A Discovery Company

# How Bits and Bytes Work

by Marshall Brain

Browse the article **How Bits and Bytes Work**

## Introduction to How Bits and Bytes Work

If you have used a computer for more than five minutes, then you have heard the words **bits** and **bytes**. Both RAM and hard disk capacities are measured in bytes, as are file sizes when you examine them in a file viewer.

You might hear an advertisement that says, "This computer has a **32-bit** Pentium processor with 64 **megabytes** of RAM and 2.1 **gigabytes** of hard disk space." And many HowStuffWorks articles talk about bytes (for example, How CDs Work). In this article, we will discuss bits and bytes so that you have a complete understanding.

**Decimal Numbers**

The easiest way to understand bits is to compare them to something you know: **digits**. A digit is a single place that can hold numerical values between 0 and 9. Digits are normally combined together in groups to create larger numbers. For example, 6,357 has four digits. It is understood that in the number 6,357, the 7 is filling the "1s place," while the 5 is filling the 10s place, the 3 is filling the 100s place and the 6 is filling the 1,000s place. So you could express things this way if you wanted to be explicit:

(6 * 1000) + (3 * 100) + (5 * 10) + (7 * 1) = 6000 + 300 + 50 + 7 = 6357

Another way to express it would be to use **powers of 10**. Assuming that we are going to represent the concept of "raised to the power of" with the "^" symbol (so "10 squared" is written as "10^2"), another way to express it is like this:

(6 * 10^3) + (3 * 10^2) + (5 * 10^1) + (7 * 10^0) = 6000 + 300 + 50 + 7 = 6357

What you can see from this expression is that each digit is a **placeholder** for the next higher power of 10, starting in the first digit with 10 raised to the power of zero.

That should all feel pretty comfortable -- we work with decimal digits every day. The neat thing about number systems is that there is nothing that forces you to have 10 different values in a digit. Our **base-10** number system likely grew up because we have 10 fingers, but if we happened to evolve to have eight fingers instead, we would probably have a base-8 number system. You can have base-anything number systems. In fact, there are lots of good reasons to use different bases in different situations.

Computers happen to operate using the base-2 number system, also known as the **binary number system** (just like the base-10 number system is known as the decimal number system). Find out why and how that works in the next section.

## The Base-2 System and the 8-bit Byte

The reason computers use the base-2 system is because it makes it a lot easier to implement them with current electronic technology. You could wire up and build computers that operate in base-10, but they would be fiendishly expensive right now. On the other hand, base-2 computers are relatively cheap.

So computers use binary numbers, and therefore use **binary digits** in place of decimal digits. The word **bit** is a shortening of the words "Binary digIT." Whereas decimal digits have 10 possible values ranging from 0 to 9, bits have only two possible values: 0 and 1. Therefore, a binary number is composed of only 0s and 1s, like this: 1011. How do you figure out what the value of the binary number 1011 is? You do it in the same way we did it above for 6357, but you use a base of 2 instead of a base of 10. So:

(1 * 2^3) + (0 * 2^2) + (1 * 2^1) + (1 * 2^0) = 8 + 0 + 2 + 1 = 11

You can see that in binary numbers, each bit holds the value of increasing powers of 2. That makes counting in binary pretty easy. Starting at zero and going through 20, counting in decimal and binary looks like this:

```
 0 =     0
 1 =     1
 2 =    10
 3 =    11
 4 =   100
 5 =   101
 6 =   110
 7 =   111
 8 =  1000
 9 =  1001
10 =  1010
11 =  1011
12 =  1100
13 =  1101
14 =  1110
15 =  1111
16 = 10000
17 = 10001
18 = 10010
19 = 10011
20 = 10100
```

When you look at this sequence, 0 and 1 are the same for decimal and binary number systems. At the number 2, you see carrying first take place in the binary system. If a bit is 1, and you add 1 to it, the bit becomes 0 and the next bit becomes 1. In the transition from 15 to 16 this effect rolls over through 4 bits, turning 1111 into 10000.

Bits are rarely seen alone in computers. They are almost always bundled together into 8-bit collections, and these collections are called **bytes**. Why are there 8 bits in a byte? A similar question is, "Why are there 12 eggs in a dozen?" The 8-bit byte is something that people settled on through trial and error over the past 50 years.

With 8 bits in a byte, you can represent 256 values ranging from 0 to 255, as shown here:

```
  0 = 00000000
  1 = 00000001
  2 = 00000010
   ...
254 = 11111110
255 = 11111111
```

In the article How CDs Work, you learn that a CD uses 2 bytes, or 16 bits, per sample. That gives each sample a range from 0 to 65,535, like this:

```
    0 = 0000000000000000
    1 = 0000000000000001
    2 = 0000000000000010
     ...
65534 = 1111111111111110
65535 = 1111111111111111
```

Next, we'll look at one way that bytes are used.

## The Standard ASCII Character Set

Bytes are frequently used to hold individual characters in a text document. In the **ASCII character set**, each binary value between 0 and 127 is given a specific character. Most computers extend the ASCII character set to use the full range of 256 characters available in a byte. The upper 128 characters handle special things like accented characters from common foreign languages.

