APPENDIX D

NUMBER SYSTEMS

You will learn about the following in this appendix:

- The four important number systems in computing—binary, octal, decimal, and hexadecimal.
- Converting a number from one number system to another.
- How to abbreviate binary numbers using the octal and hexadecimal number systems, and why this is useful.
- A number system converter program written in C# and based on recursion.
- The concepts two's complement and one's complement to represent negative values by the binary number system.

Introduction

Chapter 1, "Computers and Computer Programming: Basic Concepts." introduced you to the nature of computer hardware and its close connection to the binary number system. It also hinted at the usefulness of other number systems to help programmers overcome the awkwardness of the binary number system. This appendix gives a thorough introduction to each of the number systems and will show you how to utilize the octal and hexadecimal number systems to cut through the cumbersome binary number system.

Binary Number System

The binary number system, also referred to as *base 2*, makes use of only two digits—1 and 0. "Bi" in binary is analogous to bi in bicycle (two wheels). Each digit of the binary system is called a *bit* originating from **bi**nary digi**t.**

If we count from zero to eleven in the decimal system, as shown in Table D.1, we can observe how the binary number system compares with the decimal number system.

TABLE D.1 Counting with the Binary and Decimal Number Systems

Binary	Decimal
0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8
1001	9
1010	10
1011	11

When we reach 1 in the binary system, we run out of digits and are forced to increase the next number to 10. Similarly, the number after 11 is 100. So 10 is greater than 1, 100 is greater than 11, and 1000 is greater than 111. 1 in the third place from the right in 100 is worth more than the two 1s in 11.

In the decimal system, we run out of digits at 9 and, accordingly, the next number is 10. Similarly, the number after 99 is 100, and 1000 comes after 999. Again, 1 in the fourth position from the right in 1000 is worth more than 999.

Here we are using *positional notation* in which digits written in different positions of the number have a different *positional value*. As we move to the left, the positional value increases. Most conventional number systems apply the positional notation, the systems discussed in this appendix in particular.

Then what exactly is the value ascribed to each position of a number in the binary system? To answer this question, it is useful first to have a look at the positional values of the decimal system displayed in Table D.2.

TABLE D.2 Positional Values in the Binary and Decimal Systems

Position	8	7	6	5	4	3	2	1
Binary								
Positional Value	128	64	32	16	8	4	2	1

TABLE D.2 continued

Position	8	7	6	5	4	3	2	1
Decimal								
Positional Value	10000000	1000000	100000	10000	1000	100	10	1

Position in Table D.2 refers to the position of each digit in an arbitrary number. The rightmost digit is in position 1, and, as we move to the left, we increase the position by one for each new digit we meet. For example, in the decimal number 768594, we have the following positions:

Digit	7	6	8	5	9	4
Position	6	5	4	3	2	1

We can establish a pattern for the positional value of a decimal number by looking at Table D.2. In general, the value of each position in the decimal system can be viewed as

$$\text{Positional value of position x in decimal system} = 10^{x-1}$$

Consequently the positional value of position 3 is $10^{3-1} = 100$ as shown in Table D.2.

We can now understand exactly what we mean when we write an arbitrary decimal number, such as 7684:

$$7684 = 7x10^3 + 6x10^2 + 8x10^1 + 4x10^0$$

The decimal system is built around powers of 10. This explains its name; decimal is derived from the Latin word decimalis meaning of tithes—a 10% tax.

Each position has a positional name and is derived from the corresponding positional value. In the last decimal number, we can say that 4 is written in the *ones position* ($10^0 = 1$), 8 is written in the *tens position* ($10^1 = 10$), 6 is written in the *hundreds position* ($10^2 = 100$), and 7 is written in the *thousands position* ($10^3 = 1000$).

Of course, this way of looking at decimal numbers is merely our unconscious thoughts made explicit. We have been reasoning like this since we were small kids. However, this emphasis on the obvious paves the way to understanding how values are calculated using the base 2, base 8, and base 16 number systems.

