectly valid program that makes the computer sit there doing precisely nothing as fast as it possibly can, for as long as the power is applied. We want to modify this program so that the computer will blink an LED attached to pin 1 of port F. As we will see later, there are two different ways that an LED can be connected. One way. that was historically common, turns on the LED when the bit is 0 and off when the bit is 1. Most people find that a little weird so we will assume the more modern 1 (HIGH) to turn on and 0 (LOW) to turn off.

Energia normally opens onto the most recent sketch. If it has not been run before then you get this empty sketch.
You can always get a new empty sketch either by selecting `New` from the File menu or by pushing the `New` button in the toolbar. That is the button that looks like a sheet of paper with the corner turned over.

Our code needs to perform two separate kinds of operation. There is some stuff that needs to be done only once, at the start of the program. In our case, it needs to set up bit 1 of Port F as an output. Then there is the main task that runs a loop that forever. In our case, we need to turn the bit on and off.

As we learned in section 2.5.2, there is a special pinMode subroutine that takes care of this for us. All we need do is tell it which pin we want (PF_1) and that we want an output and not an input.

```
pinMode(PF_1, OUTPUT);
```

We put the cursor on the blank body line of the setup() routine and type this in. Remember that every complete statement, essentially every line of code, must end in a semicolon (unless it is just a line with a } on it).

Next we need to write 1's and 0s to bit 6 in the data register itself. The

digitalWrite subroutine allows us to control the state of any output pin. We can turn the LED on with the line

```
digitalWrite(PF_1,HIGH);
```

and then off again with the line

```
digitalWrite(PF_1,HIGH);
```

If we put these two lines in the body of the loop() subroutine, the computer will turn the bit off, turn it on, then go back to the start of the loop and repeat, turning the bit on and off forever.

Because each statement ends with a semicolon C allows you to write several statements on one line. I consider it poor style as it tends to obscure the simple top-to-bottom flow of a program.

At this point you should have a sketch that looks something like this.

```
/*
* Blink.c
* A program to blink on and off an LED attached to
* Port F pin 1 of the LaunchPad board.
* Brian Collett, 1/25/16
*/
void setup()
{
    pinMode(PF_1,OUTPUT);
}

void loop()
{
     digitalWrite(PF_1,HIGH);
    digitalWrite(PF_1,LOW);
}
```

Now we have a program that blinks an LED but is in a file that is still called something like *sketch_Nov05a*. You can change the name when you save the new program to the computer.Either go to the file menu and select the `Save As`.... option or press the `Save button`, the one that looks like a down arrow. Either will bring up a file save dialog where you can choose a new name for the program. I called mine Blink and Energia saves the program as a file *Blink.ino* in a directory called Blink..

If you did all this correctly, you should now have your first Energia program.

StarWars memory of the day:
Owen Lars: "What I really need is a droid that understands the binary language of moisture vaporators."
C-3PO: "Vaporators? Sir, my first job was programming binary loadlifters—very similar to your vaporators in most respects."
Only the binary language part is real, I am afraid.

3.2.3 Verify and Upload

The code you have written so far is a piece of human readable text in the Energia/Arduino dialect of the language called C++. Before it can be sent to the TM4C123 to be run, it must be converted from text into the binary machine language that the TM4C123 speaks. This is a rather complicated process but Energia takes care of all the work. All you have to do is to press

either the **Verify** button on the toolbar–the one with the big check mark–or the **Upload** button–the one with the right arrow icon.

Verify
As the name implies, this button *verifies* that the code that you have entered is valid C++ and roughly makes sense. When you press this button Energia goes through a series of steps to convert your text program into an executable form and prints the commands and their results to the black Message pane. You are welcome to scroll through and see the work that goes into turning these few lines of text into executable code. You will know that all is well when the last couple of lines are something like

```
Binary sketch size:  2,424 bytes (of a 262,144 byte maximum)
```

Upload
This button is the one that you will use most often. Like Verify, it goes through the complex sequence of operations required to compile your code, that is, to turn it into executable code. Once the code has compiled successfully, Upload tranfers the executable version from the PC to the LaunchPad, using the USB cable that joins them, and then sets the code running.

Press the `Upload` button and watch the progress of the program and the behavior of the LED. Interestingly, the LED does indeed light up, but it does not go out. Instead of a blinking LED we have an always on LED. So we have a program that is correct in the sense that it compiles but incorrect in that it does not do the right thing. We call this a **run-time error**.

At the moment this is a bit of a mystery (unless you remember Chapter 2). We can learn more if we look at the signal on pin PF_1 with an oscilloscope. It is then clear that the problem with the program is that it turns the LED on and off too fast. With single instructions taking a fraction of a microsecond, our program is turning the LED on and off more than 1,000,000 times per second. No wonder we can't see it blink!

3.3 Slowing Down the Blinking

While there may be times when we would be very happy to generate a output that turns on and off more than 1 million times per second, this is not one of them. The human eye cannot see a light that blinks faster than about 20-30 times per second. This means that we need to insert some wait time into our program. As we learned in Chapter 2, the *Wiring* library provides us with a command to waste time. The subroutine (remember that is just a fancy word for a command) `delay` inserts a pause for some number of milliseconds. Let's look at what the documentation says about delay. Figure 3.2 shows a slightly edited version of the Energia documentation for delay.

Well, I fixed a typo, since the actual documentation says "miliseconds", and I trimmed out a reference to an alternate way of pausing.

If you want to be precise, then the **parameter** is the variable that appears in the code for the subroutine. The **argument** is the value that is given at the point where the subroutine is called. We are often not careful about this difference.

There are two key ideas to talk about here. First is the idea of a **parameter** or **argument**. These are different names for a way to pass information from

delay()
Pauses the program for the amount of time (in milliseconds) specified as parameter.
(There are 1000 milliseconds in a second.)
Syntax
`delay(ms)`
Parameters
ms: the number of milliseconds to pause (unsigned long)
Returns
nothing

Figure 3.2: Energia Documentation for delay

your program to the code inside the subroutine. Parameters are values that are put into the parentheses that follow the name of the subroutine. So that in the call

```
delay(1000);
```

the number 1000 is the single parameter that is passed down. A single subroutine may have many parameters, separated by commas, but they must match what the designer of the subroutine wants. In this case the documentation tells us that there should be one parameter, referred to by the name mS, and that it will be treated as an unsigned long integer. That means that we can pass in any number from 0 all the way to 2^{32}-1= 4,294,967,295. That is a lot of milliseconds, about 50 days worth!

In addition to parameters that carry information down from the main program into the bowels of a subroutine, a subroutine may pass 1 piece of information back as a result. There is nothing that delay needs to tell us so it does not return a return a result; the documentation says that it returns nothing. Operations that do return results are quite common. For example there is a Wiring subroutine millis() that returns the number of milliseconds that have elapsed since the computer started running. In general we call a subroutine that returns a value a function.

3.3.1 Let it Blink

I said earlier that and LED that turns on and off faster than about 20 times per second will be perceived as a steady light. If we want to see our LED blink then we need to make sure that it turns on and off more slowly than this. For example, we could get 1 blink per second if we turned the LED on for 0.5 seconds and then turned it off for 0.5 seconds.

0.5 s is the same as 500 mS so we can wait for 0.5 s with a call of

```
delay(500);
```

Adding two such delays to our program gives us

```
1. /*
2. * Blink.c
3. * A program to blink on and off an LED attached to
4. * Port F pin 1 of the LaunchPad board.
5. * Brian Collett, 1/25/16
6. */
7. void setup()
8. {
9.  pinMode(PF_1,OUTPUT);
10. }
11. void loop()
12. {
13.  digitalWrite(PF_1,HIGH);
14.  delay(500);
15.  digitalWrite(PF_1,LOW);
16.  delay(500);
17. }
```

Change your code to match this program, press Upload, and you should be rewarded with a blinking LED!

3.4 More Energia

So far we have looked carefully at two of the buttons on the Energia toolbar, Verify and Upload. Let's look at the rest of them. Here is the full toolbar.

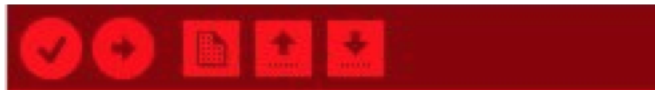

Figure 3.3: Energia Toolbar

We recognize the left two icons as the Verify and Upload icons. The other three are for working with files.

New This is the button with the icon of a sheet of paper. It opens a new sketch and populates it with the default code. It is a common way to start a new program. This is the same as clicking the New item on the File menu. The new sketch is given a default name that will be made from the word Sketch and a date.

Open This is the up-arrow icon and it presents you with a choice of all the sketches that it knows about on the computer. You can then pick one and it will open it in a new window and let you work on that sketch. This is the same as **File | Open...**; that is, the same as selecting the **Open...** item from the **File** menu.

Save This is the down arrow icon. It allows you to save your sketch to the computer. If the sketch is a new one then it will let you change the name to something more useful than *Sketch_Jan16a*. This is the same as **File|Save. . . .**

3.5 Talk to me. Simple text I/O.

Blinking LEDs is all very well but it would be really nice if we could talk to our microcomputers. Unlike their big desktop cousins, microcomputers don't have keyboards and screens to make communication easy. It would be nice if the desktops could help out their little cousins, sort of lend them their keyboards and screens. This is often not practical for a finished embedded computer. After all, if it looks like a computer then it is not really an *embedded* computer. But a keyboard and screen can be extremely powerful tools during development and debugging. So Arduino/Energia provides tools to let the microcomputer access the keyboard and screen of its desktop host.

3.5.1 PuttyTel

The key to this trick is a simple digital interconnection called a **serial port**. As we shall see in more detail later (Chapter **??**), a serial port uses two signal wires and a ground wire to carry binary data between two computers. Our LaunchPads have several such ports and our desktop PCs have one (you can add more using USB if you have to). The first thing that we need is a program to talk to the serial port on the PC. A good choice is a free program called **PuttyTel** (*http:/www.putty.org*). This opens a window on the PC screen and arranges that any keystroke typed into the window is sent out the serial port and any characters that arrive through the serial port are displayed on the screen.

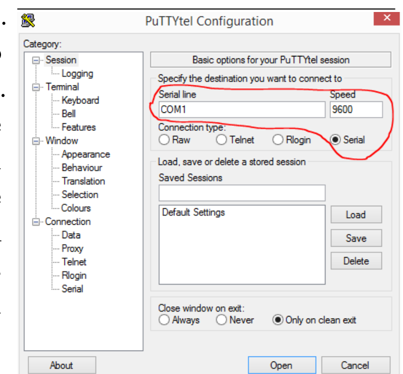

Figure 3.4: PutTTYTel setup

When you first launch PuttyTel you will get a setup window that looks like Figure 3.4.

I have circled the only bits that we care about. We start by clicking the `Serial` button. This should make the program fill in **COM1** in the left-hand text box and the number 9600 in the right-hand text box. We will look at these values a bit more later. For now just remember the 9600 and can start the program by clicking the **OPEN** button. That should make the dialog disappear and leave us with an empty black window, like this.

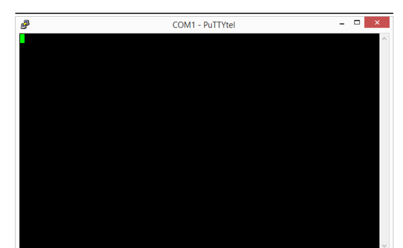

Figure 3.5: PutTTYTel first opens

When this is the frontmost window, we can type on the PC keyboard and know that every character is being sent out the serial port. If you try this then you will find that nothing seems to happen; no characters appear in the window. This is because there is nothing on the other end of the serial connection. The only characters that appear in the window are ones that

are arrive over the serial port. Since there is nothing connected to the serial port, nothing will arrive.

3.5.2 Making the Connection

Although our serial connection only uses three wires, most serial connections are made using cables with 9-pin **DB-9** connectors on their ends. They look like Figure 3.6.

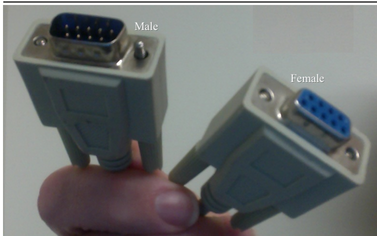

Figure 3.6: DB-9 Connector

Each connector has an outer metal shell that also serves as a ground connection. One end has a series of 9 metal pins arranged in two rows. We call this the **male** end. The other end has a black plastic block with 9 holes, also arranged in two rows. This is the **female** end. Your desktop PC should have a male DB-9 connector on the back and you need to connect plug the female end of your cable in to that. It helps to tighten the locking screws so that the plug won't fall out.

Your LaunchPad does not have a DB-9 connector on it. Instead, you have a little PC board with such connector on one side and 4 pins on the other side. Connect this to the male end of the cable and plug the pins into the breadboard so that the pin names on the computer match those on your little adapter board. The result should look a bit like this.

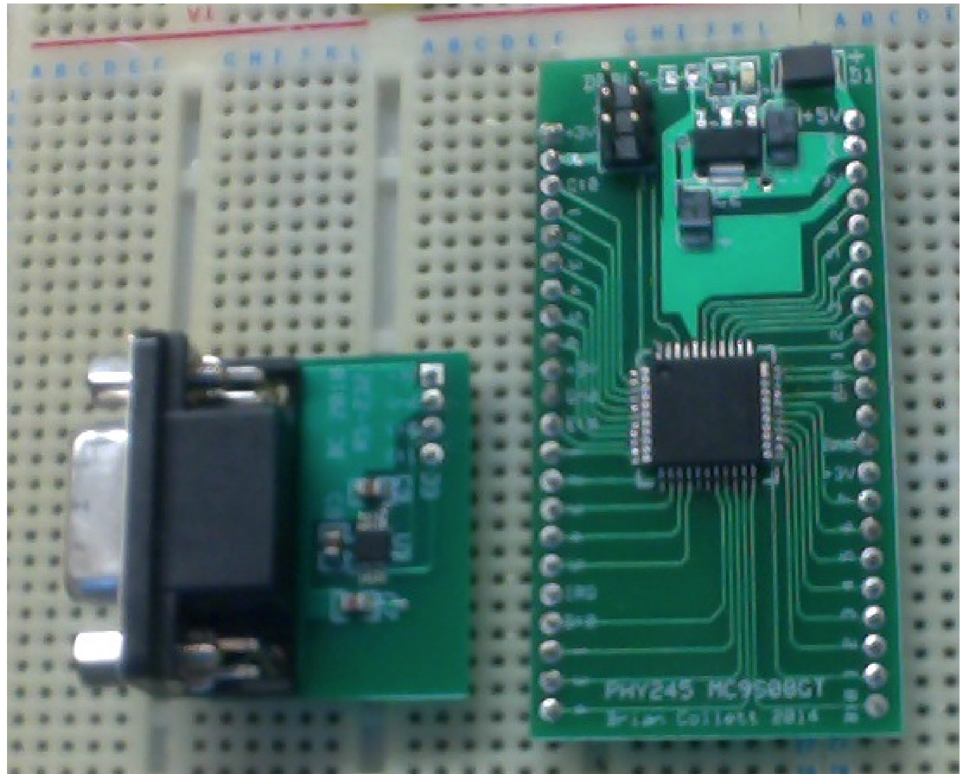

Figure 3.7: Serial Adapter Board

Even with everything turned on, this will still do nothing. We need a program at the TM4C123G end to talk to its serial port.

3.5.3 A First Program

The standard C/C++ library provides a range of routines to handle text input and output. On a desktop computer these routines would use the window and keyboard of the computer on which the program was running.

Our little embedded processors do not have keyboards or displays so that the ordinary C/C++ routines aren't much use. Instead, Energia/Arduino provides a set of routines that talk to the host PC over a serial port. Our first simple program will make the serial connection and then send a character to the PC. For this we shall need two routines, `Serial.begin()` and `Serial.write()`.

The first thing that we will need is

Serial.begin(<baudRate>);

This routine steals two pins from port A (PA_0 and PA_1) and converts them into the two serial pins. It needs one extra piece of information, the speed at which to send data, what is known as the **Baud rate**. Back when we started PuttyTel we accepted the default speed of 9600 so we have to use that same number now. So the first line of our program is

```
Serial.begin(9600);
```

Our program is going to be simplest possible. It will sit sending a single 'a' character out the serial port. After the call to `Serial.begin()` the Tiva is can send characters over the serial line to PuttyTel and receive characters from it. We send a character out with the routine `Serial.write( <character> )`, which must be passed the ASCII code for the character. We looked at the ASCII code back in Chapter 1. There we found that the binary code for the lower case letter 'a' is 1100001_2. C allows us to write numbers in any of the forms that we met in chapter 1, decimal (97), hexadecimal (0x61), binary (0b01100001), but it also provides a special form for ASCII characters. We can simply put the character in single quotes and C will replace it by the correct ASCII code. Thus we can write our lower case 'a' with any of the lines

```
Serial.write('a');
Serial.write(97);
Serial.write(0x61);
Serial.write(0b01100001);
```

This gives us our first program.

```
void setup(void) {
    Serial.begin(9600);
}

void loop(void) {
    Serial.write('a');
}
```

If you run this program then the PuttyTel window should start to fill up with 'a's. You can experiment with putting different values in the call to `Serial.write` and seeing what letters they produce.

Writing strings|

It would be very tedious if we had to use one `Serial.write()` call for each letter that we want to write so the Arduino provides Serial.print(<string>), which sends an entire string of character in one call. As we shall shall see later, this print command is quite powerful and can do a lot more than just print simple strings, but this is where we start.

Strings in C are in general somewhat complicated and we shall learn a lot more about them later (??ref??). Constant strings, however, are very easy. Just as we can make a single character constant by surrounding a letter with *single* quotes, so we can make a string constant by surrounding the string with *double* quotes. Thus we have strings such as

```
"Hello"
"Hello, world!"
"How's it going?"
```

We can send these to the PC by passing them to 'print', like these

```
Serial.print("Hello!");
Serial.print("Hello, Hello!\r\n");
```

The last of those is a bit weird. It has some extra stuff at the end. If you cast your mind back to Chapter 1 you may remember that the first 32 ASCII characters are not not printable symbols but represent directions to the display to do fancy things. In particular the character with ASCII value 0x0D is a carriage return character while 0x0A is a linefeed. C provides a way to put such special characters into string using special two-character sequences called **escape sequences** such as these. `'\r'` is the ASCII value 0x0D that tells the display program to move the cursor back to the start of the line. It is the carriage return sequence (`'r'` for return). `'\n'` is the ASCII value 0x0A that moves the cursor down one line (`'n'` for newline). If we want to move the cursor down to the start of the next line then we need both of these special characters so it is very common to end a string intended as output with the `"\r\n"` sequence.

Arduino provides an alternative way to print a string and move onto the next line. In addition to `Serial.print(<string>)` there is a second form that prints the string and moves to the start of a new line. It is called `Serial.println(<string>)`. To demonstrate this version, here is the classic first C program, HelloWorld. It opens a serial port and prints a string to PuttyTel, moves to a new line, and then just sits doing nothing until you turn it off.

```
void setup(void) {
    //
    // Put any one-time setup code here.
    //
    Serial.begin(9600);
    Serial.println("Hello, world!");
}

void loop(void) {
    //
    //  Code here runs in an infinite loop until
    //  something external stops it.
    //
}
```

You may wish to experiment with different combinations of printable strings and escape sequences and see what sorts of effect you can achieve.

3.5.4 Talking to ourselves

Serial.read

So far this has been a one-way conversation. The microcomputer can talk to the PC but we can't say anything back. It is time to meet `Serial.read`. When you call this routine looks to see if a character has arrived from the PC and returns its ASCII code if it has. Otherwise it returns the value -1, which does not correspond to any ASCII character (it is an 8-bit value and all the ASCII codes fit in 7 bits and so have the msb set to zero). This is called a **non-blocking read**. It can be very useful in embedded programs that rarely get characters and spend most of their time doing something else.

Example 3.5.1

Consider of a traffic-light program. If the pedestrian button sent a character to the traffic light controller then the controller could go about its business of keeping the lights changing on time and would only need to process a button push when one was present.

Contrast this with the **blocking read**.It will not return to the caller until a character is ready to be read. This would bring our traffic light program to a complete halt. However, many kinds of program that interact with people

want to spend most of their time waiting for the person to press some keys. In that case a blocking read can be much more convenient.

Example 3.5.2

A simple calculator program is much easier to write with a blocking read. It wants to sit doing nothing, waiting for the user to press buttons. Once the buttons have been pressed it performs a calculation that takes it almost no time and goes back to waiting for input.

We need to find a way to provide a *non-blocking read* that will return immediately if there isn't a character so that the program can go on working while it waits. Because `Serial.read` returns a unique value (-1) when there is no character, it is pretty to make a blocking version. Here is a subroutine that will do the trick.

```
char WaitForChar(void) {
    char ch = Serial.read();
    while(ch == -1) {
        ch = Serial.read();
    };
    return ch
}
```

This keeps reading from the serial port until it gets a valid character. It is infinitely patient and will keep trying until someone has pity on it and send it a character.

