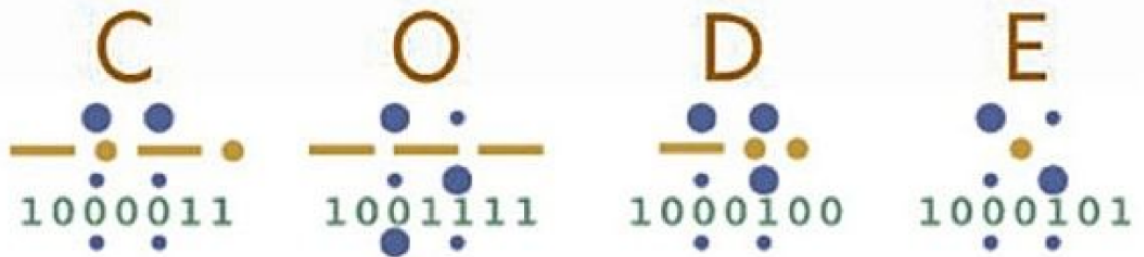


The Hidden Language of Computer Hardware and Software



Charles Petzold

Table of Contents

Preface to the Paperback Edition

1. Best Friends

2. Codes and Combinations

3. Braille and Binary Codes

4. Anatomy of a Flashlight

5. Seeing Around Corners

6. Telegraphs and Relays

7. Our Ten Digits

8. Alternatives to Ten

9. Bit by Bit by Bit

10. Logic and Switches

11. Gates (Not Bill)

12. A Binary Adding Machine

13. But What About Subtraction?

14. Feedback and Flip-Flops

15. Bytes and Hex

16. An Assemblage of Memory

17. Automation

18. From Abaci to Chips

19. Two Classic Microprocessors

20. ASCII and a Cast of Characters

21. Get on the Bus

22. The Operating System

23. Fixed Point, Floating Point

[24. Languages High and Low](#)

[25. The Graphical Revolution](#)

[A. Acknowledgments](#)

[B. Bibliography](#)

[Index](#)

[About the Author](#)

[Colophon](#)

Preface to the Paperback Edition

Code rattled around in my head for about a decade before I started writing it. As I was contemplating *Code* and then writing it, and even after the book was published, people would ask me, “What’s the book about?”

I was always reluctant to answer this question. I’d mumble something about “a unique journey through the evolution of the digital technologies that define the modern age” and hope that would be sufficient.

But finally I had to admit it: “*Code* is a book about how computers work.”

As I feared, the reactions weren’t favorable. “Oh, I have a book like that,” some people would say, to which my immediate response was, “No, no, no, you don’t have a book like this one.” I still think that’s true. *Code* is not like other how-computers-work books. It doesn’t have big color illustrations of disk drives with arrows showing how the data sweeps into the computer. *Code* has no drawings of trains carrying a cargo of zeros and ones. Metaphors and similes are wonderful literary devices but they do nothing but obscure the beauty of technology.

The other comment I heard was, “People don’t want to know how computers work.” And this I’m sure is true. I personally happen to enjoy learning how things work. But I also like to choose which things I learn about and which I do not. I’d be hard pressed to explain how my refrigerator works, for example.

Yet I often hear people ask questions that reveal a need to know something about the inner workings of personal computers. One such common question is, “What’s the difference between storage and memory?”

That’s certainly a critical question. The marketing of personal computers is based on such concepts. Even novice users are expected to know how many *megas* of the one thing and *gigas* of the other thing will be necessary for their particular applications. Novice users are also expected to master the concept of the computer “file” and to visualize how files are loaded from storage into memory and saved from memory back to storage.

The storage-and-memory question is usually answered with an analogy:

“Memory is like the surface of your desk and storage is like the filing cabinet.” That’s not a bad answer as far as it goes. But I find it quite unsatisfactory. It makes it sound as if computer architecture were patterned after an office. The truth is that the distinction between memory and storage is an artificial one and exists solely because we don’t have a single storage medium that is both fast and vast as well as nonvolatile. What we know today as “von Neumann architecture”—the dominant computer architecture for over 50 years—is a direct result of this technical deficiency.

Here’s another question that someone once asked me: “Why can’t you run Macintosh programs under Windows?” My mouth opened to begin an answer when I realized that it involved many more technical issues than I’m sure my questioner was prepared to deal with in one sitting.

I want *Code* to be a book that makes you understand these things, not in some abstract way, but with a depth that just might even rival that of electrical engineers and programmers. I also hope that you might recognize the computer to be one of the crowning achievements of twentieth century technology and appreciate it as a beautiful thing in itself without metaphors and similes getting in the way.

Computers are constructed in a hierarchy, from transistors down at the bottom to the information displayed on our computer screens at the top. Moving up each level in the hierarchy—which is how *Code* is structured—is probably not as hard as most people might think. There is certainly a lot going on inside the modern computer, but it is a lot of very common and simple operations.

Although computers today are more complex than the computers of 25 years or 50 years ago, they are still fundamentally the same. That’s what’s so great about studying the history of technology: The further back in time you go, the simpler the technologies become. Thus it’s possible to reach a point where it all makes relatively easy sense.

In *Code*, I went as far back as I could go. Astonishingly, I found that I could go back into the nineteenth century and use early telegraph equipment to show how computers are built. In theory at least, everything in the first 17 chapters of *Code* can be built entirely using simple electrical devices that have been around for over a century.

This use of antique technology gives *Code* a fairly nostalgic feel, I think. *Code* is a book that could never be titled *The Faster New Faster Thing* or *Business @ the Speed of a Digital Nervous System*. The “bit” isn’t defined until page 68; “byte” isn’t defined until page 180. I don’t mention transistors until page

142, and that's only in passing.

So, while *Code* goes fairly deep into the workings of the computer (few other books show how computer processors actually work, for example), the pace is fairly relaxed. Despite the depth, I tried to make the trip as comfortable as possible.

But without little drawings of trains carrying a cargo of zeros and ones.

Charles Petzold

August 16, 2000

Chapter 1. Best Friends

code (kōd) ...

3.a. A system of signals used to represent letters or numbers in transmitting messages.

b. A system of symbols, letters, or words given certain arbitrary meanings, used for transmitting messages requiring secrecy or brevity.

4. A system of symbols and rules used to represent instructions to a computer...

—*The American Heritage Dictionary of the English Language*

You're 10 years old. Your best friend lives across the street. In fact, the windows of your bedrooms face each other. Every night, after your parents have declared bedtime at the usual indecently early hour, you still need to exchange thoughts, observations, secrets, gossip, jokes, and dreams. No one can blame you. After all, the impulse to communicate is one of the most human of traits.

While the lights are still on in your bedrooms, you and your best friend can wave to each other from the windows and, using broad gestures and rudimentary body language, convey a thought or two. But sophisticated transactions seem difficult. And once the parents have decreed "Lights out!" the situation seems hopeless.

How to communicate? The telephone perhaps? Do you have a telephone in your room at the age of 10? Even so, wherever the phone is you'll be overheard. If your family personal computer is hooked into a phone line, it might offer soundless help, but again, it's not in your room.

What you and your best friend *do* own, however, are flashlights. Everyone knows that flashlights were invented to let kids read books under the bed covers; flashlights also seem perfect for the job of communicating after dark. They're certainly quiet enough, and the light is highly directional and probably won't seep out under the bedroom door to alert your suspicious folks.

Can flashlights be made to speak? It's certainly worth a try. You learned how to write letters and words on paper in first grade, so transferring that knowledge to the flashlight seems reasonable. All you have to do is stand at

your window and draw the letters with light. For an O, you turn on the flashlight, sweep a circle in the air, and turn off the switch. For an I, you make a vertical stroke. But, as you discover quickly, this method simply doesn't work. As you watch your friend's flashlight making swoops and lines in the air, you find that it's too hard to assemble the multiple strokes together in your head. These swirls and slashes of light are not *precise* enough.

You once saw a movie in which a couple of sailors signaled to each other across the sea with blinking lights. In another movie, a spy wiggled a mirror to reflect the sunlight into a room where another spy lay captive. Maybe that's the solution. So you first devise a simple technique. Each letter of the alphabet corresponds to a series of flashlight blinks. An A is 1 blink, a B is 2 blinks, a C is 3 blinks, and so on to 26 blinks for Z. The word BAD is 2 blinks, 1 blink, and 4 blinks with little pauses between the letters so you won't mistake the 7 blinks for a G. You'll pause a bit longer between words.

This seems promising. The good news is that you no longer have to wave the flashlight in the air; all you have to do is point and click. The bad news is that one of the first messages you try to send ("How are you?") turns out to require a grand total of 131 blinks of light! Moreover, you forgot about punctuation, so you don't know how many blinks correspond to a question mark.

But you're close. Surely, you think, somebody must have faced this problem before, and you're absolutely right. With daylight and a trip to the library for research, you discover a marvelous invention known as Morse code. It's *exactly* what you've been looking for, even though you must now relearn how to "write" all the letters of the alphabet.

Here's the difference: In the system you invented, every letter of the alphabet is a certain number of blinks, from 1 blink for A to 26 blinks for Z. In Morse code, you have two kinds of blinks—short blinks and long blinks. This makes Morse code more complicated, of course, but in actual use it turns out to be much more efficient. The sentence "How are you?" now requires only 32 blinks (some short, some long) rather than 131, and that's *including* a code for the question mark.

When discussing how Morse code works, people don't talk about "short blinks" and "long blinks." Instead, they refer to "dots" and "dashes" because that's a convenient way of showing the codes on the printed page. In Morse code, every letter of the alphabet corresponds to a short series of dots and dashes, as you can see in the following table.

A	·--	J	·----	S	...
B	--...	K	--·	T	--
C	--·..	L	·...	U	··--
D	---·	M	--	V	····
E	·	N	--·	W	·--·
F	·····	O	---·	X	---·
G	---·	P	·---·	Y	---·
H	····	Q	---·	Z	---·
I	··	R	·--·		

Although Morse code has absolutely nothing to do with computers, becoming familiar with the nature of codes is an essential preliminary to achieving a deep understanding of the hidden languages and inner structures of computer hardware and software.

In this book, the word *code* usually means a system for transferring information among people and machines. In other words, a code lets you communicate. Sometimes we think of codes as secret. But most codes are not. Indeed, most codes must be well understood because they're the basis of human communication.

In the beginning of *One Hundred Years of Solitude*, Gabriel Garcia Marquez recalls a time when “the world was so recent that many things lacked names, and in order to indicate them it was necessary to point.” The names that we assign to things usually seem arbitrary. There seems to be no reason why cats aren't called “dogs” and dogs aren't called “cats.” You could say English vocabulary is a type of code.

The sounds we make with our mouths to form words are a code intelligible to anyone who can hear our voices and understands the language that we speak. We call this code “the spoken word,” or “speech.” We have other code for words on paper (or on stone, on wood, or in the air, say, via skywriting). This code appears as handwritten characters or printed in newspapers, magazines, and books. We call it “the written word,” or “text.” In many languages, a strong correspondence exists between speech and text. In English, for example, letters and groups of letters correspond (more or less) to spoken sounds.

For people who can't hear or speak, another code has been devised to help in face-to-face communication. This is sign language, in which the hands and arms form movements and gestures that convey individual letters of words or whole words and concepts. For those who can't see, the written word can be replaced with Braille, which uses a system of raised dots that correspond to letters, groups of letters, and whole words. When spoken words must be transcribed into text very quickly, stenography or shorthand is useful.

We use a variety of different codes for communicating among ourselves because some codes are more convenient than others. For example, the code of the spoken word can't be stored on paper, so the code of the written word is used instead. Silently exchanging information across a distance in the dark isn't possible with speech or paper. Hence, Morse code is a convenient alternative. A code is useful if it serves a purpose that no other code can.

As we shall see, various types of codes are also used in computers to store and communicate numbers, sounds, music, pictures, and movies. Computers can't deal with human codes directly because computers can't duplicate the ways in which human beings use their eyes, ears, mouths, and fingers. Yet one of the recent trends in computer technology has been to enable our desktop personal computers to capture, store, manipulate, and render all types of information used in human communication, be it visual (text and pictures), aural (spoken words, sounds, and music), or a combination of both (animations and movies). All of these types of information require their own codes, just as speech requires one set of human organs (mouths and ears) while writing and reading require others (hands and eyes).

Even the table of Morse code shown on page 4 is itself a code of sorts. The table shows that each letter is represented by a series of dots and dashes. Yet we can't actually send dots and dashes. Instead, the dots and dashes correspond to blinks.

When sending Morse code with a flashlight, you turn the flashlight switch on and off very quickly (a fast blink) for a dot. You leave the flashlight turned on somewhat longer (a slower on-off blink) for a dash. To send an A, for example, you turn the flashlight on and off very quickly and then on and off at a lesser speed. You pause before sending the next character. By convention, the length of a dash should be about three times that of a dot. For example, if a dot is one second long, a dash is three seconds long. (In reality, Morse code is transmitted much faster than that.) The receiver sees

the short blink and the long blink and knows it's an A.

Pauses between the dots and dashes of Morse code are crucial. When you send an A, for example, the flashlight should be off between the dot and the dash for a period of time equal to about one dot. (If the dot is one second long, the gap between dots and dashes is also a second.) Letters in the same word are separated by longer pauses equal to about the length of one dash (or three seconds if that's the length of a dash). For example, here's the Morse code for "hello," illustrating the pauses between the letters:



Words are separated by an off period of about two dashes (six seconds if a dash is three seconds long). Here's the code for "hi there":



The lengths of time that the flashlight remains on and off aren't fixed. They're all relative to the length of a dot, which depends on how fast the flashlight switch can be triggered and also how quickly a Morse code sender can remember the code for a particular letter. A fast sender's dash may be the same length as a slow sender's dot. This little problem could make reading a Morse code message tough, but after a letter or two, the receiver can usually figure out what's a dot and what's a dash.

At first, the definition of Morse code—and by *definition* I mean the correspondence of various sequences of dots and dashes to the letters of the alphabet—appears as random as the layout of a typewriter. On closer inspection, however, this is not entirely so. The simpler and shorter codes are assigned to the more frequently used letters of the alphabet, such as E and T. Scrabble players and *Wheel of Fortune* fans might notice this right away. The less common letters, such as Q and Z (which get you 10 points in Scrabble), have longer codes.

Almost everyone knows a little Morse code. Three dots, three dashes, and three dots represent SOS, the international distress signal. SOS isn't an abbreviation for anything—it's simply an easy-to-remember Morse code sequence. During the Second World War, the British Broadcasting Corporation prefaced some radio broadcasts with the beginning of Beethoven's Fifth Symphony—BAH, BAH, BAH, BAHMMMMM—which Ludwig didn't know at the time he composed the music is the Morse code V, for Victory.

One drawback of Morse code is that it makes no differentiation between uppercase and lowercase letters. But in addition to representing letters, Morse code also includes codes for numbers by using a series of five dots and dashes:

1	• — — — —	6	— — — — •
2	• • — — —	7	— — — — •
3	• • • — —	8	— — — — • •
4	• • • • —	9	— — — — — •
5	• • • • •	0	— — — — —

These codes, at least, are a little more orderly than the letter codes. Most punctuation marks use five, six, or seven dots and dashes:

.	• — — — — •	'	• — — — — — •
,	— — — — — • —	(— — — — — •
?	• • — — — • •)	— — — — — — •
:	— — — — — • •	=	— — — — — —
;	— — — — — • • •	+	• • — — — •
-	— — — — — — •	\$	• • — — — — —
/	— — — — — • •	¶	• — — — — • •
"	• — — — — — • •	_	• • — — — — — —

Additional codes are defined for accented letters of some European languages and as shorthand sequences for special purposes. The SOS code is one such shorthand sequence: It's supposed to be sent continuously with only a one-dot pause between the three letters.

You'll find that it's much easier for you and your friend to send Morse code if you have a flashlight made specifically for this purpose. In addition to the normal on-off slider switch, these flashlights also include a pushbutton switch that you simply press and release to turn the flashlight on and off. With some practice, you might be able to achieve a sending and receiving

speed of 5 or 10 words per minute—still much slower than speech (which is somewhere in the 100-words-per-minute range), but surely adequate.

When finally you and your best friend memorize Morse code (for that's the only way you can become proficient at sending and receiving it), you can also use it vocally as a substitute for normal speech. For maximum speed, you pronounce a dot as *dih* (or *dit* for the last dot of a letter) and a dash as *dah*. In the same way that Morse code reduces written language to dots and dashes, the spoken version of the code reduces speech to just two vowel sounds.

The key word here is *two*. Two types of blinks, two vowel sounds, two different anything, really, can with suitable combinations convey all types of information.

Chapter 2. Codes and Combinations

Morse code was invented by Samuel Finley Breese Morse (1791–1872), whom we shall meet more properly later in this book. The invention of Morse code goes hand in hand with the invention of the telegraph, which we'll also examine in more detail. Just as Morse code provides a good introduction to the nature of codes, the telegraph provides a good introduction to the hardware of the computer.

Most people find Morse code easier to send than to receive. Even if you don't have Morse code memorized, you can simply use this table, conveniently arranged in alphabetical order:

A	·--	J	·----	S	...
B	---··	K	--·-	T	--
C	---·-	L	·---·	U	··--
D	--··	M	--	V	··--
E	·	N	--·	W	·--
F	··---	O	---	X	---·-
G	--·-	P	·---·	Y	--·--
H	····	Q	---·-	Z	---··
I	··	R	·--·		

Receiving Morse code and translating it back into words is considerably harder and more time consuming than sending because you must work backward to figure out the letter that corresponds to a particular coded sequence of dots and dashes. For example, if you receive a dash-dot-dash-dash, you have to scan through the table letter by letter before you finally discover that the code is the letter Y.

The problem is that we have a table that provides this translation:

But we *don't* have a table that lets us go backward:

In the early stages of learning Morse code, such a table would certainly be convenient. But it's not at all obvious how we could construct it. There's nothing in those dots and dashes that we can put into alphabetical order.

So let's forget about alphabetical order. Perhaps a better approach to organizing the codes might be to group them depending on how many dots and dashes they have. For example, a Morse code sequence that contains either one dot or one dash can represent only two letters, which are E and T:

·	E
—	T

A combination of exactly two dots or dashes gives us four more letters—I, A, N, and M:

··	I	—·	N
·—	A	— —	M

A pattern of three dots or dashes gives us eight more letters:

···	S	—··	D
··—	U	—·—	K
·—·	R	— —·	G
·— —	W	— — —	O

And finally (if we want to stop this exercise before dealing with numbers and punctuation marks), sequences of four dots and dashes give us 16 more characters:

....	H	----	B
...-	V	----	X
....	F	----.	C
...-	Ü	----	Y
....	L	---.	Z
...-	Ä	----	Q
....	P	---.	Ö
...-	J	----	Ş

Taken together, these four tables contain 2 plus 4 plus 8 plus 16 codes for a total of 30 letters, 4 more than are needed for the 26 letters of the Latin alphabet. For this reason, you'll notice that 4 of the codes in the last table are for accented letters.

These four tables might help you translate with greater ease when someone is sending you Morse code. After you receive a code for a particular letter, you know how many dots and dashes it has, and you can at least go to the right table to look it up. Each table is organized so that you find the all-dots code in the upper left and the all-dashes code in the lower right.

Can you see a pattern in the *size* of the four tables? Notice that each table has twice as many codes as the table before it. This makes sense: Each table has all the codes in the previous table followed by a dot, and all the codes in the previous table followed by a dash.