You can see the 127 standard ASCII codes below. Computers store text documents, both on disk and in memory, using these codes. For example, if you use Notepad in Windows 95/98 to create a text file containing the words, "Four score and seven years ago," Notepad would use 1 byte of memory per character (including 1 byte for each space character between the words -- ASCII character 32). When Notepad stores the sentence in a file on disk, the file will also contain 1 byte per character and per space.

Try this experiment: Open up a new file in Notepad and insert the sentence, "Four score and seven years ago" in it. Save the file to disk under the name **getty.txt**. Then use the explorer and look at the size of the file. You will find that the file has a size of 30 bytes on disk: 1 byte for each character. If you add another word to the end of the sentence and re-save it, the file size will jump to the appropriate number of bytes. Each character consumes a byte.

If you were to look at the file as a computer looks at it, you would find that each byte contains not a letter but a number -- the number is the ASCII code corresponding to the character (see below). So on disk, the numbers for the file look like this:

```
 F   o   u   r     a   n   d      s   e   v   e   n
70 111 117 114 32 97 110 100 32 115 101 118 101 110
```

By looking in the ASCII table, you can see a one-to-one correspondence between each character and the ASCII code used. Note the use of 32 for a space -- 32 is the ASCII code for a space. We could expand these decimal numbers out to binary numbers (so 32 = 00100000) if we wanted to be technically correct -- that is how the computer really deals with things.

The first 32 values (0 through 31) are codes for things like carriage return and line feed. The space character is the 33rd value, followed by punctuation, digits, uppercase characters and lowercase characters. To see all 127 values, check out Unicode.org's chart.

We'll learn about byte prefixes and binary math next.

## Byte Prefixes and Binary Math

When you start talking about lots of bytes, you get into **prefixes** like kilo, mega and giga, as in kilobyte, megabyte and gigabyte (also shortened to K, M and G, as in Kbytes, Mbytes and Gbytes or KB, MB and GB). The following table shows the **binary** multipliers:

**Kilo (K)**
2^10 = 1,024

**Mega (M)**
2^20 = 1,048,576

**Giga (G)**
2^30 = 1,073,741,824

**Tera (T)**
2^40 = 1,099,511,627,776

**Peta (P)**
2^50 = 1,125,899,906,842,624

**Exa (E)**
2^60 = 1,152,921,504,606,846,976

**Zetta (Z)**
2^70 = 1,180,591,620,717,411,303,424

**Yotta (Y)**
2^80 = 1,208,925,819,614,629,174,706,176

You can see in this chart that kilo is about a thousand, mega is about a million, giga is about a billion, and so on. So when someone says, "This computer has a 2 gig hard drive," what he or she means is that the hard drive stores 2 gigabytes, or approximately 2 billion bytes, or exactly 2,147,483,648 bytes. How could you possibly need 2 gigabytes of space? When you consider that one CD holds 650 megabytes, you can see that just three CDs worth of data will fill the whole thing! Terabyte databases are fairly common these days, and there are probably a few petabyte databases floating around the Pentagon by now.

Binary math works just like decimal math, except that the value of each bit can be only **0 or 1**. To get a feel for binary math, let's start with decimal addition and see how it works. Assume that we want to add 452 and 751:

```
  452
+ 751
  ---
 1203
```

To add these two numbers together, you start at the right: 2 + 1 = 3. No problem. Next, 5 + 5 = 10, so you save the zero and carry the 1 over to the next place. Next, 4 + 7 + 1 (because of the carry) = 12, so you save the 2 and carry the 1. Finally, 0 + 0 + 1 = 1. So the answer is 1203.

Binary addition works exactly the same way:

```
  010
+ 111
  ---
 1001
```

Starting at the right, 0 + 1 = 1 for the first digit. No carrying there. You've got 1 + 1 = 10 for the second digit, so save the 0 and carry the 1. For the third digit, 0 + 1 + 1 = 10, so save the zero and carry the 1. For the last digit, 0 + 0 + 1 = 1. So the answer is 1001. If you translate everything over to decimal you can see it is correct: 2 + 7 = 9.

To see how boolean addition is implemented using gates, see How Boolean Logic Works.

To sum up, here's what we've learned about bits and bytes:

- Bits are binary digits. A bit can hold the value 0 or 1.
- Bytes are made up of 8 bits each.
- Binary math works just like decimal math, but each bit can have a value of only 0 or 1.

There really is nothing more to it -- bits and bytes are that simple.

For more information on bits, bytes and related topics, check out the links on the next page.

## Lots More Information

### Related Articles

- How Boolean Logic Works
- How Electronic Gates Work
- How Computer Memory Works
- How C Programming Works
- How Java Works
- How File Compression Works
- How Hard Disks Work
- How CDs Work
- How PCs Works
- How BIOS Works

### More Great Links

- Microcomputer/DOS Tutorial: Bits and Bytes
- The Binary System
- Tutorial: Data lines, bits, nibbles, bytes, words, binary and HEX

- Beginner's Programming Tutorial in QBasic - Numbering systems