A simple Echo program

As a demonstration, here is a program that will sit waiting for characters to come from the PC and, every time it finds one, send it back. I am writing this with the non-blocking read so we need to test each result from read before we send it out. Otherwise we send a huge number of -1 characters (the delete character) to the PC. Since the program needs to call read repeatedly, we will put the code into the loop of our program. Our first try looks like this.

```
void setup(void) {
    //
    // Put any one-time setup code here.
    //
    Serial.begin(9600);
    Serial.println("Program echoes characters back to user.");
}

void loop(void) {
    char ch = Serial.read();
    if (ch != -1) {
        Serial.write(ch);
```

```
    }
}
```

If you run this code with you LaunchPad connected to PuttyTel then you should find that any key that you type on the keyboard will show up in the window. You can check that this is due to your computer either by stopping the program or by turning off the power to the LaunchPad.

A Command Driven Program

Now that we can read characters from the user, we can write programs that change their behavior in response to typed commands. This will become useful once we start to interface our computers to our robots. You can imagine controlling a robot's behavior by sending it commands such as the letter 'f' to move forward, the letter 'l' to turn to the left, and so on. Here is the outline of such a program.

```
void setup(void) {
    //
    // Put any one-time setup code here.
    //
    Serial.begin(9600);
    Serial.println("Simple Robot Controller.");
    Serial.println("Responds to single letter commands as follows:.");
    Serial.println("f Move forward.");
    Serial.println("b Move backward.");
    Serial.println("l Turn to the left.");
    Serial.println("r Turn to the right.");
}

void loop(void) {
    char ch = Serial.read();
    if (ch != -1) {
        Serial.write(ch); \\ Echo so user sees what they typed
        if (ch == 'f') {
            // Code to move forward
        } else if(ch == 'f') {
            // Code to move forward
        } else if(ch == 'b') {
            // Code to move backward
        } else if(ch == 'l') {
            // Code to turn left
        } else if(ch == 'r') {
            // Code to turn right
        } else {
            Serial.println("");
            Serial.print("Unknown command ");
```

```
                Serial.write(ch);
                Serial.println("");
            }
        }
    }
```

Note the use of `Serial.println("");` to just print a newline and note how we combined a regular print, a write, and a println to build a more complicated output line.

3.5.5 Serial.readBytesUntil

You don't have to want to do anything very fancy before you need to be able to enter more than one character at a time. Energia/Arduino provides a way to read a whole bunch of characters into a string with the routine

```
Serial.readBytesUntil(<termchar>, <buffer>, <length>);
```

This is a lot more complicated than any previous routine. Ideally, it reads characters from the input and stores them into the buffer until either it runs out of space or it sees the special character <termchar>. For example, we could read a name into a string, stopping when we see a space, with the code

```
char nameBuffer[16];
Serial.readBytesUntil(' ', nameBuffer, 15);
```

There we see a space character used as the terminator.

Timeout

I said ideally. In reality, the routine is extremely impatient. Unless we take special measures it will only wait 1 second for us to type our name in. That is probably *not* what we want. This is because the `Serial` system comes with a built-in setting that means that it will only wait for 1 second before giving up on *any* input command. We will normally want a *much* more patient system. We tell `Serial` to be patient by setting the `Timeout` value to a very large number. Because of the weird behavior of signed and unsigned numbers the best way is to call

```
Serial.setTimeout(-1);
```

immediately after we call `Serial.begin`. We shall see a lot more of this.

This works because the internal representation of -1 is a bit pattern with every bit set to 1, exactly the same as the bit pattern for the largest possible number of that size.

3.5.6 I/O with Numbers

The three I/O routines that we have met so far allow us to move characters between the PC and our LaunchPad computer. These are the basis of all

serial interaction. However, we often want to deal with numeric information instead of character information. We need ways to write numbers out and read them in.

3.5.6.1 Printing numbers

The `Serial.print()` routine can do more than just print fixed character strings. This is an example of an **overloaded** function, a function that can take different kinds of arguments and do something appropriate with them. So far we have used it to print strings but if we pass in a single integer instead then it will interpret the bit pattern as a number and print the value in decimal. As before, there are two forms

```
Serial.print(<int>)
Serial.println(<int>)
```

that differ only in whether or not they start a new line. For example, we can use the two different forms to print a message with a number in it like this:

```
int age = 21;
Serial.print("Your age is ");
Serial.println(age);
```

If you put this into a program then it will print out

```
Your age is 21.
```

In this simple case I printed the value of a single variable. I can also print the value of an expression, like this:

```
int nEgg = 27;
Serial.print("You have ");
Serial.print( nEgg / 12 );
Serial.print(" dozen eggs with ");
Serial.print( nEgg % 12 );
Serial.println(" left over");
```

which will print

```
You have 2 dozen eggs with 3 left over.
```

and will move to the start of a new line.

3.5.7 Printing in Hex

Because we are mostly going to be thinking about values inside our microcontrollers as bit patterns, we often want to see them in binary. Energia/Arduino supports this by passing an extra argument to the `print` routine. In fact, not only does it support printing in binary, but it also allows you to print in

hexadecimal and in octal (old-fashioned base 8), as well as to explicitly ask for a decimal value. The following forms are supported:

```
Serial.print(<int>, DEC); // Explicitly print decimal
Serial.print(<int>, HEX); // Print in hexadecimal
Serial.print(<int>, OCT); // Print in octal
Serial.print(<int>, BIN); // Print in binary
```

Thus we can print the same number in all these ways and get all these different answers.

```
Serial.print(123, DEC); // Prints as "123"
Serial.print(123, HEX); // Print as "7B"
Serial.print(123, OCT); // Print as "173"
Serial.print(123, BIN); // Print as "1111011"
```

Note that the non-decimal formats do not add any information to tell you about the base that is being printed. In C/C++ we have to write that number as `0x7B`, but when we print it all we get is the `7B`.

3.5.8 Reading numbers

Energia/Arduino also makes it easy to read numbers from the input. To read a single decimal number from the input we use the routine

```
Serial.parseInt();
```

It will read characters from the serial port until it sees one that is not a valid digit and then it will stop and return the value.

BEWARE
Serial.parseInt only behaves in this sensible way if you have set the timeout value to some large value. Otherwise parseInt stops when it sees a non-digit OR when it times out, whichever happens first.
This is why I recommend always using Serial.setTimeout(-1).

```
void setup() {
    Serial_begin(9600);
    Serial.setTimeout(-1);
}
void loop() {
    int value;
    while (1) {
        Serial.print("Enter a decimal number ");
        value = Serial.parseInt();
        Serial.print(value); Serial.print(" in hex is ");Serial.println
    }
}
```

I have made a rare use of the ability to put multiple commands on a single line, separated by semicolons. It sort of makes sense here because the three commands print a single line on the output. I would not try this for anything longer than this.

3.6 Summary

We can write numeric values in C using decimal (100) hex (0x64), binary (0b01100100), and ASCII ('d') representations. We store values that change in *variables* of types such as

```
char             // 8-bit number (range -128 -> 127)
int              // 16-bit signed number (-32768 -> 32767)
unsigned int     // 16-bit unsigned number (0 -> 65535)
```

All variables must be *declared* before any other statements in a program.

We change the values of variables using *assignment* statements of the form

```
 <var> = <expression>;
```

We can talk over a serial cable to a PC running a terminal program such as PuttyTel using the routines

```
Serial.begin(int baudRate);      // Makes  connection at speed.
Serial.write(int theChar);       // Writes one character to PC
Serial.read();                   // Waits for 1 char and returns it
Serial.print(char* theString)    // Writes whole string to PC
Serial.printf(int arg)           // Writes integer to PC in decimal
Serial.printf(int arg, int base) // base is one of DEC, BIN, OCT, HEX
Serial.parseInt()                // Reads a single number
```

Chapter 4

Basic Ideas of Programming in C

4.1 Introduction

Fundamentally, a computer is a tool for executing programs. A program is a set of instructions for performing some task. A familiar, non-computer, example of a program would be a recipe. It consists of a list of the resources that you need, the ingredients list, and then a set of instructions that tell the cook what to do with the ingredients. A computer program is like a recipe. It has a list of resources, the data items that we process, and a list of instructions that tell the computer what to do with the resources.

From the CPU's point of view, the program must be expressed as a series of machine instructions, because those are the only things that the computer knows how to manipulate. Unfortunately, even with an assembler to help us, programming at the machine level requires that we deal with a level of detail that is very difficult for humans. Computers have tiny minds that work on one detail at a time and see only the details. People have minds that get confused by too many details and work best thinking in terms of larger, higher level, more abstract, ideas than the tiny details that satisfy computers. In order to bridge the gap between the human programmers and the computers that execute their programs, the humans have developed tools to let them do their thinking at the more abstract and then translate the ideas into the detailed code for the computer. The basic abstract tools are high-level languages and the compilers that translate them into the machine code that the computer can understand.

Natural languages, such as English, evolved to allow humans to communicate with each other. They allow humans to express the full range of ideas to each other (indeed, there is evidence that languages are so integral to the way that human brains work that it may be impossible for us to think any thought that cannot be expressed in our language). A computer language is

an artificial language developed to make it easier for human programmers to tell computers what they want them to do. That is a much more restricted assignment than supporting the expression of all thought and so computer languages are much simpler than human languages.

C vs C++
I mostly talk about C because pretty much everything that we will need in this course is available within the C language. However, Arduino and Energia actually use the full GCC C++ compiler and so you can actually use any of the fancier features that C++ provides. I will only mention two. First, C++ allows us to declare variables anywhere we like, so long as it is before their first use. C requires that all declarations come before any executable statements. Second, some of the Arduino libraries, such as the serial library, make simple use of objects.

This chapter starts our formal study of the computer language called C (and yes, it did come after B and it has descendants called C++, C#, and D). This is a language that was designed for writing programs that are closely matched to the hardware upon which they run. While, in its fullest form, it is capable of expressing any kind of program that can possibly be written, it lacks many of the tools that more sophisticated languages provide for modeling more abstract ideas in the real world. However, since the language stays quite close to the hardware it is especially suitable for writing programs that manipulate the hardware, embedded programs of the type we wish to write. We shall build our understanding of the language a little at a time starting in this chapter with some of the most basic ideas, the procedures from which a program is built and the variables and their types in which we store our data.

4.2 Programming

A **program** is a detailed set of instructions for carrying out one or more algorithms. It is usually a set of instructions for a computer, but people can use programs too. For example, a recipe is a program; a set of more or less detailed instructions for constructing a particular dish. Another example of a program for humans would be the directions for getting to someone's house. There is room for a certain amount of imprecision in these human based examples–e.g. "a pinch of salt", "cook until golden", "about half a mile"–because the human using them is supposed to possess intelligence and initiative. There is no such room in computer programs because computers possess neither—they are extremely careful and persistent morons. A computer will do *exactly* what you tell it to do. It is up to *you* to tell it exactly what you want it to do.

Typically, we can identify four stages, or steps, to programming a computer to perform a task;

1. High-level Design: figuring out how you would do the task,
2. Detailed Design: splitting the task into its basic operations and decisions,
3. Coding: expressing all this in a language that the computer can understand.
4. Debugging: figuring out why the program does not do what it should, and altering it so that it does.

The second and third steps are the subject of this chapter. These are relatively straightforward steps and can be taught. Unfortunately, they are

in some ways the least important. The most important step is the first one, making sure that you understand the task completely, followed closely by the fourth, fixing the mistakes in the first. The trouble is that these are difficult to describe; you have to learn them by experience.

The fourth step, **debugging**, is particularly frustrating in this regard. You put hours of work into designing and implementing a program and it doesn't work correctly. You spend ages looking at it, fiddling with it, trying what you know of debugging, and it still doesn't work. So you ask an experienced programmer for help and she spends 3 seconds looking at the program then does something simple and tells you which stupid mistake you have made. It has happened to us all. It goes on happening. The only thing that changes is that eventually you are the one who gets asked for help, gives it, and tells the person you helped what a silly mistake they made. It doesn't stop you making your own mistakes though!

Debugging
The term debugging, meaning getting the problems or 'bugs' out of a program, has a history which predates the computer. The Oxford English Dictionary traces the first written mention of the term to an article in the Journal of the Royal Aeronautical Institute in 1946 that speaks of "debugging an engine". Apparently the term was sufficiently established in spoken currency to appear in the literature by then. It seems likely that the term came to computing with the engineers who worked on the first wartime computers.
The most famous case of a computer bug was recorded by Admiral Grace Hopper (one of the designers of COBOL and a pioneer in program testing). In 1948 a technician traced a problem in the Mark II Relay computer at Harvard to a dead moth trapped in the contacts of a relay. Grace Hopper duly recorded the event in the computer log book as the "first actual case of a bug being found" and stuck the moth in with sticky tape. The book and the moth are on display in the Smithsonian Institution.

Figure 4.1: The first actual bug!

4.2.1 Flowcharts and Pseudocode

While I will concentrate in this chapter on presenting the details of the C language needed to execute step three of our general scheme, I want to start by mentioning some tools to help in steps one and two; flowcharts and pseudocode.

A **flowchart** is a diagram that represents a piece of a program. It consists of a set of boxes joined together by arrows. The boxes contain short English or mathematical statements that describe actions that the program must take and the arrows tell you how to navigate round the diagram. Flowcharts can help us visualize the operation of a computer program, especially one that makes many decisions.

Pseudocode is a kind of fake high-level language. It is a way of expressing mid-level programming ideas that is reasonably easy for humans to understand without bothering with the details needed to make a valid C program. Once programs get more than a few lines long it can become hard to both express the main ideas and get all the details right. In that case it can be helpful to sketch down the ideas in a mixture of bits of code and fragments of English without worrying about details of syntax. Once the ideas are clear then it is usually pretty simple to translate the pseudocode into real C.

Flowcharts are the best tools for designing programs that must make a lot of decisions. In them, the decisions are clearly visible as are the resulting paths through the code. Pseudocode is not as good at making the flow of control visible but it is a lot easier to translate into real code. Simple programs are best designed using pseudocode that you then translate into C. More complex programs are often best designed with flowcharts that you express first as pseudocode and then turn into real C.

Most new programmers are reluctant to spend the few extra minutes that

Programming Discipline
Edsger Dijkstra coined the phrase 'A Discipline of Programming' for a famous and very beautiful book that he wrote on the subject. It is a good phrase to remember. Good programming requires discipline as well as creativity. Unfortunately, the real benefits of discipline don't become apparent until you are writing fairly long and complex programs By then, if you have learned poor habits, they are very hard to break. In particular, the habits of careful design and of rigorous commenting are tedious and boring to acquire and seem like a real waste of time at first. It is only when you have to write or modify a long program that their benefits become apparent. By then it is too late. This is why I stress the need for discipline from the beginning so that you build good habits and don't have to relearn later. That is what teaching is all about; trying to prevent students from repeating the mistakes of their teachers!

it takes to design a program using these methods. Early programs are easy enough that they charge straight in and start writing code because it is obviously so much faster to do it that way. And at the beginning it works pretty well. Pretty soon, however, programs get complicated enough that this technique becomes a hindrance. It still seems fastest to skip the flowcharts and pseudocode and just write the code. But if you do it that way, then you bog down in details and actually spend twice as long getting to a working program as the people who did it right from the very beginning.

As I present the control structures of C in the next few chapters I will use flowcharts to illustrate many of the structures. Then, as I present more complex programs, I will often use pseudocode to express the ideas before translating them into the final C code.

4.3 Introduction to C

It is time to study the details of C and explore the basic structure of a C program. Since a computer program exists to manipulate bit-strings in the computer memory, the basic C entity is a named block of memory. There are two basic classes of memory block, ones that store data, called **variables**, and ones that store code, called **functions** or **subroutines**. Named entities are used to construct various types of **statement**, the building blocks of almost all programming languages. We have already encountered some kinds of statement such as the function call and the while loop. Before we examine the details of variables, statements, and procedures we need to develop a little notation.

4.3.1 Notation–Meta-syntactic elements.

A syntactic element is a legitimate piece of a language. For example, in english, nouns, verbs, subjects, predicates, etc. are all syntactic elements. A meta-syntactic element is something that stands in place of a real syntactic element. Thus, we might describe the form of a simple sentence using meta-syntactic elements

<subject> <verb> <object>.

The elements in angle brackets, and the brackets the delimit them, are the meta-syntactic elements. A real sentence would be made by replacing each element by an actual element of the appropriate kind. For example, the sentence

The dog drinks the martini.

is of this form, where "The dog" is the subject, "drinks" is the verb, and "the martini" is the object.

From now on, I shall often use angle brackets to surround such elements. The words inside the brackets are chosen to tell you what sort of thing the real element is. Of course, the brackets do NOT appear in the real form. For example, an assignment statement has the form

Occasionally we shall run into the problem that our meta-syntactic elements such as <, >, [, |, and], are also valid symbols in C. If there is any chance of confusion then we shall surround the C symbols with single quotes.

<variable> = <expression>;

and so an example of a real assignment statement would be

```
nByte = 2 * nWord  5;+
```

where the <variable> is nByte and the <expression> is 2 * nWord + 5.

In addition to angle brackets we will also need square brackets, [and], to mark optional elements. For example, we shall meet the following line:

<type> <name> [= <expression>];

Here the square brackets tell that the equals sign and the expression are optional elements. You can have a valid statement with them or without them. However, if either is present then both must be. You can't have the = without the <expression> or the <epxression> without the =.

4.3.2 Names in C

Every program needs a way to represent different kinds of object that are specific to the task in hand and thus not built into the language. Since a C program is made up from text, we use text strings called **names** to represent these objects. Examples of objects would include memory locations to hold program data and pieces of code to manipulate those data.

Although it is legal to begin names with the underscore I *strongly* advise against doing so. Compilers regularly use names the start with an underscore for their own behind-the-scenes purposes so it is wise for us to avoid them.

C requires that names start with a letter and may contain any mixture of letters, numbers, and the underscore character '_'. C names are case sensitive so myName and MyName are two completely separate variables. Note also that C reserves a small number of words for its own use and will not let you use any of these as variables. These **reserved words** are

```
auto      break   case      char    const    continue default
do        double  else      enum    extern   float    for
goto      if      int       long    register return   short
signed    sizeof  static    struct  switch   typedef  union
unsigned  void    volatile  while
```

There are times when a variable is so short lived or so generic in its application that it does not really need a long-winded detailed name. In such cases the old standby names like i, j, k, etc. are perfectly adequate. For example, in code that uses many for loops (see below) it may be just fine to use i, j, etc. for the loop variables. Any variable that is visible over more than a few lines of code or that holds any interesting kind of variable should have a sensible name. Named pieces of code are always sufficiently important that they must have meaningful names.

We shall not encounter all of these words in this book, but it is a good idea to know that they all exist. Even if we don't expect to need them ourselves, the C compiler will still not let us use them as names.

If you use short, easy to type, names such as j, k, or a3, then you completely obscure the meaning and intended use of the name. Instead, make a practice of using names that describe the meaning of the named object. For example, if you have a value that is used to count the number of words in a sentence

Capitalization
C++ is extremely case sensitive. 'W' and 'w' are totally different letters from its point of view. Thus `numwords` and `numWords` are different names. It would be a *very* bad idea to use both of them in the same file. C++ would see them as completely different but *you* might well get them mixed up.

then give it a name such as `numWords` or `wordCount`. If you have a string used to ask the user for their name then call it `askNameString` or something like that.

4.3.3 Variables

A variable is a named box into which we store a value. Most variables, as their names indicate, have values that change during the course of a program's execution. Variables have three kinds of information associated with them;

- **The actual value**: stored as a bit pattern in some bytes of memory and manipulated by the CPU.
- **A range of acceptable values**: represented in the language by the **type** and communicated to the compiler in the **declaration**.
- **A meaning**: which exists only in the mind of the programmer and which can be made visible only by the choice of a good name!

All these different pieces of information are represented by the name, so it pays to choose a good one.

In a flowchart or in a pseudocode version of a program we introduce a variable by naming it and giving it a value. All the other information is stored in our minds. So we might introduce a value at first use like this,

```
numWords = 0
```

which introduces the name and gives it a starting value.

In C/C++ we have to provide more information. Before we use a variable, we must not only give it a name but must also tell the compiler what **type** of value it will hold. The **type** is an abstract representation of the range of values that the variable can take.

A **declaration** is a a simple statement that introduces the name of the variable, defines its type, and may optionally give it a value. It has the syntax

```
<type> <name> [= <expression>];
```

While you are not required to give the variable a starting value it almost always a good idea. If you don't specify an initial value then the variable starts off with an unknown value that almost certainly nothing that you want.

where, as discussed above, the brackets tell us that the "= <expression>" is optional. Thus, a possible C++ definition of texttnumWords might be

```
int numWords = 0;
```

The full C language offers a wide variety of types, including an elaborate mechanism for building user defined types. However, in our small systems we will need only a few. For the moment the type must be one of:-

`char`	a signed 8-bit object that can hold a single ASCII character or any integer in the range −128 to +127
`unsigned char`	an unsigned 8-bit object than can hold a positive integer in the range 0 to 255
`short`	a 16-bit signed object than can hold an integer in the range -32,768 to +32,767
`unsigned short`	an unsigned 16-bit object than can hold an integer in the range 0 to 65535
`int`	a 32-bit signed object that can hold an integer in the range -2,147,483,648 to 2,147,483,647
`unsigned int`	he unsigned version of an `int`. It can hold a positive integer in the range 0 to to 4,294,967,295

As you see, these types map directly onto underlying 1, 2, and 4-byte machine values. On desktop computers, it is common practice to use int for most general purpose variables unless memory space is very tight. On micro-computers we try to use the smallest type that will accommodate the range of values expected. Using longer types wastes space and may also run slower.

