APPENDIX A

Computer Arithmetic

In the chapters of this book, I've deliberately kept discussion of arithmetic to a minimum. However, it's important, so I'm going to quickly go over the subject in this appendix. If you feel confident in your math skills, this review will be old hat to you. If you find the math parts tough, this section should show you how easy it really is.

Binary Numbers

First, let's consider exactly what you intend when you write a common, everyday decimal number such as 324 or 911. Obviously what you mean is *three hundred and twenty-four* or *nine hundred and eleven*. Put more precisely, you mean the following:

324 is $3 \times 10^2 + 2 \times 10^1 + 4 \times 10^0$, which is $3 \times 10 \times 10 + 2 \times 10 + 4$

911 is $9 \times 10^2 + 1 \times 10^1 + 1 \times 10^0$, which is $9 \times 10 \times 10 + 1 \times 10 + 1$

We call this **decimal notation** because it's built around powers of 10. (This is derived from the Latin **decimalis**, meaning "of tithes," which was a tax of 10 percent. Ah, those were the days . . .)

Representing numbers in this way is very handy for people with 10 fingers and/or 10 toes, or indeed 10 of any kind of appendage. However, your PC is rather less handy, being built mainly of switches that are either on or off. It's OK for counting up to 2, but not spectacular at counting to 10. I'm sure you're aware that this is the primary reason why your computer represents numbers using base 2 rather than base 10. Representing numbers using base 2 is called the **binary** system of counting. With numbers expressed using base 10, digits can be from 0 to 9 inclusive, whereas with binary numbers digits can only be 0 or 1, which is ideal when you only have on/off switches to represent them. In an exact analogy to the system of counting in base 10, the binary number 1101, for example, breaks down like this:

$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$

which is

$1 \times 2 \times 2 \times 2 + 1 \times 2 \times 2 + 0 \times 2 + 1$

This amounts to 13 in the decimal system. In Table A-1 you can see the decimal equivalents of all the possible numbers you can represent using eight binary digits (a binary digit is more commonly known as a **bit**).

Notice that using the first seven bits you can represent numbers from 0 to 127, which is a total of 2^7 numbers, and that using all eight bits you get 256 or 2^8 numbers. In general, if you have n bits, you can represent 2^n integers, with values from 0 to $2^n - 1$.

Table A-1. *Decimal Equivalents of 8-Bit Binary Values*

Binary	Decimal	Binary	Decimal
0000 0000	0	1000 0000	128
0000 0001	1	1000 0001	129
0000 0010	2	1000 0010	130
. . .	. . .	. . .	. . .
0001 0000	16	1001 0000	144
0001 0001	17	1001 0001	145
. . .	. . .	. . .	. . .
0111 1100	124	1111 1100	252
0111 1101	125	1111 1101	253
0111 1110	126	1111 1110	254
0111 1111	127	1111 1111	255

Adding binary numbers inside your computer is a piece of cake, because the "carry" from adding corresponding digits can only be 0 or 1, and very simple circuitry can handle the process. Figure A-1 shows how the addition of two 8-bit binary values would work.

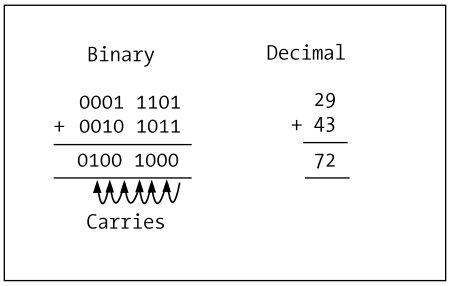

Figure A-1. *Adding binary values*

Hexadecimal Numbers

When you start dealing with larger binary numbers, a small problem arises. Look at this one:

1111 0101 1011 1001 1110 0001

Binary notation here starts to be more than a little cumbersome for practical use, particularly when you consider that if you work out what this is in decimal, it's only 16,103,905—a miserable eight decimal digits. You can sit more angels on the head of a pin than that. Clearly we need a more economical way of writing this, but decimal isn't always appropriate. Sometimes (as you saw in Chapter 3) you might need to be able to specify that the 10th and 24th bits from the right are set to 1, but without the overhead of writing out all the bits in binary notation. To figure out the decimal integer required to do this sort of thing is hard work, and there's a good chance you'll get it wrong

anyway. A much easier solution is to use hexadecimal notation in which the numbers are represented using base 16.