We can summarize this interesting trend this way:

Number of Dots and Dashes	Number of Codes
1	2
2	4
3	8
4	16

Each of the four tables has twice as many codes as the table before it, so if the first table has 2 codes, the second table has 2×2 codes, and the third table has $2 \times 2 \times 2$ codes. Here's another way to show that:

Number of Dots and Dashes	Number of Codes
1	2
2	2×2
3	$2 \times 2 \times 2$
4	$2 \times 2 \times 2 \times 2$

Of course, once we have a number multiplied by itself, we can start using exponents to show powers. For example, $2 \times 2 \times 2 \times 2$ can be written as 2^4 (2 to the 4th power). The numbers 2, 4, 8, and 16 are all powers of 2 because you can calculate them by multiplying 2 by itself. So our summary can also be shown like this:

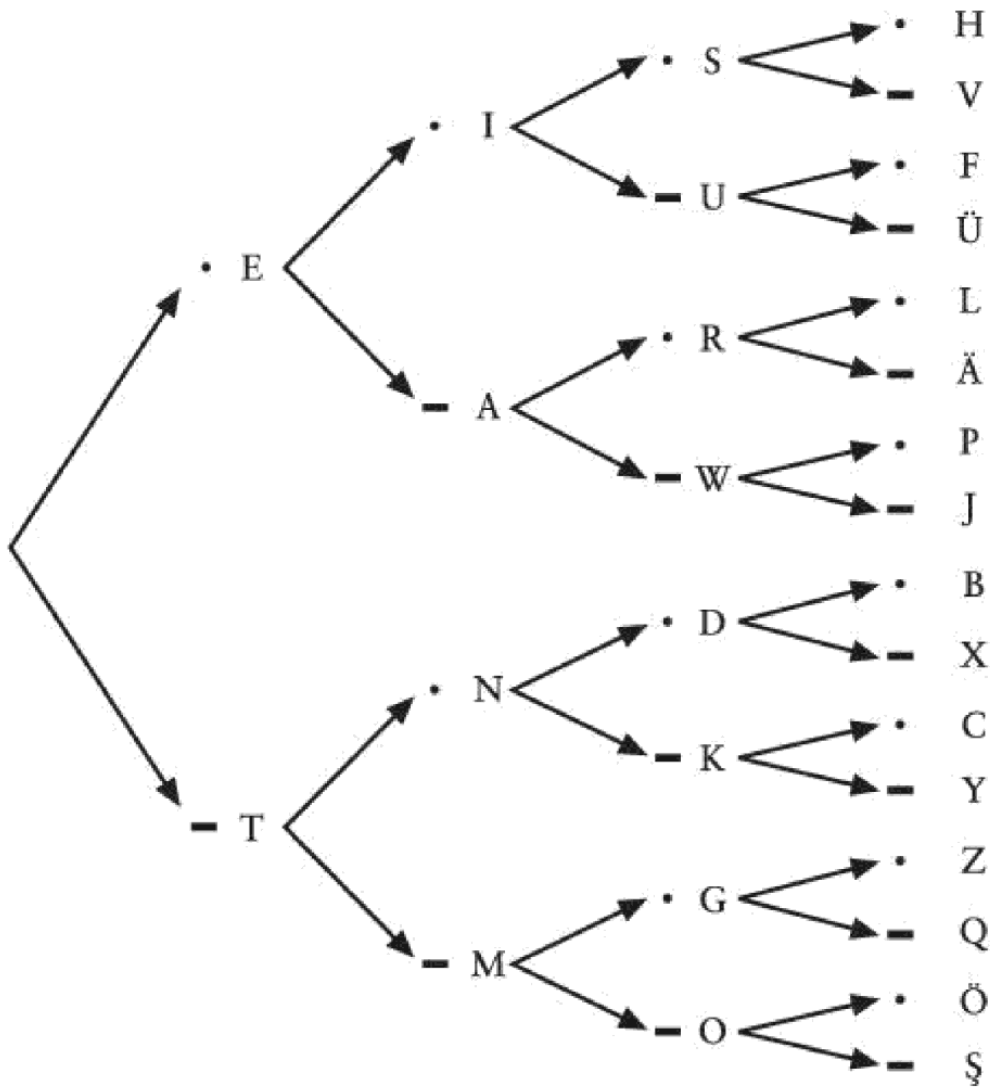
Number of Dots and Dashes	Number of Codes
1	2^1
2	2^2
3	2^3
4	2^4

This table has become very simple. The number of codes is simply 2 to the power of the number of dots and dashes. We might summarize the table data in this simple formula:

$$\text{number of codes} = 2^{\text{number of dots and dashes}}$$

Powers of 2 tend to show up a lot in codes, and we'll see another example in the next chapter.

To make the process of decoding Morse code even easier, we might want to draw something like the big treelike table shown here.



This table shows the letters that result from each particular consecutive sequence of dots and dashes. To decode a particular sequence, follow the arrows from left to right. For example, suppose you want to know which letter corresponds to the code dot-dash-dot. Begin at the left and choose the dot; then continue moving right along the arrows and choose the dash and then another dot. The letter is R, shown next to the last dot.

If you think about it, constructing such a table was probably necessary for defining Morse code in the first place. First, it ensures that you don't make the dumb mistake of using the same code for two different letters! Second, you're assured of using all the possible codes without making the sequences of dots and dashes unnecessarily long.

At the risk of extending this table beyond the limits of the printed page, we could continue it for codes of five dots and dashes and more. A sequence of exactly five dots and dashes gives us 32 ($2 \times 2 \times 2 \times 2 \times 2$, or 2^5) additional codes.

Normally that would be enough for the 10 numbers and the 16 punctuation symbols defined in Morse code, and indeed the numbers are encoded with five dots and dashes. But many of the other codes that use a sequence of five dots and dashes represent accented letters rather than punctuation marks.

To include all the punctuation marks, the system must be expanded to six dots and dashes, which gives us 64 ($2 \times 2 \times 2 \times 2 \times 2 \times 2$, or 2^6) additional codes for a grand total of $2+4+8+16+32+64$, or 126, characters. That's overkill for Morse code, which leaves many of these longer codes "undefined." The word *undefined* used in this context refers to a code that doesn't stand for anything. If you were receiving Morse code and you got an undefined code, you could be pretty sure that somebody made a mistake.

Because we were clever enough to develop this little formula,

$$\text{number of codes} = 2^{\text{number of dots and dashes}}$$

we could continue figuring out how many codes we get from using longer sequences of dots and dashes:

Number of Dots and Dashes	Number of Codes
1	$2^1 = 2$
2	$2^2 = 4$
3	$2^3 = 8$
4	$2^4 = 16$
5	$2^5 = 32$
6	$2^6 = 64$
7	$2^7 = 128$
8	$2^8 = 256$
9	$2^9 = 512$
10	$2^{10} = 1024$

Fortunately, we don't have to actually write out all the possible codes to determine how many there would be. All we have to do is multiply 2 by itself over and over again.

Morse code is said to be a *binary* (literally meaning *two by two*) code because the components of the code consist of only two things—a dot and a dash. That's similar to a coin, which can land only on the head side or the tail side. Binary objects (such as coins) and binary codes (such as Morse code) are always described by powers of two.

What we're doing by analyzing binary codes is a simple exercise in the branch of mathematics known as *combinatorics* or *combinatorial analysis*. Traditionally, combinatorial analysis is used most often in the fields of probability and statistics because it involves determining the number of

ways that things, like coins and dice, can be combined. But it also helps us understand how codes can be put together and taken apart.

Chapter 3. Braille and Binary Codes

Samuel Morse wasn't the first person to successfully translate the letters of written language to an interpretable code. Nor was he the first person to be remembered more as the name of his code than as himself. That honor must go to a blind French teenager born some 18 years after Samuel Morse but who made his mark much more precociously. Little is known of his life, but what is known makes a compelling story.

Louis Braille was born in 1809 in Coupvray, France, just 25 miles east of Paris. His father was a harness maker. At the age of three—an age when young boys shouldn't be playing in their fathers' workshops—he accidentally stuck a pointed tool in his eye. The wound became infected, and the infection spread to his other eye, leaving him totally blind. Normally he would have been doomed to a life of ignorance and poverty (as most blind people were in those days), but young Louis's intelligence and desire to learn were soon recognized. Through the intervention of the village priest and a schoolteacher, he first attended school in the village with the other children and at the age of 10 was sent to the Royal Institution for Blind Youth in Paris.



One major obstacle in the education of the blind is, of course, their inability to read printed books. Valentin Haüy (1745–1822), the founder of the Paris school, had invented a system of raised letters on paper that could be read by touch. But this system was very difficult to use, and only a few books had been produced using this method.

The sighted Haüy was stuck in a paradigm. To him, an A was an A was an A, and the letter A must look (or feel) like an A. (If given a flashlight to communicate, he might have tried drawing letters in the air as we did before we discovered it didn't work very well.) Haüy probably didn't realize that a type of code quite different from the printed alphabet might be more appropriate for sightless people.

The origins of an alternative type of code came from an unexpected source. Charles Barbier, a captain of the French army, had by 1819 devised a system of writing he called *écriture nocturne*, or “night writing.” This system used a pattern of raised dots and dashes on heavy paper and was intended for use by soldiers in passing notes to each other in the dark when quiet was necessary. The soldiers were able to poke these dots and dashes into the back of the paper using an awl-like stylus. The raised dots could then be read with the fingers.

The problem with Barbier's system is that it was quite complex. Rather than using patterns of dots and dashes that corresponded to letters of the alphabet, Barbier devised patterns that corresponded to sounds, often requiring many codes for a single word. The system worked fine for short

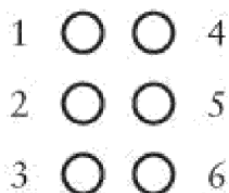
messages in the field but was distinctly inadequate for longer texts, let alone entire books.

Louis Braille became familiar with Barbier's system at the age of 12. He liked the use of raised dots, not only because it proved easy to read with the fingers but also because it was easy to *write*. A student in the classroom equipped with paper and a stylus could actually take notes and read them back. Louis Braille diligently tried to improve the system and within three years (at the age of 15) had come up with his own, the basics of which are still used today. For many years, the system was known only within the school, but it gradually made its way to the rest of the world. In 1835, Louis Braille contracted tuberculosis, which would eventually kill him shortly after his 43rd birthday in 1852.

Today, enhanced versions of the Braille system compete with tape-recorded books for providing the blind with access to the written word, but Braille still remains an invaluable system and the only way to read for people who are both blind and deaf. In recent years, Braille has become more familiar in the public arena as elevators and automatic teller machines are made more accessible to the blind.

What we're going to do in this chapter is dissect Braille code and see how it works. We don't have to actually *learn* Braille or memorize anything. We just want some insight into the nature of codes.

In Braille, every symbol used in normal written language—specifically, letters, numbers, and punctuation marks—is encoded as one or more raised dots within a two-by-three cell. The dots of the cell are commonly numbered 1 through 6:



In modern-day use, special typewriters or embossers punch the Braille dots into the paper.

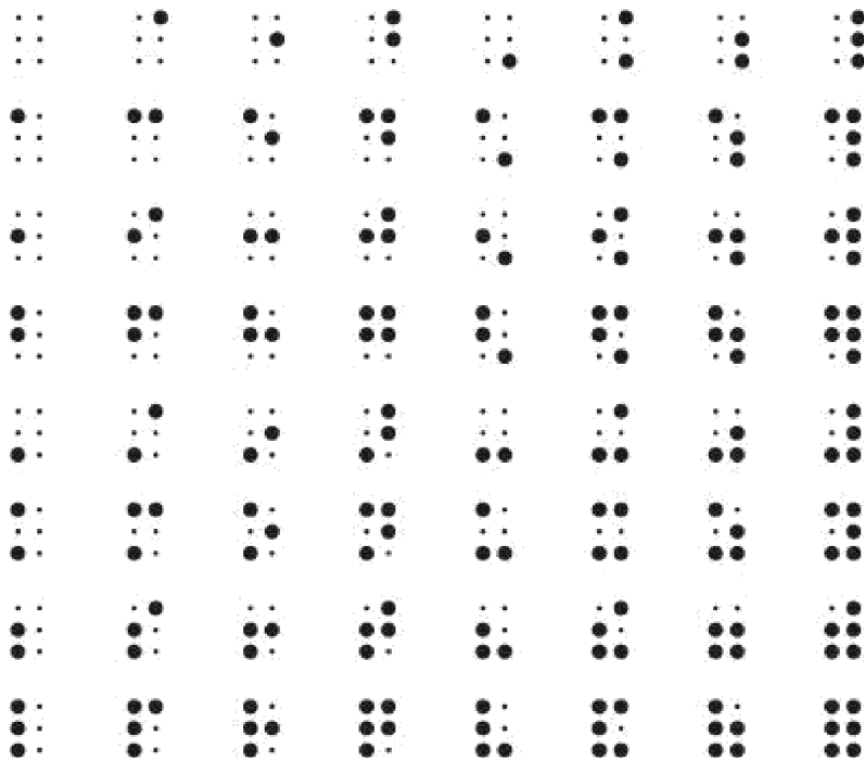
Because embossing just a couple pages of this book in Braille would be prohibitively expensive, I've used a notation common for showing Braille on the printed page. In this notation, all six dots in the cell are shown. Large dots indicate the parts of the cell where the paper is raised. Small dots

indicate the parts of the cell that are flat. For example, in the Braille character dots 1, 3, and 5 are raised and dots 2, 4, and 6 are not.



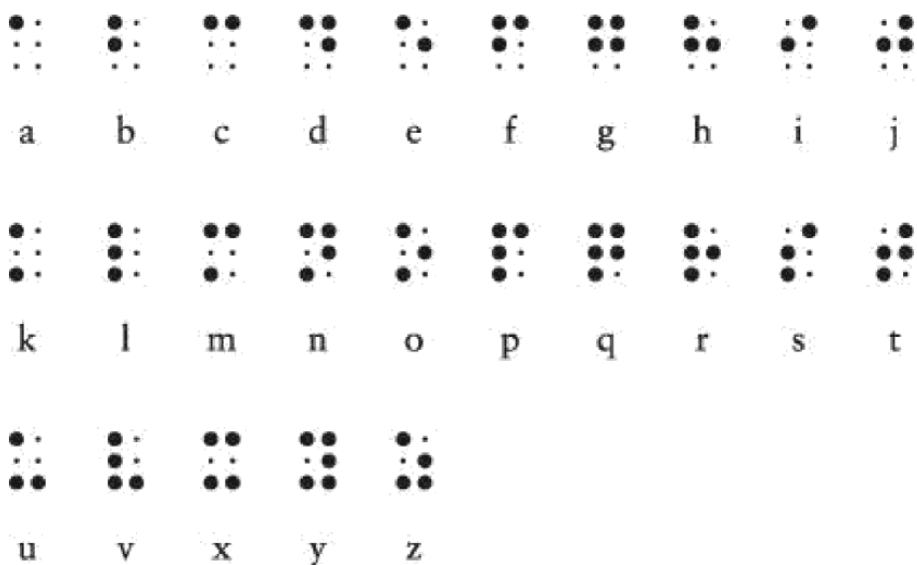
What should be interesting to us at this point is that the dots are *binary*. A particular dot is either flat or raised. That means we can apply what we've learned about Morse code and combinatorial analysis to Braille. We know that there are 6 dots and that each dot can be either flat or raised, so the total number of combinations of 6 flat and raised dots is $2 \times 2 \times 2 \times 2 \times 2 \times 2$, or 2^6 , or 64.

Thus, the system of Braille is capable of representing 64 unique codes. Here they are—all 64 possible Braille codes:



If we find fewer than 64 codes used in Braille, we should question why some of the 64 possible codes aren't being used. If we find *more* than 64 codes used in Braille, we should question either our sanity or fundamental truths of mathematics, such as 2 plus 2 equaling 4.

To begin dissecting the code of Braille, let's look at the basic lowercase alphabet:



For example, the phrase “you and me” in Braille looks like this:













Notice that the cells for each letter within a word are separated by a little bit of space; a larger space (essentially a cell with no raised dots) is used between words.











This is the basis of Braille as Louis Braille devised it, or at least as it applies to the letters of the Latin alphabet. Louis Braille also devised codes for letters with accent marks, common in French. Notice that there’s no code for *w*, which isn’t used in classical French. (Don’t worry. The letter will show up eventually.) At this point, only 25 of the 64 possible codes have been accounted for.

Upon close examination, you’ll discover that the three rows of Braille illustrated above show a pattern. The first row (letters *a* through *j*) uses only the top four spots in the cell—dots 1, 2, 4, and 5. The second row duplicates the first row except that dot 3 is also raised. The third row is the same except that dots 3 and 6 are raised.











Since the days of Louis Braille, the Braille code has been expanded in various ways. Currently the system used most often in published material in English is called Grade 2 Braille. Grade 2 Braille uses many contractions in order to save trees and to speed reading. For example, if letter codes appear by themselves, they stand for common words. The following three rows (including a “completed” third row) show these word codes:

 (none) but can do every from go have (none) just

 knowledge like more not (none) people quite rather so that











 us very it you as and for of the with

Thus, the phrase “you and me” can be written in Grade 2 Braille as this:



So far, I’ve described 31 codes—the no-raised-dots space between words and the 3 rows of 10 codes for letters and words. We’re still not close to the 64 codes that are theoretically available. In Grade 2 Braille, as we shall see, nothing is wasted.

First, we can use the codes for letters *a* through *j* combined with a raised dot 6. These are used mostly for contractions of letters within words and also include *w* and another word abbreviation:

 ch gh sh th wh ed er ou ow w
 (or “will”)

For example, the word “about” can be written in Grade 2 Braille this way:



Second, we can take the codes for letters *a* through *j* and “lower” them to use only dots 2, 3, 5, and 6. These codes are used for some punctuation marks and contractions, depending on context:

ea	bb	cc	dis	en	to	gg	his	in	was
,	;	:	.		!	()	“	”	”

The first four of these codes are the comma, semicolon, colon, and period. Notice that the same code is used for both left and right parentheses but that two different codes are used for open and closed quotation marks.

We're up to 51 codes so far. The following 6 codes use various unused combinations of dots 3, 4, 5, and 6 to represent contractions and some additional punctuation:

st	ing	ble	ar	'	com
/		#			,

The code for “ble” is very important because when it's not part of a word, it means that the codes that follow should be interpreted as numbers. These number codes are the same as those for letters *a* through *j*:

1	2	3	4	5	6	7	8	9	0

Thus, this sequence of codes means the number 256.

If you've been keeping track, we need 7 more codes to reach the maximum of 64. Here they are:

--	--	--	--	--	--	--

The first (a raised dot 4) is used as an accent indicator. The others are used as prefixes for some contractions and also for some other purposes: When dots 4 and 6 are raised (the fifth code in this row), the code is a decimal point in numbers or an emphasis indicator, depending on context. When dots 5 and 6 are raised, the code is a letter indicator that counterbalances a

number indicator.

And finally (if you've been wondering how Braille encodes capital letters) we have dot 6—the capital indicator. This signals that the letter that follows is uppercase. For example, we can write the name of the original creator of this system as



This is a capital indicator, the letter l, the contraction ou, the letters i and s, a space, another capital indicator, and the letters b, r, a, i, l, l, and e. (In actual use, the name might be abbreviated even more by eliminating the last two letters, which aren't pronounced.)

In summary, we've seen how six binary elements (the dots) yield 64 possible codes and no more. It just so happens that many of these 64 codes perform double duty depending on their context. Of particular interest is the number indicator and the letter indicator that undoes the number indicator. These codes alter the meaning of the codes that follow them—from letters to numbers and from numbers back to letters. Codes such as these are often called *precedence*, or *shift*, codes. They alter the meaning of all subsequent codes until the shift is undone.

The capital indicator means that the following letter (and only the following letter) should be uppercase rather than lowercase. A code such as this is known as an *escape* code. Escape codes let you “escape” from the humdrum, routine interpretation of a sequence of codes and move to a new interpretation. As we'll see in later chapters, shift codes and escape codes are common when written languages are represented by binary codes.

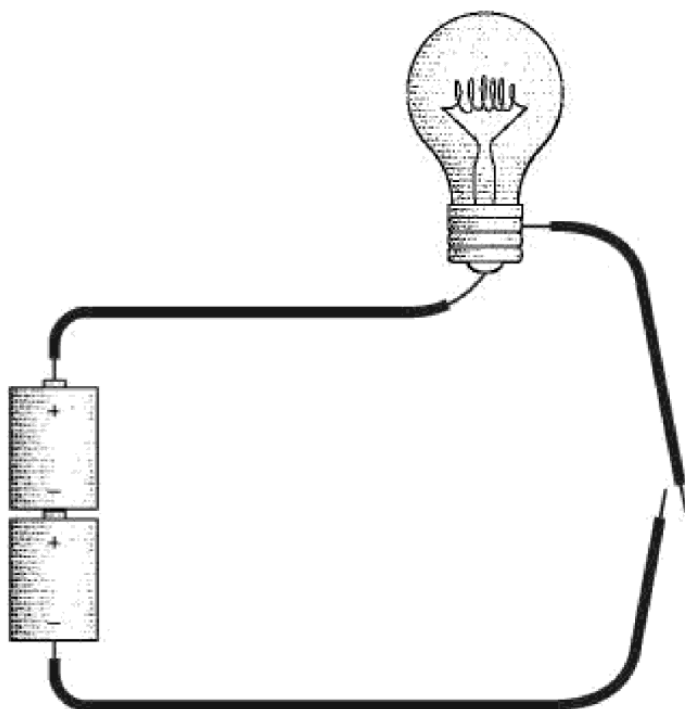
Chapter 4. Anatomy of a Flashlight

Flashlights are useful for numerous tasks, of which reading under the covers and sending coded messages are only the two most obvious. The common household flashlight can also take center stage in an educational show-and-tell of the magical stuff known as electricity.

Electricity is an amazing phenomenon, managing to be pervasively useful while remaining largely mysterious, even to people who pretend to know how it works. But I'm afraid we must wrestle with electricity anyway. Fortunately, we need to understand only a few basic concepts to comprehend how it's used inside computers.

The flashlight is certainly one of the simpler electrical appliances found in most homes. Disassemble a typical flashlight, and you'll find it consists of a couple of batteries, a bulb, a switch, some metal pieces, and a plastic case to hold everything together.

You can make your own no-frills flashlight by disposing of everything except the batteries and the lightbulb. You'll also need some short pieces of insulated wire (with the insulation stripped from the ends) and enough hands to hold everything together.



Notice the two loose ends of the wires at the right of the diagram. That's our switch. Assuming that the batteries are good and the bulb isn't burned out, touching these loose ends together will turn on the light.