We can now utilize the logic applied to the positional values of the decimal number system to work out the positional values of the binary number system. By looking at the values in Table D.2, we discover the following pattern for determining base 2 positional values:

$$\text{Positional value of position x in binary number system: } 2^{x-1}$$

Converting from the binary to the decimal number system can then be performed as illustrated by converting the binary number 100110 to base 10.

$$1x2^5 + 0x2^4 + 0x2^3 + 1x2^2 + 1x2^1 + 0x2^0 = 38$$

Like the decimal number system, each position in the binary number system has a name, shown in Table D.3. The names are found simply by applying the 2^{x-1} rule and writing the result with text. For example $2^{6-1} = 32$ makes thirty twos.

TABLE D.3 Positional Names for Digits in 100110

Digit	1	0	0	1	1	0
Position Name	Thirty-twos	Sixteens	Eights	Fours	Twos	Ones

Bits and Bytes

The byte concept is often used as a unit for memory size. One *byte* refers to a unit of usually eight adjacently positioned bits in computer memory. A kilobyte is equal to 1,024 bytes, and a megabyte is equal to 1,024 kilobytes.

Octal Number System

Table D.4 gives an overview of the digits used in the four number systems discussed in this appendix. Octal numbers use the digits 0–7. Table D.5 compares the first 15 decimal numbers to their octal counterparts.

TABLE D.4 Digits of the Four Number Systems

Binary Digits	Octal Digits	Decimal Digits	Hexadecimal Digits
0	0	0	0
1	1	1	1
	2	2	2
	3	3	3
	4	4	4
	5	5	5
	6	6	6
	7	7	7
		8	8
		9	9

TABLE D.4 continued

Binary Digits	Octal Digits	Decimal Digits	Hexadecimal Digits
			A
			B
			C
			D
			E
			F

TABLE D.5 Counting with the Octal and Decimal Number Systems

Octal	Decimal
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
10	8
11	9
12	10
13	11
14	12
15	13
16	14
17	15

The octal numbers are based on powers of 8, just as the binary system is based on powers of 2. Table D.6 shows the relationship between the positional values of base 8 and base 10 systems.

It is easy to confuse an octal number with a decimal number. For example, nothing in 126 hints as to whether this number should be interpreted as belonging to base 10 or base 8. Consequently, we will use a common notation form to indicate an octal number—a 0 (zero) prefix. Thus, 06542 is an octal number.

Note

It is not possible to specify a literal in C# to be a base 8 value by using the 0 prefix or any other notation.

When converting a number of base 8 to a number of base 10, we apply the same logic as with the previous binary and decimal numbers. This is illustrated with 06542 as follows:

$$06542 = 6x8^3 + 5x8^2 + 4x8^1 + 2x8^0 = 6x512 + 5x64 + 4x8 + 2x1 = 3426$$

TABLE D.6 Positional Values in the Octal and Decimal Systems

Position	Octal Positional Value	Decimal Positional Value
8	2097152	10000000
7	262144	1000000
6	32765	100000
5	4096	10000
4	512	1000
3	64	100
2	8	10
1	1	1

Hexadecimal Number System

Hexadecimal numbers use the digits 0–9 and the letters A–F (you can also use lowercase a–f), as shown in Table D.4. The first six letters of the alphabet were arbitrarily chosen but are preferred because they are easy to remember and easy to find on a keyboard.

To avoid confusion, hexadecimal numbers have a 0x or 0X prefix, for the same reason that we use the prefix 0 with octal numbers. Thus, 0x4B2 is a base 16 number.

Note

You can specify a number to be hexadecimal in C# by using the 0x (or 0X) prefix, as shown in the following:

```
int distance;
distance = 0x15;
Console.WriteLine("The distance is: " + distance);
```

This prints out

```
The distance is: 21
```

to the console, because 0×15 is equal to 21 in the decimal system.

Following the logic of bases 2, 8, and 10, we don't experience any surprises when looking at the positional values of base 16 numbers in Table D.8. Using 0x4B2 as an example, we say that 2 is written in the *ones position,* B is written in the *sixteens position,* and 4 is written in the *twohundredandfiftysixths position*.

Hexadecimal numbers are based on powers of 16, shown in Table D.8, which then provides us with enough information to calculate the base 10 value of 0x4B2.

$0x4B2 = 4 \times 16^2 + 11 \times 16^1 + 2 \times 16^0 = 1202$

Notice how 11 is substituted for B in the formula. In Table D.7, you can find the values for the other letters A–F.