Arithmetic to base 16 is a much more convenient option, and it fits rather well with binary. Each hexadecimal digit can have values from 0 to 15 (the digits from 10 to 15 being represented by letters A to F, as shown in Table A-2), and values from 0 to 15 correspond nicely with the range of values that four binary digits can represent.

Table A-2. *Hexadecimal Digits and Their Values in Decimal and Binary*

Hexadecimal	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Because a hexadecimal digit corresponds to four binary digits, you can represent a large binary number as a hexadecimal number simply by taking groups of four binary digits starting from the right and writing the equivalent hexadecimal digit for each group. Look at the following binary number:

1111 0101 1011 1001 1110 0001

Taking each group of four bits in turn and replacing it with the corresponding hexadecimal digit from the table, this number expressed in hexadecimal notation will come out as follows:

F 5 B 9 E 1

You have six hexadecimal digits corresponding to the six groups of four binary digits. Just to prove that it all works out with no cheating, you can convert this number directly from hexadecimal to decimal by again using the analogy with the meaning of a decimal number. The value of this hexadecimal number therefore works out as follows:

F5B9E1 as a decimal value is given by

$15 \times 16^5 + 5 \times 16^4 + 11 \times 16^3 + 9 \times 16^2 + 14 \times 16^1 + 1 \times 16^0$

This turns out to be

15,728,640 + 327,680 + 45,056 + 2,304 + 224 + 1

Thankfully, this adds up to the same number you get when converting the equivalent binary number to a decimal value: 16,103,905.

Negative Binary Numbers

There's another aspect to binary arithmetic that you need to understand: negative numbers. So far, you've assumed that everything is positive—the optimist's view, if you will—and so the glass is still half full. But you can't avoid the negative side of life—the pessimist's perspective—that the glass is already half empty. How can a negative number be represented inside a computer? Well, you have only binary digits at your disposal, so the solution has to be to use one of those to indicate whether the number is negative or positive.

For numbers that you want to allow to have negative values (referred to as **signed numbers**), you must first decide on a fixed length (in other words, the number of binary digits) and then designate the leftmost binary digit as a **sign bit**. You have to fix the length to avoid any confusion about which bit is the sign bit.

Because your computer's memory consists of 8-bit bytes, the binary numbers are going to be stored in some multiple (usually a power of 2) of 8 bits. Thus, you can have some numbers with 8 bits, some with 16 bits, and some with 32 bits (or whatever), and as long as you know what the length is in each case, you can find the sign bit—it's just the leftmost bit. If the sign bit is 0, the number is positive, and if it's 1, the number is negative.

This seems to solve the problem, and in some computers it does. Each number consists of a sign bit that is 0 for positive values and 1 for negative values, plus a given number of bits that specify the absolute value of the number—unsigned, in other words. Changing +6 to –6 just involves flipping the sign bit from 0 to 1. Unfortunately this representation carries a lot of overhead with it in terms of the complexity of the circuits that are needed to perform arithmetic with this number representation. For this reason, most computers take a different approach.

Ideally when two integers are added, you don't want the computer to be messing about, checking whether either or both of the numbers are negative. You just want to use simple **add** circuitry, regardless of the signs of the operands. The add operation will combine corresponding binary digits to produce the appropriate bit as a result, with a carry to the next digit where necessary. If you add –8 in binary to +12, you would really like to get the answer +4, using the same circuitry that would apply if you were adding +3 and +8.

If you try this with your simplistic solution, which is just to set the sign bit of the positive value to 1 to make it negative, and then perform the arithmetic with conventional carries, it doesn't quite work:

12 in binary is 0000 1100.