All of these types are restricted to holding integer values. Most embedded computers provide hardware support for 8-bit and 16-bit integers and some, such as ours, also support 32-bit integers. Support for non-integer values, for values with decimal points, is *much* rarer in microcontrollers. Our TM4C123 is one of the rare microcontrollers that can handle non-integers. It allows you to use the **float** type to hold floating point number with about 6-7 decimal digits of precision.

4.3.4 Constants

While a name stands for a value, a constant is an object whose value is obvious from its representation. For example, `123`, `0x3BC`, `‘t’`, `3.141592653`, and `"a short string"` are all constants. C supports four kinds of constants; **integers**, **reals**, **characters**, and **strings**.

Integers.

Integers are the most common kind of constants because they are the natural objects to represent in a binary bit pattern. The most obvious integer constant is a string of decimal digits, the digits

```
0,1,2,3,4,5,6,7,8,9
```

that does not begin with a zero. Such a constant can have any number of digits, but when it is used the value will be truncated to the number of bits that fit in a storage location. Thus single byte values can range from 0 to 255, two-byte constants from 0-65538, and so on.

You can’t begin a decimal number with a leading zero because C interprets numbers that start with a 0 as being in base 8, or octal. This is a rather obsolete way to represent binary bit patterns that has been superceded by hex.

Strictly, there are no negative constants. You make a negative number by preceding a constant with a ‘-’ sign, but the sign is treated as a separate operator that changes the value from positive to negative. This distinction is normally not worth worrying about and you can treat -1028 as a valid negative constant without any problems.

Because decimal constants are not simply related to the underlying bit patterns it is often convenient to express constants in hex. A hex value in C

begins with the marker 0x or 0X and then consists of a string of hex digits,

```
0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f,A,B,C,D,E,F.
```

Thus `0x1ac` and `0x00FC` are valid hex constants. These are commonly used when we are interested in the underlying bit pattern.

Although standard C stops at this point, CodeWarrior has extended the language to support binary constants written directly as strings of 1's and 0's with a leading 0b or 0B marker. Thus the hex constant 0xC5 could also be written as the binary constant 0b11000101. Because you can't use commas in any kind of numeric constant it is generally not a good idea to write numbers of more than 8-bits in binary. They are just too hard to read.

Real numbers

In addition to integer constants, C also supports floating point numbers as a representation of real numbers. They add the decimal point to the list of acceptable characters and also allow you to specify a power of ten, using the same 'E' notation as a spreadsheet. Thus the following are valid real constants.

```
3.14159, 2.0E3, 1.995E-5
```

BEWARE: Just because you can type a real number, does not mean that the computer will do what you expect. Unless you specifically created a floating point variable to hold the value, C will simply turn the value into an integer, throwing away any fractional part *without telling you*!

Characters

As discussed in Chapter 1, computers use small integers to encode characters. C understand the standard coding and so supports character constants. A character constant consists of a single ASCII character surrounded by single quotes. Thus 'a', 'c', '&', and 'L' are all character constants. Internally, of course, each is represented by the appropriate bit pattern. Thus the character 'a' and the small constant 0x61 are internally exactly the same bit pattern. C provides a way to enter some of the control codes using special codes called 'escape' codes. These are two-character sequences that are internally replaced by the correct ASCII bit patterns.

Table 4.1 lists the valid escape codes. Note that it includes escape sequences for the quotation marks and the backslash character that would otherwise get interpreted specially and so not be able to appear as characters.

Table 4.1: Escape Codes

Code	Meaning
\n	linefeed
\t	horizontal tab
\r	carriage return
\\	the \ character
\'	a single quote, '
\"	a double quote, "

Strings

The last kind of constant is a string constant. A string is collection of characters, each stored as ASCII values, terminated by a NULL character (value 0x00). A C string constant consists of the list of characters enclosed in double quotes, '"'. You can put any character constant inside a string, so that the escape sequences are commonly found inside strings. For example, a string that would be used to print the text

```
That's all Folks!
```

on a line by itself would appear in C as

```
"That\'s all Folks!\r\n".
```

Note both the use of an escape sequence to include a single quote, or apostrophe, in the string and the use of the `'\r'` and `'\n'` escape sequences to end the string. These are needed to force the display down to a new line. The carriage return character, `'\r'`, moves the cursor to the start of the current line while the linefeed character, `'\n'`, moves it down to the next line. Most strings that are designed to be printed or displayed end this way.

Some C compilers automagically interpret the '\n' escape sequence, producing both a carriage return and a linefeed code in the actual string. Energia leaves this up to us and puts in only a linefeed. It gives us greater control at the expense of some convenience.

4.3.5 Assignment Statements and Expressions

'='?
From a mathematical point of view this is a somewhat perverse use of the equals sign. It does not specify a relationship between the two sides but the operation of replacing the old value of a variable with a new value. It is best to pronounce the single equals sign as **becomes** to remind ourselves that it alters the variable on its left.

In C we change the values of variables by **assignment**. An assignment statement has an '=' sign with a variable name on the left and a value on the right. The simplest form just assigns a constant to a variable.

```
numWords = 1;
```

The power of a computer really becomes apparent, however, when we note that the value on the right can be any normal mathematical expression. This may even involve that value of the variable on the left. This does not present a problem since the whole expression on the right is always evaluated *before* the assignment is made. Thus the value that is used is always the old value and there is no contradiction set up. Thus, any of the following would be legal.

We already learned that simple statements in C must end with a semi-colon so that the compiler can tell that we are done. This allows us to put multiple assignments on one line if we really want to. Personally I discourage this because I find it hard to read.

```
numWords = numWords + 1; // adds one to numWords
value = (value * 16) + newValue; // used in translating
                     // strings of hex into actual numbers.
voltage = (ADCValue * 5) / 255;+
```

An expression in C is basically the same as an expression in mathematics; a pattern of values and operators that can be evaluated to give a single result, usually a number.

C provides a fairly robust set of operations that you can use to build expressions and it organizes them by mathematical **precedence**.

The mathematical precedence controls the order in which operations are performed. For example, multiplication, *, has higher precedence that addition, +. That means that multiplications are performed before additions. Thus 7 + 3 * 4 is evaluated as 7 + (3*4) and gives the result 19 rather than being evaluated as (7+3)*4, giving 40. Note that you can always use parentheses to control the order of evaluation.

Arithmetic Operators

In later chapters we shall meet a variety of other operators, including ones to compare numbers and ones to manipulate the individual bits of the numbers. These will allow us to perform many other kinds of operation but will make things more complicated, so I have left them till later.

C supports the usual arithmetic operators of addition (+), subtraction (-), multiplication (*), and division (/). As in standard mathematical practice multiplication and division have the higher precedence and addition and subtraction a lower precedence, so that all multiplications and divisions are performed before additions and subtractions (subject always to parentheses). These operators are all **binary** operators, meaning that each takes two operands. There are also **unary** forms of + and - that take only a single argument. The unary forms have higher precedence than multiplication or division. The unary minus operator combined with a numeric constant gives the effect of a negative constant. Thus the following is a legal expression in C

```
4 + 3 * -2
```

where the + and * are binary operators and the - is unary. This expression would evaluate to the value -2.

There are also some fancier operators. In particular, you can combine any of the binary operators with an "=" sign to make an operator that modifies a value in one go. For example

```
aVal += 3;
```

is exactly equivalent to

```
aVal = aVal + 3;
```

but is shorter to type.

You can go even further with the operators "++" and "–". These allow you to modify a single value without needing an "=" sign at all. They are called **increment** and **decrement** operators. They either increase the value by one (++) or decrease it by one (–). Thus

```
aVal++:
```

has exactly the same effect as

```
aVal += 1;
```

They are very often used in the last clause of `for` loops, like this

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

```
        // do something ten times
    }
```

4.4 Compound Statements

So far we have met the two simplest kinds of C statement, the declaration and the assignment. C is a member of a large family of languages that all trace their ancestry to an obsolete language called Algol. Such languages share a number of features including the idea of a **block** of code.

Other current block structured languages include the elderly Pascal, Java, and derivatives of C such as C++, C#, GO. Hints of block structure have also been added to other languages such as BASIC and modern version of FORTRAN, though the underlying languages have very different roots.

A **block** is unit of code with a single entry point and (normally) one exit point. In C we group the lines of a block together by enclosing them in curly brackets '{'and '}'. and we can use a block pretty well anywhere that a statement would be legal. Because blocks bind several simpler statements into one more complex entity, they are also called compound statements.

One of the key ideas of a block is the idea of scope. Names that are declared inside a particular are visible only from within that block, including within any enclosed blocks. This is best understood with an example.

```
1. while (true) {      // start of outer block
2.     char A = 0;     // declare A visible from here on
3.     A = A + 1;      // A is visible here but B is NOT
4.     if (A > 4) {    // Start of inner block
5.         char B = 2; // B is visible only in inner block
6.         A = 2 * B;  // A is also visible in inner block
7.     }               // end of inner block
8.     putchar(A);     // A is still visible but B is NOT
9. } // end inner block
```

There are several important things to note about this (quite useless) example.

- The numbers at the beginnings of the lines are NOT part of the program. I added them to make it easier to refer to the individual lines. A C program does **not** use line numbers.
- You would never normally comment every line but it makes sense here as this one of the first piece of C that you seen.
- All simple statements in C must end with a semicolon, ';'. This terminates the statement and would allow us to put multiple statements on a single text line, though I strongly discourage this practice; it makes programs harder to read.

NOTE: While simple statements end with semicolons, blocks do not. It is not usually an error to follow the ending '}' with a semi-colon, but it is pointless.

- I have used indentation, white space at the beginning of lines, to make the block structure easily visible. This is an extremely common and useful practice. Computer programs are densely packed with

information and anything that we can do with the formatting to make the structure more visible is a good idea!

The popular Python programming language takes this idea one step further. It requires the indentation and dispenses with the curly brackets. There the indentation enforces the grouping of lines into blocks.

- The variable B is declared only in the inner block, lines 5 and 6, and any reference to it before or after the block would be an error. The compile would fail to compile the code and would put up a message telling us that the name B was not defined at that point.
- The variable A is declared in the outer block, at line 2. It is visible throughout the rest of the code, including the inner block. Variable A ceases to exist at line 9, when the outer block ends.
- We say that the inner block is **nested** within the outer block. Blocks in C can be nested to any depth.
- You **can** put a block anywhere a single statement is legal but blocks are normally used to enclose the bodies of conditionals, loops and procedures (see below).

Conditionals are statements that begin with the keyword “if” and they allow the code to behave differently depending the values of variables.
Loops are statements that can cause a block of code to be repeated some number of times.

4.5 Procedures

A **subroutine** or **procedure** is a self-contained piece of code that performs some useful task. The code is given a name and can be used by calling on that name. The advantage of a subroutine is that its code only appears in one place in the program but it can be used in many places. This reduces the size of the complete program and also reduces the amount of work for the programmer. In particular, many subroutines perform tasks that are common to many programs. In such a case we don’t have to rewrite the subroutine, instead we include a copy in our program and use it as needed. Even better, we don’t even have to write a subroutine if we can find one that already does the job. Indeed, one of the great advantages of programming in high-level languages such as C or Java is that they come with libraries of subroutines to perform most of the common tasks for us. For example, subroutines take care of such complex tasks as writing text to the screen, reading text from the keyboard, reading and writing disk data, and converting numbers from their internal binary form into human readable strings for printing.

If a subroutine always had to do exactly the same thing every time that it was called, for example printing the string “Hello world.”, then it would be of only limited utility. We need to be able to pass some information to the subroutine so that it can perform a slightly different task each time. For example, we might want a subroutine PutString that can print *any* string that we ask it to. We use **arguments** to pass such information.

Every subroutine has a list (possibly an empty list) of values that it gets from the program that calls it. We call that list the argument list. When we invoke (call) a subroutine, we write the name of the subroutine and write

the list of arguments in brackets after the name. Thus, a call to a PutString routine might look like this

```
PutString("A string to write.");
```

Each subroutine defines its own list of arguments and we have to look at the documentation for the subroutine to know what those arguments do and in which order they must be written.

Sometimes a subroutine needs to send information back to its caller. In the common case that we need to return only one piece of information we describe that information as the result of the subroutine and we access the return value by putting the subroutine call into an expression such as the right hand side of an assignment. Thus, a routine called StrLen that takes a string as its only argument and returns the number of characters in the string could be used like this

```
nChar = StrLen("How long is this string?");
```

After this statement was executed, nChar would have the value 24, the number of characters in the string (including the question mark).

The C standard provides a large number of subroutines to perform common tasks but we can also write our own subroutines to perform tasks that we find useful. We need to identify the arguments that our subroutine will need and we need to name the subroutine and the arguments. As usual, we should give them names that suggest their uses.

4.5.1 Declaration and Definition

When we make up our own subroutines we need to provide two kinds of information. First we need to tell any code that uses the subroutine what types of arguments it takes and what result it returns, if any. Second, we need to tell the compiler what it is that we want the subroutine to do. We call the first of these operations **declaration** and the second **definition**. A subroutine must be declared before it can be used anywhere and that declaration may be given several times if the program is split across several files. The subroutine must also be defined exactly once, in a file with other code or in a file all by itself.

Declaration

In C a subroutine declaration statement looks like this

```
<type> <name>(<type> <arg1>, <type> <arg2>...);
```

The first <type> specifies the type of the subroutine result. This will either be one of the arithmetic types that we met earlier or special type void if the subroutine does not need to return a value.

< and >
Remember that the angle brackets mark placeholder items. In a real declaration <type> will be replaced by a real type name, such as int., <name> will be replaced by the actual name of the subroutine, and so on.

The <name> is the name by which the subroutine will be known throughout the program. This should be a sensible name that tells you about what the subroutine does. I normally like to begin subroutine names with a capital letter, but that is a matter of style. The compiler knows that we are declaring a subroutine and not a simple variable because of the parentheses that follow the name.

The parentheses hold a comma separated list of **arguments**. Each argument is given with its name and its type. The name is the name by which the argument will be known inside the subroutine. It has nothing to do with the actual value that we put in when we call the subroutine and the name is invisible from outside the subroutine. If the subroutine does not need any arguments then you should use the `void` keyword to insist that you didn't forget anything, thus,

Strictly, you can declare a subroutine with no arguments simply by omitting the arguments. Some compilers, however, are quite picky and will issue a warning if you don't use the void keyword

```
int GetKey(void);
```

Because all names have to be declared before they are used, we have to put a subroutine declaration somewhere before the first use of the subroutine. This means that we tend to put declarations at the start of each file of C so that they are visible to all the code below. If a subroutine is likely to be used in more than one program then we often puts its declaration into a separate file, called a header file, with a name of the form <name>.h, and then use a special command

The hash sign, '#' at the beginning of the line marks this is a rather special sort of C command. The compiler does not do all its work in one go, instead it breaks the job up into several phases called 'passes'. The first pass through the file is called preprocessing. During this phase lines that begin with a '#' are examined and acted upon. In this case the preprocessor replaces the line with the entire contents of the file. So when the next phase of the compiler runs it sees a much larger file with all the header information included.

```
#include ``name.h''
```

at the start of every file that uses the subroutine. This include command tells the C compiler to read the contents of the file name.h before continuing to process the main file. Thus we can put common declarations in a header file and then use that header file in lots of different programs.

Definition

We define a function in a slightly redundant way. A function definition starts off with a repeat of the declaration and then follows that with the body, a compound statement wrapped in curly brackets, thus

```
<type> <name>(<type> <arg1>, <type> <arg2>...)  {
<body, declarations then actions>;
}
```

If the subroutine has a result, i.e. if it is a function, then we need to include a return statement in the body. This is a line with the keyword return followed by an expression that evaluates to the result. For example,

```
return sum/count;{
```

Example 4.5.1

Here is a routine to blink the LED connected to bit 1 of Port F, turning it on briefly then turning it off. We will make the length of time into an argument to the routine.. This subroutine will not need to return a value.

```
/*
* BlinkF1 will turn on bit 1 of port F for onTime milliseconds.
*/
void BlinkF1(int onTime) {
    digitalWrite(PF_1, HIGH);      // Turn LED on
    delay(onTime);                 // Wait onTime mS
    digitalWrite(PF_1, LOW){;      // Turn LED back off
}  // End of subroutine
```

Note that I rather overdid the commenting again since this is the first complete procedure we have seen.

There must be at least one return statement for every subroutine that has a result and there may be more than one if there are different paths through the routine. Thus we might have a function that finds the maximum of two numbers

Yes, I know we haven't formally met the `if` statement yet, but it should be pretty obvious what this does!

```
int MaxVal(int val1, int val2) {
    if (val1 > val2) {
        return val1;
    } else {
        return val2;
    }
}
```

Calling a Subroutine or Function Procedure

Once a procedure has been declared we can use it simply by using its name. If we have a subroutine or are not interested in the value of a function then we we can use a **subroutine call** statement. This is the simplest kind of statement, consisting only of the name of the subroutine, the parentheses, and any arguments. For example, I can call our BlinkF1 subroutine with some different times to generate a pattern of blinks with a series of statements like this.

We only need to have declared the procedure before using it. The actual definition could be anywhere, later in the same file or even in a totally separate file. We do not even need to have access to the definition, merely to the code the does the work. It possible for someone else to define and compile the code and give it to us in object form. So long as they tell us how the subroutine is declared then we can use it without knowing how it works.

```
void delay(int time);  // Declare Wait before use
pinMode(PF_1, OUTPUT); // Make Port A bit 0 an output
BlinkF1(100);          // Short blink
delay(100);            // Wait 0.1S of dark
BlinkF1(200);          // Long blink
delay(100);            // Another 0.1S of dark
BlinkF1(100);          // and another short blink
```

This example also illustrates the idea that we do not need to have the definition of a function in order to use it. Someone must have created `delay` and have provided the code in some form. We may only have the compiled

code, probably in a library, and need have no idea how it works so long as we know how to call it and what it does.

If we want to use the value of a function then we simply put the name of the function, with parentheses and any arguments, wherever we want the value of the function. Each time the computer sees the name it will go off and perform the calculations in the function then come back with the value. ready for further calculation. For example, if we have a function called GetNumKey() that returns numbers typed on a numeric keypad then we can write a terribly primitive calculator like this.

```
sum = GetnumKey() + GetNumKey();
```

This will read the value of a key, waiting for us to press it as needed, then do it again for a second key before adding the two values together and storing them in sum.

This is **not** the best way to perform this function. There is some overhead associated with a function call and this needs n function calls to calculate n! A simple loop can calculate the value more efficiently, but the process is less mathematically elegant.

Example 4.5.2

A classic example calculates the factorial of a number using a function that calls itself (a **recursive** function). Because the argument of the internal call to factorial is not the same as the original argument this does not produce an infinite loop.

```
int factorial(int n) {
    if (n == 0) return 1;
    else return (n * factorial(n-1));
}
```

4.6 The Structure of a Complete Program

We now have all the pieces we need to write complete small programs. Let me remind you of a few key ideas, pretty much in the order in which they go in a file.

Comments, especially a comment at the start of each program or procedure that fully describes what the code does, are a vital part of writing readable programs.

Header files allow us to bring in information about the chip we are using and any pre-written code that we use. They are made part of the program using the `#include` directive.

Every procedure that we use in the program must be declared before its first use, either in a header file or in the space before the main program. You may also choose to define the procedures there but you don't have to.

Programs for Arduino or Energia, technically known as **sketches**, *must* consist of at least two procedures, which you do *not* need to declare, `setup()`, which will be executed exactly once before the program goes on to call the other procedure, `loop()`, infinitely.

Pure C and C++ programs must instead contain a procedure called `main()`. This procedure marks the start of the program but may call all sorts of other

procedures to do its work. Behind our backs, Arduino/Energia provides a main program that basically looks like this

```
int main() {
    setup();
    while (true) {    \\ A loop that will never end
        loop();
    }
}
```

With these rules in mind, Figure 4.2 on the next page is is a complete Energia program to blink an LED attached to Port A pin 0 in a simple SOS pattern. I have used the `BlinkF1` and `delay` routines tand have created another helper that performs three blinks of the same length.

4.7 Designing a Program

It is very difficult to set down rules and methods for designing programs. Programming is an art that one learns by experience. So, if the only way to learn to program is by experience, and the only way to get experience is to program, it seems as though we have a serious chicken-and-egg problem. Fortunately, we can break this vicious circle by learning from other people's experience. I have found that studying the programs of others is the most effective way to get started programming.

There are a number of ways to gain experience. One obvious one is to read books that describe how to solve particular programming problems. Fortunately, the world is full of these. A good place to start is books with names containing words like Data Structures, Algorithms, and Programming, especially ones that call themselves an introduction or a first course or something similar. If you can handle the rather abstract approaches there are few rivals to Donald Knuth's series "The Art of Programming" and Edsger Dijkstra's "A Discipline of Programming".