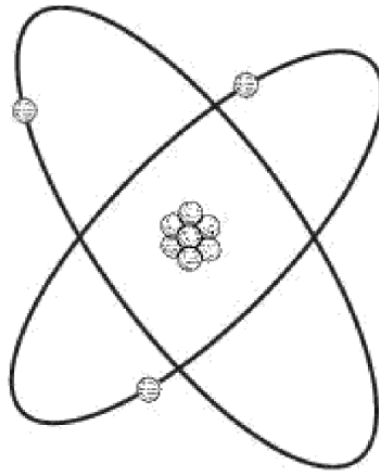
What we've constructed here is a simple electrical circuit, and the first thing to notice is that a *circuit* is a *circle*. The lightbulb will be lit only if the path from the batteries to the wire to the bulb to the switch and back to the batteries is continuous. Any break in this circuit will cause the bulb to go out. The purpose of the switch is to control this process.

The circular nature of the electrical circuit suggests that something is moving around the circuit, perhaps like water flowing through pipes. The "water and pipes" analogy is quite common in explanations of how electricity works, but eventually it breaks down, as all analogies must. Electricity is like nothing else in this universe, and we must confront it on its own terms.

The prevailing scientific wisdom regarding the workings of electricity is called the *electron theory*, which says that electricity derives from the movement of electrons.

As we know, all matter—the stuff that we can see and feel (usually)—is made up of extremely small things called atoms. Every atom is composed of three types of particles; these are called neutrons, protons, and electrons. You can picture an atom as a little solar system, with the neutrons and protons

bound into a nucleus and the electrons spinning around the nucleus like planets around a sun:



I should mention that this isn't exactly what you'd see if you were able to get a microscope powerful enough to see actual atoms, but it works as a convenient model.

The atom shown on the preceding page has 3 electrons, 3 protons, and 4 neutrons, which means that it's an atom of lithium. Lithium is one of 112 known *elements*, each of which has a particular *atomic number* ranging from 1 to 112. The atomic number of an element indicates the number of protons in the nucleus of each of the element's atoms and also (usually) the number of electrons in each atom. The atomic number of lithium is 3.

Atoms can chemically combine with other atoms to form *molecules*. Molecules usually have very different properties from the atoms they comprise. For example, water is composed of molecules that consist of two atoms of hydrogen and one atom of oxygen (hence, H_2O). Obviously water is appreciably different from either hydrogen or oxygen. Likewise, the molecules of table salt consist of an atom of sodium and an atom of chlorine, neither of which would be particularly appetizing on French fries.

Hydrogen, oxygen, sodium, and chlorine are all elements. Water and salt are called *compounds*. Salt water, however, is a *mixture* rather than a compound because the water and the salt maintain their own properties.

The number of electrons in an atom is usually the same as the number of protons. But in certain circumstances, electrons can be dislodged from atoms. That's how electricity happens.

The words *electron* and *electricity* both derive from the ancient Greek word

ηλεκτρον (*elektron*), which you might expect means something like “little tiny invisible thing.” But no—ηλεκτρον is actually the Greek word for “amber,” which is the glasslike hardened sap of trees. The reason for this unlikely derivation is that the ancient Greeks experimented with rubbing amber with wool, which produces something we now call static electricity. Rubbing wool on amber causes the wool to pick up electrons from the amber. The wool winds up with more electrons than protons, and the amber ends up with fewer electrons than protons. In more modern experiments, carpeting picks up electrons from the soles of our shoes.

Protons and electrons have a characteristic called *charge*. Protons are said to have a positive (+) charge and electrons are said to have a negative (–) charge. Neutrons are neutral and have no charge. But even though we use plus and minus signs to denote protons and electrons, the symbols don’t really mean plus and minus in the arithmetical sense or that protons have something that electrons don’t. The use of these symbols just means that protons and electrons are opposite in some way. This opposite characteristic manifests itself in how protons and electrons relate to each other.

Protons and electrons are happiest and most stable when they exist together in equal numbers. An imbalance of protons and electrons will attempt to correct itself. When the carpet picks up electrons from your shoes, eventually everything gets evened out when you touch something and feel a spark. That spark of static electricity is the movement of electrons by a rather circuitous route from the carpet through your body back to your shoes.

Another way to describe the relationship between protons and electrons is to note that opposite charges attract and like charges repel. But this isn’t what we might assume by looking at the diagram of the atom. It looks like the protons huddled together in the nucleus are attracting each other. The protons are held together by something stronger than the repulsion of like charges, and that something is called the *strong force*. Messing around with the strong force involves splitting the nucleus, which produces nuclear energy. In this chapter, we’re merely fooling around with the electrons to get electricity.

Static electricity isn’t limited to the little sparks produced by fingers touching doorknobs. During storms, the bottoms of clouds accumulate electrons while the tops of clouds lose electrons; eventually, the imbalance is evened out with a stroke of lightning. Lightning is a lot of electrons moving very quickly from one spot to another.

The electricity in the flashlight circuit is obviously much better mannered than a spark or a lightning bolt. The light burns steadily and continuously because the electrons aren't just jumping from one place to another. As one atom in the circuit loses an electron to another atom nearby, it grabs another electron from an adjacent atom, which grabs an electron from another adjacent atom, and so on. The electricity in the circuit is the passage of electrons from atom to atom.

This doesn't happen all by itself. We can't just wire up any old bunch of stuff and expect some electricity to happen. We need something to precipitate the movement of electrons around the circuit. Looking back at our diagram of the no-frills flashlight, we can safely assume that the thing that begins the movement of electricity is not the wires and not the lightbulb, so it's probably the batteries.

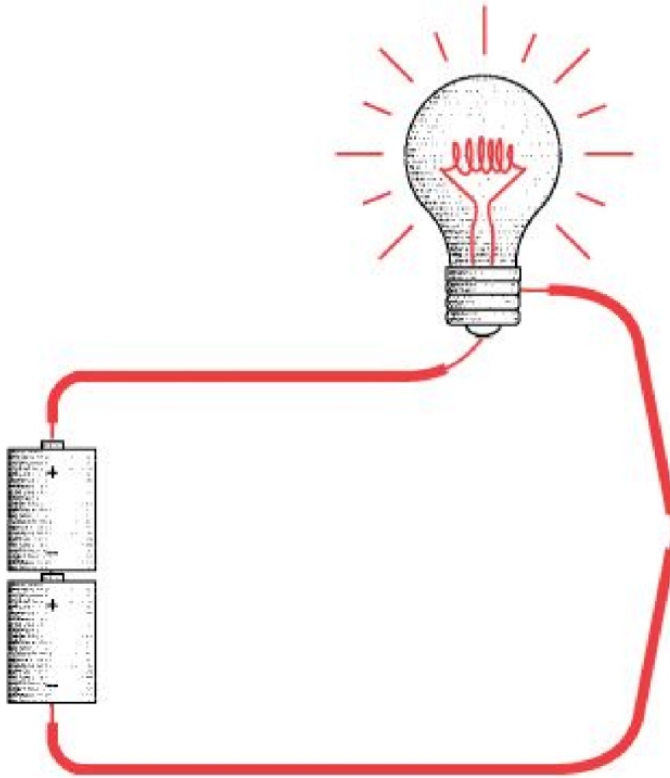
Almost everybody knows a few things about the types of batteries used in flashlights:

- They're tubular in shape and come in different sizes, such as D, C, A, AA, and AAA.
- Regardless of the battery's size, they're all labeled "1.5 volts."
- One end of the battery is flat and is labeled with a minus sign (-); the other end has a little protrusion and is labeled with a plus sign (+).
- If you want your appliance to work right, it's a good idea to install the batteries correctly with the plus signs facing the right way.
- Batteries wear out eventually. Sometimes they can be recharged, sometimes not.
- And finally, we suspect that in some weird way, batteries produce electricity.

In all batteries, chemical reactions take place, which means that some molecules break down into other molecules, or molecules combine to form new molecules. The chemicals in batteries are chosen so that the reactions between them generate spare electrons on the side of the battery marked with a minus sign (called the negative terminal, or *anode*) and demand extra electrons on the other side of the battery (the positive terminal, or *cathode*). In this way, chemical energy is converted to electrical energy.

The chemical reaction can't proceed unless there's some way that the extra

electrons can be taken away from the negative terminal of the battery and delivered back to the positive terminal. So if the battery isn't connected to anything, nothing much happens. (Actually the chemical reactions still take place, but very slowly.) The reactions take place only if an electrical circuit is present to take electrons away from the negative side and supply electrons to the positive side. The electrons travel around this circuit in a counterclockwise direction:

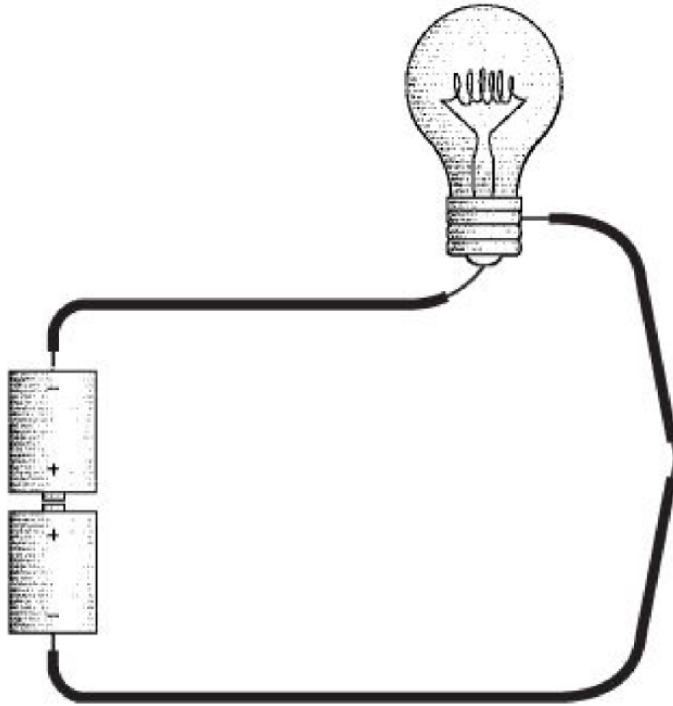


In this book, the color red is used to indicate that electricity is flowing through the wires.

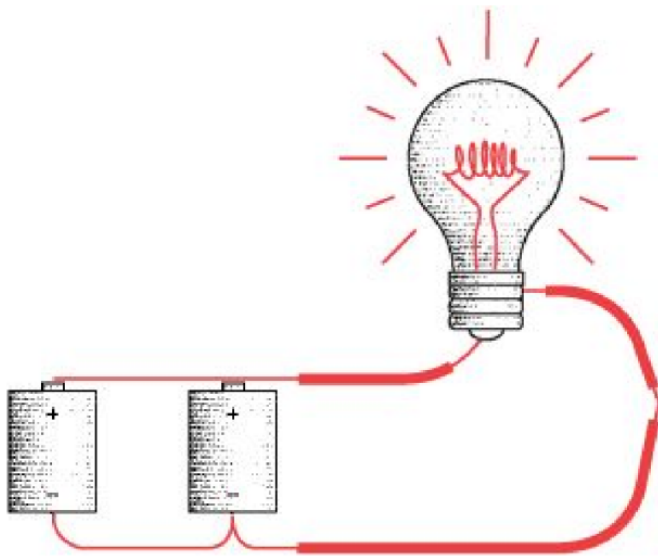
Electrons from the chemicals in the batteries might not so freely mingle with the electrons in the copper wires if not for a simple fact: All electrons, wherever they're found, are identical. There's nothing that distinguishes a copper electron from any other electron.

Notice that both batteries are facing the same direction. The positive end of the bottom battery takes electrons from the negative end of the top battery. It's as if the two batteries have been combined into one bigger battery with a positive terminal at one end and a negative terminal at the other end. The combined battery is 3 volts rather than 1.5 volts.

If we turn one of the batteries upside down, the circuit won't work:



The two positive ends of the battery need electrons for the chemical reactions, but there's no way electrons can get to them because they're attached to each other. If the two positive ends of the battery are connected, the two negative ends should be also:



This works. The batteries are said to be connected *in parallel* rather than *in series* as shown earlier. The combined voltage is 1.5 volts, which is the same as the voltage of each of the batteries. The light will probably still glow, but not as brightly as with two batteries in series. But the batteries will last twice as long.

We normally like to think of a battery as providing electricity to a circuit. But we've seen that we can also think of a circuit as providing a way for a battery's chemical reactions to take place. The circuit takes electrons away from the negative end of the battery and delivers them to the positive end of the battery. The reactions in the battery proceed until all the chemicals are exhausted, at which time you throw away the battery or recharge it.

From the negative end of the battery to the positive end of the battery, the electrons flow through the wires and the lightbulb. But why do we need the wires? Can't the electricity just flow through the air? Well, yes and no. Yes, electricity can flow through air (particularly wet air), or else we wouldn't see lightning. But electricity doesn't flow through air very readily.

Some substances are significantly better than others for carrying electricity. The ability of an element to carry electricity is related to its subatomic structure. Electrons orbit the nucleus in various levels, called shells. An atom that has just one electron in its outer shell can readily give up that electron, which is what's necessary to carry electricity. These substances are conducive to carrying electricity and thus are said to be *conductors*. The best conductors are copper, silver, and gold. It's no coincidence that these three elements are found in the same column of the periodic table. Copper is the most common substance for making wires.

The opposite of conductance is *resistance*. Some substances are more resistant to the passage of electricity than others, and these are known as *resistors*. If a substance has a very high resistance—meaning that it doesn't conduct electricity much at all—it's known as an *insulator*. Rubber and plastic are good insulators, which is why these substances are often used to coat wires. Cloth and wood are also good insulators as is dry air. Just about anything will conduct electricity, however, if the voltage is high enough.

Copper has a very low resistance, but it still has *some* resistance. The longer a wire, the higher the resistance it has. If you tried wiring a flashlight with wires that were miles long, the resistance in the wires would be so high that the flashlight wouldn't work.

The thicker a wire, the lower the resistance it has. This may be somewhat counterintuitive. You might imagine that a thick wire requires much more electricity to "fill it up." But actually the thickness of the wire makes available many more electrons to move through the wire.

I've mentioned voltage but haven't defined it. What does it mean when a battery has 1.5 volts? Actually, voltage—named after Count Alessandro Volta (1745–1827), who invented the first battery in 1800—is one of the

more difficult concepts of elementary electricity. Voltage refers to a *potential* for doing work. Voltage exists whether or not something is hooked up to a battery.

Consider a brick. Sitting on the floor, the brick has very little potential. Held in your hand four feet above the floor, the brick has more potential. All you need do to realize this potential is drop the brick. Held in your hand at the top of a tall building, the brick has much more potential. In all three cases, you're holding the brick and it's not doing anything, but the *potential* is different.

A much easier concept in electricity is the notion of *current*. Current is related to the number of electrons actually zipping around the circuit. Current is measured in *amperes*, named after André Marie Ampère (1775–1836), but everybody calls them *amps*, as in “a 10-amp fuse.” To get one amp of current, you need 6,240,000,000,000,000 electrons flowing past a particular point per second.

The water-and-pipes analogy helps out here: Current is similar to the *amount* of water flowing through a pipe. Voltage is similar to the water *pressure*. Resistance is similar to the width of a pipe—the smaller the pipe, the larger the resistance. So the more water pressure you have, the more water that flows through the pipe. The smaller the pipe, the less water that flows through it. The amount of water flowing through a pipe (the current) is directly proportional to the water pressure (the voltage) and inversely proportional to the skinniness of the pipe (the resistance).

In electricity, you can calculate how much current is flowing through a circuit if you know the voltage and the resistance. Resistance—the tendency of a substance to impede the flow of electrons—is measured in *ohms*, named after Georg Simon Ohm (1789–1854), who also proposed the famous Ohm's Law. The law states

$$I = E / R$$

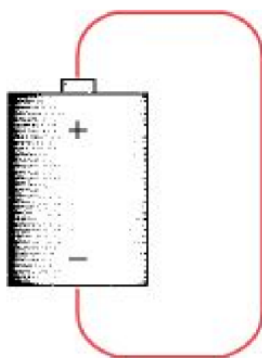
where I is traditionally used to represent current in amperes, E is used to represent voltage (it stands for *electromotive force*), and R is resistance.

For example, let's look at a battery that's just sitting around not connected to anything:



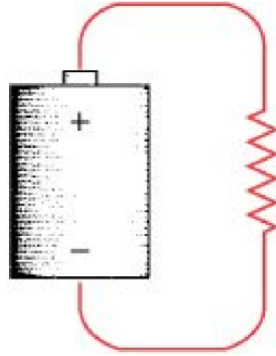
The voltage E is 1.5. That's a potential for doing work. But because the positive and negative terminals are connected solely by air, the resistance (the symbol R) is very, very, very high, which means the current (I) equals 1.5 volts divided by a large number. This means that the current is just about zero.

Now let's connect the positive and negative terminals with a short piece of copper wire (and from here on, the insulation on the wires won't be shown):



This is known as a *short circuit*. The voltage is still 1.5, but the resistance is now very, very low. The current is 1.5 volts divided by a very small number. This means that the current will be very, very high. Lots and lots of electrons will be flowing through the wire. In reality, the actual current will be limited by the physical size of the battery. The battery will probably not be able to deliver such a high current, and the voltage will drop below 1.5 volts. If the battery is big enough, the wire will get hot because the electrical energy is being converted to heat. If the wire gets very hot, it will actually glow and might even melt.

Most circuits are somewhere between these two extremes. We can symbolize them like so:



The squiggly line is recognizable to electrical engineers as the symbol for a resistor. Here it means that the circuit has a resistance that is neither very low nor very high.

If a wire has a low resistance, it can get hot and start to glow. This is how an incandescent lightbulb works. The lightbulb is commonly credited to America's most famous inventor, Thomas Alva Edison (1847–1931), but the concepts were well known at the time he patented the lightbulb (1879) and many other inventors also worked on the problem.

Inside a lightbulb is a thin wire called a filament, which is commonly made of tungsten. One end of the filament is connected to the tip at the bottom of the base; the other end of the filament is connected to the side of the metal base, separated from the tip by an insulator. The resistance of the wire causes it to heat up. In open air, the tungsten would get hot enough to burn, but in the vacuum of the lightbulb, the tungsten glows and gives off light.

Most common flashlights have two batteries connected in series. The total voltage is 3.0 volts. A lightbulb of the type commonly used in a flashlight has a resistance of about 4 ohms. Thus, the current is 3 volts divided by 4 ohms, or 0.75 ampere, which can also be expressed as 750 milliamperes. This means that 4,680,000,000,000,000 electrons are flowing through the lightbulb every second.

(A brief reality check: If you actually try to measure the resistance of a flashlight lightbulb with an ohmmeter, you'll get a reading much lower than 4 ohms. The resistance of tungsten is dependent upon its temperature, and the resistance gets higher as the bulb heats up.)

As you may know, lightbulbs you buy for your home are labeled with a certain wattage. The watt is named after James Watt (1736–1819), who is best known for his work on the steam engine. The watt is a measurement of power (P) and can be calculated as

$$P = E \times I$$

The 3 volts and 0.75 amp of our flashlight indicate that we're dealing with a 2.25-watt lightbulb.

Your home might be lit by 100-watt lightbulbs. These are designed for the 120 volts of your home. Thus, the current that flows through them is equal to 100 watts divided by 120 volts, or about 0.83 ampere. Hence, the resistance of a 100-watt lightbulb is 120 volts divided by 0.83 ampere, or 144 ohms.

So we've seemingly analyzed everything about the flashlight—the batteries, the wires, and the lightbulb. But we've forgotten the most important part!

Yes, the switch. The switch controls whether electricity is flowing in the circuit or not. When a switch allows electricity to flow, it is said to be *on*, or *closed*. An *off*, or *open*, switch doesn't allow electricity to flow. (The way we use the words *closed* and *open* for switches is opposite to the way we use them for a door. A closed door prevents anything from passing through it; a closed switch allows electricity to flow.)

Either the switch is closed or it's open. Either current flows or it doesn't. Either the lightbulb lights up or it doesn't. Like the binary codes invented by Morse and Braille, this simple flashlight is either on or off. There's no in-between. This similarity between binary codes and simple electrical circuits is going to prove very useful in the chapters ahead.

Chapter 5. Seeing Around Corners

You're twelve years old. One horrible day your best friend's family moves to another town. You speak to your friend on the telephone now and then, but telephone conversations just aren't the same as those late-night sessions with the flashlights blinking out Morse code. Your second-best friend, who lives in the house next door to yours, eventually becomes your new best friend. It's time to teach your new best friend some Morse code and get the late-night flashlights blinking again.

The problem is, your new best friend's bedroom window doesn't face your bedroom window. The houses are side by side, but the bedroom windows face the same direction. Unless you figure out a way to rig up a few mirrors outside, the flashlights are now inadequate for after-dark communication.

Or are they?