TABLE D.7 Counting with the Hexadecimal and Decimal Number Systems

Hexadecimal	Decimal
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
A	10

TABLE D.7 continued

Hexadecimal	Decimal
B	11
C	12
D	13
E	14
F	15
10	16
11	17
12	18

TABLE D.8 Positional Values in the Hexadecimal and Decimal Systems

Position	8	7	6	5	4	3	2	1
Hexadecimal Positional Value	268435456	16777216	1048576	65635	4096	256	16	1
Decimal Positional Value	10000000	1000000	100000	10000	1000	100	10	1

The Practical Use of Octal Numbers and Hexadecimal Numbers

Octal and hexadecimal numbers make it more convenient to work with the long cumbersome binary numbers. In fact, we are able to abbreviate binary numbers by utilizing base 8 and base 16 numbers, as you will see shortly.

To understand how, we need to look at equivalent values of bases 2, 8, and 16 as shown in Table D.9. One of the first things to notice is that each lengthy binary number can be expressed concisely in either the octal or the hexadecimal number systems. For example 1111 is equivalent to 17 in base 8 and simply F in base 16. Even though each base 2 number also has a shorter equivalent in base 10, this latter number system is not suitable for abbreviating binary numbers.

TABLE D.9 Base 2, Base 8, Base 16, and Base 10 Equivalents

Binary Number System	Octal Number System	Hexadecimal Number System	Decimal Number System
0	0	0	0
1	1	1	1
10	2	2	2
11	3	3	3
100	4	4	4
101	5	5	5
110	6	6	6
111	7	7	7
1000	10	8	8
1001	11	9	9
1010	12	A	10
1011	13	B	11
1100	14	C	12
1101	15	D	13
1110	16	E	14
1111	17	F	15
10000	20	10	16

The underlying reason why base 8 and base 16 can be used to abbreviate base 2 numbers can be found in the fact that 8 (being the base of base 8 numbers) and 16 (being the base of base 16 numbers) are powers of the base of the binary system (being 2). This is illustrated in Figure D.1.

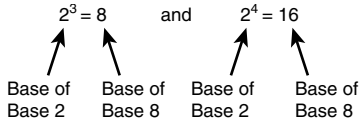


FIGURE D.1 Relationships between bases 2, 8, and 16.

This same relationship does not hold for base 10, which renders it useless for abbreviating binary numbers.

Consider the binary number 100110010100; its equivalent base 16 number is 0x994.

To see how we can easily convert 100110010100 to 0x994, we need to separate the binary number into three parts with each containing four consecutive digits. Underneath each part we write the corresponding base 16 number, which can be found in Table D.9.

Base 2	1001	1001	0100
Base 16	9	9	4

This works because the base of base 16 is a power of the base of the binary system, as was shown previously, enabling us to write:

```
100100000000  =  900    +
    10010000  =   90    +
        0100  =    4    +
=============================
100110010100      994
```

Applying the same logic, we can use the octal number system to abbreviate our binary number 100110010100 from the previous example. This time, we need to break the binary number into four parts each with three consecutive digits:

Base 2	100	110	010	100
Base 8	4	6	2	4

Thus, 100110010100 converts to 04624

Note

"Old" computers often use 3 bits to store a number; this makes the octal abbreviation method, which uses 3 bits for each octal digit, very useful. However, because older computers are becoming a rarity, the octal number system is not used as often today.

The hexadecimal system is the preferred number system today. Accordingly, only hexadecimal values can be defined in C#, not values of base 8.

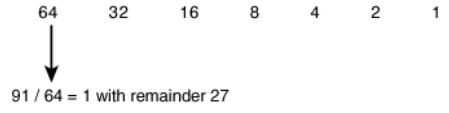
It is now a simple matter to reverse the process and convert base 8 and base 16 numbers to base 2 numbers. For example, the octal number 04624 is converted to base 2 by writing the 4 as its binary equivalent 100, 6 converts to 110, 2 to 010, and 4 to 100. Combining them gives us 100110010100. Exactly the same procedure is used when converting from base 16 numbers to base 2 numbers.

Converting from Base 10 to Bases 2, 8, and 16

We have already seen how we can convert from bases 2, 8, and 16 to base 10. Those conversions followed naturally from the positional values. Albeit not as straightforward, the process of converting from base 10 to bases 2, 8 and 16 is based on the same positional values. The easiest way to understand the process is by looking at an example. Following are the steps you need to take when converting 91 of base 10 to base 2.