The approach I shall take in this book is to try to provide lots of examples of programs to solve common tasks. In particular, chapters 9 and 10 provide a large number of program fragments and subroutines that I have found useful in programming embedded systems. The exercises then provide suggestions for modifying these examples in various ways because I find that modifying existing code is an excellent way to get started producing your own code. Once you have a little practice with this you can move on to writing your own small programs that use the subroutines I have provided. Finally, you will find that you are able to write your own subroutines to solve new problems.

```
/*
* BlinkSOS.c
* A program to blink an LED attached to pin 1 of Port F on
* a TM4C123G LaunchPad in an SOS pattern.
* BCollett 1/9/17
*/
//
// Global variable control the duration of dot and dash.
// Values are times in mS.
int gDotTime = 100;         // duration of a single dot
int gDashTime = 3*gDotTime; // dash is required to be three times dot
//
// BlinkF1(int onTime) turns the LED on for onTime milli-
// seconds and then turns it off again,
//
void BlinkA0(int onTime) {
    digitalWrite(PF_1, HIGH);    // Turn LED on
    delay((onTime);
    digitalWrite(PF_1, LOW);    // Turn LED off
}
//
// Blink3F1(int time) uses BlinkF1 to perform 3 flashes,
// each of duration time, separated by 0.1 s gaps.
//
void Blink3F1(int time) {
    int count;
    for {count = 0; count < 3; count++} {
        BlinkF1(time);
        delay(100);
    }
}
//
// Main program has to set port F pin 1 to be an output and
// then do three short, three long, and three short flashes
// forever. It delegates much of the work to Blink3F1.
//
void setup(void) {
    pinMode{PF_1, OUTPUT);
}
void loop(void) {
    Blink3A0(dotTime);   // dot dot dot
    delay(100);          // extra time between characters
    Blink3A0(dashTime);  // dash dash dash
    delay(100);
    Blink3A0(dotTime);   // dot dot dot
    delay(400);          // extra time between words
}
```

Figure 4.2: The Complete SOS Program

4.8 Summary

A C/C++ program consists of **statements** grouped into blocks by curly brackets.

Data items are represented in C by **variables**, each with a type, a value, and a unique name. Procedures are also represented by names and types and may optionally have a list of arguments that will be passed to the procedure.

Statements include variable declarations,

```
<type> <name> [= <expression>];
```

procedure declarations,

```
<type> <subroutine name>([<argument>[,<argument>]...]);
```

assignment statements,

```
<variable name> = <expression>;
```

and subroutine calls.

```
<subroutine name>([<argument>[,<argument>]...]);
```

Types describe the storage requirements and the allowed values of variable. Allowed types include

```
char         unsigned char
short        unsigned short
int          unsigned int
void (only for subroutines and argument lists)
float (not well supported by Arduino/Energia)
```

Expressions are normal arithmetic expressions using addition, +, subtraction, -, multiplication, *, division, /, and remainder, %, as well as parentheses, (and), for grouping. Valid elements in an expression include variables, constant, and function calls.

Procedures are self-contained pieces of code that can be executed as independent units called from another procedure. A **function** is a procedure that returns a value. A **subroutine** is a procedure that returns no value. It has type **void**. Both kinds of procedure have a **body** consisting of a block. Functions must include at least one **return** statement within their body.

Every variable must be declared before it is used. Similarly, all procedures must be declared before they are used. The **declaration** and **definition** may either be separate or combined into a single entity.

Exercises

1. Do a little research on the Web to learn something about the Morse code. Adapt the BlinkSOS program to blink your name or your initials in Morse code.
2. Write a C function that returns the sum of the first n integers. You may assume that the answer will fit in an int. Your function should have the definition

   ```
   int SumFirstN(int n);
   ```

Chapter 5

The TM4C123G Microcomputer

5.1 Introduction

The TM4C123G is a member of the **Tiva** family of microcontrollers made by Texas Instruments (TI). They are considered "performance" microcontrollers, capable of handling tasks of some complexity. They wed an ARM Cortex-M4F core to an extensive set of peripherals, a moderate amount of memory, and a power control system that supports a variety of low power modes to enable battery-powered systems with long run times. TI suggests that they are good for tasks that include

- Low power, hand-held smart devices
- Gaming equipment
- Home and commercial site monitoring and control
- Motion control
- Medical instrumentation
- Test and measurement equipment
- Factory automation
- Fire and security
- Transportation

Family
A family of processors is a set of devices that all share common architecture but that differ in various details. Different members of the family come in different packages, have different numbers of I/O ports, and may have different amounts of memory. Because they all share the same CPU, any program that runs on the smallest member will also run on the larger members.
Since these are intended for quite complex tasks they come in quite large packages, from the 64-pin package of our chip up to ones with twice that many pins.

We will focus on the TM4C123G model which comes in a 64-pin package where 43 pins can be used as inputs or outputs, of which 37 are actually available for us to use on the LaunchPad board.

This chapter aims to provide a bird's eye overview of the chip and the LaunchPad development board. This is a chapter to skim through and maybe look back at from time to time as you encounter new sections of the chip.

5.2 The Heart of the TM4C123G

The CPU core of the TM4C123G is based on ARM's Cortex-M4F cpu core. This is the cadillac of ARM's microcontroller lineup, providing support for memory protection and for working with floating point numbers as well as a flexible interrupt controller and a low-level timer. Texas Instruments has added their own clock generator module so that the chip can run with a minimum of external components. The TM4C123G can run with clock speeds up to 80 MHz, so that the fastest individual instructions take only 12.5 nS to execute!

The Cortex-M4F core is a complete 32-bit computer core, with hardware to add, subtract, and multiply two 32-bit numbers in a single tick of the clock. The CPU is an example of the RISC philosophy of CPU design. This stands for **R**educed **I**nstruction **S**et **C**omputer. Basically, this means a machine that has a certain amount of ultra-fast memory built into the CPU, called **registers**, and all its instructions either move information between the registers and the main memory or do arithmetic on values in the registers. This contrasts with the Intel CPUs that espouse the **C**omplex **I**nstruction **S**et **C**omputer (CISC) philosophy that allows arithmetic on values that live in main memory and usually have many fewer registers. The ARM design provides thirteen 32-bit registers for use in computation plus three that store the addresses of important locations in memory (such as the location of the next instruction to execute) and some with information about the current state of the CPU and the current computation.

Abbreviations from Chapter 1
lsb—Least Significant Bit: the rightmost bit of a binary number.
msb—Most Significant Bit: the leftmost bit of a binary number.

To make it easy to talk about individual bits in a register or memory location, we number the bits from the right to left. The rightmost, least significant, bit is bit 0 and the leftmost, most significant, is bit 31. We refer to a single bit in a register by giving the register name, a colon, and the bit number. So the most significant bit of the stack pointer is SP:31 and the least significant bit of the data register 0 is R0:0.

The CPU core is the invariable centre of the Tiva family. The different chips in the family have different amounts of ROM and RAM and have different numbers and kinds of interfaces but they all have the same CPU. This means that once we have learned to program one chip we can transfer easily to any other. All we have to learn is names and uses of the new peripherals.

5.3 On-chip memory

Since it is a microcomputer and not just a microprocessor, each Tiva device has some memory built into the chip. This means that it can operate as a complete computer-on-a-chip without any other support circuits. Since the Cortex-M4F is a 32-bit chip each individual address is 32-bits long so that we can address about 4 billion memory addresses. That is enormously more

than are actually implemented so there are whole ranges of address that are invalid. If you try to access one of these the CPU will notice and will throw up its hands and declare an error, ending your program instantly.

The TM4C123G contains five different kinds of memory in addition to the registers built into the CPU.

First there are the obvious kinds. There are 256kB of non-volatile FLASH memory, that is, memory that retains its contents when the power is removed. FLASH is a special case of ROM memory that can be erased and programmed in chunks through either special external hardware or by using extra programming hardware built into the chip. We normally store our programs in the FLASH so that the computer can start working as soon as the power is applied. The FLASH in the TM4C123G is special high-speed FLASH that can run at the full 80MHz processor speed.

Second there are 32kB of RAM memory. Like the FLASH, this runs at full speed and we usually store all our changeable data here. When you declare a variable in C++ you are giving a name to a location in this RAM. Unlike the FLASH and the ROM and EPROM that I will describe next, the RAM loses its memory when the power is removed. This can happen because the whole chip has been turned off but it can also happen when the CPU shuts down bits of the chip to lower power consumption. Obviously, when the RAM is powered down you can't actually do anything!

Third, there is a rather vague amount of ROM memory that TI has preprogrammed with some useful subroutines. The Energia library that we use often does its work by calling some of these built-in ROM routines. The idea here is that TI can tweak the code on ROM for each slightly new chip and the user will not have to alter their program so long as it relies on the ROM to do the job.

Fourth, there are 2kB of EPROM. This is another kind of non-volatile memory that is intended for storing small bits of data that will survive a power down. For example, if you used the chip as the heart of a fancy programmable thermostat then you would store the time and temperature settings in the EPROM. You can erase and re-program individual memory locations in the EPROM but it takes special code and a few mS to do it.

Fifth, there are several hundred things called Special Registers. These are pieces of hardware that appear to the computer as ordinary memory locations (RAM) but they are also connected to the peripherals. Bit patterns written into these special registers control the behavior of the peripheral hardware. For example, there is one such special location that is connected to the pins of Port B so that when you write a value to that location it sets the voltages on any pins of port B that are configured as OUTPUTs.

5.4 TM4C123G On-Chip Peripherals

Around the CPU core there are a large number of support circuits.

Figure 5-3 on the facing page shows the block diagram for the TM4C123GH6PZ chip that is the focus of this chapter. At the top are the CPU, the with memory, and some system control circuits. The CPU talks over the system bus to the blck labelled Bus Matrix. It funnels the information over two more buses, the APB and AHB, to the various peripherals that are shown in the grey blocks. They, in turn, talk to the outside world over the pins of the device, all of which are shared by several different functions, though only one function can be active at any time.

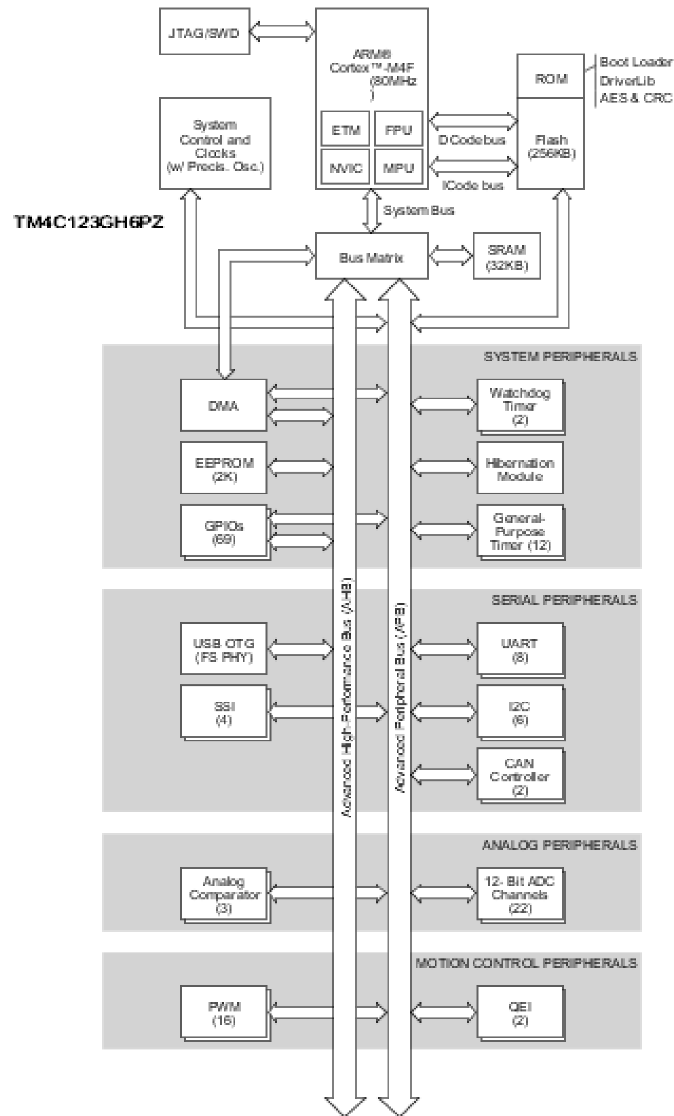

Figure 5.1: Internal Block Diagram of the TM4C123GH6P

5.4.1 CPU Support Subsytems.

Intimately integrated with the CPU is the debug module (JTAG/SWD). It uses a single wire to communicate to a desktop PC running software that allows the host to send programs to the ARM through an interface module, in this case a second TM4C123G built into the LaunchPad. The PC software can then cause the ARM, called the target computer, to run the program. The Energia package uses this interface to transfer the program to the target and run it. The more powerful CodeComposer that we shall meet later in the semester can do all that and then allow us to control and watch the program as it runs. Thus we can for example step through the program one line at a time watching the values of the registers and memory of the target change as the program executes.

The next block, the System Control & Clocks module, both provides the clock signal for the CPU and handles unexpected or rare events. System Control knows how to start the CPU up when the power is first applied and how to stop the CPU if the supply voltage should get dangerously low.

The Clock Generator provides a stable clock to control the timing of all the chip's internal operations. In an effort to make a self-contained computer with the minimum number of components, TI have built a sophisticated clock generator into the chip. There is a completely internal 16MHz clock that is used when power is first applied. For low power, lower accuracy, tasks this clock can be left in place all the time. Many systems need a more accurate and stable clock such as that provided by a specially constructed quartz crystal. TI allows you to use either an external clock signal or a an external quartz crystal to generate a high accuracy clock. Energia switches from the internal clock to the crystal clock before calling our `setup()` routine so that we normally run from the crystal built into the LaunchPad.

LaunchPad Clocks
The LaunchPad board provides two crystals, the flat, square black one labelled Y2 and the little shiny cylinder labelled Y1. Y2 is the 16MHz crystal for the clock that we use. Y1 is a much slower 32kHz crystal for use when the chip is running in very low-power mode. The Y1 signal can also be used by the real-time clock that works rather like wristwatch for the computer.

The remaining blocks in the upper part of the figure are the FLASH, ROM, and RAM to store programs and data that were discussed in section 5.3 above. These complete the basic CPU component. Various peripherals are connected to the CPU over the two peripheral busses, the **A**dvanced **P**eripheral **B**us (APB) that conveys control signals to and from the special registers built into the various peripheral I/O substems and the **A**dvanced **H**igh-Performance **B**us (AHB) that moves high speed data between them.

Each of these subsystems appears to the CPU as a block of memory locations that can be read an written. Each register has different rules for what the bits mean and what will be the effect of reading or writing them. These rules are detailed in the Reference Manual for the TM4C123G. Some idea of the complexity of working directly with the special registers is gained by noting that the manual is over 1400 pages long! We will normally let Energia handle the details for us. The manual for that is only a few dozen pages long and a lot easier to read.

The grouping of the peripherals into four different grey blocks is more of a

way for us to organize our view of the different kinds of I/O than a reflection of any real difference inside the chip. We shall explore most of the subsystems in more detail in later chapters, but here is a quick look at the functions that they provide. The ones with asterisks after them can be skipped until you have a reason to read about them.

5.4.2 System Peripherals

The first four of these blocks provide advanced functions that are well beyond anything we can explore in this introduction. However, the remaining two, the GPIOs and timers, are the basis of most our work.

DMA* The acronym stands for **D**irect **M**emory **A**ccess. This block can be programmed to move data between some of the peripherals and the memory without using the CPU. It is quite tricky to use and I will not discuss it further.

Watchdog Timers* The two watchdog timers can be used to catch a run-away program and bring it back to normal operation. A program can set one of these running for some length of time and go off and perform its normal operations. If the timer ever runs down then the chip will be reset. The idea is that so long as the program is operating normally then it keeps restarting the watchdog so that it never runs down. If a flaw stops the main program from running normally then the watchdog will intervene and restart the whole system. We will never use them.

EEPROM* As I explained above, this is a place to store information that you want to survive across a power down. I have never used this function.

Hibernation Module* This one is for really low-power battery operated systems. It allows you to turn off the clocks to almost all of the chip, reducing its power consumption almost to nothing. The hibernation module remains awake and looking for events that would re-wake the CPU. These might include a timer running out or a button being pressed on a digital pin. Again, I have never needed this.

GPIOs This acronym stands for **G**eneral **P**urpose **I**nput **O**utput. These are the digital input/output signals of the chips. They are organized into six ports, Port A - F. Each is a set of up to 8 lines, each of which can serve either as a 1-bit digital input or output. These are the signals that we control with the `pinMode`, `digitalWrite`, and `digitalRead` routines.

Most pins are shared between the GPIO ports and various special interface functions. When the computer first starts up most of the interfaces are disabled so the pins act as GPIOs and are all configured as inputs. We shall return to them in Chapter ??.

General Purpose Timers These incredibly useful peripherals can perform all sorts of time-related functions. We can use them to generate streams

of pulses and to measure streams of pulses as well as simply to time the operation of a piece of code. Energia uses these to support the `analogWrite` command. We will learn more about them in Chapter ??.

5.4.3 Serial Peripherals

These provide various ways for our chip to talk to other chips. A serial interface is one where data are sent one bit at a time over a small number of wires (2-5) rather being sent over as many wires as there are bits. They come in several forms to handle different sorts of communication.

USB* This is the omnipresent **U**niversal **S**erial **B**us. This is one of the more powerful kinds, known as a USB **O**n **T**he **G**o (OTG) device. It can operate as either a controller device (like a computer) or as a peripheral device (like a mouse or keyboard). The LaunchPad board uses the USB device on the debugging CPU to transfer information quickly between the main computer and our CPU. You could use the main CPUs USB port to build your own pointing device controller, like a mouse, or maybe to create your own MIDI device. This is both very powerful and very complicated to use; well beyond the scope of this course.

UART The **U**niversal **A**synchronous **R**eceiver and **T**ransmitter interfaces are an older kind of serial interface. They were very common in the days before USB became popular. They were designed to allow two computers to talk to each other over a small number of wires (3-6) at moderate speeds and long distances. They used to be the most popular way to connect devices to computers. For example, the mouse and keyboard used to connect over UARTs. They are still a good way for humans to use the keyboard and screen of a PC to talk to a little embedded system such the LaunchPad that has no keyboard or display of its own. The TM4C123G has 8 of these UARTs and Energia provides fairly sophisticated support for them. We shall make extensive use of this interface to communicate with our programs and so it is described in Chapter 10.

SSI The **S**ynchronous **S**erial **I**nterface is a very simple 3-4 wire interface for an embedded CPU to talk to simple, nearby, chips. It requires very little hardware on the other end and so is ideal for really cheap external chips such as digital-to-analog converters. Energia provides support for some of the functions of this interface through the routines of the **SPI** library. This interface is intended to talk to chips no more than a few inches from the CPU.

SPI
TIs SSi interface is an expanded version of the more standard **S**erial **P**eripheral **I**nterface (SPI). Energia only supports the SPI version of the interface, hence the name of the Energia routines.

IIC The **I**nter-**I**ntegrated **C**ircuit Interface is a souped up version of the SSI designed to serve pretty much the same functions. Where the SSI is fine for one CPU to talk one external chip, the IIC makes it easy for a single CPU to talk to several external chips without anyone getting confused. It is somewhat more expensive to implement so you are more likely to find it in

more complicated external chips such as LCD displays and 3-axis compass chips. It is supported by Energia through the routines of the **Wire** library.

CAN* The **C**ontrol **A**rea **N**etwork is the most sophisticated of these interfaces. It is mostly used in complicated systems such as automobiles where it allows the many computers that control all the fancy gadgets in a modern car to talk to each other. It is well beyond the scope of this work.

5.4.4 Analog Peripherals

These allow the CPU to interact with continuously varying voltages instead of the strict 0V or 3.3V of the binary signals that are natural to the CPU.

Analog Comparator An analog comparator allows the system to notice when the voltage on a pin crosses some preset value, either going from less to more or vice versa. You might use an analog comparator to monitor the battery voltage in a battery operated system so that the system knows to shut itself down if the voltage gets too small.

Analog-to-Digital Converter The Analog to Digital Converters provide up to 22 separate inputs that can measure the value of the input voltage. It uses a analog multiplexers to share two 12-bit successive approximation ADCs between various external and internal signals. It is a very sophisticated unit and we shall explore some of its functions in Chapter 11 but you will need to read the TI documentation for full details on its capabilities. Energia makes it easy for us to make simple measurements with this using the `analogRead()` command.

5.4.5 Motion Control Peripherals

These are two sets of peripherals that are intended to make it easy to control motors of various kinds.

PWM* This means **P**ulse **W**idth **M**odulation, and we shall explore it in some detail in Chapter ?? The general purpose timers can generate simple pulse-width modulated signals but some kinds of motors need to have several PWM signals that are correlated in fancy ways. There are two separate systems each of which can generate four pairs of correlated signals. I don't think that we will need to use these.

QEI* A **Q**uadrature **E**ncoder **I**nterface is a piece of hardware that can be connected to the shaft of a motor and provide information about the rotation of the motor. Quadrature encoders are quite expensive pieces of hardware and usually found only an large, expensive motors. It might be possible to use cheaper versions of these to improve our little robots but it is beyond the current scope of the course.

5.5 The LaunchPad board

The TM4C123GXL LaunchPad board is an inexpensive introduction to the TM4C12 family of computers. The price is kept low ($13 per board in 2016) to make the platform attractive in a competitive market. TI wants to encourage designers to experiment with these devices so that they will adopt them for their money-making products.

The LaunchPad combines a TM4C123GH6 chip that has most of its I/O pins brought out to convenient connectors with a few simple inputs and outputs and an on-board debugging interface. Figure 5.2 shows a top view of the LaunchPad board.