Maybe you have learned something about electricity by this time, so you decide to make your own flashlights out of batteries, lightbulbs, switches, and wires. In the first experiment, you wire up the batteries and switch in your bedroom. Two wires go out your window, across a fence, and into your friend's bedroom, where they're connected to a lightbulb:



Although I'm showing only one battery, you might actually be using two. In this and future diagrams, this will be an off (or open) switch:



and this will be the switch when it's on (or closed):

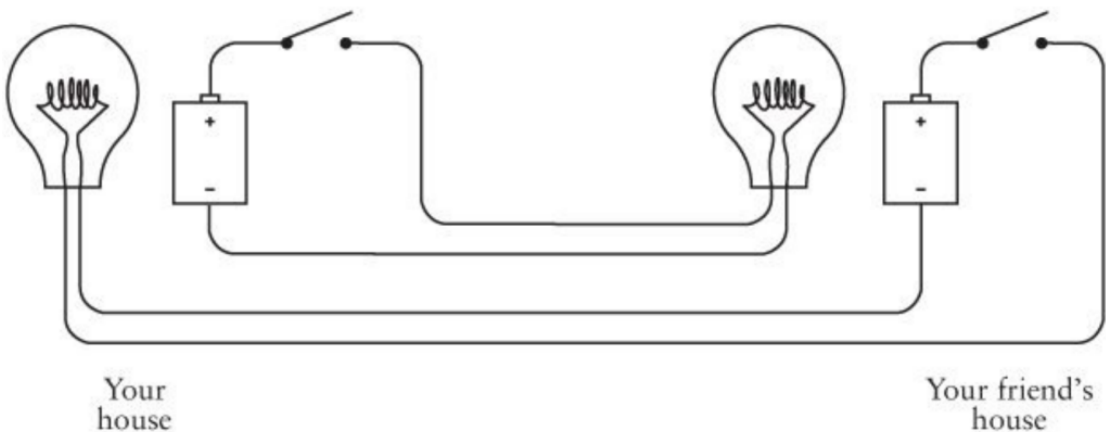


The flashlight in this chapter works the same way as the one illustrated in the previous chapter, although the wires connecting the components for this chapter's flashlight are a bit longer. When you close the switch at your end, the light goes on at your friend's end:



Now you can send messages using Morse code.

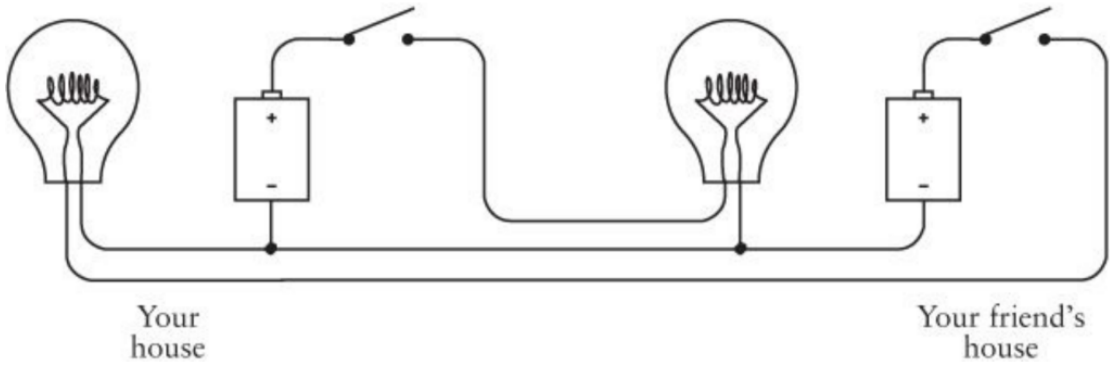
Once you have one flashlight working, you can wire another long-distance flashlight so that your friend can send messages to you:



Congratulations! You have just rigged up a bidirectional telegraph system. You'll notice that these are two identical circuits that are entirely independent of and unconnected to each other. In theory, you can be sending a message to your friend while your friend is sending a message to

you (although it might be hard for your brain to read and send messages at the same time).

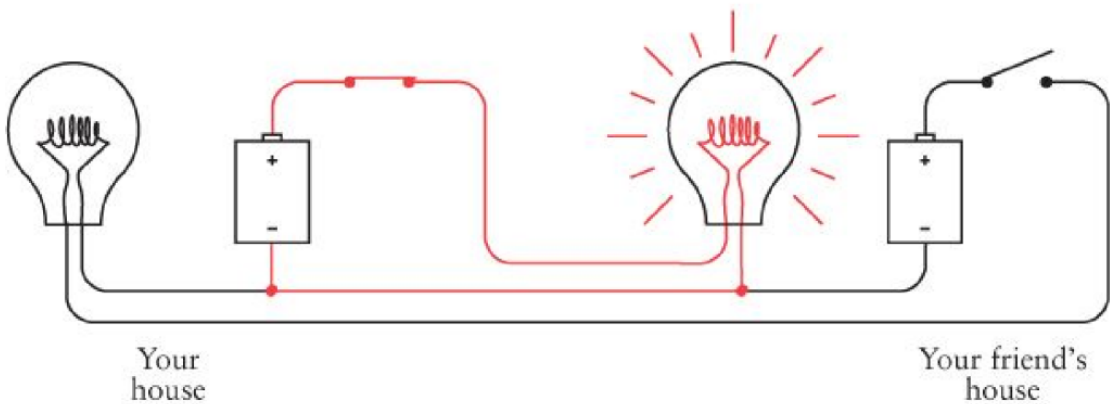
You also might be clever enough to discover that you can reduce your wire requirements by 25 percent by wiring the configuration this way:



Notice that the negative terminals of the two batteries are now connected. The two circular circuits (battery to switch to bulb to battery) still operate independently, even though they're now joined like Siamese twins.

This connection is called a *common*. In this circuit the common extends from the point where the leftmost lightbulb and battery are connected to the point where the rightmost lightbulb and battery are connected. These connections are indicated by dots.

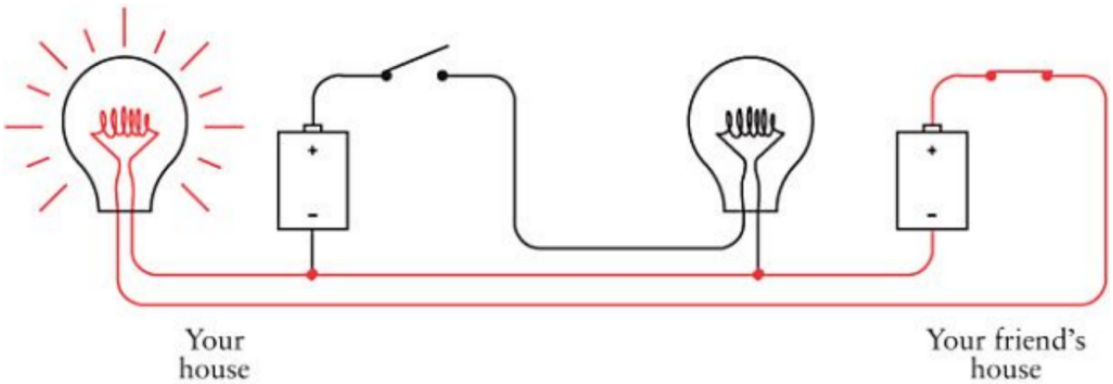
Let's take a closer look to assure ourselves that nothing funny is going on. First, when you depress the switch on your side, the bulb in your friend's house lights up. The red wires show the flow of electricity in the circuit:



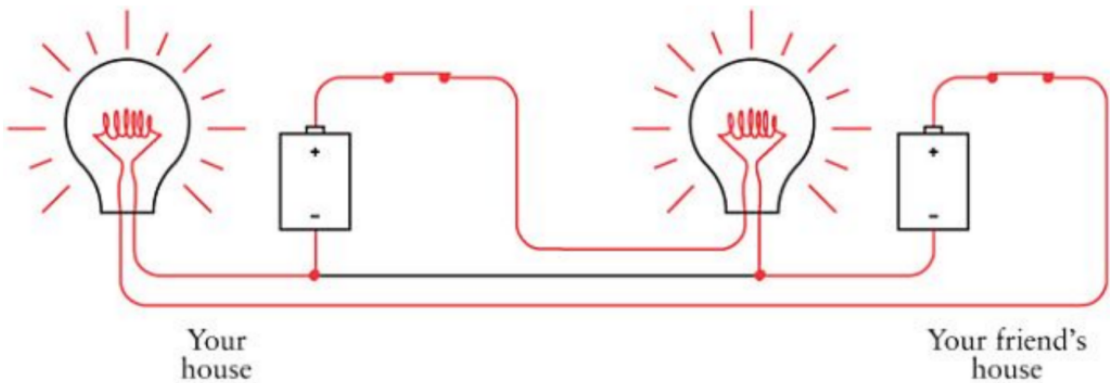
No electricity flows in the other part of the circuit because there's no place for the electrons to go to complete a circuit.

When you're not sending but your friend is sending, the switch in your

friend's house controls the lightbulb in your house. Once again, the red wires show how electricity flows in the circuit:



When you and your friend both try to send at the same time, sometimes both switches are open, sometimes one switch is closed but the other is open, and sometimes both switches are depressed. In that case, the flow of electricity in the circuit looks like this:



No current flows through the common part of the circuit.

By using a common to join two separate circuits into one circuit, we've reduced the electrical connection between the two houses from four wires to three wires and reduced our wire expenses by 25 percent.

If we had to string the wires for a very long distance, we might be tempted to reduce our expenses even more by eliminating another wire. Unfortunately, this isn't feasible with 1.5-volt D cells and small lightbulbs. But if we were dealing with 100-volt batteries and much larger lightbulbs, it could certainly be done.

Here's the trick: Once you have established a common part of the circuit, you don't have to use wire for it. You can replace the wire with something

else. And what you can replace it with is a giant sphere approximately 7900 miles in diameter made up of metal, rock, water, and organic material, most of which is dead. The giant sphere is known to us as Earth.

When I described good conductors in the last chapter, I mentioned silver, copper, and gold, but not gravel and mulch. In truth, the earth isn't such a hot conductor, although some kinds of earth (damp soil, for example) are better than others (such as dry sand). But one thing we learned about conductors is this: The larger the better. A very thick wire conducts much better than a very thin wire. That's where the earth excels. It's really, really, really big.

To use the earth as a conductor, you can't merely stick a little wire into the ground next to the tomato plants. You have to use something that maintains a substantial contact with the earth, and by that I mean a conductor with a large surface area. One good solution is a copper pole at least 8 feet long and $\frac{1}{2}$ inch in diameter. That provides 150 square inches of contact with the earth. You can bury the pole into the ground with a sledgehammer and then connect a wire to it. Or, if the cold-water pipes in your home are made of copper and originate in the ground outside the house, you can connect a wire to the pipe.

An electrical contact with the earth is called an *earth* in Great Britain and a *ground* in America. A bit of confusion surrounds the word *ground* because it's also often used to refer to a part of a circuit we've been calling the *common*. In this chapter, and until I indicate otherwise, a ground is a physical connection with the earth.

When people draw electrical circuits, they use this symbol to represent a ground:



Electricians use this symbol because they don't like to take the time to draw an 8-foot copper pole buried in the ground.

Let's see how this works. We began this chapter by looking at a one-way configuration like this:



If you were using high-voltage batteries and lightbulbs, you would need only one wire between your house and your friend's house because you could use the earth as one of the connectors:



When you turn the switch on, electricity flows like this:



The electrons come out of the earth at your friend's house, go through the lightbulb and wire, the switch at your house, and then go into the positive terminal of the battery. Electrons from the negative terminal of the battery go into the earth.

You might also want to visualize electrons leaping from the 8-foot copper pole buried in the backyard of your house into the earth, then scurrying through the earth to get to the 8-foot copper pole buried in the backyard of

your friend's house.

But if you consider that the earth is performing this same function for many thousands of electrical circuits around the world, you might ask: How do the electrons know where to go? Well, obviously they don't. A different image of the earth seems much more appropriate.

Yes, the earth is a massive conductor of electricity, but it can also be viewed as both a source of and a repository for electrons. *The earth is to electrons as an ocean is to drops of water.* The earth is a virtually limitless source of electrons and also a giant sink for electrons.

The earth, however, does have *some* resistance. That's why we can't use the earth ground to reduce our wiring needs if we're playing around with 1.5-volt D cells and flashlight bulbs. The earth simply has too much resistance for low-voltage batteries.

You'll notice that the previous two diagrams include a battery with the negative terminal connected to the ground:



I'm not going to draw this battery connected to the ground anymore. Instead, I'm going to use the capital letter V, which stands for *voltage*. The one-way lightbulb telegraph now looks like this:

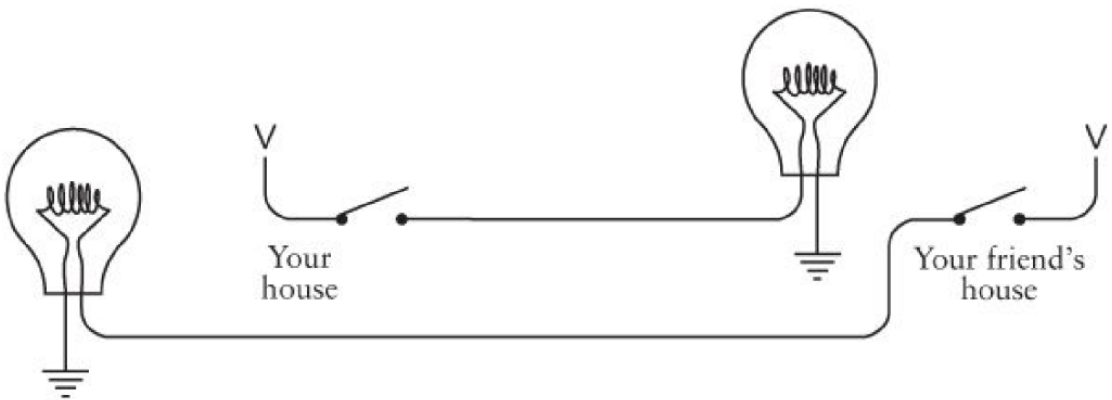


The *V* stands for *voltage*, but it could also stand for *vacuum*. Think of the *V* as an electron vacuum and think of the ground as an ocean of electrons. The electron vacuum pulls the electrons from the earth through the circuit, doing work along the way (such as lighting a lightbulb).

The ground is sometimes also known as the point of *zero potential*. This means that no voltage is present. A voltage—as I explained earlier—is a potential for doing work, much as a brick suspended in the air is a potential source of energy. Zero potential is like a brick sitting on the ground—there’s no place left for it to fall.

In **Chapter 4**, one of the first things we noticed was that circuits were circles. Our new circuit doesn’t look like a circle at all. It still is one, however. You could replace the *V* with a battery with the negative terminal connected to ground, and then you could draw a wire connecting all the places you see a ground symbol. You’d end up with the same diagram that we started with in this chapter.

So with the help of a couple of copper poles (or cold-water pipes), we can construct a two-way Morse code system with just two wires crossing the fence between your house and your friend’s:



This circuit is functionally the same as the configuration shown previously, in which three wires crossed the fence between the houses.

In this chapter, we’ve taken an important step in the evolution of communications. Previously we had been able to communicate with Morse code but only in a straight line of sight and only as far as the beam from a flash-light would travel.

By using wires, not only have we constructed a system to communicate around corners beyond the line of sight, but we’ve freed ourselves of the

limitation of distance. We can communicate over hundreds and thousands of miles just by stringing longer and longer wires.

Well, not exactly. Although copper is a very good conductor of electricity, it's not perfect. The longer the wires, the more resistance they have. The more resistance, the less current that flows. The less current, the dimmer the lightbulbs.

So how long exactly can we make the wires? That depends. Let's suppose you're using the original four-wire, bidirectional hookup without grounds and commons, and you're using flashlight batteries and lightbulbs. To keep your costs down, you may have initially purchased some 20-gauge speaker wire from Radio Shack at \$9.99 per 100 feet. Speaker wire is normally used to connect your speakers to your stereo system. It has two conductors, so it's also a good choice for our telegraph system. If your bedroom and your friend's bedroom are less than 50 feet apart, this one roll of wire is all you need.

The thickness of wire is measured in *American Wire Gauge*, or AWG. The smaller the AWG number, the thicker the wire and also the less resistance it has. The 20-gauge speaker wire you bought has a diameter of about 0.032 inches and a resistance of about 10 ohms per 1000 feet, or 1 ohm for the 100-foot round-trip distance between the bedrooms.

That's not bad at all, but what if we strung the wire out for a mile? The total resistance of the wire would be more than 100 ohms. Recall from the last chapter that our lightbulb was only 4 ohms. From Ohm's Law, we can easily calculate that the current through the circuit will no longer be 0.75 amp (3 volts divided by 4 ohms) as before, but will now be less than 0.03 amp (3 volts divided by more than 100 ohms). Almost certainly, that won't be enough current to light the bulb.

Using thicker wire is a good solution, but that can be expensive. Ten-gauge wire (which Radio Shack sells as Automotive Hookup Wire at \$11.99 for 35 feet, and you'd need twice as much because it has only one conductor rather than two) is about 0.1 inch thick but has a resistance of only 1 ohm per 1000 feet, or 5 ohms per mile.

Another solution is to increase the voltage and use lightbulbs with a much higher resistance. For example, a 100-watt lightbulb that lights a room in your house is designed to be used with 120 volts and has a resistance of about 144 ohms. The resistance of the wires will then affect the overall circuitry much less.

These are problems faced 150 years ago by the people who strung up the first telegraph systems across America and Europe. Regardless of the thickness of the wires and the high levels of voltage, telegraph wires simply couldn't be continued indefinitely. At most, the limit for a working system according to this scheme was a couple hundred miles. That's nowhere close to spanning the thousands of miles between New York and California.

The solution to this problem—not for flashlights but for the clicking and clacking telegraphs of yesteryear—turns out to be a simple and humble device, but one from which entire computers can be built.

Chapter 6. Telegraphs and Relays

Samuel Finley Breese Morse was born in 1791 in Charleston, Massachusetts, the town where the Battle of Bunker Hill was fought and which is now the northeast part of Boston. In the year of Morse's birth, the United States Constitution had been ratified just two years before and George Washington was serving his first term as president. Catherine the Great ruled Russia. Louis XVI and Marie Antoinette would lose their heads two years later in the French Revolution. And in 1791, Mozart completed *The Magic Flute*, his last opera, and died later that year at the age of 35.

Morse was educated at Yale and studied art in London. He became a successful portrait artist. His painting *General Lafayette* (1825) hangs in New York's City Hall. In 1836, he ran for mayor of New York City on an independent ticket and received 5.7 percent of the vote. He was also an early photography buff. Morse learned how to make daguerreotype photographs from Louis Daguerre himself and made some of the first daguerreotypes in America. In 1840, he taught the process to the 17-year-old Mathew Brady, who with his colleagues would be responsible for creating the most memorable photographs of the Civil War, Abraham Lincoln, and Samuel Morse himself.



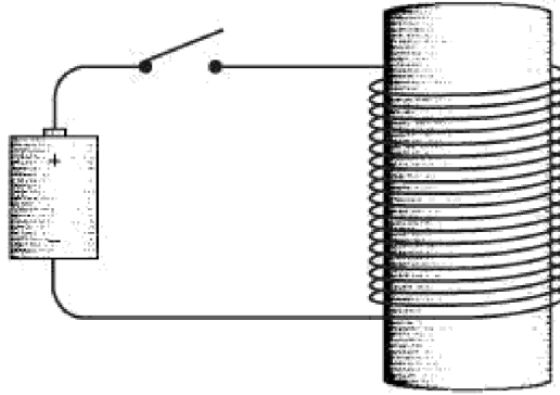
But these are just footnotes to an eclectic career. Samuel F. B. Morse is best known these days for his invention of the telegraph and the code that bears his name.

The instantaneous worldwide communication we've become accustomed to is a relatively recent development. In the early 1800s, you could communicate instantly and you could communicate over long distances, but you couldn't do both at the same time. Instantaneous communication was limited to as far as your voice could carry (no amplification available) or as far as the eye could see (aided perhaps by a telescope). Communication over longer distances by letter took time and involved horses, trains, or ships.

For decades prior to Morse's invention, many attempts were made to speed long-distance communication. Technically simple methods employed a relay system of men standing on hills waving flags in semaphore codes. Technically more complex solutions used large structures with movable arms that did basically the same thing as men waving flags.

The idea of the telegraph (literally meaning "far writing") was certainly in the air in the early 1800s, and other inventors had taken a stab at it before Samuel Morse began experimenting in 1832. In principle, the idea behind an electrical telegraph was simple: You do something at one end of a wire that causes something to happen at the other end of the wire. This is exactly what we did in the last chapter when we made a long-distance flashlight. However, Morse couldn't use a lightbulb as his signaling device because a practical one wouldn't be invented until 1879. Instead, Morse relied upon the phenomenon of *electromagnetism*.

If you take an iron bar, wrap it with a couple hundred turns of thin wire, and then run a current through the wire, the iron bar becomes a magnet. It then attracts other pieces of iron and steel. (There's enough thin wire in the electromagnet to create a resistance great enough to prevent the electromagnet from constituting a short circuit.) Remove the current, and the iron bar loses its magnetism:



The electromagnet is the foundation of the telegraph. Turning the switch on and off at one end causes the electromagnet to do something at the other end.

Morse's first telegraphs were actually more complex than the ones that later evolved. Morse felt that a telegraph system should actually write something on paper, or as computer users would later phrase it, "produce a hard copy." This wouldn't necessarily be words, of course, because that would be too complex. But *something* should be written on paper, whether it be squiggles or dots and dashes. Notice that Morse was stuck in a paradigm that required paper and reading, much like Valentin Haüy's notion that books for the blind should use raised letters of the alphabet.