1. Starting at the right side of the paper and moving to the left, write the positional values of the base 2 number system beginning at position 1. Stop when we reach a position with a value greater than the base 10 number 91. In this case, we stop at 128.

2. Remove the column with the value greater than our base 10 number. This leaves us



   Take the number in the leftmost column and divide it into the base 10 number. Here, we divide 64 into 91. The result is 1 with 27 as a remainder. We then write 1 under the 64 column (see Table D.10).

3. Find the rightmost column that is greater than 27; this is 32. Discard 32 and write a 0 under 32 (see Table D.10). Use the number on the right side of 32 to divide into our remainder 27. 27/16 yields 1 with a remainder of 11. Now write 1 under 16. We repeat the same process over and over to obtain the binary number 1011011.

TABLE D.10 Converting from Base 2 to Base 10 Using Positional Values

Positional values	64	32	16	8	4	2	1
Binary digits	1	0	1	1	0	1	1

Converting from base 10 to base 8 and base 16 follows the same pattern. Just exchange the positional values of base 2 used in Table D.10 to the positional values corresponding to either base 8 or base 16.

A Recursive C# Number System Converter Program

Even though the process of converting numbers of base 10 to bases 2, 8 and 16 is straightforward, it is somewhat slow and cumbersome. To speed up the process, I have provided a C# program, displayed in Listing D.1, that uses recursion as the fundamental mechanism to perform the conversions. If you haven't yet read the chapter on recursion, don't despair. You can just copy in the program and use it as is without trying to understand it.

Notice that the program further utilizes exception handling discussed in Chapter 19 to manage invalid user input.

Pause for a moment and look at lines 49–57 of Listing D.1. These are, in fact, the few essential lines of source code needed to perform our conversions. One can then only marvel at the elegance and power of recursion.

The program is straightforward to use. It allows you to convert numbers of any base 2–16 to numbers of base 10, but it cannot handle negative numbers.

LISTING D.1 Source Code of `NumberSystemConverter.cs`

```
01: using System;
02:
03:  // Class enabling conversions from base 10 to any number system
04:  // with base between 2 and 16.
05: public class NumberSystemConverter
06: {
07:     const string digits = "0123456789abcdef";
08:
09:     public static void Main()
10:     {
11:         int decimalNumber;
12:         int baseNumber;
13:         string answer = "Y";
14:
15:         do
16:         {
17:             try
18:             {
19:                 Console.Write("Enter decimal number: ");
20:                 decimalNumber = Convert.ToInt32(Console.ReadLine());
21:                 Console.Write("Enter base: ");
22:                 baseNumber = Convert.ToInt32(Console.ReadLine());
23:                 if (baseNumber < 2 || baseNumber > 16)
24:                 {
25:                  throw (new InvalidBaseNumberException("Invalid base number"));
26:                 }
27:                 Console.Write("Result: ");
28:                 printBase(decimalNumber, baseNumber);
29:                 Console.WriteLine();
30:                 Console.Write("Another conversion? y(es) n(o) ");
31:                 answer = Console.ReadLine().ToUpper();
32:                 Console.WriteLine();
33:             }
34:
35:             catch (InvalidBaseNumberException ex)
36:             {
37:                 Console.WriteLine("InvalidBaseNumberException: {0}", ex);
38:             }
39:
40:             catch (Exception ex)
```

LISTING D.1 continued

```
41:             {
42:                 Console.WriteLine("Exception: {0}", ex);
43:             }
44:
45:         } while (answer == "Y");
46:         Console.WriteLine("Have a good day. Bye Bye!");
47:     }
48:
49:     public static void printBase(int n, int nBase)
50:     {
51:          // Recursive method to print n in any base
52:          // Assumes 2 <= nBase <= 16
53:         if (n >= nBase)
54:              // Method is calling itself, making it a recursive method.
55:             printBase(n / nBase, nBase);
56:         Console.Write(digits[n % nBase]);
57:     }
58: }
59:
60:
61: public class InvalidBaseNumberException : Exception
62: {
63:     public InvalidBaseNumberException()
64:     {
65:     }
66:
67:     public InvalidBaseNumberException(string message) : base(message)
68:     {
69:     }
70:
71:     public override string ToString()
72:     {
73:         return "Base is restricted to: 2 <= base <= 16";
74:     }
75: }
```

The following is sample output from Listing D.1.

```
Enter decimal number: 20<enter>
Enter base: 2<enter>
Result: 10100
Another conversion? y(es) n(o) y<enter>

Enter decimal number: 420<enter>
Enter base: 8<enter>
Result: 644
Another conversion? y(es) n(o) y<enter>

Enter decimal number: 1202<enter>
Enter base: 16<enter>
Result: 4b2
Another conversion? y(es) n(o) n<enter>
```

The `Main` method is defined in lines 9–47. It interacts with the user and makes sure, in line 23, that a correct base between 2 and 16 is entered.