–8 in binary (you suppose) is 1000 1000.

If you now add these together, you get 1001 0100.

This seems to be –20, which isn't what you wanted at all. It's definitely not +4, which you know is 0000 0100. "Ah," I hear you say, "you can't treat a sign just like another digit." But that is just what you *do* want to do.

Let's see how the computer would like you to represent –8 by trying to subtract +12 from +4, as that should give you the right answer:

+4 in binary is 0000 0100.

+12 in binary is 0000 1100.

Subtract the latter from the former and you get 1111 1000.

For each digit from the fourth from the right onward, you had to "borrow" 1 to do the subtraction, just as you would when performing ordinary decimal arithmetic. This result is supposed to be –8, and even though it doesn't look like it, that's exactly what it is. Just try adding it to +12 or +15 in binary, and you'll see that it works.

Of course, if you want to produce –8 you can always do so by subtracting +8 from 0.

What *exactly* did you get when you subtracted 12 from 4 or +8 from 0, for that matter? It turns out that what you have here is called the **2's complement** representation of a negative binary number, and you can produce this from any positive binary number by a simple procedure that you can perform in your head. At this point, I need to ask a little faith on your part and avoid getting into explanations of *why* it works. I'll just show you how the 2's complement form of a negative number can be constructed from a positive value, and you can prove to yourself that it does work. Let's return to the previous example, in which you need the 2's complement representation of –8.

You start with +8 in binary:

0000 1000

You now "flip" each binary digit, changing 0s to 1s and vice versa:

1111 0111

This is called the **1's complement** form, and if you now add 1 to this, you'll get the 2's complement form:

1111 1000

This is exactly the same as the representation of –8 you get by subtracting +12 from +4. Just to make absolutely sure, let's try the original sum of adding –8 to +12:

+12 in binary is 0000 1100.

Your version of –8 is 1111 1000.

If you add *these* together, you get 0000 0100.

The answer is 4—magic. It works! The "carry" propagates through all the leftmost 1s, setting them back to 0. One fell off the end, but you shouldn't worry about that—it's probably compensating for the one you borrowed from the end in the subtraction operation that produced the –8 in the first place. In fact, what's happening is that you're making an assumption that the sign bit, 1 or 0, repeats forever to the left. Try a few examples of your own; you'll find it always works automatically. The really great thing about using 2's complement representation of negative numbers is that it makes arithmetic very easy (and fast) for your computer.

Big-Endian and Little-Endian Systems

As I've discussed, integers generally are stored in memory as binary values in a contiguous sequence of bytes, commonly groups of 2, 4, or 8 bytes. The question of the sequence in which the bytes appear can be very important—it's one of those things that doesn't matter until it matters, and then it *really* matters.

Let's consider the decimal value 262,657 stored as a 4-byte binary value. I chose this value because in binary it happens to be

0000 0000 0000 0100 0000 0010 0000 0001

Each byte has a pattern of bits that is easily distinguished from the others.

If you're using an Intel PC, the number will be stored as follows:

Byte address:	00	01	02	03
Data bits:	0000 0001	0000 0010	0000 0100	0000 0000

As you can see, the most significant eight bits of the value—the one that's all 0s—are stored in the byte with the highest address (last, in other words), and the least significant eight bits are stored in the byte with the lowest address, which is the leftmost byte. This arrangement is described as **little-endian**.

If you're using a mainframe computer, a RISC workstation, or a Mac machine based on a Motorola processor, the same data is likely to be arranged in memory as follows:

Byte address:	00	01	02	03
Data bits:	0000 0000	0000 0100	0000 0010	0000 0001

Now the bytes are in reverse sequence with the most significant eight bits stored in the leftmost byte, which is the one with the lowest address. This arrangement is described as **big-endian**.

■ **Note** Within each byte, the bits are arranged with the most significant bit on the left and the least significant bit on the right, regardless of whether the byte order is big-endian or little-endian.