Although Samuel Morse notified the patent office in 1836 that he had invented a successful telegraph, it wasn't until 1843 that he was able to persuade Congress to fund a public demonstration of the device. The historic day was May 24, 1844, when a telegraph line rigged between Washington, D.C., and Baltimore, Maryland, successfully carried the biblical message: "What hath God wrought!"

The traditional telegraph "key" used for sending messages looked something like this:

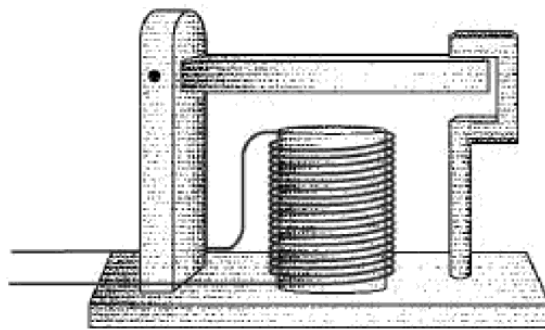


Despite the fancy appearance, this was just a switch designed for maximum speed. The most comfortable way to use the key for long periods of time was to hold the handle between thumb, forefinger, and middle finger, and

tap it up and down. Holding the key down for a short period of time produced a Morse code dot. Holding it down longer produced a Morse code dash.

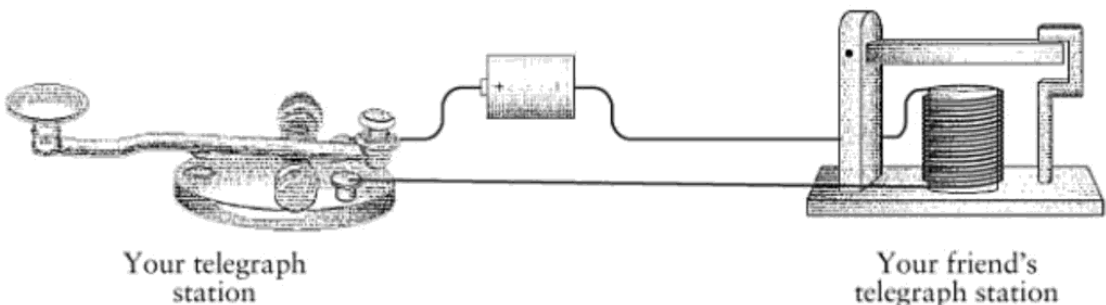
At the other end of the wire was a receiver that was basically an electromagnet pulling a metal lever. Originally, the electromagnet controlled a pen. While a mechanism using a wound-up spring slowly pulled a roll of paper through the gadget, an attached pen bounced up and down and drew dots and dashes on the paper. A person who could read Morse code would then transcribe the dots and dashes into letters and words.

Of course, we humans are a lazy species, and telegraph operators soon discovered that they could transcribe the code simply by listening to the pen bounce up and down. The pen mechanism was eventually eliminated in favor of the traditional telegraph “sounder,” which looked something like this:



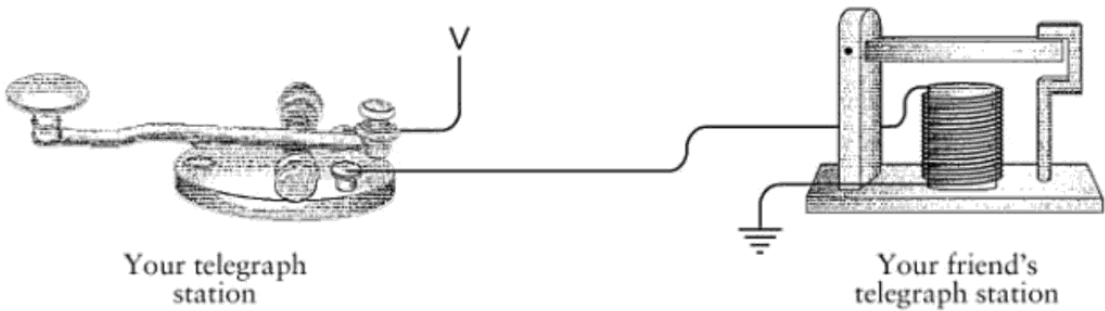
When the telegraph key was pressed, the electromagnet in the sounder pulled the movable bar down and it made a “click” noise. When the key was released, the bar sprang back to its normal position, making a “clack” noise. A fast “click-clack” was a dot; a slower “click...clack” was a dash.

The key, the sounder, a battery, and some wires can be connected just like the lightbulb telegraph in the preceding chapter:



As we discovered, you don't need two wires connecting the two telegraph stations. One wire will suffice if the earth provides the other half of the circuit.

As we did in the previous chapter, we can replace the battery connected to the ground with a capital V. So the complete one-way setup looks something like this:



Two-way communication simply requires another key and sender. This is similar to what we did in the preceding chapter.

The invention of the telegraph truly marks the beginning of modern communication. For the first time, people were able to communicate further than the eye could see or the ear could hear and faster than a horse could gallop. That this invention used a binary code is all the more intriguing. In later forms of electrical and wireless communication, including the telephone, radio, and television, binary codes were abandoned, only to later make an appearance in computers, compact discs, digital videodiscs, digital satellite television broadcasting, and high-definition TV.

Morse's telegraph triumphed over other designs in part because it was tolerant of bad line conditions. If you strung a wire between a key and a sounder, it usually worked. Other telegraph systems were not quite as forgiving. But as I mentioned in the last chapter, a big problem with the telegraph lay in the resistance of long lengths of wire. Although some telegraph lines used up to 300 volts and could work over a 300-mile length, wires couldn't be extended indefinitely.

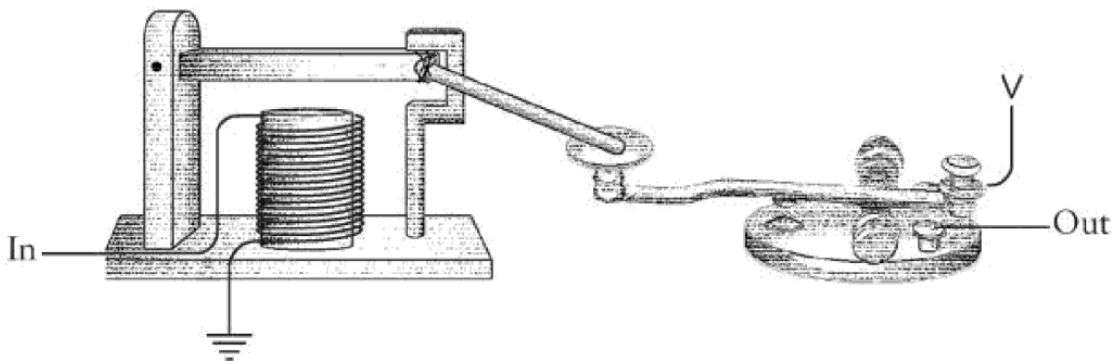
One obvious solution is to have a relay system. Every couple hundred miles or so, a person equipped with a sounder and a key could receive a message and resend it.

Now imagine that you have been hired by the telegraph company to be part of this relay system. They have put you out in the middle of nowhere

between New York and California in a little hut with a table and a chair. A wire coming through the east window is connected to a sounder. Your telegraph key is connected to a battery and wire going out the west window. Your job is to receive messages originating in New York and to resend them, eventually to reach California.

At first, you prefer to receive an entire message before resending it. You write down the letters that correspond to the clicks of the sounder, and when the message is finished, you start sending it using your key. Eventually you get the knack of sending the message as you're hearing it without having to write the whole thing down. This saves time.

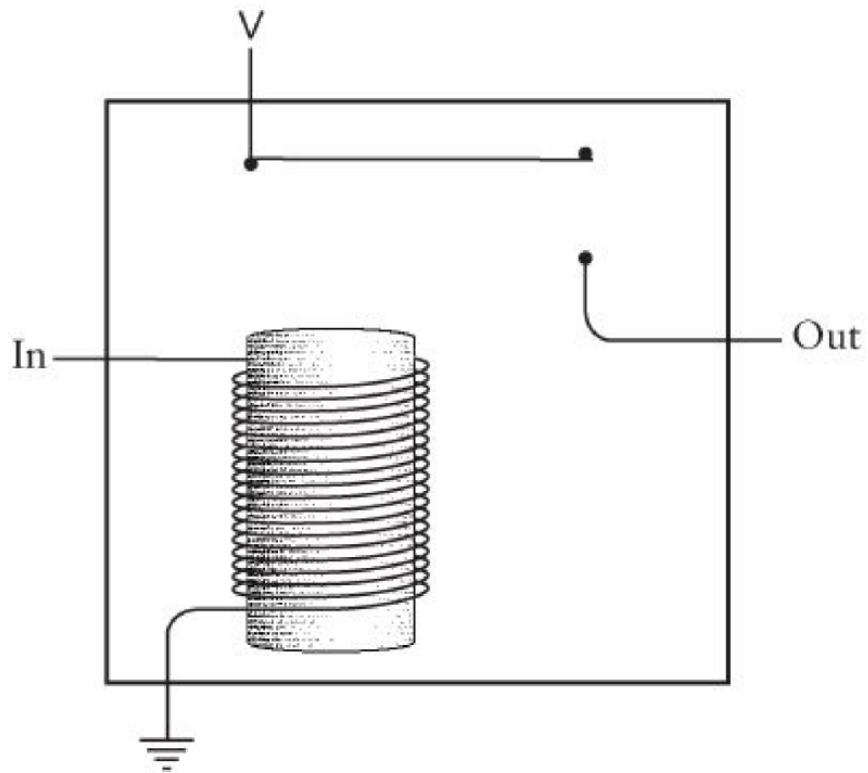
One day while resending a message, you look at the bar on the sounder bouncing up and down, and you look at your fingers bouncing the key up and down. You look at the sounder again and you look at the key again, and you realize that the sounder is bouncing up and down the same way the key is bouncing up and down. So you go outside and pick up a little piece of wood and you use the wood and some string to physically connect the sounder and the key:



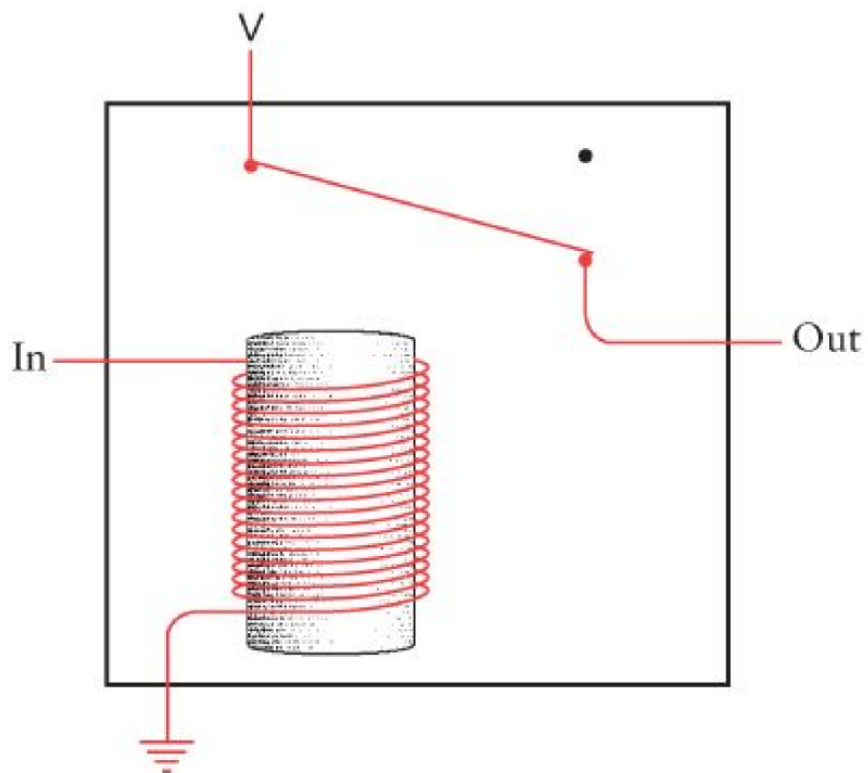
Now it works by itself, and you can take the rest of the afternoon off and go fishing.

It's an interesting fantasy, but in reality Samuel Morse had understood the concept of this device early on. The device we've invented is called a *repeater*, or a *relay*. A relay is like a sounder in that an incoming current is used to power an electromagnet that pulls down a metal lever. The lever, however, is used as part of a switch connecting a battery to an outgoing wire. In this way, a weak incoming current is "amplified" to make a stronger outgoing current.

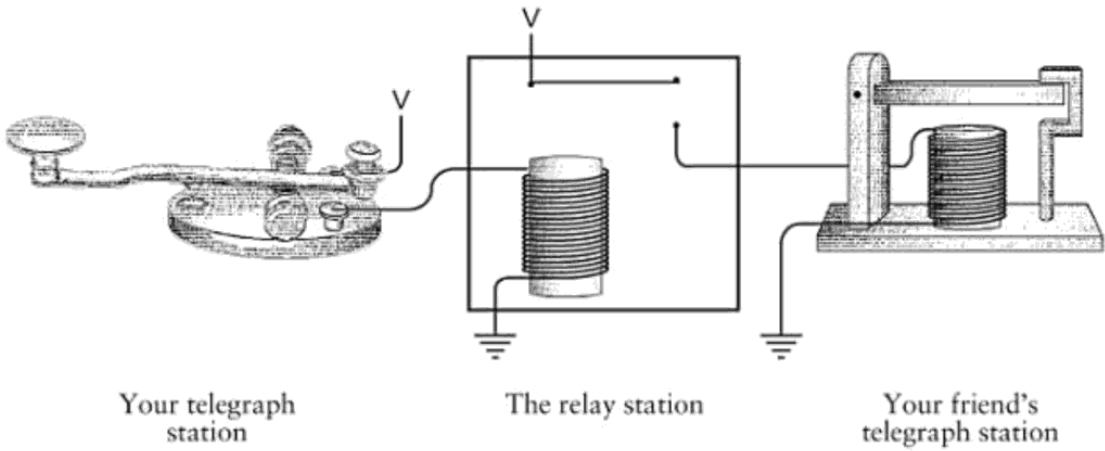
Drawn rather schematically, the relay looks like this:



When an incoming current triggers the electromagnet, the electromagnet pulls down a flexible strip of metal that acts like a switch to turn on an outgoing current:



So a telegraph key, a relay, and a sounder are connected more or less like this:



The relay is a remarkable device. It's a switch, surely, but a switch that's turned on and off not by human hands but by a current. You could do amazing things with such devices. You could actually assemble much of a computer with them.

Yes, this relay thing is much too sweet an invention to leave sitting around the telegraphy museum. Let's grab one and stash it inside our jacket and walk quickly past the guards. This relay will come in very handy. But before we can use it, we're going to have to learn to count.

Chapter 7. Our Ten Digits

The idea that language is merely a code seems readily acceptable. Many of us at least attempted to learn a foreign language in high school, so we're willing to acknowledge that the animal we call a cat in English can also be a *gato*, *chat*, *Katze*, *KOIIIK*, *cat* .

Numbers, however, seem less culturally malleable. Regardless of the language we speak and the way we pronounce the numbers, just about everybody we're likely to come in contact with on this planet writes them the same way:

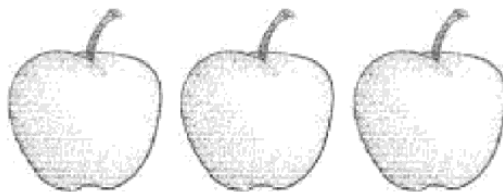
1 2 3 4 5 6 7 8 9 10

Isn't mathematics called "the universal language" for a reason?

Numbers are certainly the most abstract codes we deal with on a regular basis. When we see the number

3

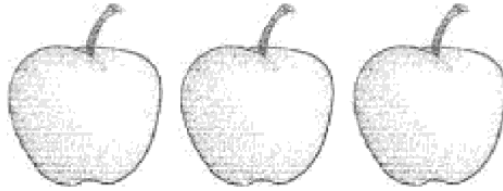
we don't immediately need to relate it to anything. We might visualize 3 apples or 3 of something else, but we'd be just as comfortable learning from context that the number refers to a child's birthday, a television channel, a hockey score, or the number of cups of flour in a cake recipe. Because our numbers are so abstract to begin with, it's more difficult for us to understand that this number of apples



doesn't necessarily have to be denoted by the symbol

3

Much of this chapter and the next will be devoted to persuading ourselves that this many apples



can also be indicated by writing

11

Let's first dispense with the idea that there's something inherently special about the number ten. That most civilizations have based their number systems around ten (or sometimes five) isn't surprising. From the very beginning, people have used their fingers to count. Had our species developed possessing eight or twelve fingers, our ways of counting would be a little different. It's no coincidence that the word *digit* can refer to fingers or toes as well as numbers or that the words *five* and *fist* have similar roots.

So in that sense, using a *base-ten*, or *decimal* (from the Latin for *ten*), number system is completely arbitrary. Yet we endow numbers based on ten with an almost magical significance and give them special names. Ten years is a decade; ten decades is a century; ten centuries is a millennium. A thousand thousands is a million; a thousand millions is a billion. These numbers are all powers of ten:

$$10^1 = 10$$

$$10^2 = 100$$

$$10^3 = 1000 \text{ (thousand)}$$

$$10^4 = 10,000$$

$$10^5 = 100,000$$

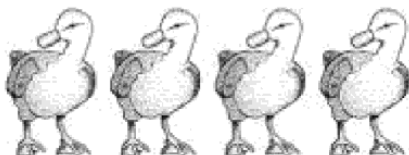
$$10^6 = 1,000,000 \text{ (million)}$$

$$10^7 = 10,000,000$$

$$10^8 = 100,000,000$$

$$10^9 = 1,000,000,000 \text{ (billion)}$$

Most historians believe that numbers were originally invented to count things, such as people, possessions, and transactions in commerce. For example, if someone owned four ducks, that might be recorded with drawings of four ducks:



Eventually the person whose job it was to draw the ducks thought, “Why do I have to draw four ducks? Why can’t I draw one duck and indicate that there are four of them with, I don’t know, a scratch mark or something?”



And then there came the day when someone had 27 ducks, and the scratch marks got ridiculous:



Someone said, “There’s got to be a better way,” and a number system was born.

Of all the early number systems, only Roman numerals are still in common use. You find them on the faces of clocks and watches, used for dates on monuments and statues, for some page numbering in books, for some items in an outline, and—most annoyingly—for the copyright notice in movies. (The question “What year was this picture made?” can often be answered only if one is quick enough to decipher MCMLIII as the tail end of the credits goes by.)

Twenty-seven ducks in Roman numerals is



The concept here is easy enough: The X stands for 10 scratch marks and the V stands for 5 scratch marks.

The symbols of Roman numerals that survive today are

I V X L C D M

The I is a one. This could be derived from a scratch mark or a single raised finger. The V, which is probably a symbol for a hand, stands for five. Two V's make an X, which stands for ten. The L is a fifty. The letter C comes from the word *centum*, which is Latin for a hundred. D is five hundred. Finally, M comes from the Latin word *mille*, or a thousand.

Although we might not agree, for a long time Roman numerals were considered easy to add and subtract, and that's why they survived so long in Europe for bookkeeping. Indeed, when adding two Roman numerals, you simply combine all the symbols from both numbers and then simplify the result using just a few rules: Five I's make a V, two V's make an X, five X's make an L, and so forth.

But multiplying and dividing Roman numerals is difficult. Many other early number systems (such as that of the ancient Greeks) are similarly inadequate for working with numbers in a sophisticated manner. While the Greeks developed an extraordinary geometry still taught virtually unchanged in high schools today, the ancient Greeks aren't known for their algebra.

The number system we use today is known as the Hindu-Arabic or Indo-Arabic. It's of Indian origin but was brought to Europe by Arab mathematicians. Of particular renown is the Persian mathematician Muhammed ibn-Musa al-Khwarizmi (from whose name we have derived the word *algorithm*) who wrote a book on algebra around A.D. 825 that used the Hindu system of counting. A Latin translation dates from A.D. 1120 and was influential in hastening the transition throughout Europe from Roman numerals to our present Hindu-Arabic system.

The Hindu-Arabic number system was different from previous number systems in three ways:

- The Hindu-Arabic number system is said to be *positional*, which means that a particular digit represents a different quantity depending on where it is found in the number. Where digits appear in a number is just as significant (actually, more significant) than what the digits actually are. Both 100 and 1,000,000 have only a single 1 in them, yet we all know that a million is much larger than a hundred.
- Virtually all early number systems have something that the Hindu-Arabic system does *not* have, and that's a special symbol for the number ten. In our number system, there's *no* special symbol for ten.
- On the other hand, virtually all of the early number systems are missing something that the Hindu-Arabic system has, and which turns out to be much more important than a symbol for ten. And that's the zero.

Yes, the zero. The lowly zero is without a doubt one of the most important inventions in the history of numbers and mathematics. It supports positional notation because it allows differentiation of 25 from 205 and 250. The zero also eases many mathematical operations that are awkward in nonpositional systems, particularly multiplication and division.