If a correct base is entered, the conversion process is initiated by calling `printBase` in line 28. `printBase` is the essential recursive method performing the actual conversion.

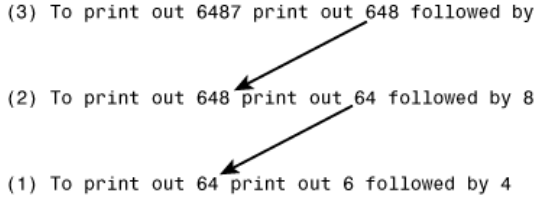
Proper exception handling using `try` and `catch` is employed to handle incorrect user input in lines 17 and 35–43. For more information about exception handling, see Chapter 19.

A user-defined exception class is defined in lines 61–75 and thrown in line 25 in the case of invalid user input.

Lines 49–57 are the essential lines of the program, performing the actual conversions through the use of recursion. You can attempt to understand these lines without having read Chapter 23, "Recursion Fundamentals," but reading this chapter certainly makes for a gentler introduction to the subject.

The path to an understanding of the `printBase()` method begins with a problem of displaying numbers on the console.

Consider the task of printing out a non-negative number `n` on the console. In this scenario, we assume that we do not have a number output method available (such as `WriteLine()`), but that we can only print out one digit at a time on the console. A first hunch for designing an algorithm to perform under these conditions might be to first isolate the leftmost digit and print it, followed by the second leftmost, and so on. For example, to print the number 6487, the algorithm needs to somehow isolate 6 and print it followed by 4, and so on. The downside to this procedure is its sloppiness. It requires a loop to isolate each digit to be printed next, from left to right. For example, to isolate 6 in 6487, the algorithm needs to move left from the rightmost digit (7) until the leftmost digit (6) is detected. It is much more efficient to acquire the rightmost digit (7) first, which can be found by simply calculating `n%10`. Recursion comes to the rescue here and allows us to find the last digit with `n%10` without reversing the order of the digits in the number. Using 6487 as an example, the process is as follows:



Each step involves exactly the same actions. The only difference is the value the actions are applied to. This means just one method with different arguments when called can solve 3, 2, and 1.

The results of lower steps are used to accomplish tasks at higher steps. For example, to accomplish 3, we must execute 2; to execute 2, we must execute 1.

Because just one method can execute all three steps, and because every step contains the result of computations performed on a lower step, it is possible to solve the problem at hand recursively.

Each step is now easily calculated. For example, in step 1, 7 can be isolated with 6487%10. To isolate 648, we simply calculate 6487/10 (.7 is discarded). This process can be expressed recursively in C# as shown in Listing D.2. Notice that the arrows above illustrate calls to the `printBaseTen()` method (see line 4).

If *n* is smaller than 10, line 5 is executed by writing only one digit (`n%10`). If *n* is larger than or equal to 10, the last digit of the number is discarded and passed on to `printBaseTen` with the recursive call in line 4. Notice that 6487/10 = 648, when passed to a variable of type `int`, not 648.7.

Remember from Chapter 23, "Recursion Fundamentals" that the recursive calls must progress toward the base case for the recursive method to be valid. The base case in the `printBaseTen` method is an *n* with just one digit (0–9), which makes (n < 10) `true` and, hence, the condition `(n >= 10)` in line 3 of `printBaseTen` `false(see Listing D.2)`, preventing the recursive calls of line 4 to go any deeper. Because every recursive call to `printBaseTen` removes one digit from *n*, we are clearly progressing toward the base case.