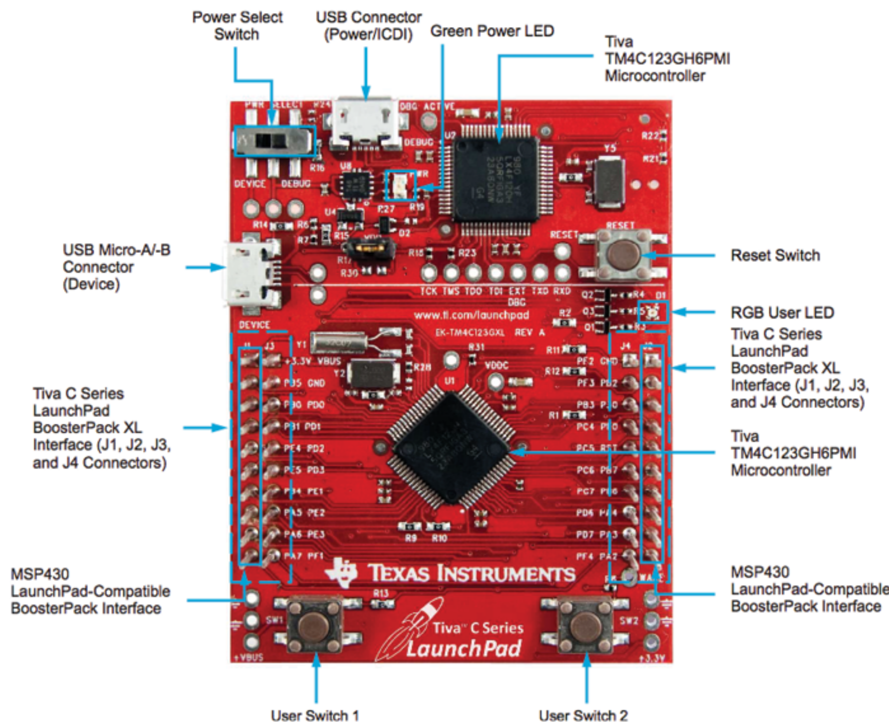

Figure 5.2: The TM4C123GXL LaunchPad

The most prominent components are the two large square black packages. These are *both* TM4C123G chips. (This is one reason why we want to buy complete boards.) Each of the TM4C123G chips has 64 very tiny pins, each of which has to be soldered down to the PC board. While this is possible with small tools, a steady hand, and a good magnifying glass, it is quite challenging; another reason to use a commercial system.

If you look carefully then you will see that the board is pretty much split into two sections by a white line running across the board at the level of the left-hand USB plug. The upper third of the board, above the white line, is home to the upper, square-on, TM4C123G chip, a green power

LED, a switch, and the top micro-USB connector. This section is called an In-Circuit Debugger Interface (ICDI). It provides a way for Energia to put our programs into the lower computer and, later in the semester, to allow us to debug the programs as they run.

The ICDI
TI refers to this incarnation of a debugging interface as a Stellaris ICDI. The name Stellaris is a left-over from a previous generation of TI ARM-based microcontrollers. The Tiva family has replaced the Stellaris family but the name lingers since the two families share the same debugging interface.

The larger, lower, section contains the diagonal chip and a whole bunch of connectors on it. This is the chip that you will actually be programming. The connectors are where you will attach wires to the pins of the computer. There are 40 pins in the connectors and they all have little names printed next to them so you know which pin does what. Alas, they are pretty well randomly organized. It would be nice if all the pins of one port were next to each other but this is *not* the case. Instead they are wired up according to some plan that escapes me. Even though it spans the white line, the left-hand micro-USB connector logically belongs to this part of the board.

In addition to the computer and the connectors, the lower part of the board hosts a pair of switches (down at the bottom left and right) and a rather snazzy three-colour LED (just under the reset switch on the middle left). These are connected to the pins of Port F, but do not usually interfere with other uses of those pins. They do mean that we can write some simple programs without having to wire anything up to the LaunchPad.

5.6 Summary

All TM4C12x microcomputers have a 32-bit ARM Cortex-M4F CPU running at up to 80-120MHz. They provide moderate amounts of memory and large sets of peripherals.

In addition to the special registers used to control the extensive on-chip interface hardware, the TM4C123GH6 has 256kB of FLASH for storing programs and 32kB of RAM for storing data. These memories can be programmed by an external host computer using the built-in debug interface and a special debugging chip connected to the host over USB.

Other than the power pins and a few CPU control pins, every pin on the chip is available as an I/O pin. In addition, most I/O pins have one or more alternate special functions that are used if the appropriate on-chip interface is enabled and suitably programmed. Each pin can serve only one function at a time.

Later chapters of the book will explore the GPIOs, the timers, the analog-to-digital converters, and the serial ports in more detail.

Chapter 6

Logical Operations, Conditionals and Loops in C

So far we have some knowledge of variables and expressions, we have assignment statements and really simple while loops in which to use them, and we know how to package them in procedures. It is time to build some more control into our programs.

There is really no repetitive task that we can't solve with only a while loop but sometimes another form can make the program shorter and easier to read.

At the heart of almost all control is the idea of a **conditional**, an expression that has one value or does one thing if some condition is met but has a different value or effect if the condition is not met. We will start with a look at some ways to express conditions and then explore C's conditional statements. Finally, we will look at some other kinds of loops that will allow us to express some ideas a little more neatly than our simple while loop.

6.1 Truth Values

We have met the two standard Energia/Arduino truth values, the constants `true` and `false`.Under most circumstances these are the best way to deal with logic values because they are the most readable. However, there are times when it is useful to understand what is going on underneath.

A value of 0 means false.
A non-zero value means true.

C represents the two possible truth values, false and true, by numbers. It regards zero as false and anything else as true. This means that the Energia constant `false` is really just a name for the number 0. Although anything else is treated as true, Energia uses 1 as the standard value for the `true` constant.

Later we will see a way to decrease the value of count in the same expression as we test its value. This method is a little more obvious and readable but the later method is probably more popular.

Occasionally we make explicit use of the fact that any non-zero value is treated as true. For example, here is a way to make a loop that executes exactly 10 times.

```
int count = 10;
```

```
while (count) {
    count = count -1;
}
```

The first time round the loop count has the value 10 and so evaluates to true. The loop runs and count gets set to 9. 9 is also not zero and so the process continues until, on the tenth trip round, the loop count becomes 0. At that point the while loop sees the value as false and so the loop comes to an end.

That form of the expression relies on us knowing that C treats 0 as false and everything else as true. The underlying logic is somewhat hidden by the brevity of the form in which it is expressed. We can make the logic more apparent by using a comparison expression.

6.1.1 Comparison and Logical Operators

C provides comparison operators that operate on numeric values and yield logic valued results. These operators are

`>` greater than
`>=` greater than or equal to
`==` equal to
`<=` less than or equal to
`<` less than
`!=` not equal to

Because any non-zero value is treated as true C compilers can choose what value they use to represent true. 1 is probably the most common but you can't count on that.

These can be used to compare any two expressions of the same type. The most usual use is to compare two numbers. Thus $5 > 4$ yields the value true while $5 >= 7$ yields the value false (0). Similarly, since characters are represented by their ASCII values we can compare characters and ‘a’ $<$ ‘z’ will be true.

Logical values can be combined into logical expressions with a second set of logical operators. There are three of these operators

`&&` logical AND
`||` logical OR
`!` logical NOT

which act on logic values. They combine individual comparisons to make an extended comparison.

Logic Operator Precedence
It would be really nice if we could rely on this. However, my experience suggests that including the brackets is *always* a good idea.

For example, we can test whether a character is a valid decimal digit with the expression

```
char >= '0' && char <= '9'.
```

For this to work correctly, the logic operations have to have lower precedence than the comparison operators so that the above expression is equivalent to

```
(char \textgreater{}= `0') \&\& (char \textless{}= `9').
```

We call such a mixture of comparison and logical operators a **conditional expression**. Thus the full syntax for our while loop is

```
while (<conditional expression>) {
    <body>
}
```

We can then put in any conditional expression to make controls for the loop. Thus a more readable way to express our earlier example is

```
int count = 10;
while (count > 0) {
    count = count-1;
}
```

This does exactly the same job as the earlier example but it is significantly more readable. The conditional expression evaluates to true so long as count is greater than zero, exactly what the original expression did.

Let's look at a more sophisticated example. Consider the task of recognizing numbers in a string of characters. A number is any consecutive string of digit characters. This little loop fragment will read in keys from the user so long as the user types only digits and will add their values up into a number.

```
char ch = GetChar();
int number = 0;
while ((ch >= '0') && (ch <= '9')) {
    number = number * 10 + ch - '0';
}
```

We shall see more of this trick later.

The comparison and logical combination operators are well suited to operate on numbers but they don't give us any easy way to investigate the state of individual bits in a number. Since we are sometimes very interested in looking at individual bits, we need a second set of operators.

6.1.2 Bitwise Logical Operators

These operators act on their arguments one bit at a time so these are called bitwise operators. C provides a full set of bitwise operators:-

`<<` left shift
`>>` right shift
`&` logical AND
`|` inclusive OR
`^` exclusive XOR
`~` NOT.

The first five are binary operators, combining the bits in two separate numbers into a third number. The last is a unary operator, inverting all the bits of a single number to give a single result. Neither kind of operator alters its arguments. Each makes a copy of the arguments, operates on the copy, and then returns a result that can either be tested or stored in a variable.

Let's look at the operators one at a time.

Left Shift

A left shift operator moves the bits of its left argument some number of bits to the left, filling in the spaces that are created with zero. Arithmetically, it has exactly the same effect as multiplying by some power of two. Thus a one bit shift multiplies by 2, a two bit shift by 4, a three bit shift by 8, and so on. Since the total number of bits is fixed by the type this must throw away the bits that are shifted off the lefthand end. Let's look at an example.

```
short val = 136;          // val holds 0b0000000010001000
short newVal = val << 2; // newVal is 0b0000001000100000
```

Since these are short values, each one has 16 bits. We started with 136 = 128 + 8. The shifting moves the upper 1 bit from bit 7 (128) to bit 9 (512) and the lower 1 bit from bit 3 (8) to bit 5 (32) so that the result is 512+32 = 544. This demonstrates the equivalence of the 2-bit left shift and multiplication by 4 since 4 * 136 = 544.

One of the most common uses of a left shift operation is to turn a bit number into a value. For example, if we want to make the 8-bit value that has only bit 6 set to 1 and all other bits left as zero then we can make that number like this

```
char bit6 = 1 << 6; // 0b00000001<< 6 = 0b01000000
```

Right Shift

The right shift operator is a little more complicated. It behaves slightly differently with signed and unsigned numbers.

Unsigned numbers are easy. The newly created high-order bits at the left hand end of the number are just filled with zeros, exactly as you would expect from the behavior of the left shift.

That would cause problems with signed numbers. Remember that the sign of signed number depends on its top bit. If the top is 0 then the number is positive. If the top bit is a 1 then the number is negative. The right shift operator has been designed so that it does not alter the sign of the number on which it operates. This means that instead of always filling in the high bits with zeros, the signed-number right shift fills them in with copies of the high bit.

The end result of the signed/unsigned rules is that the right shift has exactly the same effect as dividing by some power of two. A one bit right shift divides by 2, a two bit shift by 4, a three bit shift by 8, and so on. This time is is the old low-order bits that are thrown away. Let's look at an example.

```
short val = 136;          // val holds 0b0000000010001000
short newVal = val >> 5; // newVal is 0b0000000000000100
```

Again, we started with $136 = 128 + 8$. This time, the shift moved the upper 1 bit from bit 7 (128) to bit 2 (4) and the lower 1 bit has been shifted off into oblivion. We see that $136 >> 5$ gives the same result as $136 / 32 = 4$.

Logical NOT

I know that it is out of order, but NOT is the simplest of the logic operations so maybe it is sense to look at it soon. This simply reverses the state of every bit in the value. It is important to distinguish between negating a value and complementing it. The unary minus operation changes the sign of a number, which changes most of the bits of a number, but preserves the magnitude in the 2s complement notation. The NOT operation just flips every bit. It does not make sense for a value that is thought of as a number but it is very useful for finding the complement of a bit pattern.

For example, if we have the bit pattern 0x0f, that (see below) is useful for extracting the bottom 4 bits of an 8-bit number, then its complement is 0xf0, that is

```
~0x0f = 0xf0
```

This pattern removes the bottom 4 bits and keeps the top 4. The exact opposite of the original.

Logical AND

The logical AND operation combines two single bits using the rule that the result is true only if BOTH inputs are true. We often represent this with a table of all the possible inputs and outputs called a **truth table**, as shown on the right.

AND

Left bit	Right bit	AND result
0	0	0
0	1	0
1	0	0
1	1	1

The logical AND operator, like the rest of the logical operators, works simultaneously on all the bits of its operands. This means that the two operands must be the same size. If one operand is smaller then C expands it by padding it on the left with zeros to match the larger.

The magic of the AND operation is that it can pick a subset of the bits from a word using a **mask**, a bit pattern with 1s in all the places you want to keep and 0s in all the places you want to ignore. We will see one example of this later in this chapter, where we will use a mask to extract the four lowest bits from a number.

Let's see how this works. Let's assume that we start with a 16-bit short and that we want to extract only the lowest four bits, the lowest hex nibble. In that case we start with a mask that has 0s in all the upper bits and 1s in just the four lower bits. That gives us a mask value

```
0x000F = 0b0000 0000 0000 1111
```

Now we AND this with a number and see the effect. Consider the initial value 0x13B7. Then we have, since 0x13B7 = 0b0001 0011 1011 0111,

```
0x13B7 & 0x000F = 0b0001 0011 1011 0111
                & 0b0000 0000 0000 1111
                  0b0000 0000 0000 7111 = 0x0007
```

and we have, as promised, extracted only the bottom hex nibble.

We can use the AND operation to force some bits in a word to 0 while leaving all the rest untouched. For example, we might force bits 5 and 6 of an 8-bit value to zero with the mask 0b1001 1111 = 0x9F. For example, if we start with the value 0xFE and we AND it with this mask then we get

```
0xFE & 0x9F = 0b1111 1110
            & 0b1001 1111
              0b1001 1110 = 0x9E
```

OR

Left bit	Right bit	OR result
0	0	0
0	1	1
1	0	1
1	1	1

Inclusive OR

We use the English word OR to represent two different logical operations. The first is known as the inclusive OR. It produces a true result when either or both of its inputs are true. The output is false only when both inputs are false. We see the truth table in the figure on the left.

Just as the AND operation can be used to clear some bits of a pattern without altering the rest, so the OR operation can set some bits without altering others. If we put a 1 bit in the mask then there will be a 1 in the result regardless of the original value. Where we have a 0 bit in the mask, that bit will just be copied into the result. So I can set bits 0 and 5 of a byte by ORing it with the mask 0b0010 0001 = 0x22. For example

```
0x55 | 0x22 = 0b0101 0101
            | 0b0010 0001
              0b0111 0101 = 0x75
```

Here we see the difference between addition and the OR operation. If we added the value and the mask then there would be a carry out of the bottom bit and several bits in the result would be modified by one bit in the mask. Because OR treats each bit position separately this does not happen. We just end up with the original value with a couple of bits that we are assured are now 1.

Exclusive OR

XOR

Left bit	Right bit	OR result
0	0	0
0	1	1
1	0	1
1	1	0

This is the other form of OR. It is slightly more usual to mean this in English. It represents a choice; either one thing *or* the other *but not* both. Because the English word is ambiguous, logic uses the word XOR (usually pronounced *ecks-or*) for this operation to distinguish it from the inclusive OR. This one is quite clever. If you look carefully at the truth table then you will see that it contains two relations that we can write with equations

```
A XOR 0 = A
A XOR 1 = !A
```

where ! is the logical not operation. This means that we can change the state of one or more bits in a word without knowing their state. For example, I might have a variable LightOn that tells me whether a light is on or off. So long as I use 0 to mean off and 1 to mean on, I can change the state of the variable with the single statement

```
LightOn = LightOn ^ 1;
```

Now light will have changed state. This is a lot easier than the more straightforward

```
if (LightOn == true) {
     LightOn = false;
} else {
    LightOn = true;
}
```

Useful Tricks with Logic Operators

These logical operators are very useful for affecting some bits of a value without altering others. This is a particularly powerful trick for working with the Special Registers that control the inner workings of the on-chip peripherals. It is normal to have each bit, or small set of bits, have a different meaning and function, quite independent of the other bits in the same word. Thus, it is common to wish to alter some bits while leaving the rest alone.

The simplest cases are those that set a specific set bits to zero or that set some bits to 1. For example, we can set the bottom three bits of a value to zero using the AND operator. Whatever the value of SReg is, the following code will zero the bottom 3 bits and leave the rest untouched.

```
SReg = SReg & 0xF8;
```

The numeric argument, called the mask, has zeros in those bits that we which to clear and ones on the others, 0xF8 = %11111000. Because it is the bottom bits that are important, you might also see this written

```
reg = reg & ~0x07;
```

using the unary NOT operator to make the mask out of its complement.

Similarly, we can use the OR operator to set one or more bits of a register to 1 without altering the other bits. This time we put 1s in the bits that we wish to set and 0s in the others. Thus

```
SReg1 = SReg1 | 0x02;
```

will set bit 1 of the memory location SReg1.

In the most complex case we combine these. For example, to set bits 3, 4, and 5 of SReg3 to the bit pattern 101 we can use the following

```
SReg3 = (SReg3 & 0b1100011) | 0b0010100;
```

6.1.3 Conditional Statements

Conditional statements allow our programs to make decisions as they run. At the heart of all conditional statements is some kind of logical test, a true/false question, usually built from the comparison operators and logical operators described above.

In these comparisons it is important that A and B are either both signed values or both unsigned values. Unexpected things happen if you mix signed and unsigned values. For example, depending on the implementation, you might easily find that -1 was greater than 10!

In a flowchart, a conditional statement is shown as a diamond shaped box or as a box with pointed ends (Figure 6.1). The box has one entrance (usually from the top) and two exits marked true and false, or yes and no, or y and n. If the condition is satisfied then the true (yes, y) exit is taken. Otherwise the false (no, n) exit is taken.

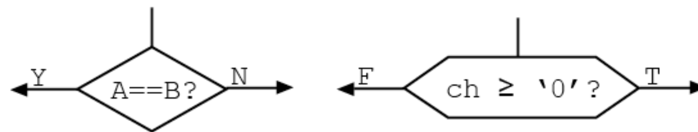

Figure 6.1

In C, a conditional statement begins with the keyword "if" and may take several forms. The simplest form is

```
if (<conditional expression>) <statement>;
```

Note In this and in the other conditional and iteration statements the parentheses are required, a part of the syntax of the statement. They *must* be there.

It is relatively unusual to use a simple statement as the body of an if and I strongly prefer to always use a compound statement, or block. Thus the preferred form of the simple if statement is

```
if ( <conditional expression\> ) {
    <statements making the body of the if>
}
```

Note As usual, the block does not need a semi-colon after it, although a simple statement would.

In this case the body of the loop is only executed if the expression evaluates as true. If the expression is false then that whole block of statements is skipped over completely.

There is an extended version of the `if` that has two bodies, one to execute if the test passes and one to execute if the test fails. Again, each of the bodies could be a simple statement but I strongly prefer the use of blocks for bodies. So the extended if has the form

```
if ( <conditional expression> ) {
    <list of statements to execute when true>
} else {
    <list of statements to execute when false>
}
```

In such a statement, we first evaluate the condition and then take one of two paths. In the first form of the statement, the true path leads us to the list of statements inside the curly brackets and the false path takes us to the next instruction after the closing bracket. In the second form, both true and false paths have their own lists of statements. As soon as we reach the end of the list, we go to the next statement after the closing curly bracket of the else clause. Thus only one of the two lists of statements is executed and the other has no effect.

For example the following C code and flowchart (Figure 6.2) examine the value in myNumber (a short) and print a message telling us whether or not the value will fit in a single signed byte.

```
if (myNumber < -128) {
    Serial.println("Your number is too small to fit in 1 byte.");
} else {
    if ((myNumber > 127) {
         Serial.println("Your number is too big to fit in 1 byte.");
    } else {
        Serial.print("Your number will fit in 1 byte.");
     }
}
```

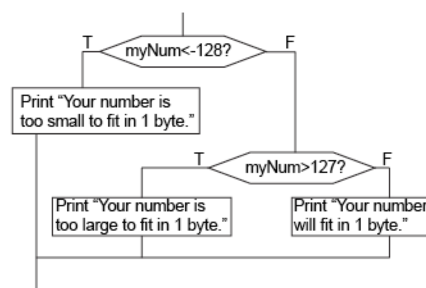

Figure 6.2

You can concatenate if-else statements to form longer tests where exactly one of a set of bodies will be executed. You get chains of ifs like this.

```
if ( <test1> ) {
    <body 1>;
```

```
} else if ( <test2> ) {
     <body2\>;
} else if ( <test3> ) {
    <body3>;
} else {
    <last body>;
}
```

where there could be as many else-if pieces as you need and there need not be a final else.

Example 6.1.1

One common way to design the software running on a scientific instrument uses a simple control language. An external computer, usually a PC, uses a serial link to talk to the instrument using strings of characters. The software on the instrument looks at the first character of the string to decide how to interpret the rest of the string. Thus, if the first character of the string coming from the PC is put in the variable `ch`, a piece of code like the following is very common.

```
if (ch == `a') {
    <do the work for command a>;
} else if (ch == `d') {
    <code to do the work for command d>;
} else if (ch == `n') {
    <code for command n>;
} else {
    <}code to tell the caller there is no such command>;
}
```

6.2 Iteration

Iteration is the act of doing something again and again. It is one of the fundamental concepts that we use to make long programs out of short statements. We shall use four basic kinds of iteration, all slight variations on the idea of doing the same thing many times in a controlled way.