The whole structure of Hindu-Arabic numbers is revealed in the way we pronounce them. Take 4825, for instance. We say “four thousand, eight hundred, twenty-five.” That means

four thousands

eight hundreds

two tens and

five.

Or we can write the components like this:

$$4825 = 4000 + 800 + 20 + 5$$

Or breaking it down even further, we can write the number this way:

$$4825 = 4 \times 1000 +$$

$$8 \times 100 +$$

$$2 \times 10 +$$

$$5 \times 1$$

Or, using powers of ten, the number can be rewritten like this:

$$4825 = 4 \times 10^3 +$$

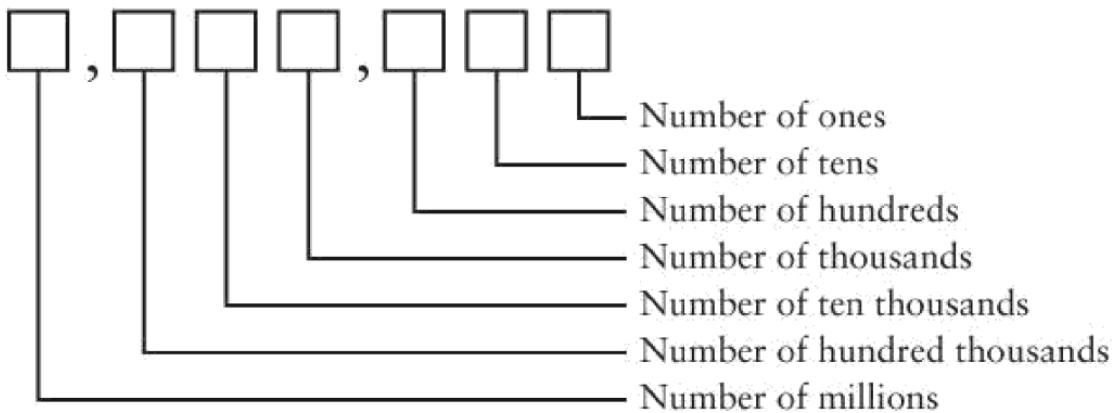
$$8 \times 10^2 +$$

$$2 \times 10^1 +$$

$$5 \times 10^0$$

Remember that any number to the 0 power equals 1.

Each position in a multidigit number has a particular meaning, as shown in the following diagram. The seven boxes shown here let us represent any number from 0 through 9,999,999:



Each position corresponds to a power of ten. We don't need a special symbol for ten because we set the 1 in a different position and we use the 0 as a placeholder.

What's also really nice is that fractional quantities shown as digits to the right of a decimal point follow this same pattern. The number 42,705.684 is

$$4 \times 10,000 +$$

$$2 \times 1000 +$$

$$7 \times 100 +$$

$$0 \times 10 +$$

$$5 \times 1 +$$

$$6 \div 10 +$$

$$8 \div 100 +$$

$$4 \div 1000$$

This number can also be written without any division, like this:

$$4 \times 10,000 +$$

$$2 \times 1000 +$$

$$7 \times 100 +$$

$$0 \times 10 +$$

$$5 \times 1 +$$

$$6 \times 0.1 +$$

$$8 \times 0.01 +$$

$$4 \times 0.001$$

Or, using powers of ten, the number is

$$4 \times 10^4 +$$

$$2 \times 10^3 +$$

$$7 \times 10^2 +$$

$$0 \times 10^1 +$$

$$5 \times 10^0 +$$

$$6 \times 10^{-1} +$$

$$8 \times 10^{-2} +$$

$$4 \times 10^{-3}$$

Notice how the exponents go down to zero and then become negative numbers.

We know that 3 plus 4 equals 7. Similarly, 30 plus 40 equals 70, 300 plus 400 equals 700, and 3000 plus 4000 equals 7000. This is the beauty of the Hindu-Arabic system. When you add decimal numbers of any length, you follow a

procedure that breaks down the problem into steps. Each step involves nothing more complicated than adding pairs of single-digit numbers. That's why someone a long time ago forced you to memorize an addition table:

+	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	10
2	2	3	4	5	6	7	8	9	10	11
3	3	4	5	6	7	8	9	10	11	12
4	4	5	6	7	8	9	10	11	12	13
5	5	6	7	8	9	10	11	12	13	14
6	6	7	8	9	10	11	12	13	14	15
7	7	8	9	10	11	12	13	14	15	16
8	8	9	10	11	12	13	14	15	16	17
9	9	10	11	12	13	14	15	16	17	18

Find the two numbers you wish to add in the top row and the left column. Follow down and across to get the sum. For example, 4 plus 6 equals 10.

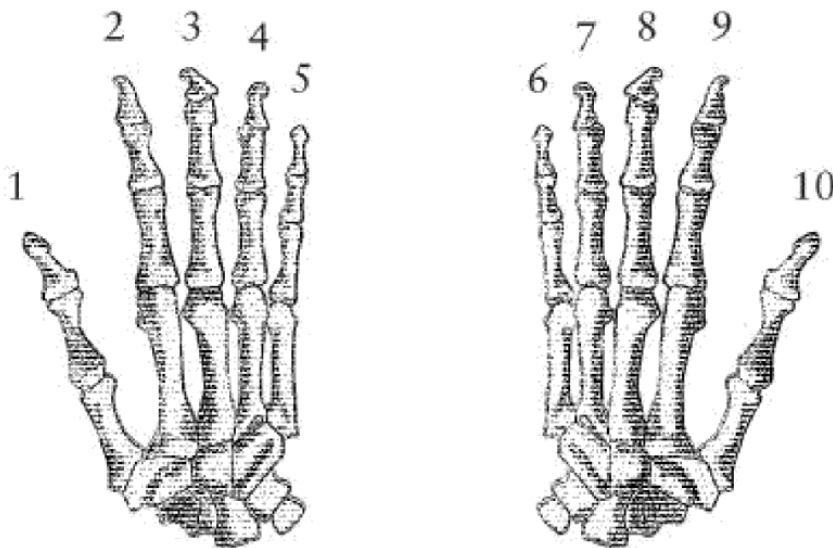
Similarly, when you need to multiply two decimal numbers, you follow a somewhat more complicated procedure but still one that breaks down the problem so that you need do nothing more complex than adding or multiplying single-digit decimal numbers. Your early schooling probably also entailed memorizing a multiplication table:

x	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

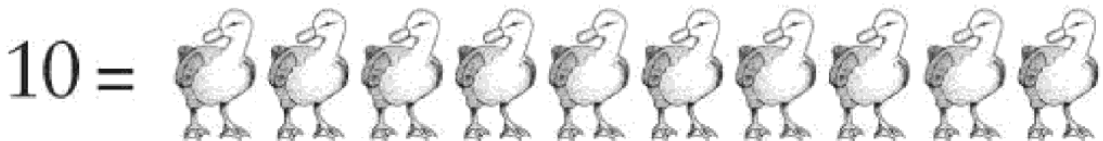
What's best about the positional system of notation isn't how well it works, but how well it works for counting systems *not* based on ten. Our number system isn't necessarily appropriate for everyone. One big problem with our base-ten system of numbers is that it doesn't have any relevance for cartoon characters. Most cartoon characters have only four fingers on each hand (or paw), so they prefer a number system that's based on eight. Interestingly enough, much of what we know about decimal numbering can be applied to a numbering system more appropriate for our friends in cartoons.

Chapter 8. Alternatives to Ten

Ten is an exceptionally important number to us humans. Ten is the number of fingers and toes most of us have, and we certainly prefer to have all ten of each. Because our fingers are convenient for counting, we humans have adapted an entire number system that's based on the number 10.



As I mentioned in the previous chapter, the number system that we use is called *base ten*, or *decimal*. The number system seems so natural to us that it's difficult at first to conceive of alternatives. Indeed, when we see the number *10* we can't help but think that this number refers to this many ducks:



But the only reason that the number 10 refers to this many ducks is that this many ducks is the same as the number of fingers we have. If human beings had a different number of fingers, the way we counted would be different, and 10 would mean something else. That same number 10 could refer to this many ducks:

$$10 = \text{[duck]} \text{[duck]} \text{[duck]} \text{[duck]} \text{[duck]} \text{[duck]} \text{[duck]} \text{[duck]} \text{[duck]}$$

or this many ducks:

$$10 = \text{[duck]} \text{[duck]} \text{[duck]} \text{[duck]}$$

or even this many ducks:

$$10 = \text{[duck]} \text{[duck]}$$

When we get to the point where 10 means just two ducks, we'll be ready to examine how switches, wires, lightbulbs, and relays (and by extension, computers) can represent numbers.

What if human beings had only four fingers on each hand, like cartoon characters? We probably never would have thought to develop a number system based on ten. Instead, we would have considered it normal and natural and sensible and inevitable and incontrovertible and undeniably proper to base our number system on eight. We wouldn't call this a *decimal* number system. We'd call it an *octal* number system, or *base eight*.

If our number system were organized around eight rather than ten, we wouldn't need the symbol that looks like this:

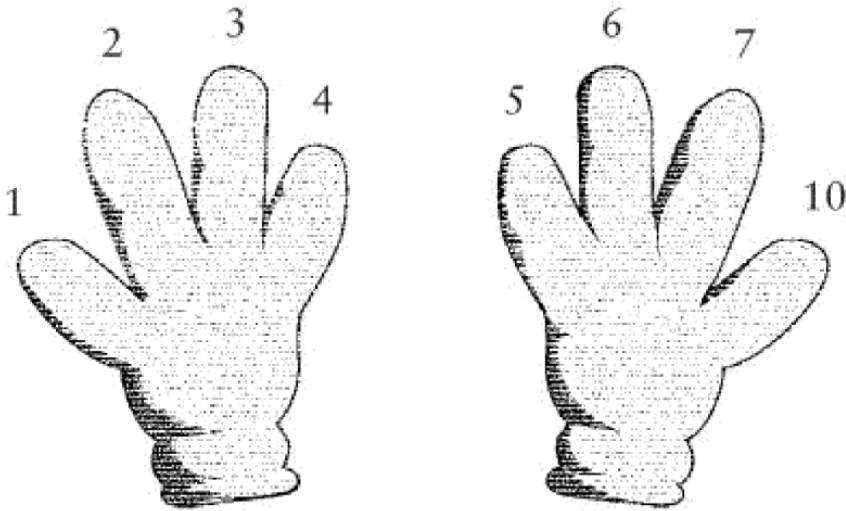
9

Show this symbol to any cartoon character and you'll get the response, "What's that? What's it for?" And if you think about it a moment, we also wouldn't need the symbol that looks like this:

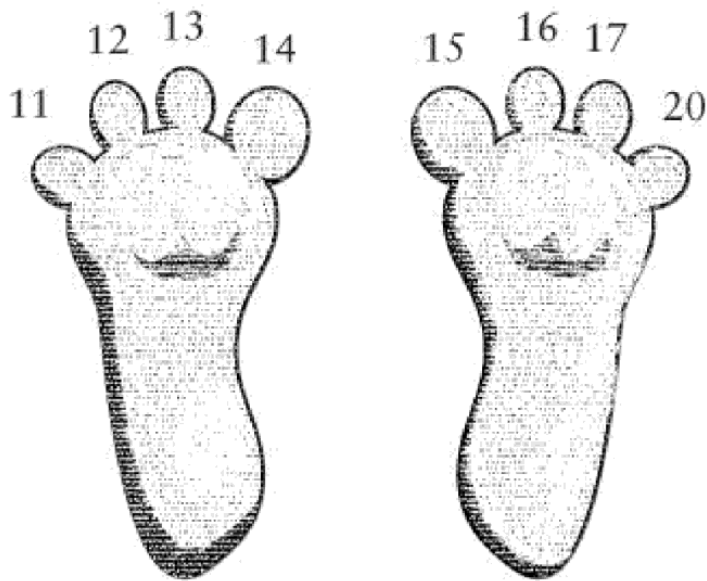
8

In the decimal number system, there's no special symbol for ten, so in the octal number system there's no special symbol for eight.

The way we count in the decimal number system is 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and then 10. The way we count in the octal number system is 0, 1, 2, 3, 4, 5, 6, 7, and then what? We've run out of symbols. The only thing that makes sense is 10, and that's correct. In octal, the next number after 7 is 10. But this 10 doesn't mean the number of fingers that humans have. In octal, 10 refers to the number of fingers that cartoon characters have.



We can continue counting on our four-toed feet:



When you're working with number systems other than decimal, you can avoid some confusion if you pronounce a number like 10 as *one zero*. Similarly, 13 is pronounced *one three* and 20 is pronounced *two zero*. To *really* avoid confusion, we can say *two zero base eight* or *two zero octal*.

Even though we've run out of fingers and toes, we can still continue counting in octal. It's basically the same as counting in decimal except that we skip every number that has an 8 or a 9 in it. And of course, the actual numbers refer to different quantities:

0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, 20, 21, 22,
 23, 24, 25, 26, 27, 30, 31, 32, 33, 34, 35, 36, 37, 40, 41, 42, 43,
 44, 45, 46, 47, 50, 51, 52, 53, 54, 55, 56, 57, 60, 61, 62, 63, 64,
 65, 66, 67, 70, 71, 72, 73, 74, 75, 76, 77, 100...

That last number is pronounced *one zero zero*. It's the number of fingers that cartoon characters have, multiplied by itself.

When writing decimal and octal numbers, we can avoid confusion and denote which is which by using a subscript to indicate the numbering system. The subscript TEN means base ten or decimal, and EIGHT means base eight or octal.

Thus, the number of dwarfs that Snow White meets is 7_{TEN} or 7_{EIGHT}

The number of fingers that cartoon characters have is 8_{TEN} or 10_{EIGHT}

The number of symphonies that Beethoven wrote is 9_{TEN} or 11_{EIGHT}

The number of fingers that humans have is 10_{TEN} or 12_{EIGHT}

The number of months in a year is 12_{TEN} or 14_{EIGHT}

The number of days in a fortnight is 14_{TEN} or 16_{EIGHT}

The “sweet” birthday celebration is 16_{TEN} or 20_{EIGHT}

The number of hours in a day is 24_{TEN} or 30_{EIGHT}

The number of letters in the Latin alphabet is 26_{TEN} or 32_{EIGHT}

The number of fluid ounces in a quart is 32_{TEN} or 40_{EIGHT}

The number of cards in a deck is 52_{TEN} or 64_{EIGHT}

The number of squares on a chessboard is 64_{TEN} or 100_{EIGHT}

The most famous address on Sunset Strip is 77_{TEN} or 115_{EIGHT}

The number of yards in an American football field is 100_{TEN} or 144_{EIGHT}

The number of starting women singles players at Wimbledon is 128_{TEN} or 200_{EIGHT}

The number of square miles in Memphis is 256_{TEN} or 400_{EIGHT}

Notice that there are a few nice round octal numbers in this list, such as 100_{EIGHT} and 200_{EIGHT} and 400_{EIGHT} . By the term *nice round number* we usually mean a number that has some zeros at the end. Two zeros on the end of a decimal number means that the number is a multiple of 100_{TEN} , which is 10_{TEN} times 10_{TEN} . With octal numbers, two zeros on the end means that the number is a multiple of 100_{EIGHT} , which is 10_{EIGHT} times 10_{EIGHT} (or 8_{TEN} times 8_{TEN} , which is 64_{TEN}).

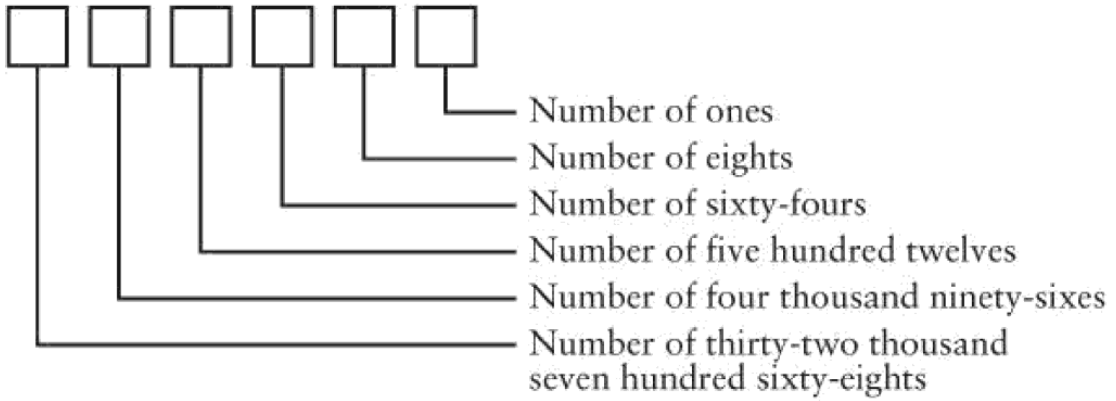
You might also notice that these nice round octal numbers 100_{EIGHT} and 200_{EIGHT} and 400_{EIGHT} have the decimal equivalents 64_{TEN} , 128_{TEN} , and 256_{TEN} , all of which are powers of two. This makes sense. The number 400_{EIGHT} (for example) is 4_{EIGHT} times 10_{EIGHT} times 10_{EIGHT} , all of which are powers of two. And anytime we multiply a power of two by a power of two, we get another power of two.

The following table shows some powers of two with the decimal and octal representations:

Power of	Two Decimal	Octal
2^0	1	1
2^1	2	2
2^2	4	4
2^3	8	10
2^4	16	20
2^5	32	40
2^6	64	100
2^7	128	200
2^8	256	400
2^9	512	1000
2^{10}	1024	2000
2^{11}	2048	4000
2^{12}	4096	10000

The nice round numbers in the rightmost column are a hint that number systems other than decimal might help in working with binary codes.

The octal system isn't different from the decimal system in any structural way. It just differs in details. For example, each position in an octal number is a digit that's multiplied by a power of eight:



Thus, an octal number such as 3725_{EIGHT} can be broken down like so:

$$3725_{\text{EIGHT}} = 3000_{\text{EIGHT}} + 700_{\text{EIGHT}} + 20_{\text{EIGHT}} + 5_{\text{EIGHT}}$$

This can be rewritten in any of several ways. Here's one way, using the powers of eight in their decimal forms:

$$3725_{\text{EIGHT}} = 3 \times 512_{\text{TEN}} +$$

$$7 \times 64_{\text{TEN}} +$$

$$2 \times 8_{\text{TEN}} +$$

$$5 \times 1$$

This is the same thing with the powers of eight shown in their octal form:

$$3725_{\text{EIGHT}} = 3 \times 1000_{\text{EIGHT}} +$$

$$7 \times 100_{\text{EIGHT}} +$$

$$2 \times 10_{\text{EIGHT}} +$$

$$5 \times 1$$

Here's another way of doing it:

$$3725_{\text{EIGHT}} = 3 \times 8^3 +$$

$$7 \times 8^2 +$$

$$2 \times 8^1 +$$

$$5 \times 8^0$$

If you work out this calculation in decimal, you'll get 2005_{TEN} . This is how you can convert octal numbers to decimal numbers.

We can add and multiply octal numbers the same way we add and multiply decimal numbers. The only real difference is that we use different tables for adding and multiplying the individual digits. Here's the addition table for octal numbers:

+	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	10
2	2	3	4	5	6	7	10	11
3	3	4	5	6	7	10	11	12
4	4	5	6	7	10	11	12	13
5	5	6	7	10	11	12	13	14
6	6	7	10	11	12	13	14	15
7	7	10	11	12	13	14	15	16

For example, $5_{\text{EIGHT}} + 7_{\text{EIGHT}} = 14_{\text{EIGHT}}$. So we can add two longer octal numbers the same way we add decimal numbers:

$$\begin{array}{r} 135 \\ + 643 \\ \hline 1000 \end{array}$$

To begin with the right column, 5 plus 3 equals 10. Put down the 0, carry the 1. One plus 3 plus 4 equals 10. Put down the 0, carry the 1. One plus 1 plus 6 equals 10.

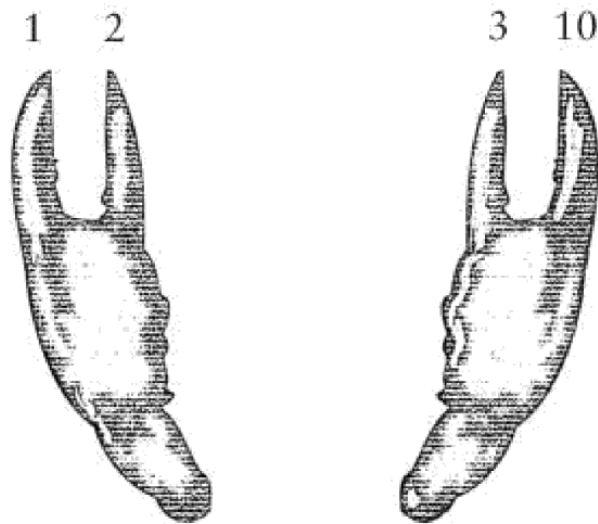
Similarly, 2 times 2 is still 4 in octal. But 3 times 3 isn't 9. How could it be?