This is all very interesting, you may say, but why should it matter? Most of the time it doesn't. More often than not you can happily write your C program without knowing whether the computer on which the code will execute is big-endian or little-endian. It *does* matter, however, when you're processing binary data that comes from another machine. Binary data will be written to a file or transmitted over a network as a sequence of bytes. It's up to you how you interpret it. If the source of the data is a machine with a different endian-ness from the machine on which your code is running, you must reverse the order of the bytes in each binary value. If you don't, you have garbage.

■ **Note** For those who collect curious background information, the terms **big-endian** and **little-endian** are drawn from the book *Gulliver's Travels* by Jonathan Swift. In the story, the emperor of Lilliput commands all his subjects to always crack their eggs at the smaller end. This is a consequence of the emperor's son having cut his finger following the traditional approach of cracking his egg at the big end. Ordinary law-abiding Lilliputian subjects who cracked their eggs at the smaller end were described as Little Endians. The Big Endians were a rebellious group of traditionalists in the Lilliputian kingdom who insisted on continuing to crack their eggs at the big end. Many were put to death as a result.

Floating-Point Numbers

We often have to deal with very large numbers—the number of protons in the universe, for example—which need around 79 decimal digits. Clearly there are a lot of situations in which you'll need more than the 10 decimal digits you get from a 4-byte binary number. Equally, there are a lot of very small numbers—for example, the amount of time in minutes it takes the typical car salesperson to accept your generous offer on a 1982 Ford LTD (and it's covered only 380,000 miles . . .). A mechanism for handling both these kinds of numbers is, as you may have guessed, **floating-point** numbers.

A floating-point representation of a number in decimal notation is a decimal value 0 with a fixed number of digits multiplied by a power of 10 to produce the number you want. It's easier to demonstrate than to explain, so let's look at some examples. The number 365 in normal decimal notation could be written in floating-point form as follows:

0.3650000E03

The *E* stands for **exponent** and precedes the power of 10 that the 0.3650000 (the mantissa) part is multiplied by to get the required value. That is

$0.3650000 \times 10 \times 10 \times 10$

which is clearly 365.

The mantissa in the number here has seven decimal digits. The number of digits of precision in a floating-point number will depend on how much memory it is allocated. A single precision floating-point value occupying 4 bytes will provide approximately seven decimal digits accuracy. I say *approximately* because inside your computer these numbers are in binary floating-point form, and a binary fraction with 23 bits doesn't exactly correspond to a decimal fraction with seven decimal digits.

Now let's look at a small number:

0.3650000E-04

This is evaluated as $.365 \times 10^{-4}$, which is .0000365—exactly the time in minutes required by the car salesperson to accept your cash.

Suppose you have a large number such as 2,134,311,179. How does this look as a floating-point number? Well, it looks like this:

0.2134311E10

It's not quite the same. You've lost three low-order digits and you've approximated your original value as 2,134,311,000. This is a small price to pay for being able to handle such a vast range of numbers, typically from 10^{-38} to 10^{+38} either positive or negative, as well as having an extended representation that goes from a minute 10^{-308} to a mighty 10^{+308}. They're called floating-point numbers for the fairly obvious reason that the decimal point "floats" and its position depends on the exponent value.

Aside from the fixed precision limitation in terms of accuracy, there's another aspect you may need to be conscious of. You need to take great care when adding or subtracting numbers of significantly different magnitudes. A simple example will demonstrate this kind of problem. You can first consider adding .365E–3 to .365E+7. You can write this as a decimal sum:

0.000365 + 3,650,000.0

This produces the following result:

3,650,000.000365

When converted back to floating-point with seven digits of precision, this becomes the following:

0.3650000E+7

So you might as well not have bothered. The problem lies directly with the fact that you carry only seven digits precision. The seven digits of the larger number aren't affected by any of the digits of the smaller number because they're all further to the right. Funnily enough, you must also take care when the numbers are nearly equal. If you compute the difference between such numbers, you may end up with a result that has only one or two digits precision. It's quite easy in such circumstances to end up computing with numbers that are totally garbage.