6.2.1 The while loop

We have already met the simplest loop construct in C, the while loop, a loop that executes while some condition remains true. The while loop actually comes in two forms. In the form that we have already met we test the condition just before we execute the body of the loop. In the other form we test the condition after we have executed the body. The key difference between these two forms is that the second form always executes its body at least once. The first form will never execute if the test fails on the first try.

Note: While the parentheses that we use to make expressions more readable are completely optional, the parentheses in the while loop are required by the syntax of the language. They are NOT optional.

In a flowchart, we make these loops by putting a conditional box inside the loop. That gives us the two forms of Figure 6.3.

In C we use the while and do keywords to build these loops. The first kind of loop has the form

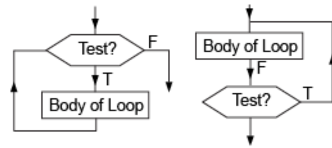

Figure 6.3: While Loop Flow Charts

```
while ( <conditional expression> ) {
    <body of the loop>;
}
```

and executes the statement in the body so long as the test is true. This form may or may not execute the loop body depending on the result of the test.

The second form moves the while to the end to show that the test is now performed after the body is executed.

```
do {
    <body of loop>
} while ( <conditional expression> );
```

This form of the loop is guaranteed to execute the body of the loop at least once, since the body executes before the test is first performed.

Example 6.2.1

As we shall see in later chapters, a computer sometimes needs to wait for something to happen in the outside world. For example, if a program interacts with a user through a keyboard then the program will often want to wait until a key is ready. Energia provides the Serial.available() command that returns the number of characters ready to be read from the serial port. The following code will sit in a while loop waiting until there is a character ready for us to read. Once one is ready we.can read it in and echo it back to the port so that the user can see what they typed.

```
char ch;
while (Serial.available() == 0) {
}
ch = Serial.read(); // Read character into ch
```

Note that the body of the while loop is empty. The loop does nothing while it is waiting.

Another way to write this which does exactly the same thing is.

```
char ch;
while (Serial.available() == 0); // Idle while no characters}
ch = Serial.read(); // Read character into ch
```

6.2.2 The for loop

Many computer languages provide a special form for the counting loop, a loop that must execute a fixed number of times. Such a loop can always be constructed from a while loop, but it is common enough to have a special form of its own. C provides a somewhat expanded version of the counting loop called a for loop, which has the form

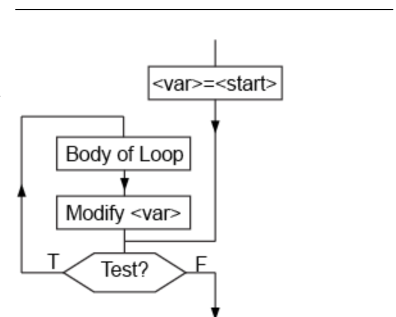

Figure 6.4: For Loop Flow Chart

```
for ([<var> = <start>];[ <test>]; [<change var>]) {
    <body of loop>
}
```

where <var> is the variable that counts through the loop, <start> is its initial value, <test> is a test for the end of the loop, and <change var> is a statement to go from one value of <var> to the next. The body is executed so long as the test is true. The control flow is a little fiddly and is best understood with a flowchart.

Here we can see that the test is made once before we enter the body of the loop and then again after every trip through the loop. This means that the loop will not be executed at all if the test fails that first time.

for loop variables
C++ allows you to declare a new value in the expression list of a for loop. Such a new value only exists during the lifetime of the for loop and vanishes as soon as the loop exits. Thus if you want to access the value of the variable after the loop then you must use a variable declared elsewhere.
I recommend sticking to variables declared elsewhere until you get really comfortable with programming. Then you can make your own choices.

This is a rather general form that allows us to construct some rather fancy loops but more than 90 % of the time we just want var to count up from zero or down to zero by some constant step. In those cases we get the forms

```
for (<var>=0; <var> '<' <limit>; <var> += <step>) {
    <body of loop>;
}
```

and

```
for (<var>=<limit>; <var> '>'  0; <var> -= <step>) {
    <body of loop>;
}
```

In the really common cases when the step is one we can replace the fancy <var> += <step> by the simpler `<var>++` form, or `<var>-` for the decrement case.

Note Here we have the common case that <step> == 1 so I have used the <var>++. In such a case each of these loops will execute exactly <limit> times In this case we get exactly 100 interations. This piece of code finds the average of the numbers from 1 to 100.

Example 6.2.2

```
short sum = 0;
short num, count=0;
for (num=0; num < 100; num++) {
     sum += num;
     count += 1;
}
short average = sum / count;
```

6.2.3 The infinite loop

So far in C we have created an infinite loop with a while loop that cannot end,

```
while (true) {
    // Body of the loop
}
```

A common alternative is a special form of the for loop,

```
for (;;) {
    // Body of the loop
}
```

Some C compilers have special code for this for version that maximizes the speed of the loop, so that it is probably the preferred form.

Example 6.2.3

Here is a simple example of an infinite loop program in both C and flowchart form (Figure 6.5). The program blinks an LED on and off. We are using an LED connected to Port B bit 1.

```
pinMode(PB_1, OUTPUT); // Make port B bit 1 an output
for (;;) {
     digitalWrite(PB_1, HIGH); // Turn the bit, and LED, on.
     delay(100); // Wait 100mS}
     digitalWrite(PB_1, LOW); // Turn bit and LED off
     delay}(100); // Wait 0.1 seconds.
} // End for loop. Never get past here.
```

I have made a common simplification in the flowchart. Instead of writing each of the body statements (corresponding to lines 2-5 in the pseudocode) in its own box, I have grouped them in a single box. It saves a lot of box drawing.

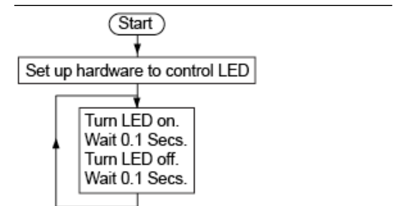

Figure 6.5: Flowchart for Blinking an LED

6.2.4 Leaving early?

Sometimes you need to be able to get out of a loop before reaching the normal end. For example, you could scan through all the characters in a line looking for a special character. The loop would normally terminate when it ran out of characters but there is no point in going on looking once you have found what you seek. C provides two different ways to change what you are doing without working through the body of a loop.

The break command stops the loop and takes you to the next statement outside the loop. This is much the most common way to end a loop early. This would be the solution to our previous example. So if we had a string of characters (see below) in an array called `buffer` we could search through it for the first occurrence of the character '*' like this

```
int index, foundIndex = -1;
for (index = 0; buffer[index] != 0;++index) {
    if (buffer[index] == '*') {
        foundIndex = index;
        break;
    }
 }
// Come here when either the loop ends or we
// found the char. foundIndex will be -1 if
// loop ended but will be the index if we
// found the `*'.
```

More rarely, we want to stop processing the rest of this trip round the body of a loop and skip to the next regular time round. We can do this with the `continue` command. The general format is something like this

```
for (index = 0; index > 100; ++index) {
     if ( <test> ) {     // Test for some condition
          continue;    // and dump the rest of the
     }                      // loop if it is met.
    //
    // Do something long and complicated
    //
    ...
}
```

In this example, suppose that we have just started the third trip round the loop, when `index` has the value 2, and we decide that this one is not worth bothering with. The `continue` statement skips over the rest of the processing in the loop, takes us to back the point where the test is made, and then takes us round the loop again, this time with `index` having the value 3.

6.3 Summary

C represents truth values as numbers with the meanings:-

zero → false
non-zero → true.

Conditional expressions produce truth values as results using the operators:-

`>` greater than
`>=` greater than or equal to
`==` equal to
`<=` less than or equal to
`<` less than
`!=` not equal to

Individual conditional expressions can be combined with the logical operators:-

`&&` logical AND
`||` logical OR
`!` logical NOT

which have lower precedence than the comparisons.

C also provides a full set of bitwise logical operators:-

`<<` left shift
`>>` right shift
`&` logical AND
`|` inclusive OR

^ exclusive XOR
~ NOT.

These act on their arguments one bit at a time to yield a bit pattern as a result.

C provides a general conditional statement that is quite flexible:-

```
if ( <conditional expression> ) {
    <list of statements to execute when true>
} else {
    <list of statements to execute when false>
}
```

Here the square brackets indicate optional elements. Since the if is itself a statement this means that you can form extended chains of if like this

```
if ( <test1> ) {
    <body 1>;
} else if ( <test2> ) {
     <body2>;
} else if ( <test3> ) {
    <body3>;
} else {
    <last body>;
}
```

where there could be as many else-if pieces as you need and there need not be a final else.

C provides a wide variety of iteration statements. The two forms of the while loop differ in when they perform the test.

```
while ( <conditional expression> ) {
    <body of the loop>;
}
```

and

```
do {
    <body of loop>
} while ( <conditional expression> );
```

The body of the second form will be executed at least once, while the first form could skip over its body altogether.

The for statement is a special form that combines initialization, test, and advance state into one shorthand syntax.

```
for ([<var> = <start>];[ <test>]; [<change var>]) {
    <body of loop>
}
```

This is exactly equivalent to

```
<var> = <start>;
while (<test>)) {
    < body of loop >
     <change var>;
}
```

The most common forms of the for loop count a single variable up or down a fixed number of times.

```
for (<var>=0; <var> '<' <limit>; <var> += <step>) {
    <body of loop>;
}
```

and

```
for (<var>=<limit>; <var> '>'  0; <var> -= <step>) {
    <body of loop>;
}
```

C provides two statements to alter the flow of loops.

```
break;
```

stops execution of the loop and takes you to the next statement after the end of the loop.

```
continue;
```

prematurely ends this trip round the loop and starts the next trip at the either the test (while loops) or the variable increment (for loops).

Chapter 7

Digital I/O and the TM4C123GXL LaunchPad

7.1 Introduction

Back in Chapter 5 we saw that the TM4C123G is a complex chip with a large set of on-chip peripherals in addition to the CPU and memory. Together, those peripherals form the connection between the abstract world of the programs we write and the concrete world of the circuits to which we connect the computer. The peripherals are the computer's senses and its limbs. Through its input wires it receives its impression of the outside world. With its output wires it works its will in that world. This chapter takes a more detailed look ways to use digital I/O with Energia.

GPIO
The TM4C123G documentation usually refers to the digital inputs and outputs as GPIOs, General Purpose Input and Outputs, and also uses talks about whole ports as GPIOs. Thus another way to refer to PortA is as GPIOA. You should be familiar with both notations.

Internally the TM4C123G chip groups individual I/Os into blocks called **ports**. Energia and Arduino do not really recognize this organization and like to think of them as individual signals each associated with a pin. Thus, the Energia functions expect to refer to the pins by their, rather arbitrary, pin number. The numbers in question are the numbers of the pins in the interface connectors on the LaunchPad. You can see the relationship between Energia pin number and the GPIO signal name in Figure 7.1 below.

The Energia pin numbers are the ones on the connectors, J1-J4. Thus, the internal number to refer to port F pin 1 is 30. We can usually get away without using these numbers directly because Energia provides names for the constants. Thus we usually refer to port F pin 1 as PF_1, but this is just a name that is defined to have the value 30.

For reference, is a table to convert pin names to pin numbers and is a table to convert pin numbers to pin names.

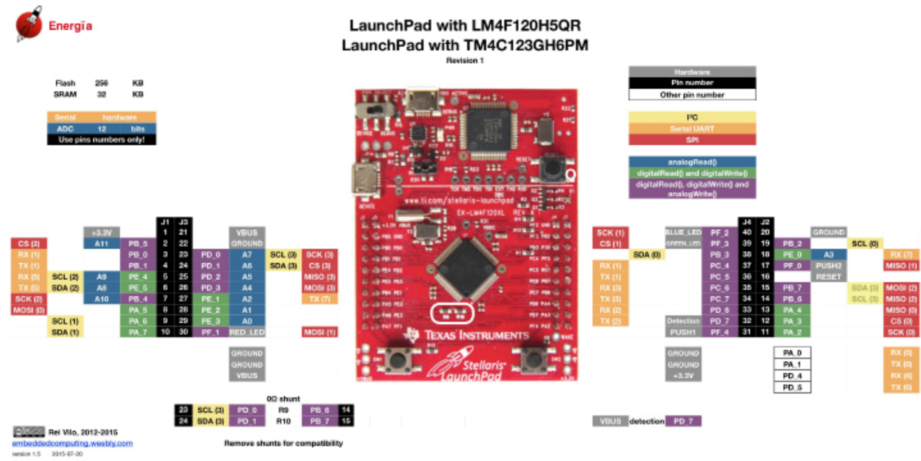

Figure 7.1: LaunchPad Pin Numbering

7.2 A Single Digital I/O

A single digital I/O is a signal that can take on only one of two states, HIGH (3.3V) or LOW (0V). If the voltage is controlled by the external circuitry then the I/O should be programmed to be an INPUT and then programs can test its state. If the voltage is controlled by the TM4C123G and is to affect the state of the external world then the I/O should be programmed to be an OUTPUT and then programs can set its state. We set the direction of a signal, its mode, with the `pinMode` command.

pinMode(<pinNo>, <mode>)
Sets the mode of a single digital signal. The signal is referred to by its pin number in the LaunchPad connectors. It is an integer between 1 and 40.

As mentioned above, we usually do not use the pin numbers directly. Instead, we use the more friendly names provided by Energia.

The mode is one of the special constants INPUT, OUTPUT, INPUT_PULLUP, or INPUT_PULLDOWN. We have already met the self-explanatory INPUT and OUTPUT modes. The other two modes are modifications of INPUT.

`INPUT`
In this mode the pin appears to the outside world as if it were a very high (>1M Ohm) resistor to ground. The voltage is determined by the external circuitry and the state can be read.

`INPUT_PULLDOWN`
As noted above, the normal INPUT mode makes the pin look like a huge resistor to ground. Sometimes it is useful to make the input look like a smaller resistor to ground. In the INPUT_PULLDOWN mode the input looks like a 10k resistor to ground so that it will tend to pull the input voltage down to ground unless the external circuitry pulls it up harder. We will see some uses for this when we look at how to connect switches to inputs.

`INPUT_PULLUP`
This is sort of the opposite. Instead of looking like a resistor connected to ground, this makes the input look like a 10k resistor connected to 3.3V. Again, it is useful for wiring switches to inputs without additional circuitry. As always, the progam can read the state of the pin and learn whether the voltage is 0V (LOW) or 3.3V (HIGH).

`OUTPUT`
This, obviously, makes the pin into an output. Specifically, it makes it into a standard CMOS output that is capable of sourcing or sinking 2mA of current. That is not a lot. You can just about light a high-efficiency LED with that much current but not much more. You pretty much always need some kind of external switch (an FET) if you are to control anything more demanding than a CMOS input.

It is perfectly legal to change the mode of a pin at any time but it is quite rare. Most programs set the direction in the `setup` routine and leave it that way. Once the mode is set then you can either write to the pin or read from it.

digitalWrite(<pinNo>,<state>)
This forces the the signal either to 3.3V if state is HIGH or to 0V if state is LOW. If the pin is configured as an input then this has no effect.

Well, no immediate effect.
If you set the state of an input and then later change the mode to OUTPUT then the signal will go to the previously set state.

digitalRead(<pinNo>)
This queries the instantaneous state of the pin and return HIGH if the voltage is near 3.3V and LOW if the voltage is near ground. It cannot return any other value. If you read from a pin that is configured as an output then you simply learn what value you wrote to the pin.

7.3 Using a Digital IO

Let's look at some of the things that we can do with our I/O pins now that we have them.

7.3.1 Driving a single LED

There are two different ways to drive a single LED, because there are two different ways that you can connect the LED to the pin (Figure 7-3). Of course on the TM4C123 you can only do this with a very high efficiency LED since the pins cannot supply enough current to drive most older LEDs directly.

The Blink example in section 6.2.3 showed how to turn on and off such an LED.

The upper LED is connected, through a current limiting resistor, between the port pin and the positive supply. In order to turn it on we must set the port pin to 0, so that there is a voltage across the LED. A 1 on the port will turn the LED off.

The lower LED is connected, again through a resistor, between the port pin and ground. This time, a 1 turns the LED on and a 0 turns the LED off.

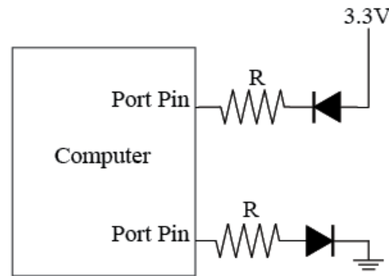

Figure 7.2: Driving an LED with a Port Pin

In either case, the resistor is there to limit the current flowing in the LED when it is turned on. The turn-on voltage for an LED varies somewhat with the color of the LED, but values around 2 V are typical. With a computer running from a 3.3 V power supply this means that we can't just connect the LED directly to the pin. Far too much current will flow and the pin, and probably the LED, will be damaged or destroyed. We need to include a resistor to limit the current to a safe value.

Standard brightness LEDs are usually rated at 20mA drive current so that driving them with only 2mA does not produce the brightest LEDs. If we want more light then we need either to used external LED drivers, such as FETs, or to switch to high-brightness LEDs that will give more light for the same current.

With a 3.3V power supply and about 2 V across the LED we are left with about 1.3V across the resistor meaning that we need a resistor value of 1.3 V/2 mA = 650 W.. The next standard value of 680 W is probably a good choice and 750 W would be a conservative choice.

7.3.2 Driving a Higher Current Load

While we can find LEDs that will light up sufficiently with <2 mA of current drive, there is not much else that you can turn on and off with that little current. We can add an external FET switch or an external driver IC to allow us to control something beefier.

The HIGH state output voltage of our chip is only 3.3V so that we can turn on an FET with a threshold voltage in the 1-2V range. Our common 2N7000 FET is a reasonable choice. According to Figure 11-7 in the Electronics volume, a 2N7000 can turn on about 50mA with a 3.3V gate drive. That is enough to drive a couple of standard brightness LEDs or energize a small relay. Figure 7.3 shows how to use an FET to allow us to turn on and off a standard LED that draws about 10mA, far too much for the chip itself to drive.

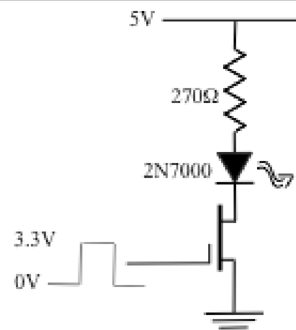

Figure 7.3: Controlling an FET

7.3.3 Reading a Switch.

The most common input task is reading the state of one or more switches. One of the simplest ways to connect a single switch to a digital input is

shown in Figure 7.4. We can obviously extend this to many switches.

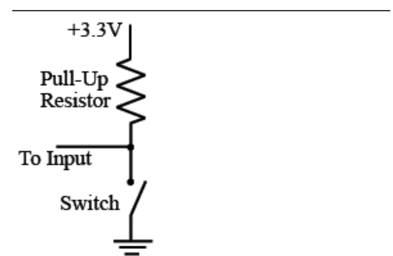

Figure 7.4: Reading a switch

When the switch is open, the resistor pulls the input wire up to the full supply voltage, 3.3 V for our chip. So long as the pin is programmed to be an input, a read of the port will see this bit as a logic 1.

When the switch is closed, the very low resistance of the switch pulls the input wire down all the way to ground. A read of the port will show this bit to be a logic 0.

We can test the value of the switch with the `digitalRead` command. For example, if we connect a switch to bit 2 of port A then we can test the value of the bit like this

As explained earlier, we do not even have to provide an external pull-up resistor if we enable the built-in pull-ups for the input bits by setting the pinMode to INPUT_PULLUP.

```
    if (digitalRead(PA_2) == LOW) {
        // Come here if the button is pushed
    } else {
        // Come here when button is not pushed
    }
```

In order to make testing the switch as transparent as possible, we might choose to package this code up into a subroutine. For example,

```
int ReadSwitch() {    // Returns true when button is pushed
    if (digitalRead(PA_2) == LOW) {
        return true;
    }
    return false;
}
```

Of course, both of these chunks of code assume that you have already made sure that the port bit is set up as an input and that there is either an internal or external pullup resistor.

7.3.4 De-bouncing switches

So long as we are only interested in whether the switch is open or closed, and do not care what happens when it changes state, that is all there is to reading a switch. However, if the behavior when the switch changes is important then we must look more carefully at what happens when a switch opens or closes. Many mechanical switches consist of two pieces of springy metal that meet to close the switch and part to open it. The trouble with springy pieces of metal is that they can bounce around when you open or close the switch. So, instead of getting a clean rise from 0V to 5V when you open the switch, you actually see something like Figure 7.5.

Figure 7.5: Switch Contact Bounce

This contact bounce happens too fast for mere people to notice but it happens extremely to a computer. The time scale is milliseconds and a TM4C123G can test the value of a port thousands of times in 1 mS. Thus you can't just connect up a switch like this and hope to do something like count the

number of times the switch is pressed. Each time the person puts her finger on the button, or takes it off, the computer will see a random number of on-off transitions. If we want to get a sensible answer out of the switch then we must **debounce** the switch. It is possible to do this with hardware but hardware costs money and a few extra instructions in a program cost nothing. We need to debounce the switch in software.