Instead 3 times 3 is 11_{EIGHT} , which is the same amount as 9_{TEN} . You can see the entire octal multiplication table at the top of the following page.

x	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	10	12	14	16
3	0	3	6	11	14	17	22	25
4	0	4	10	14	20	24	30	34
5	0	5	12	17	24	31	36	43
6	0	6	14	22	30	36	44	52
7	0	7	16	25	34	43	52	61

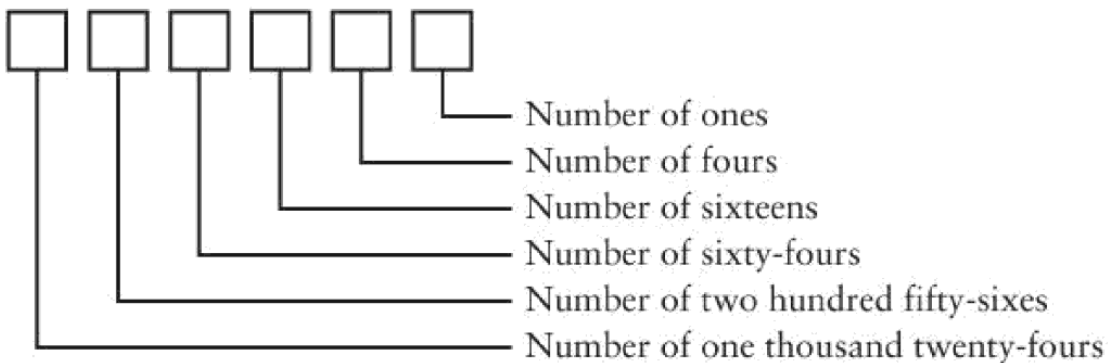
Here we have 4×6 equaling 30_{EIGHT} , but 30_{EIGHT} is the same as 24_{TEN} , which is what 4×6 equals in decimal.

Octal is as valid a number system as decimal. But let's go further. Now that we've developed a numbering system for cartoon characters, let's develop something that's appropriate for lobsters. Lobsters don't have fingers exactly, but they do have pincers at the ends of their two front legs. An appropriate number system for lobsters is the *quaternary* system, or base four:



Counting in quaternary goes like this: 0, 1, 2, 3, 10, 11, 12, 13, 20, 21, 22, 23, 30, 31, 32, 33, 100, 101, 102, 103, 110, and so forth.

I'm not going to spend much time with the quaternary system because we'll be moving on shortly to something much more important. But we can see how each position in a quaternary number corresponds this time to a power of *four*:



The quaternary number 31232 can be written like this:

$$\begin{aligned}
 31232_{\text{FOUR}} &= 3 \times 256_{\text{TEN}} + \\
 &1 \times 64_{\text{TEN}} + \\
 &2 \times 16_{\text{TEN}} + \\
 &3 \times 4_{\text{TEN}} + \\
 &2 \times 1_{\text{TEN}}
 \end{aligned}$$

which is the same as

$$\begin{aligned} 31232_{\text{FOUR}} &= 3 \times 10000_{\text{FOUR}} + \\ &1 \times 1000_{\text{FOUR}} + \\ &2 \times 100_{\text{FOUR}} + \\ &3 \times 10_{\text{FOUR}} + \\ &2 \times 1_{\text{FOUR}} \end{aligned}$$

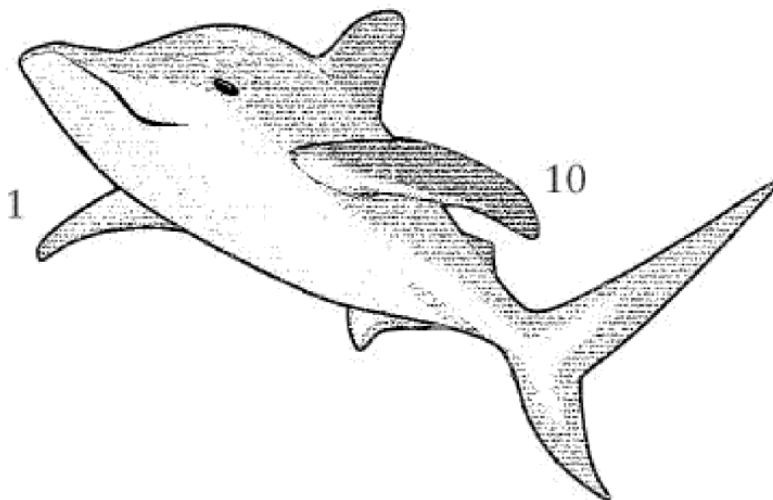
And it's also the same as

$$\begin{aligned} 31232_{\text{FOUR}} &= 3 \times 4^4 + \\ &1 \times 4^3 + \\ &2 \times 4^2 + \\ &3 \times 4^1 + \\ &2 \times 4^0 \end{aligned}$$

If we do the calculations in decimal, we'll find that 31232_{FOUR} equals 878_{TEN} .

Now we're going to make another leap, and this one is extreme. Suppose we were dolphins and must resort to using our two flippers for counting. This is the number system known as base two, or *binary* (from the Latin for *two by two*). It seems likely that we'd have only two digits, and these two digits would be 0 and 1.

Now, 0 and 1 isn't a whole lot to work with, and it takes some practice to get accustomed to binary numbers. The big problem is that you run out of digits very quickly. For example, here's how a dolphin counts using its flippers:



Yes, in binary the next number after 1 is 10. This is startling, but it shouldn't really be a surprise. No matter what number system we use, whenever we run out of single digits, the first two-digit number is always 10. In binary we count like this:

0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011, 1100,
1101, 1110, 1111, 10000, 10001...

These numbers might look large, but they're really not. It's more accurate to say that binary numbers get *long* very quickly rather than large:

The number of heads that humans have is 1_{TEN} or 1_{TWO}

The number of flippers that dolphins have is 2_{TEN} or 10_{TWO}

The number of teaspoons in a tablespoon is 3_{TEN} or 11_{TWO}

The number of sides to a square is 4_{TEN} or 100_{TWO}

The number of fingers on one human hand is 5_{TEN} or 101_{TWO}

The number of legs on an insect is 6_{TEN} or 110_{TWO}

The number of days in a week is 7_{TEN} or 111_{TWO}

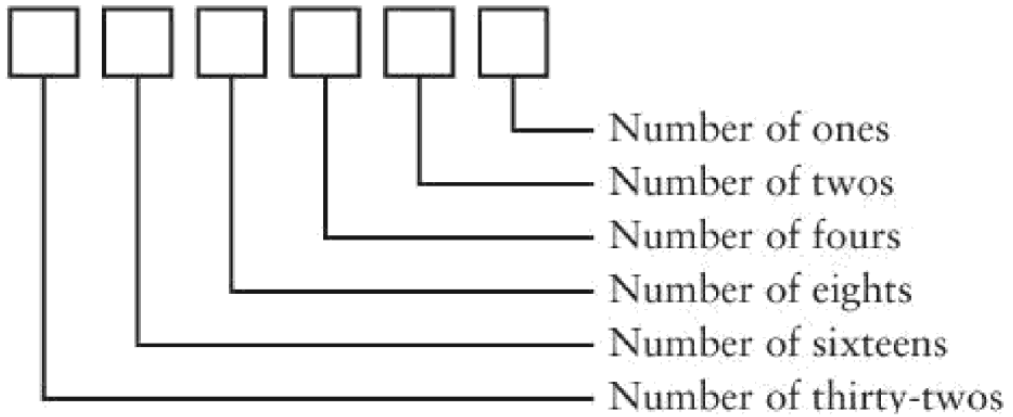
The number of musicians in an octet is 8_{TEN} or 1000_{TWO}

The number of planets in our solar system (including Pluto) is 9_{TEN} or 1001_{TWO}

The number of gallons in a cowboy hat is 10_{TEN} or 1010_{TWO}

and so forth.

In a multidigit binary number, the positions of the digits correspond to powers of two:



So anytime we have a binary number composed of a 1 followed by all zeros, that number is a power of two. The power is the same as the number of zeros in the binary number. Here's our expanded table of the powers of two demonstrating this rule:

Power of Two	Decimal	Octal	Quaternary	Binary
2^0	1	1	1	1
2^1	2	2	2	10
2^2	4	4	10	100
2^3	8	10	20	1000
2^4	16	20	100	10000
2^5	32	40	200	100000
2^6	64	100	1000	1000000
2^7	128	200	2000	10000000
2^8	256	400	10000	100000000
2^9	512	1000	20000	1000000000
2^{10}	1024	2000	100000	10000000000
2^{11}	2048	4000	200000	100000000000
2^{12}	4096	10000	1000000	1000000000000

Let's say we have the binary number 101101011010. This can be written as

$$101101011010_{\text{TWO}} = 1 \times 2048_{\text{TEN}} +$$

$$0 \times 1024_{\text{TEN}} +$$

$$1 \times 512_{\text{TEN}} +$$

$$1 \times 256_{\text{TEN}} +$$

$$0 \times 128_{\text{TEN}} +$$

$$1 \times 64_{\text{TEN}} +$$

$$0 \times 32_{\text{TEN}} +$$

$$1 \times 16_{\text{TEN}} +$$

$$1 \times 8_{\text{TEN}} +$$

$$0 \times 4_{\text{TEN}} +$$

$$1 \times 2_{\text{TEN}} +$$

$$0 \times 1_{\text{TEN}}$$

The same number can be written this way:

$$101101011010_{\text{TWO}} = 1 \times 2^{11} +$$

$$0 \times 2^{10} +$$

$$1 \times 2^9 +$$

$$1 \times 2^8 +$$

$$0 \times 2^7 +$$

$$1 \times 2^6 +$$

$$0 \times 2^5 +$$

$$1 \times 2^4 +$$

$$1 \times 2^3 +$$

$$0 \times 2^2 +$$

$$1 \times 2^1 +$$

$$0 \times 2^0$$

If we just add up the parts in decimal, we get $2048 + 512 + 256 + 64 + 16 + 8 + 2$, which is $2,906_{\text{TEN}}$.

To convert binary numbers to decimal more concisely, you might prefer a method that uses a template I've prepared:

<input style="width: 50px; height: 30px;" type="text"/>	<input style="width: 50px; height: 30px;" type="text"/>	<input style="width: 50px; height: 30px;" type="text"/>	<input style="width: 50px; height: 30px;" type="text"/>	<input style="width: 50px; height: 30px;" type="text"/>	<input style="width: 50px; height: 30px;" type="text"/>	<input style="width: 50px; height: 30px;" type="text"/>	<input style="width: 50px; height: 30px;" type="text"/>
x128	x64	x32	x16	x8	x4	x2	x1
<input style="width: 50px; height: 30px;" type="text"/>	<input style="width: 50px; height: 30px;" type="text"/>	<input style="width: 50px; height: 30px;" type="text"/>	<input style="width: 50px; height: 30px;" type="text"/>	<input style="width: 50px; height: 30px;" type="text"/>	<input style="width: 50px; height: 30px;" type="text"/>	<input style="width: 50px; height: 30px;" type="text"/>	<input style="width: 50px; height: 30px;" type="text"/>

+ + + + + + + =

This template allows you to convert numbers up to eight binary digits in length, but it could easily be extended. To use it, put up to eight binary digits in the 8 boxes at the top, one digit to a box. Do the eight multiplications and put the products in the 8 lower boxes. Add these eight boxes for the final result. This example shows how to find the decimal equivalent of 10010110:

<input style="width: 50px; height: 30px; border: 1px solid black;" type="text" value="1"/>	<input style="width: 50px; height: 30px; border: 1px solid black;" type="text" value="0"/>	<input style="width: 50px; height: 30px; border: 1px solid black;" type="text" value="0"/>	<input style="width: 50px; height: 30px; border: 1px solid black;" type="text" value="1"/>	<input style="width: 50px; height: 30px; border: 1px solid black;" type="text" value="0"/>	<input style="width: 50px; height: 30px; border: 1px solid black;" type="text" value="1"/>	<input style="width: 50px; height: 30px; border: 1px solid black;" type="text" value="1"/>	<input style="width: 50px; height: 30px; border: 1px solid black;" type="text" value="0"/>
x128	x64	x32	x16	x8	x4	x2	x1
<input style="width: 50px; height: 30px; border: 1px solid black;" type="text" value="128"/>	<input style="width: 50px; height: 30px; border: 1px solid black;" type="text" value="0"/>	<input style="width: 50px; height: 30px; border: 1px solid black;" type="text" value="0"/>	<input style="width: 50px; height: 30px; border: 1px solid black;" type="text" value="16"/>	<input style="width: 50px; height: 30px; border: 1px solid black;" type="text" value="0"/>	<input style="width: 50px; height: 30px; border: 1px solid black;" type="text" value="4"/>	<input style="width: 50px; height: 30px; border: 1px solid black;" type="text" value="2"/>	<input style="width: 50px; height: 30px; border: 1px solid black;" type="text" value="0"/>

+ + + + + + + =

Converting from decimal to binary isn't quite as straightforward, but here's a template that let's you convert decimal numbers from 0 through 255 to binary:

<input style="width: 50px; height: 30px;" type="text"/>	<input style="width: 50px; height: 30px;" type="text"/>	<input style="width: 50px; height: 30px;" type="text"/>	<input style="width: 50px; height: 30px;" type="text"/>	<input style="width: 50px; height: 30px;" type="text"/>	<input style="width: 50px; height: 30px;" type="text"/>	<input style="width: 50px; height: 30px;" type="text"/>	<input style="width: 50px; height: 30px;" type="text"/>
÷128	÷64	÷32	÷16	÷8	÷4	÷2	÷1
<input style="width: 50px; height: 30px;" type="text"/>	<input style="width: 50px; height: 30px;" type="text"/>	<input style="width: 50px; height: 30px;" type="text"/>	<input style="width: 50px; height: 30px;" type="text"/>	<input style="width: 50px; height: 30px;" type="text"/>	<input style="width: 50px; height: 30px;" type="text"/>	<input style="width: 50px; height: 30px;" type="text"/>	<input style="width: 50px; height: 30px;" type="text"/>

The conversion is actually trickier than it appears, so follow the directions carefully. Put the entire decimal number (less than or equal to 255) in the box in the upper left corner. Divide that number (the dividend) by the first divisor (128), as indicated. Put the quotient in the box below (the box at the lower left corner), and the remainder in the box to the right (the second box on the top row). That first remainder is the dividend for the next calculation, which uses a divisor of 64. Continue in the same manner through the template.

Keep in mind that each quotient will be either 0 or 1. If the dividend is less than the divisor, the quotient is 0 and the remainder is simply the dividend.

If the dividend is greater than or equal to the divisor, the quotient is 1 and the remainder is the dividend minus the divisor. Here's how it's done with 150:

150	22	22	22	6	6	2	0
÷128	÷64	÷32	÷16	÷8	÷4	÷2	÷1
1	0	0	1	0	1	1	0

If you need to add or multiply two binary numbers, it's probably easier to do the calculation in binary rather than convert to decimal. This is the part you're *really* going to like. Imagine how quickly you could have mastered addition if the only thing you had to memorize was this:

+	0	1
0	0	1
1	1	10

Let's use this table to add two binary numbers:

$$\begin{array}{r}
 1100101 \\
 + 0110110 \\
 \hline
 10011011
 \end{array}$$

Starting at the right column: 1 plus 0 equals 1. Second column from right: 0 plus 1 equals 1. Third column: 1 plus 1 equals 0, carry the 1. Fourth column: 1 (carried) plus 0 plus 0 equals 1. Fifth column: 0 plus 1 equals 1. Sixth column: 1 plus 1 equals 0, carry the 1. Seventh column: 1 (carried) plus 1 plus 0 equals 10.

The multiplication table is even simpler than the addition table because it can be entirely derived by using two of the very basic rules of multiplication: Multiplying anything by 0 gets you 0, and multiplying any number by 1 has no effect on the number.

x	0	1
0	0	0
1	0	1

Here's a multiplication of 13_{TEN} by 11_{TEN} in binary:

$$\begin{array}{r}
 1101 \\
 \times 1011 \\
 \hline
 1101 \\
 1101 \\
 0000 \\
 1101 \\
 \hline
 10001111
 \end{array}$$

The result is 143_{TEN} .

People who work with binary numbers often write them with leading zeros (that is, zeros to the left of the first 1)—for example, 0011 rather than just 11. This doesn't change the value of the number at all; it's just for cosmetic purposes. For example, here are the first sixteen binary numbers with their decimal equivalents:

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

Let's take a look at this list of binary numbers for a moment. Consider each of the four vertical columns of zeros and ones, and notice how the digits

alternate going down the column:

- The rightmost digit alternates between 0 and 1.
- The next digit from the right alternates between two 0s and two 1s.
- The next digit alternates between four 0s and four 1s.
- The next digit alternates between eight 0s and eight 1s.

This is *very* methodical, wouldn't you say? Indeed, you can easily write the next sixteen binary numbers by just repeating the first sixteen and putting a 1 in front:

Binary	Decimal
10000	16
10001	17
10010	18
10011	19
10100	20
10101	21
10110	22
10111	23
11000	24
11001	25
11010	26
11011	27
11100	28
11101	29
11110	30
11111	31

Here's another way of looking at it: When you count in binary, the rightmost digit (also called the least significant digit), alternates between 0

and 1. Every time it changes from a 1 to a 0, the digit second to right (that is, the next most significant digit) also changes, either from 0 to 1 or from 1 to 0. So every time a binary digit changes from a 1 to a 0, the next most significant digit also changes, either from a 0 to a 1 or from a 1 to a 0.

When we're writing large decimal numbers, we use commas every three places so that we can more easily know what the number means at a glance. For example, if you see 12000000, you probably have to count digits, but if you see 12,000,000, you know that means twelve million.

Binary numbers can get very long very quickly. For example, twelve million in binary is 101101110001101100000000. To make this a *little* more readable, it's customary to separate every four binary digits with a dash, for example 1011-0111-0001-1011-0000-0000 or with spaces: 1011 0111 0001 1011 0000 0000. Later on in this book, we'll look at a more concise way of expressing binary numbers.

By reducing our number system to just the binary digits 0 and 1, we've gone as far as we can go. We can't get any simpler. Moreover, the binary number system bridges the gap between arithmetic and electricity. In previous chapters, we've been looking at switches and wires and lightbulbs and relays, and any of these objects can represent the binary digits 0 and 1:

A wire can be a binary digit. If current is flowing through the wire, the binary digit is 1. If not, the binary digit is 0.

A switch can be a binary digit. If the switch is on, or closed, the binary digit is 1. If the switch is off, or open, the binary digit is 0.

A lightbulb can be a binary digit. If the lightbulb is lit, the binary digit is 1. If the lightbulb is not lit, the binary digit is 0.

A telegraph relay can be a binary digit. If the relay is closed, the binary digit is 1. If the relay is at rest, the binary digit is 0.

Binary numbers have a whole *lot* to do with computers.

Sometime around 1948, the American mathematician John Wilder Tukey (born 1915) realized that the words *binary digit* were likely to assume a much greater importance in the years ahead as computers became more prevalent. He decided to coin a new, shorter word to replace the unwieldy five syllables of *binary digit*. He considered *bigit* and *binit* but settled instead on the short, simple, elegant, and perfectly lovely word *bit*.

Chapter 9. Bit by Bit by Bit

When Tony Orlando requested in a 1973 song that his beloved “Tie a Yellow Ribbon Round the Ole Oak Tree,” he wasn’t asking for elaborate explanations or extended discussion. He didn’t want any ifs, ands, or buts. Despite the complex feelings and emotional histories that would have been at play in the real-life situation the song was based on, all the man really wanted was a simple yes or no. He wanted a yellow ribbon tied around the tree to mean “Yes, even though you messed up big time and you’ve been in prison for three years, I still want you back with me under my roof.” And he wanted the absence of a yellow ribbon to mean “Don’t even *think* about stopping here.”

These are two clear-cut, mutually exclusive alternatives. Tony Orlando did *not* sing, “Tie half of a yellow ribbon if you want to think about it for a while” or “Tie a blue ribbon if you don’t love me anymore but you’d still like to be friends.” Instead, he made it very, very simple.

Equally effective as the absence or presence of a yellow ribbon (but perhaps more awkward to put into verse) would be a choice of traffic signs in the front yard: Perhaps “Merge” or “Wrong Way.”

Or a sign hung on the door: “Closed” or “Open.”

Or a flashlight in the window, turned on or off.

You can choose from lots of ways to say yes or no if that’s all you need to say. You don’t need a sentence to say yes or no; you don’t need a word, and you don’t even need a letter. All you need is a *bit*, and by that I mean all you need is a 0 or a 1.

As we discovered in the previous chapters, there’s nothing really all that special about the decimal number system that we normally use for counting. It’s pretty clear that we base our number system on ten because that’s the number of fingers we have. We could just as reasonably base our number system on eight (if we were cartoon characters) or four (if we were lobsters) or even two (if we were dolphins).

But there *is* something special about the binary number system. What’s

special about binary is that it's the *simplest* number system possible. There are only two binary digits—0 and 1. If we want something simpler than binary, we'll have to get rid of the 1, and then we'll be left with just a 0. We can't do much of anything with just a 0.

The word *bit*, coined to mean *binary digit*, is surely one of the loveliest words invented in connection with computers. Of course, the word has the normal meaning, "a small portion, degree, or amount," and that normal meaning is perfect because a bit—one binary digit—is a very small quantity indeed.