LISTING D.2 Recursive Method to Print `n` as a Decimal Number

```
1: public static void printBaseTen(int n)
2: {
3:     if (n >= 10)
4:         printBaseTen(n / 10);
5:     Console.Write(n%10);
6: }
```

Armed with an understanding of Listing D.2, we are ready to tackle the problem of printing in any base 2–16. The resulting code is found in line 7 and lines 49–57 of Listing D.1. Line 7 declares an array containing all the digits of the hexadecimal system. When a digit is required, it is accessed and printed in line 56. Rather than merely passing *n* to `printBase`, we also pass the base of the number to which we are converting. It is interesting to note that the same logic utilized in Listing D.2 applies when `10`, in lines 4 and 5, is substituted by *n*`Base` in lines 55 and 56 of Listing D.1.

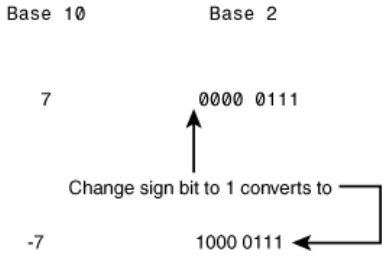
Negative Binary Numbers

So far, we have only discussed positive numbers, but the computer also somehow needs to represent and process negative numbers with its underlying binary system.

The computer uses a system whereby it allocates a specific bit, called a *sign bit*, to indicate whether a number is positive or negative. When this bit is 0, the sign is interpreted to be positive; similarly, 1 indicates a negative sign. To avoid confusion as to where this particular bit is located, the computer needs to know the exact length (the number of digits) of the number it is processing. It then designates the leftmost binary digit as the sign bit. Some numbers can contain 8 bits, others 16 bits, others 32 bits, and so on. This is fine, as long as the computer knows the length in each particular case. However, this is not the whole story. An example illustrates why.

Adding –7 (in base 2) to 15 (in base 2) should give the answer +8. Let's perform the calculations assuming the number length is 8 bits and the leftmost bit is the sign bit.

First, the value of –7 is found by changing the sign bit of 7, as illustrated next, resulting in the value 1000 0111. 15 in base 2 is 0000 1111.



Adding 1000 0111 to 0000 1111 results in the following:

```
 0000 1111
 1000 0111 +
 ==========
 1001 0110
```

The answer 1001 0110 is equivalent to –22, obviously not the correct result. Even though treating the sign just like another digit can seem somewhat shoddy, we are moving in the right direction. We need to subtract +15 from +8 to realize how we can represent –7 correctly.

```
 +8      0000 1000
+15      0000 1111  -
 ==========
-7       1111 1001
```

We apply the same method used in base-10 subtractions of borrowing. Then the claim is that –7 in base 10 is equivalent to 1111 1001 in base 2. Hmm…, it doesn't look like it, but let's do a small test. We know that 7+(–7) = 0, so 0000 0111+1111 1001 should be 0000 0000.

```
 +7      0000 0111
 -7      1111 1001 +
 ==========
  0      0000 0000
```

It's looking very promising. The carry keeps moving left until it is discarded from the leftmost bit, which gives the promising result of 0000 0000. Let's perform our previous calculation (–7)+15 to perform another check.

```
 -7      1111 1001
+15      0000 1111
 ==========
 +8      0000 1000
```

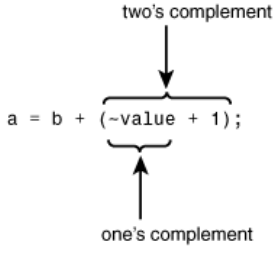
Another correct result! In fact, 1111 1001 is the correct representation for –7. It even comes with its own term—the *two's complement* representation of negative binary numbers. Earlier, we found the base 2 equivalent of –7 by calculating 8–15. Even though we got the correct result, it was fairly awkward. Fortunately, there is a much easier way to find two's complement. The following are the steps using –7 as an example:

1. Write down the positive binary number of a given decimal number—0000 0111.
2. Form *one's complement* by reversing each bit of the number. 0s become 1s and 1s become 0s. We then obtain 1111 1000.
3. Form two's complement simply by adding 1 to one's complement. The final result is the familiar 1111 1001.

C# has a built-in bitwise complement operator—~. At the machine level, `~value` is equivalent to `value` with all its bits reversed. The complement operator is very handy for the computer when performing subtractions such as the following:

```
a = b - value;
```

At the lower levels, this subtraction becomes



So, even though the computer performs a subtraction, it is done by adding the negative value of `value` to `b`.