One very nice way to handle this is to write a subroutine that tests the switch and only returns when it is sure that the switch has gone high. It can tell that the switch is really high if it reads the state twice, a mS or so apart, and gets a 1 answer both times. Assuming that we have the ReadSwitch() that we just wrote, here is the code.

```
void WaitForSwitchHi() {
    int bit;
    while (true) {
        do {
            bit = readSwitch();
        } while (bit == 0);
        delay(20);
        bit = ReadSwitch();
        if (bit == 1) return;
    }
}
```

The program sits in a loop reading and re-reading the switch until it sees a 1. That one is either the start of a bounce or the start of a real high, so the program waits a short time and tests the bit again. If the bit is still 1 then it returns, happy that the switch is stable. If the bit is back to zero then the first 1 was a bounce and could be ignored. You can easily adapt this idea to detect falling edges or even 0-1-0 pulses.

7.4 The GPIO Port

So far we have treated the digital pins as completely separate entities. As their names suggest, this is not the case. The pins are actually organized inside the chip into groups called **ports**. A port is set of 8 digital I/Os that are all connected to the same set of special registers. Basically, each pin corresponds to a single bit in the registers. This means that the hardware can change the state of all 8 bits of a single port simultaneously. There are some occasions when this is extremely useful.

While it is true that each of the special registers has 8 bits, not all of those bits are available for our use. Some are reserved for special uses by the TM4C123G chip itself while others are used by the LaunchPad board and are not brought to the connector pins. This is how a chip with 6 ports can have only 45 I/O pins and then can be used on a LaunchPad that only

makes 35 of those pins available for our use. The following table shows the fate of each bit in each port. Each entry shows either the LaunchPad pin number, a LaunchPad signal name, or a blank for pins that don't exist.

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Port A	10	9	8	13	12	11	UART->USB	
Port B	15	14	2	7	38	19	4	3
Port C	34	35	36	37	The ICDI debug pins			
Port D	32	33	USB		26	25	24	23
Port E			6	5	29	28	27	18
Port F				31	39	40	30	17

Table 7.1: Launch Pan Pin Numbers

Missing Pins
It turns out that some of the missing pins are more hidden than missing. In particular, pins PD4 and PD5 are brought to pads next to the USB port on the side of the board. It is possible to solder pins to those pads and gain the full use of Port D.
Pins PA0 and PA1 are not quite as useful. They carry serial data from the target chip to the debugging chip so that we can use the Serial commands in Energia. If we don't need Serial then we can reuse PA0. PA1 is pretty much useless, however, as it is connected to an output pin on the debug chip so we can't really use it for anything else.

Interestingly, while the Wiring library that was the predecessor of Arduino provides support for access to all the pins in a port at once, that support was left out of Arduino and thus out of Energia. I have provided the missing routines so that you can use them in just the same way as other Energia routines.

portMode(<port>, <mode>)
This is the whole port equivalent of `pinMode`. It allows you to set the mode of all the pins in a port at once. I have provided named constants for all the ports that you can use for the first argument. The mode can be any of the same constants that you can use in pinMode. Thus, we can make all the pins of port B into output pins with the command

```
portMode(PORTA, OUTPUT);
```

Note that portMode knows about the missing pins in ports A and D thru F. It is quite safe to say

```
portMode(PORTC, OUTPUT);
```

even though the bottom 4 bits are not available to you. Only the upper 4 bits will be affected.

It is perfectly possible to mix calls to pinMode and portMode. For example, we could make the top 6 bits of port B into outputs and use the bottom two bits as inputs with the three commands

```
portMode(PORTB, OUTPUT);
pinMode(PB\_0, INPUT);
pinMode(PB\_1, INPUT);
```

The portMode command makes all 8 bits of the port into outputs and then the two pinMode commands switch individual bits to inputs. It is obviously important to do the individual pins after the portMode since it must affect all the bits the same way.

portWrite(<port>, <value>)

While `portMode` and `pinMode` are nearly identical, `portWrite` is a little more different than `digitalWrite` because it takes an 8-bit value as its second argument. Remember that we want to set the state of eight individual bits all at once. However, we very rarely want to set them all to the same value. Thus we need to pass in a bit pattern, i.e. a number, to `portWrite`. This is a place where binary numbers can make a lot of sense. For example, if we want to turn on all the even number bits and turn off all the odd number bits in port E then we could use the command

```
portWrite(PORTE, 0b10101010);
```

or either of the equivalent forms

```
portWrite(PORTE, 0xAA);
portWrite(PORTE, 170);
```

Remember that all values are represented inside the computer by bit patterns. Binary, hex, and decimal are just conveniences that C++ provides to make programming easier for us.

portRead(<port>)
This command reads all 8 bits of the port in a single operation and returns their state as a bit pattern. For any bits that are configured as INPUTs of any kind the bits returned will reflect the external voltages on the pins. For any pins that are configured as OUTPUTs, `portRead` will just return the values to which you set those pins.

7.4.1 Driving a 7-segment display

Our next example makes use of the ability to set a whole set of output pins in one operation. We shall connect 7 bits of a single output port to the LEDs of a 7-segment display and then make the display count continuously.

The LEDs in a typical display are not high-brightness types so we can't drive them directly. Instead we will let each port pin drive an FET switch connected to an LED through a current limiting resistor. Since we need a full 8-bit port, I have chosen to use port B. Figure 7.6 shows the circuit we shall use.

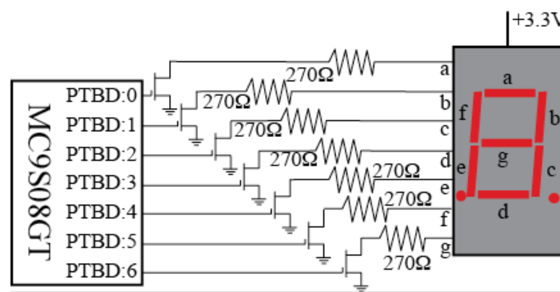

Figure 7.6: Driving a 7-segment LED

The seven-segment LED looks rather like a 14-pin IC but is really a lot simpler. It just consists of 9 LEDs set into a plastic base. It would take 18 pins to bring out all of the connections to the LEDs so the manufacturer connects together all the anodes or all the cathodes. For historical reasons, common anode LEDs are easier to find than common cathode ones and we shall use common anode LEDs for this example. As shown in Figure 7.7 below, the anodes of each LED are connected together on pin 14 and the separate cathodes brought out to various other pins.

We connect the common anode, pin 14, to the +ve power supply as shown in Figure 7.6 and then we can turn on any one of the LEDs by connecting its cathode to ground through the current limiting resistor.

Now it is easy to turn on and off any segment in the display. All we have to do is to program a 0 or a 1 into the appropriate bit in the port B data register. A 1 in the register will turn the associated segment on by turning on the FET. So, eg., the bit pattern 0b00000110 will turn segments b and c on and turn all the others off, displaying the digit '1'.

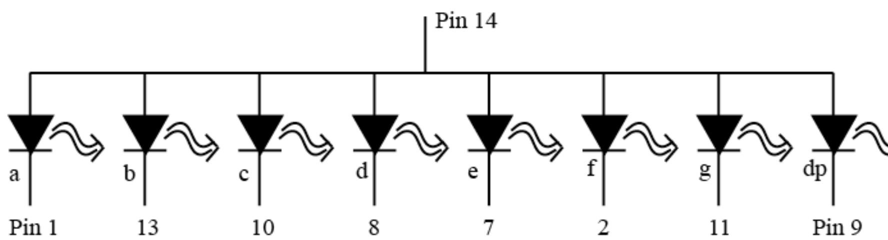

Figure 7.7: Inside a 7-Segment LED

So how do we display the digit 0? Well, in a 0, segments a-f must all be turned on while segment g must be turned off. That means that we need to put a 1 in bits 0-5 while bit 6 is a 0. It does not matter what we put in bit 7 since it is not connected to an LED. So we can display a 0 by setting port B = 0b00111111 = 0x3F.

We can derive a similar hex byte for each of the other 9 decimal digits. We get this set of codes.

0 → 0x3F, 1 → 0x06, 2 → 0x5B, 3 → 0x4F, 4 → 0x66,

5 → 0x6D, 6 → 0x7D, 7 → 0x07, 8 → 0x7F, 9 → 0x67

All that our program has to do is to cycle through these values in order and keep doing it forever. If we want to be able to see the numbers, then we shall have to slow the program down and display each number for a little while, say 1 second.

The really easy, if slightly klutzy, way to do this is to write 10 little segments, one for each digit. Then we just string them together, something like this

More Patterns
This same idea can be extended to support as many patterns as you would like. For example, it is easy to extend the system to support display of Hex numbers by adding bit patterns for the Hex digits, A, B, C, D, E, and F. Though it is usual to use a mixture of upper and lower case to avoid confusion between digits and numbers. Thus we usually use the patterns A, b, C, d, E, and F and we distinguish between the b and the 6 by turning on bit 'a' in the 6 character but not in the b character.

```
void loop() {
    Write code for `0' into port B
    Wait 1 sec
    Write code for `1' into port B
    Wait 1 sec
    ...
    ...
    Write code for `9' into port B
    Wait 1 sec
}
```

This will work perfectly well and it will fit easily into the program memory of our TM4C123G so there is nothing wrong with this solution. However, it is not elegant. A much more elegant solution is to use a loop to send each digit. If we arrange the digit codes in order in memory (like a string) then we can step through the codes one a time, like this.

```
char code[10] = {0x3F,0x06,0x5B,0x4F,0x66,
                0x6D,0x7D,0x07,0x7F,0x67};

void setup(void) {
    portMode(PORTB, OUTPUT);
}

void loop(void) {
    for(int index = 0; index < 10; index = index+1) {
        portWrite(PORTB, code[index]);   // Write current digit
        delay(1000);                     // Wait for one second
    }
}
```

I have used a for loop to step through the entire table in order and then relied on the system to keep calling the loop routine so that the process repeats forever.

7.4.2 Reading a Switch Matrix

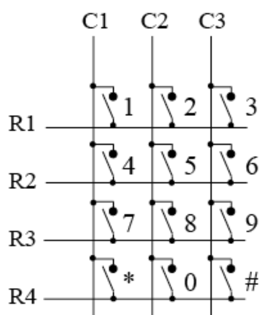

Figure 7.8: 4x3 Matrix of Switches

If you have a large number of switches to read then you can easily run out of I/O pins. For example, a standard desktop computer keyboard has more than 100 keys, each of which is a switch. Rather than connect each key to its own input pin and sit scanning through all the input pins we connect the switches into a **matrix**, where each switch is connected to both an input pin and an output pin instead of to an input pin and a fixed voltage. For example, a telephone keypad is usually laid out as a matrix switch. Figure 7.8 shows 1 possible arrangement.

Here there are 12 switches connected to only 7 wires. The savings get larger as the number of switches increases. For example, we could add another

column of four switches to this matrix at a cost of only 1 more wire–16 switches on 8 wires. The secret to reading a switch matrix like this is to deal with 1 row at a time. Instead of connecting the column wires, C1-C3, to power supplies we shall connect them to three output pins and then we shall connect the four row wires, R1-R4, to four input pins. Next, we have to make sure that the inputs are pulled to a standard state even if no switches are closed. Since all of the standard input ports have pull-up resistors built in, this is easy.

In general 2n wires can handle 2^n switches.

Now we can read the states of the switches by examining them 1 column at a time. We shall start with column C1. Since the switch-open state of each row wire is a high voltage, we need to make sure that the unused columns, C2 and C3, appear as high voltages whether the switches are closed or not. Thus we pull C2 and C3 high. Now, to make C1 different we pull it low by writing a 0 to that bit and then we read the state of the four row lines. Everywhere there is an open switch we shall see a logic 1, everywhere there is a closed switch we shall see a logic 0. Thus we can tell exactly which of the switches 1, 4, 7, and * are closed by examining which bits are 0. By repeating this for the other two columns, we can examine the state of all 12 switches.

There are several ways of reading such a switch matrix depending on whether you want to record switch openings as well as switch closures and on whether you need to be able to tell when several keys are pressed at once. For a telephone keypad, you just want to know when a single key is pressed. In this case the program can sit and scan through the matrix expecting that almost all the time all the switches will be open. Occasionally someone will push a button and then the routine can return a code describing the particular button. Let us pursue this as an example.

We will connect the four row of wires to the four low-order bits of port B and the three columns to bits 4-6 of port B, which we will program as outputs. Then a routine to scan the switch until a single button is pushed will look something like this.

```
set port B to 4-input/3-output bits
while (true) {
    for (column = 1; column < 4; column = column + 1) {
        clear current column output bit in port B
        set other output bits in port B
        read port B and mask to 4 bits
        for (row = 1; row < 5; row = row + 1)
            if (current bit is 0) return row & column number
        }
    }
}
```

There are two tricky steps in this code. The first is setting and clearing the correct column bits. We somehow need to translate a column index into a

bit-pattern. This is just like the task we faced in the LED problem and the solution is the same, use an array. We build a 3-element array that holds the three bit patterns 0b01100000, 0b01010000, and 0b00110000, and select a value from that array with the column index.

The second tricky step is working our way through the switch values that we read from port H. Here we again need to translate a row number into a particular bit. We could do this the same way, with a second array, but there is a better way to test each of the bits in a word in sequence. We can use the shift operator to slide the bits, one at a time, into a variable where we can test them. Here is a new version of the pseudocode with more of the details made explicit.

Arrays
The square brackets after Strobe tell C++ that this is to be an **array** variable, a set of adjacent boxes with a single name rather than just one box. The list of values in curly brackets tells C++ what values to put in each of the boxes. We will learn more about this in chapter 8.

```
unsigned char Strobe[3] = {0x60, 0x50, 0x30};

void setup(void) {
    portMode(PORTB, INPUT); // Make them all inputs
    pinMode(PTB_4, OUTPUT); // Then change the top 3
    pinMode(PTB_5, OUTPUT);
    pinMode(PTB_6, OUTPUT);
}

void loop(void) {
    for (col = 0; col < 3; col = col+1) {
        portWrite(PORTB, Strobe[column]);     // Drive column
        int switchVal = digitalRead(PORTB);        // Read rows
        for (row = 0; row < 4; row = row + 1) {
            char newBit = (switchVal << row) &  0x01;
           if (newBit == 0) {
                return row & column info;
            } // End if
        } // End for (row}
    } // End for (col}
} // End loop}
```

The only tricky piece of code here is the expression in line 8 that extracts the bit of interest. `switchVal >> row` takes the bit pattern in switchVal and shifts it right by `row` bits, leaving the bit of interest in the bottom bit. Then the AND operation zeros out all the other bits of the number leaving us with the value of just one bit, the bit of interest, which I placed into the `newBit` variable. I could have used the expression directly in the if statement, but this form is usually easier to debug. When you are in the debugger you can examine the value in `newBit` to make sure that it is what you expect.

The last task before we produce a final C subroutine is to figure out how to return the information. At the moment, we know where we are in the switch matrix because we know the row and column numbers of the switch. However, it would be nice to return 0x01 when the ‘1’ switch is pressed,

0x02 when the '2' switch is pressed and so on. We can return 0x0A for the '*' switch and 0x0B for the '#' switch (or we could pick any other unused values). We can easily do this with another array. We can convert the row/column pair into a unique index with the algorithm

```
index = column * number of rows + row
```

This is a general formula for converting a 2-D index into a matrix into a single index into the array. All we have to do is to put the codes into the array in the right order. That is easy, we just work up or down the columns 1 row at a time, depending on the order in which we wire the pins. Let us say that we map

PortB:0 = R1, PortB:1 = R2, PortB:2 = R3, PortB:3 = R4,
PortB:4 = C1, PortB:5 = C2, PortB:6 = C3

Then when `column == 0` and `row == 0` we must be looking at the switch at the intersection of C1 and R1, that is, at the '1'. When `column == 0` and `row == 1` we are looking at the '4' key. Following this pattern through we get the following array.

```
int KeyCode[12] = [0x01, 0x04, 0x07, 0x0A,
                   0x02, 0x05, 0x08, 0x00,
                   0x03, 0x06, 0x09, 0x0B];
```

Here is the final version.

```
/*
* GetPhoneKey()
* This routine reads the state of 3x4 matrix switch
* connected to port B. The switch must be laid out}
* as follows C1 C2 C3
* R1 1 2 3
* R2 4 5 6
* R3 7 8 9
* R4 * 0 #
* R1 goes to PH0, R2 to PH1, R3 to PH2, and R4 to PH3.
* C1 comes from PH4, C2 comes from PH5, and C3 from PH6.
* The routine scans the matrix until it sees a key down,
* then returns the integer value of the key pressed with
* the * having value 10 and the # having value 11.
* Assumes PB_6, PB_5, and PB_4 are OUTPUTs and PB_0 to PB_3 are INPUTs.
* Brian Collett 1/24/2016
*/
/*
* Strobe array is used to turn on column strobes in correct
* sequence.
*/
unsigned char Strobe[3] = [0x60, 0x50, 0x30];
/*
```

```
* KeyCode is used to translate a row and column index
* pair into the code for the switch.
*/
int KeyCode[12] = [0x01, 0x04, 0x07, 0x0A,
                   0x02, 0x05, 0x08, 0x00,
                   0x03, 0x06, 0x09, 0x0B];

void GetPhoneKey() {
    char col, row;
    while (true) {
        for (col = 0; col < 3; col = col+1) {
            portWrite(PORTB, Strobe[column]);  // Drive column
            int switchVal = portRead(PORTB);    // Read rows
            for (row = 0; row \textless{} 4; row = row + 1) {
                char newBit = (switchVal << row) & 0x01;
                if (newBit == 0) {
                    return KeyCode[col * 4 + row];
                } // End if
            } // End for (row
         } // End for (col
    } // End while
} // End GetPhoneKey
```

7.5 Digital Interrupts

So far we have seen the computer as a device that can follow a precise set of directions to perform one or more tasks in a strictly sequential fashion. The program may jump from one line to another using constructs such as if, while, and for, but the code has always obeyed the basic principle that the computer executes the lines in their pre-determined sequence. That is fine for self-contained programs, but once we write programs that interact with the world around them this simple behavior becomes a limitation.

7.5.1 Polling versus Interruption

An asynchronous event is an event that occurs when it feels like it rather when told to occur by a program. Many such events arise outside the computer and affect the computer through its peripherals. A few, such as a timer expiring, are really under computer control—after all, the program tells the timer when it should do something—but still appear asynchronous to the program. It is like a person setting an alarm on her watch. She controls when the alarm will occur, but does not then sit looking at the clock and waiting for it. Instead she goes about her business and, at the predetermined moment, is reminded by the alarm, which then appears as a

surprise, an asynchronous event.

There are two different ways for a computer to deal with asynchronous events. The first is by **polling** and the second is with an **interrupt**.

The computer **polls** for an event by sitting in a loop and continually asking "did it happen yet?" as we did in our ReadSwitch routine in Chapter 7. This is the fastest way to respond to the event because you can act upon it as soon as you detect it. So long as the computer is not trying to do anything else, polling is the best way to deal with asynchronous events.

While polling makes use of the normal ability of the computer do a simple repetitive task, an **interrupt** requires special hardware. The hardware must stop the CPU in its steady procession through its instructions, to mark its place, and go off to execute a completely different piece of code, called an **interrupt handler**. This takes a little more time than polling, since there is some overhead to save the current state of the program and to switch to the new code. It is significantly more complex to program, but it offers huge gains in flexibility as soon as the computer needs to do something else while waiting for the event. Interrupts are the key to making a computer seem to be able to do multiple things at once.

An interrupt handler is a special kind of subroutine which is called by the hardware rather than by the program. We shall see several interrupt handlers in the course of this chapter.

Example 7.5.1

Consider a simple traffic light program. Basic traffic lights are extremely easy to program. They go through a fixed set of states at a regular rate. We can easily imagine connecting the lights to some port bits to turn them on and off and then using the delay() routine to control the timing of the steps.

Now imagine trying to add pedestrian call buttons to this system. One standard method is just to use the red-red phase of the lights. If the call button has been pressed then the computer should increase the duration of the red-red phase of the lights and possibly play with some cross/don't cross lights.

This is easy in a flow-chart but how do we handle the pedestrian call buttons in a real program? The buttons provide a signal to the computer only while the person has their hand on the button. The pedestrian wants to be able to push the button at any time during the cycle and have the computer remember this at the next red-red part of the cycle. If we do the obvious thing of testing the button at the start of each red-red cycle then the pedestrian will have to keep holding the button down until the right moment or the computer will completely ignore them. Not a good design. The only way to know about the state of the button at any time is to keep checking the state. But the computer spends all its time in delay. So we would have to write a new version of delay that mixed the job of waiting with the job of checking the button and saving somehow (a global variable comes to mind) the information about whether the button was pushed. This would require a complete rewrite of the traffic light program.

7.5.2 Interrupts and TM4C123G/Energia

All processors in the TM4C12x family provide hardware support for interrupts from a variety of sources; external events, on-chip peripherals, and even a software-initiated interrupt. The details of the supported interrupts vary from peripheral to peripheral but all follow the same outline.

First one must make sure that the interrupt is able to occur. Most of the individual kinds of interrupt have to be enabled separately, usually as part of the process of setting up the associated peripheral hardware.

Many peripherals have one or more interrupts associated with them and each usually has a separate single-bit flag to enable it in one of the special registers associated with the peripheral. For example, each of the digital I/O pins can generate an interrupt when the external state of its individual input pin changes.

In addition, there is a global switch, called the interrupt flag, that controls whether interrupts are allowed to occur at all. Energia provides a pair of commands that enable or disable interrupt processing. Under normal circumstances interrupts are enabled by the time Energia runs your `setup`() routine, as they are used to maintain the millisecond clock used by the `millis`() and `delay`() commands. It is possible to disable the interrupt handling for a short period with the command

```
noInterrupts();
```

and then to turn them back on again with the command

```
interrupts();{}
```

You would normally only do this to execute a few lines of code that absolutely must run as a continuous block. This can happen when making to some of the special registers for example.

Next there is the actual code to run when the interrupt occurs. This interrupt handler code must be put into a special kind of subroutine which must contain all the code to run for that interrupt. We normally try to keep interrupt handlers as short as possible since the main program is halted while they run. Energia actually takes care of some bookkeeping associated with each interrupt and lets us write an ordinary subroutine to do the rest of the work. This subroutine will be called by the actual interrupt handler and must be declared to take no arguments and return no values. Thus it must be declared like this:

```
void myHandler();
```

The last portion of the process connects the interrupt handler to the interrupt. Energia provides the `attachInterrupt` function to make this easy, at least in the case of the interrupts connected to the digital pins. You pass the number of the (input) pin you want to use, the name of the interrupt subroutine, and a constant that tells it when to to recognize an interrupt.

Here is a summary of the structure of a simple program that uses one or more interrupts.

- Declare any globals to share info.
- Declare, define, and name the handler subroutine(s). They may change values of globals. In `setup()`
  - Define your inputs and outputs as usual.
  - Connect your handler subroutines to their interrupts.

- In `loop()`
- Do using your usual processing. You can check the globals to see if an interrupt has happened.

7.5.3 Handling Interrupts.

When an interrupt occurs, the CPU saves the current state and transfers control to the address given in a piece of program memory called the **vector table**. It is up to Energia to make sure that the correct addresses are in the table and that each points to an interrupt handler that is suitable for that interrupt.

As described above, the interrupt handler may do some housekeeping but then it looks to see if the main program has defined a handler subroutine. If one exists then the subroutine is called. While that subroutine runs the main program is halted and no other interrupt can take place, so it is a good idea to keep interrupt handlers as short as possible. They should usually do the minimum amount of work to service the hardware and save information for the main program to act on.

When the hardware starts executing an interrupt handler it first adjusts the interrupt flag so that no further interrupts can occur. In addition, many of the peripheral systems take additional care to prevent unintended multiple interrupts. Most of them set a flag in one of their special registers to prevent all further interrupts of the same type. Thus, most handlers must contain code to clear their these flag.

Nested Interrupts
Because the hardware sets the interrupt flag before it starts to run an interrupt handler, we don't have to worry about an interrupt handler itself being interrupted. It is perfectly possible for the handler to reenable interrupts which would then mean that on handler could interrupt another. An interrupt that occurs during processing of another interrupt is called a *Nested Interrupt*. ARM processors provide extensive support nested for nested interrupts but I consider them an advanced technique and will not discuss them further.

7.5.4 Installing an Interrupt Subroutine

As mentioned earlier, Energia provides a standard way for us to install the most common type of interrupt subroutine. Each one of the digital I/O pins has the capability to generate an interrupt when its state changes. Usually, this ability is turned off but we can turn it on for a particular pin and specify both when the interrupt should be taken and what to do about it with the `attachInterrupt` function.

Interrupt Modes

The digital pin interrupts can be set to occur when some kind of change occurs to the state of the affected pin. Three kinds of change are possible.

1. The interrupt can be set to happen when the pin goes from LOW to HIGH. This is called a RISING interrupt.
2. The interrupt can be set to happen when the pin goes from HIGH to LOW. This is called a FALLING interrupt.

3. The interrupt can be set to happen when the state of the pin makes either kind of change. This is called a CHANGE interrupt.

Normally, we want interrupts to occur as the result of changes external to the chip. Thus any pin that is used to generate interrupts should first be made into an input.

It is possible to enable interrupts for a pin that is set to OUTPUT. In that case the interrupt will occur when the program makes the appropriate change in the pin state. This is a very weird thing to do!

attachInterrupt(<pinNo>, <handler>, <mode>)

The attachInterrupt routine takes care of all the behind-the-scenes work needed to tell the pin to support interrupts and under what conditions they should be taken and then it arranges that the appropriate handler subroutine will be called. It is very simple to use and hides from the user the rather large amount of work needed to make all this happen.

An additional benefit of the global trick is seen in the traffic light case. There it does not matter if the switch that we are using suffers from switch bounce. If the interrupt is called several times within a single traffic light cycle then several calls function exactly the same as one; they set the global.

Example 7.5.2

The digital interrupts allow us to monitor switches at the same time that a main program is doing its own job. The LaunchPad board has a three-colour LED connected to Port F pins 1, 2 and 3, and two switches, connected to Port F pins 4 and 0. We can demonstrate the interrupt capability with a main program that flashes the LEDs in clearly recognizable pattern and use an interrupt to monitor SW1. When SW1 is pressed, the interrupt code will alter the pattern.
The main trick to notice is the use of a global variable to communicate between the main program and the interrupt handler. Since globals are visible to all subroutines and the interrupt is, from C++'s point of view, just another subroutine, this is the standard way to share information between the two different kinds of code.

```
/*
* Interrupt Demo
* Blinks the LED in either red or green at 2 flashes per
* second. Every time SW1 is pressed the colour changes.
* Brian Collett 2/6/16
*/
int SW1 = PF_4;
int RedLED = PF_1;
int GreenLED = PF_3;
int NextColour = RedLED;
//
// This is called when SW1 is pressed. It swaps the LEDs;
// switching from red to green or green to red.
//
void HandleSwitch() \{
    if (NextColour == RedLED) \{
    NextColour = GreenLED;
    } else {
    NextColour = RedLED;
    }
}
void setup()
{
    // LEDs need to be outputs, SW1 input and interrupt:
    pinMode(SW1, INPUT_PULLUP);
    pinMode(RedLED, OUTPUT);
    pinMode(GreenLED, OUTPUT);
    attachInterrupt(SW1, HandleSwitch, FALLING);
}
void loop()
{
    // Blink the current LED:
    int colour = NextColour;
    digitalWrite(colour, HIGH);
    delay(100);
    digitalWrite(colour, LOW);
    delay(100);
}
```

Note that there is no attempt to de-bounce the switches so a 'single' press may advance the pattern more than one step.

7.6 Summary

The simplest form of communication between a computer and the world is the digital I/O bit. The TM4C123GXL LaunchPad provides 35 such pins spread over 6 ports. However, almost all of the IO pins have alternate functions and will not be available as general-purpose digital IO if the alternate function is needed.

Each port appears internally as a set of memory locations in the special register area. Bit patterns values written to those memory locations either affect the working of the pins or appear as voltages on those pins that are configured as outputs. Data read from the location reflect the voltage present on those pins that are configures as inputs. We normally do not need to access the special registers directly as Energia provides routines to make digital I/O easy.

You must include the Ports library if you want to use the commands to access complete ports.

Every IO pin can be configured as either an input or an output pin. The direction is set for individual pins with the `pinMode` command and for all the pins in a port with the `portMode` command. Special forms of the input mode, `INPUT_PULLUP` and `INPUT_PULLDOWN` make it easier to connect simple switches to inputs.

You first select the direction of each I/O pin using `pinMode`. Then you can alter the values on the output pins by setting them `HIGH` or `LOW` with `digitalWrite` and can find the values on the input pins by reading them with `digitalRead`.

If a port is programmed so that some bits are input bits and some bits are output bits then writing to the port will alter only those pins that are programmed as outputs. The values of the other bits will be lost. Any bits programmed as inputs will return the values according to the voltages on the pins.

Interrupts provide a way for a program to respond to asynchronous external events. The TM4C processors support interrupts on every one of their digital I/O pins. Energia allows us to enable such interrupts and install handlers for them with the `attachInterrupt` command. Each interrupt can be set to happen when the signal goes from LOW to HIGH (`RISING`), goes from HIGH to LOW (`FALLING`), and when the signal changes state (`CHANGE`).

Exercises

1. Write a subroutine to write single hex digits onto a 7-segment LED connected to the lower 7 bits of Port B. The routine should take a 4-bit number in accumulator A and show that digit on the LED.
2. Write a program to send a digital ramp out of port B. It should send the numbers from 0-0xFF out of port B as fast as possible and keep doing it for ever.
3. The technique that we used in the matrix switch program, using one set of bits to select one row and a second set to interface to the columns, can be adapted to drive multiple 7-segment LEDs using only 1 bit per LED in addition to the 7-bits to drive segments. We connect all the segment a's to one pin, all the segment b's to another and so on and then use bits from another port to turn on the anode drive to one complete LED at a time. Then our program rapidly cycles through, turning on an LED and turning on the bits that we want to see before turning that one off and going to the next LED. If we do this fast enough then the eye won't notice that the LEDs blink. Write a program to count on 2 LEDs connected with port B driving the segments (0 means turn bit on) and two lines from port A driving the power to the individual LEDs.
4. Write a modified version of the interrupt demonstration program that switches between all three colors of LED when SW1 is pressed.

Chapter 8

Variables in C++ Revisited

We have seen that C++ uses the abstract idea of a variable to provide a view of memory that is matched to the problem rather than to the hardware. From the hardware point of view all memory locations are the same. From the programmer's point of a view an integer used to count widgets on a conveyor belt is a totally different kind of object from a character read from a keyboard.

So far we have encountered variables that hold a single object throughout the lifetime of a single procedure. In this chapter we shall explore some of the more sophisticated types of variables that C++ provides: arrays, to store many related objects, globals and statics, to share information between procedures and from one invocation to another, and pointers, which hold addresses of other variables.

8.1 Arrays

In mathematical terms our ordinary variables are like scalars and our arrays are like vectors and matrices.

Simple variables are designed for holding single, unique objects. Sometimes we have to deal with collections of similar objects such as the letters in a string or the values in a table. We cannot hold such objects in a single memory location and so cannot hold them in a single simple variable. Instead we must introduce the idea of an **array**, a collection of similar values grouped together under a single name and accessed using a number as well as a name. Where a simple variable is a named box that can hold one value, an array variable is the name of a whole set of side-by-side boxes that are numbered consecutively, starting from 0, as suggested in Figure 8.1.

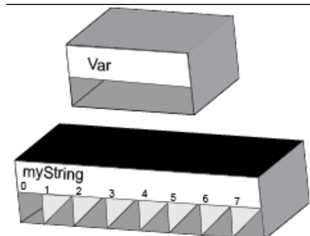

Figure 8.1: Array Storage

We can access the value of one of the individual boxes that make up an array variable by combining the name with an index thus: `xval[4]`. Such

an array reference behaves exactly like a variable and can occur on either side of an assignment. Thus, we can have statements like this

```
yval[4] = m * xval[4] + b;
```

which is part of filling in values that describe a straight line.

It is possible to be even more general and declare an array with more than one index, like a mathematical matrix, but that is beyond the scope of this book.

When we declare an array variable, we specify how many slots there are in the array by putting the number of slots in square brackets after the variable name. Thus, we would declare the integer array dataVals, with room for up to 10 numbers, like this

```
int dataVals[10];
```

The general form of an array declaration is

```
<type> <name>[ <const expr> ] = { < list of constants> };
```

where the equals sign and everything beyond it are optional. The expression in square brackets sets the number of elements (slots) in the variable and the type determines the size and interpretation of each element. Because the compiler has to know how much space to allot for the array, the number of elements must be known when the compiler runs so the expression has to be a constant that the compiler can evaluate. You can't decide how big an array is going to be when the program runs.

Larger computer systems normally do provide a way to create such dynamic arrays using a system of pointers. Such a use is very rare in embedded systems like ours.

As usual, it is possible to assign starting values to an array variable. Obviously it will take more than one value to fill an array so C++ provides a syntax for specifying any number of values (again, they must be constants) by putting a list of values in curly braces (‘{‘ and ‘}’).

It is up to the programmer to make sure that there are exactly as many initializing expressions as there are elements in the array or the compiler will usually issue a warning.

The advantage of an array variable is that the index need not be a fixed number. Instead, it can be any expression that evaluates to an integer within the range of values in the array. Thus we can refer to element `myString[theIndex]` where `theIndex` is any number in the range 0 to 9. Using variable indices we can write programs that process every one of the entries in an array in the same fashion without having to repeat the code. Instead we just put the code in a loop and execute the same piece of code many times.

Actually, C will let the index take on any value at all but does not define what happens if the index is too large (or too small) for the array. In practice such an index will reference a piece of memory that is probably used for some other purpose. Index errors like this are the primary cause of security vulnerabilities in the desktop computer world where they are often called buffer overflows.

Example 8.1.1

Here is a rather simple little example that finds the average of a set of numbers stored in an array. In this case I put the numbers there at the start of the program but obviously this would normally have happened at some earlier time in the program.

```
int data[10]={31,24,56,43,76,53,25,36,42,91};
int sum = 0;          // Will accumulate the total
int i;                   // Our loop index
int average;        // Will hold the answer
for (i = 0; i < 10; i = i + 1) {
    sum = sum + data[i];     // new data element each time!
}
```

```
average = sum / i;
```

Note the trick of dividing by the final value of the loop variable, which is at this point equal to 10. In this case I could just have divided by 10 but in a more general case I would not know how many numbers I was averaging until run time so I often use this trick.

8.1.1 Translating Codes with Arrays

One common use for arrays in embedded systems is in converting one kind of code to another. This is most useful when the codes are successive integers with few, if any, gaps. One particularly common example is printing values in hex. Each different 4-bit binary number corresponds to a single hex digit, a code of successive integers with no gaps. However, the ASCII codes for the digits are not so simply placed. The decimal digits are easy; their codes follow one another with no gaps. However there is a large gap in the ASCII table between '9' (0x39) and 'A' (0x41). A simple table allows us to perform the translation in a single C expression.

We start by creating an array of the ASCII codes for the hex digits in order

```
char HexTable[16] = {`0','1','2','3','4','5','6','7',
                     '8','9','A','B','C','D','E','F'};
```

Then we can translate a number in the range 0-15 into its corresponding ASCII hex representation with a single array reference. `HexTable[3]` returns the ASCII code for '3', `HexTable[13]` returns the ASCII code for 'D' as it should. Thus, we can write a routine to write a single byte variable out to the serial port in hex like this

```
void PrintHexByte(char ch) {
    char HexTable[16] = {`0','1','2','3','4','5','6','7',
                         '8','9','A','B','C','D','E','F'};
    //
    // Extract and write the high order hex digit
    //
    Serial.write(HexTable[ch >> 4]); // >> shifts bits right
    //
    // Extract and write out the low order hex digit.
    //
    Serial.write(HexTable[ch & 0x0f]); // Mask off top nibble
}
```

Note the use of the right shift operator >>. This takes the bits of its left argument and shifts them to the right the number of times given in its right argument. It shifts zeros in to replace them so the `ch >> 4` evaluates to an 8-bit number with the top 4-bits of ch in its bottom nibble and zero in its top nibble. It is exactly the binary representation of the top 4 bits of ch. If we started out with `ch = 0x59 = 0x01011001` then `ch >> 4` would

There are two shift operators in C, << and >>, which perform left and right shifts respectively. Each takes two integer aguments and shifts its left argument the number of bit positions in its right argument.

produce

```
0x05 = 0b00000101.
```

Similarly, ch & 0x0f masks off the top four bits of ch leaving only the bottom four bits, the second hex digit. In our example

```
ch & 0x0f = 0x09 = 0b00001001.
```

8.1.2 Strings

Probably the most common use of an array variable in our programs will be to represent a string. A string is simply a list of characters stored one after another in adjacent storage cells so it is natural to represent a string in C as an array of characters. We already have string constants to represent fixed strings but they are just that, fixed. When we need to modify a string we need to store it in an array variable. For example, this declaration will create a variable with 20 slots for characters

```
char myString[20];
```

The trouble with this is that once we have a string in the variable we need a way to mark how many of the character slots we have used since all of the 20 bytes of memory contain something whether we put it there or not. The C language does not have anything to say about this issue but tradition (enshrined in the standard C libraries) says that we mark the end of the string by putting a 0x00 byte after the last byte of the string. Thus our 20-byte variable can only hold strings of up to 19 characters and still leave room for the terminating zero.

Because strings are represented internally as array of small numbers, using the ASCII code, and are terminated by a Zero, we sometimes refer to them as ASCIIZ strings.

Example 8.1.2

The piece of code counts the number of characters in the string named **myString**. We use a while loop to search through the array for the 0 that terminates the string while we keep track of how many characters we have seen.

```
    unsigned char index = 0;

    while (myString[index] != 0) {

        index = index + 1;}

    }
// At this point index holds the number of chars
```

Since the answer, the number of characters in **myString**, is the value of the variable **index** when the program stops, another possible choice of name would have been **numChars**. The name **index** stresses the ongoing use of the variable while the name **numChars** stresses the final use.

It is important to realize that C does not provide any operations that work on complete arrays. For example, if array1 and array2 are the names of two C arrays we cannot copy the contents of one to the other by saying

```
array1 = array2;      // This does NOT work
```

instead we must copy the elements one-by-one by ourselves. We would usually use a loop to for this, resulting in code like

```
for (i = 0; i < length; ++i) {
    array1[i] = array2[i];
}
```

We shall see a number of short procedures for playing with strings in the next chapter.

8.2 Local, Global, and Static Variables

The variables that we have encountered so far are known technically as local variables. Such variables are declared at the start of a procedure or block and continue to exist and be visible until the end of the block. Once the block ends, the storage associated with the variable is re-used and the old variable no longer exists. Most of the time this is entirely adequate but just occasionally we want something with a longer life. C provides several alternatives.

8.2.1 Global Variables

A global variable is declared outside any procedure. It exists throughout the lifetime of the entire program and it is visible to all procedures in the same file. This provides a mechanism for different procedures to share information between them.

It is possible to split a C program over several files. This is a common practice in the desktop world, where programs tend to be much larger. C provides the extern keyword to let procedures in one file see global variables declared in another. Our programs are usually small enough to fit naturally in one file so we won't worry about this.

Global variables must be declared just like any other variable but the declaration comes outside any procedure, usually just after the include directives at the top of the file. They can be scalars or arrays and they can be initialized with constants in the usual way.

Used carefully, globals provide a means for different parts of a program, different procedures, to communicate indirectly. Since a global is visible to all the procedures in a single file, all the procedures can read and modify it. This is both a good and a bad thing. The path of communication is much less explicit than the normal path of argument and result between a function and its caller. That makes the code harder to read but it can also make the code more flexible.

Example 8.2.1

Let us consider the problem of writing navigation code for a small wheeled robot. The robot has to keep sending commands to its wheels to keep the wheels moving. Let us assume that there is a Move subroutine that takes care of this. Then there are other subroutines that receive information from the external world. These could include subroutines to check whether the robot has run into anything and routines to monitor external variables such as light intensity or temperature. Finally, there must be a main routine that controls the whole system, analyzing the external information and deciding on the next course of action. Several systems may want to know which way the robot is facing or what the wheels are currently doing and the best way to make that information generally available is through global variables.

Overall, the robot program might have a structure something like this.

```
/* Global variables. */
int gLeftSpeed = 0;   // Current speed of left wheel
```

I very carefully gave all the global variables names that began with a letter g. This is not a part of the C language but is a common practice that helps make clear that these are special shared variables.

```
int gRightSpeed = 0; // Current speed of right wheel
int gDirection = -1;    // Compass direction, 0=north
int gLeftLight = -1;    // Light level from left sensor
int gRightLight = -1;  // Light from right.
//
void setup(void) \{
    // initialize hardware and software
    ....
}
//
void loop(void) \{
    /* Figure out current state of world */
    leftbump = CheckLeftBumper();
    rightBump = CheckRightBumper();
    CheckIllumination();
    /* Decide what to do next and do it */
    PlanMotion(leftBump, rightBump);
    Move();
}
```

Each subroutine only needs to know how to do one limited job and yet they can share information without passing around tons of parameters and results. In this example I mixed parameters and globals to illustrate that both can play roles in the same program.

8.2.2 Static Variables

Local Variables
Local variables are created in a special region of memory called the Stack. It operates like a stack of plates in that you can only put plates on the top of the stack and can only take them off the top.
When a subroutine is called, the system creates a new 'plate' of its local variables and puts it on the top of the stack. As soon as the subroutine ends the 'plate' is taken away and the local variables cease to exist.
Global and static variables are stored in a different area of memory that never goes away so they persist through the life of the program.

Sometimes you want a variable that is visible only within a single procedure, like a local variable, but that persists from one invocation of the procedure to the next. In that case you can declare a variable to be static. This is a keyword that can be put in front of any valid variable declaration to tell the compiler that this variable is persistent. The variable will be put into the same region of RAM as the global variables but will be visible only to code within that one procedure. You could, for example, use a static variable to count how many times a subroutine has been called, or how many times it has done some special action. An ordinary local variable won't do this since it is created anew each time the procedure is invoked and then destroyed as soon as the procedure ends.

Example 8.2.2

The CheckLeftBumper procedure in the previous example could keep track of how many times the robot has run into something using a static variable, like this:

```
int CheckLeftBumper() {
    static sBumpCount;
    if ( <code to check the bumper> ) {
        sBumpCout = sBumpCount + 1;
        return 1;     // Report a hit
    }
    return 0;         // Report no hit
}
```

Note again the use of a special kind of name for a special variable.