Sometimes when a new word is invented, it also assumes a new meaning. That's certainly true in this case. A *bit* has a meaning beyond the *binary digits* used by dolphins for counting. In the computer age, the bit has come to be regarded as *the basic building block of information*.

Now that's a bold statement, and of course, bits aren't the only things that convey information. Letters and words and Morse code and Braille and decimal digits convey information as well. The thing about the bit is that it conveys very *little* information. A bit of information is the tiniest amount of information possible. Anything less than a bit is no information at all. But because a bit represents the smallest amount of information possible, more complex information can be conveyed with multiple bits. (By saying that a bit conveys a "small" amount of information, I surely don't mean that the information borders on the unimportant. Indeed, the yellow ribbon is a *very* important bit to the two people concerned with it.)

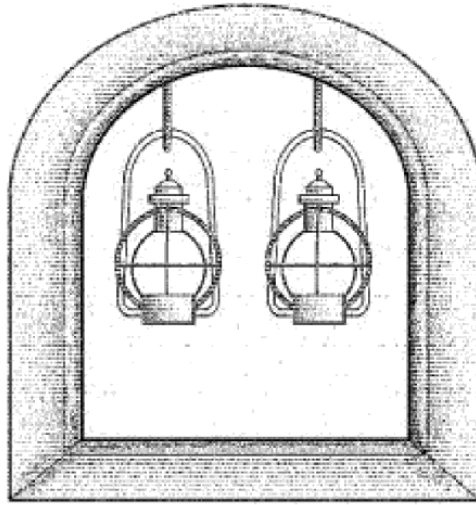
"Listen, my children, and you shall hear / Of the midnight ride of Paul Revere," wrote Henry Wadsworth Longfellow, and while he might not have been historically accurate when describing how Paul Revere alerted the American colonies that the British had invaded, he did provide a thoughtprovoking example of the use of bits to communicate important information:

*He said to his friend "If the British march
By land or sea from the town to-night,
Hang a lantern aloft in the belfry arch
Of the North Church tower as a special light,—
One, if by land, and two, if by sea..."*

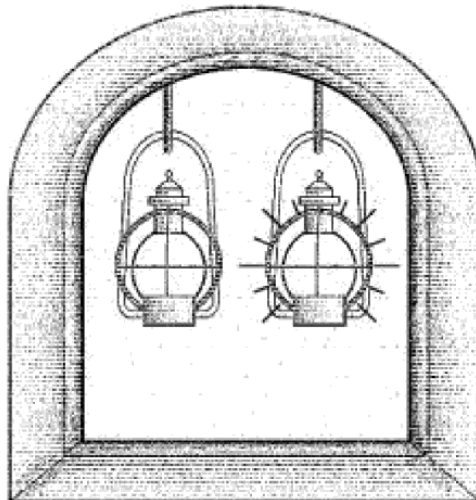
To summarize, Paul Revere's friend has two lanterns. If the British are invading by land, he will put just one lantern in the church tower. If the British are coming by sea, he will put both lanterns in the church tower.

However, Longfellow isn't explicitly mentioning all the possibilities. He left unspoken a *third* possibility, which is that the British aren't invading just yet. Longfellow implies that this possibility will be conveyed by the *absence* of lanterns in the church tower.

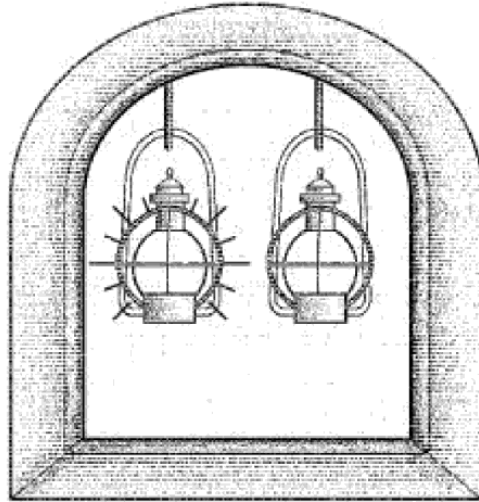
Let's assume that the two lanterns are actually permanent fixtures in the church tower. Normally they aren't lit:



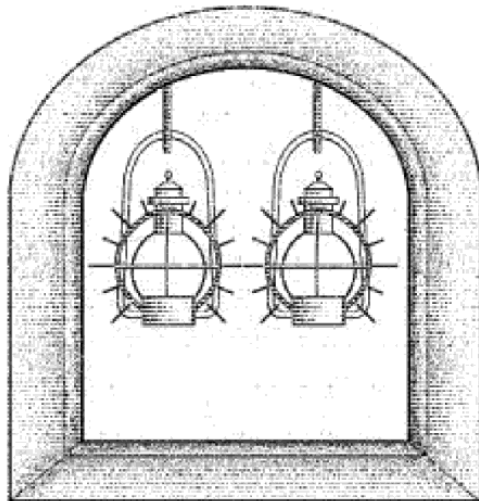
This means that the British aren't yet invading. If one of the lanterns is lit,



or



the British are coming by land. If both lanterns are lit,



the British are coming by sea.

Each lantern is a bit. A lit lantern is a 1 bit and an unlit lantern is a 0 bit. Tony Orlando demonstrated to us that only one bit is necessary to convey one of two possibilities. If Paul Revere needed only to be alerted that the British were invading (and not where they were coming from), one lantern would have been sufficient. The lantern would have been lit for an invasion and unlit for another evening of peace.

Conveying one of three possibilities requires another lantern. Once that second lantern is present, however, the two bits allows communicating one of four possibilities:

00 = The British aren't invading tonight.

01 = They're coming by land.

10 = They're coming by land.

11 = They're coming by sea.

What Paul Revere did by sticking to just three possibilities was actually quite sophisticated. In the lingo of communication theory, he used *redundancy* to counteract the effect of *noise*. The word *noise* is used in communication theory to refer to anything that interferes with communication. Static on a telephone line is an obvious example of noise that interferes with a telephone communication. Communication over the telephone is usually successful, nevertheless, even in the presence of noise because spoken language is heavily redundant. We don't need to hear every syllable of every word in order to understand what's being said.

In the case of the lanterns in the church tower, noise can refer to the darkness of the night and the distance of Paul Revere from the tower, both of which might prevent him from distinguishing one lantern from the other. Here's the crucial passage in Longfellow's poem:

And lo! As he looks, on the belfry's height

A glimmer, and then a gleam of light!

He springs to the saddle, the bridle he turns,

But lingers and gazes, till full on his sight

A second lamp in the belfry burns!

It certainly doesn't sound as if Paul Revere was in a position to figure out exactly which one of the two lanterns was first lit.

The essential concept here is that *information represents a choice among two or more possibilities*. For example, when we talk to another person, every word we speak is a choice among all the words in the dictionary. If we numbered all the words in the dictionary from 1 through 351,482, we could just as accurately carry on conversations using the numbers rather than words. (Of course, both participants would need dictionaries where the words are numbered identically, as well as plenty of patience.)

The flip side of this is that *any information that can be reduced to a choice among two or more possibilities can be expressed using bits*. Needless to say, there are plenty of forms of human communication that do *not* represent choices among discrete possibilities and that are also vital to our existence.

This is why people don't form romantic relationships with computers. (Let's hope they don't, anyway.) If you can't express something in words, pictures, or sounds, you're not going to be able to encode the information in bits. Nor would you want to.

A thumb up or a thumb down is one bit of information. And two thumbs up or down—such as the thumbs of film critics Roger Ebert and the late Gene Siskel when they rendered their final verdicts on the latest movies—convey two bits of information. (We'll ignore what they actually had to say about the movies; all we care about here are their thumbs.) Here we have four possibilities that can be represented with a pair of bits:

00 = They both hated it.

01 = Siskel hated it; Ebert loved it.

10 = Siskel loved it; Ebert hated it.

11 = They both loved it.

The first bit is the Siskel bit, which is 0 if Siskel hated the movie and 1 if he liked it. Similarly, the second bit is the Ebert bit.

So if your friend asked you, "What was the verdict from Siskel and Ebert about that movie *Impolite Encounter*?" instead of answering, "Siskel gave it a thumbs up and Ebert gave it a thumbs down" or even "Siskel liked it; Ebert didn't," you could have simply said, "One zero." As long as your friend knew which was the Siskel bit and which was the Ebert bit, and that a 1 bit meant thumbs up and a 0 bit meant thumbs down, your answer would be perfectly understandable. But you and your friend have to know the code.

We could have declared initially that a 1 bit meant a thumbs down and a 0 bit meant a thumbs up. That might seem counterintuitive. Naturally, we like to think of a 1 bit as representing something affirmative and a 0 bit as the opposite, but it's really just an arbitrary assignment. The only requirement is that everyone who uses the code must know what the 0 and 1 bits mean.

The meaning of a particular bit or collection of bits is always understood contextually. The meaning of a yellow ribbon around a particular oak tree is probably known only to the person who put it there and the person who's supposed to see it. Change the color, the tree, or the date, and it's just a meaningless scrap of cloth. Similarly, to get some useful information out of Siskel and Ebert's hand gestures, at the very least we need to know what

movie is under discussion.

If you maintained a list of the movies that Siskel and Ebert reviewed and how they voted with their thumbs, you could add another bit to the mix to include your own opinion. Adding this third bit increases the number of different possibilities to eight:

000 = Siskel hated it; Ebert hated it; I hated it.

001 = Siskel hated it; Ebert hated it; I loved it.

010 = Siskel hated it; Ebert loved it; I hated it.

011 = Siskel hated it; Ebert loved it; I loved it.

100 = Siskel loved it; Ebert hated it; I hated it.

101 = Siskel loved it; Ebert hated it; I loved it.

110 = Siskel loved it; Ebert loved it; I hated it.

111 = Siskel loved it; Ebert loved it; I loved it.

One bonus of using bits to represent this information is that we know that we've accounted for all the possibilities. We know there can be eight and only eight possibilities and no more or fewer. With 3 bits, we can count only from zero to seven. There are no more 3-digit binary numbers.

Now, during this description of the Siskel and Ebert bits, you might have been considering a very serious and disturbing question, and that question is this: What do we do about *Leonard Maltin's Movie & Video Guide*? After all, Leonard Maltin doesn't do the thumbs up and thumbs down thing. Leonard Maltin rates the movies using the more traditional star system.

To determine how many Maltin bits we need, we must first know a few things about his system. Maltin gives a movie anything from 1 star to 4 stars, with half stars in between. (Just to make this interesting, he doesn't actually award a single star; instead, the movie is rated as a BOMB.) There are seven possibilities, which means that we can represent a particular rating using just 3 bits:

000 = BOMB

001 = *½

010 = **

011 = **½

100 = ***

101 = ***½

110 = ****

“What about 111?” you may ask. Well, that code doesn’t mean anything. It’s not defined. If the binary code 111 were used to represent a Maltin rating, you’d know that a mistake was made. (Probably a computer made the mistake because people never do.)

You’ll recall that when we had two bits to represent the Siskel and Ebert ratings, the leftmost bit was the Siskel bit and the rightmost bit was the Ebert bit. Do the individual bits mean anything here? Well, sort of. If you take the numeric value of the bit code, add 2, and then divide by 2, that will give you the number of stars. But that’s only because we defined the codes in a reasonable and consistent manner. We could just as well have defined the codes this way:

000 = ***

001 = *½

010 = **½

011 = ****

101 = ***½

110 = **

111 = BOMB

This code is just as legitimate as the preceding code so long as everybody knows what it means.

If Maltin ever encountered a movie undeserving of even a single full star, he could award a half star. He would certainly have enough codes for the half-star option. The codes could be redefined like so:

000 = MAJOR BOMB

001 = BOMB

010 = *½

011 = **

100 = **½

101 = ***

110 = ***½

111 = ****

But if he then encountered a movie not even worthy of a half star and decided to award no stars (ATOMIC BOMB?), he'd need another bit. No more 3-bit codes are available.

The magazine *Entertainment Weekly* gives grades, not only for movies but for television shows, CDs, books, CD-ROMs, Web sites, and much else. The grades range from A+ straight down to F (although it seems that only Pauly Shore movies are worthy of that honor). If you count them, you see 13 possible grades. We would need 4 bits to represent these grades:

0000 = F

0001 = D-

0010 = D

0011 = D+

0100 = C-

0101 = C

0110 = C+

0111 = B-

1000 = B

1001 = B+

1010 = A-

1011 = A

1100 = A+

We have three unused codes: 1101, 1110, and 1111, for a grand total of 16.

Whenever we talk about bits, we often talk about a certain *number* of bits. The more bits we have, the greater the number of different possibilities we can convey.

It's the same situation with decimal numbers, of course. For example, how many telephone area codes are there? The area code is three decimal digits long, and if all of them are used (which they aren't, but we'll ignore that), there are 10^3 , or 1000, codes, ranging from 000 through 999. How many 7-

digit phone numbers are possible within the 212 area code? That's 10^7 , or 10,000,000. How many phone numbers can you have with a 212 area code and a 260 prefix? That's 10^4 , or 10,000.

Similarly, in binary the number of possible codes is always equal to 2 to the power of the number of bits:

Number of Bits	Number of Codes
1	$2^1 = 2$
2	$2^2 = 4$
3	$2^3 = 8$
4	$2^4 = 16$
5	$2^5 = 32$
6	$2^6 = 64$
7	$2^7 = 128$
8	$2^8 = 256$
9	$2^9 = 512$
10	$2^{10} = 1024$

Every additional bit doubles the number of codes.

If you know how many codes you need, how can you calculate how many bits you need? In other words, how do you go backward in the preceding table?

The method you use is something called the *base two logarithm*. The logarithm is the opposite of the power. We know that 2 to the 7th power equals 128. The base two logarithm of 128 equals 7. To use more mathematical notation, this statement

$$2^7 = 128$$

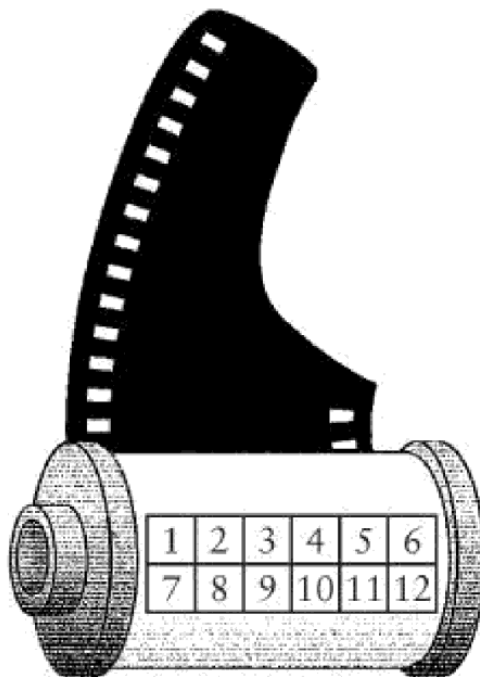
is equivalent to this statement:

$$\log_2 128 = 7$$

So if the base two logarithm of 128 is 7, and the base two logarithm of 256 is 8, then what's the base two logarithm of 200? It's actually about 7.64, but we really don't have to know that. If we needed to represent 200 different things with bits, we'd need 8 bits.

Bits are often hidden from casual observation deep within our electronic appliances. We can't see the bits encoded in our compact discs or in our digital watches or inside our computers. But sometimes the bits are in clear view.

Here's one example. If you own a camera that uses 35-millimeter film, take a look at a roll of film. Hold it this way:



You'll see a checkerboard-like grid of silver and black squares that I've numbered 1 through 12 in the diagram. This is called *DX-encoding*. These 12 squares are actually 12 bits. A silver square means a 1 bit and a black square means a 0 bit. Square 1 and square 7 are always silver (1).

What do the bits mean? You might be aware that some films are more sensitive to light than others. This sensitivity to light is often called the film *speed*. A film that's very sensitive to light is said to be *fast* because it can be exposed very quickly. The speed of the film is indicated by the film's ASA (American Standards Association) rating, the most popular being 100, 200, and 400. This ASA rating isn't only printed on the box and the film's cassette but is also encoded in bits.

There are 24 standard ASA ratings for photographic film. Here they are:

25	32	40
50	64	80
100	125	160
200	250	320
400	500	640
800	1000	1250
1600	2000	2500
3200	4000	5000

How many bits are required to encode the ASA rating? The answer is 5. We know that 2^4 equals 16, so that's too few. But 2^5 equals 32, which is more than sufficient.

The bits that correspond to the film speed are shown in the following table:

Square 2	Square 3	Square 4	Square 5	Square 6	Film Speed
-------------	-------------	-------------	-------------	-------------	---------------

0	0	0	1	0	25
0	0	0	0	1	32
0	0	0	1	1	40
1	0	0	1	0	50
1	0	0	0	1	64
1	0	0	1	1	80
0	1	0	1	0	100
0	1	0	0	1	125
0	1	0	1	1	160
1	1	0	1	0	200
1	1	0	0	1	250
1	1	0	1	1	320
0	0	1	1	0	400
0	0	1	0	1	500
0	0	1	1	1	640
1	0	1	1	0	800
1	0	1	0	1	1000
1	0	1	1	1	1250

0	1	1	1	0	1600
0	1	1	0	1	2000
0	1	1	1	1	2500
1	1	1	1	0	3200
1	1	1	0	1	4000
1	1	1	1	1	5000

Most modern 35-millimeter cameras use these codes. (Exceptions are cameras on which you must set the exposure manually and cameras that have built-in light meters but require you to set the film speed manually.) If you take a look inside the camera where you put the film, you should see six metal contacts that correspond to squares 1 through 6 on the film canister. The silver squares are actually the metal of the film cassette, which is a conductor. The black squares are paint, which is an insulator.

The electronic circuitry of the camera runs a current into square 1, which is always silver. This current will be picked up (or not picked up) by the five contacts on squares 2 through 6, depending on whether the squares are bare silver or are painted over. Thus, if the camera senses a current on contacts 4 and 5 but not on contacts 2, 3, and 6, the film speed is 400 ASA. The camera can then adjust film exposure accordingly.

Inexpensive cameras need read only squares 2 and 3 and assume that the film speed is 50, 100, 200, or 400 ASA.

Most cameras don't read or use squares 8 through 12. Squares 8, 9, and 10 encode the number of exposures on the roll of film, and squares 11 and 12 refer to the *exposure latitude*, which depends on whether the film is for black-and-white prints, for color prints, or for color slides.

Perhaps the most common visual display of binary digits is the ubiquitous Universal Product Code (UPC), that little bar code symbol that appears on virtually every packaged item that we purchase these days. The UPC has come to symbolize one of the ways computers have crept into our lives.

Although the UPC often inspires fits of paranoia, it's really an innocent little thing, invented for the purpose of automating retail checkout and inventory, which it does fairly successfully. When it's used with a well-designed checkout system, the consumer can have an itemized sales receipt, which isn't possible with conventional cash registers.

Of interest to us here is that the UPC is a binary code, although it might not seem like one at first. So it will be instructive to decode the UPC and examine how it works.

In its most common form, the UPC is a collection of 30 vertical black bars of various widths, divided by gaps of various widths, along with some digits. For example, this is the UPC that appears on the 10 $\frac{3}{4}$ -ounce can of Campbell's Chicken Noodle Soup:



We're tempted to try to visually interpret the UPC in terms of thin bars and black bars, narrow gaps and wide gaps, and indeed, that's one way to look at it. The black bars in the UPC can have four different widths, with the thicker bars being two, three, and four times the width of the thinnest bar. Similarly, the wider gaps between the bars are two, three, and four times the width of the thinnest gap.

But another way to look at the UPC is as a series of bits. Keep in mind that the whole bar code symbol isn't exactly what the scanning wand "sees" at the checkout counter. The wand doesn't try to interpret the numbers at the bottom, for example, because that would require a more sophisticated computing technique known as *optical character recognition*, or OCR. Instead, the scanner sees just a thin slice of this whole block. The UPC is as large as it is to give the checkout person something to aim the scanner at. The slice that the scanner sees can be represented like this:



This looks almost like Morse code, doesn't it?

As the computer scans this information from left to right, it assigns a 1 bit to the first black bar it encounters, a 0 bit to the next white gap. The subsequent gaps and bars are read as series of bits 1, 2, 3, or 4 bits in a row, depending on the width of the gap or the bar. The correspondence of the scanned bar code to bits is simply:



So the entire UPC is simply a series of 95 bits. In this particular example, the bits can be grouped as follows:

Bits	Meaning
101	Left-hand guard pattern
0001101	Left-side digits
0110001	
0011001	
0001101	
0001101	
0001101	
01010	Center guard pattern
1110010	Right-side digits
1100110	
1101100	
1001110	
1100110	
1000100	
101	Right-hand guard pattern

The first 3 bits are always 101. This is known as the *left-hand guard pattern*, and it allows the computer-scanning device to get oriented. From the guard pattern, the scanner can determine the width of the bars and gaps that correspond to single bits. Otherwise, the UPC would have to be a specific

size on all packages.

The left-hand guard pattern is followed by six groups of 7 bits each. Each of these is a code for a numeric digit 0 through 9, as I'll demonstrate shortly. A 5-bit center guard pattern follows. The presence of this fixed pattern (always 01010) is a form of built-in error checking. If the computer scanner doesn't find the center guard pattern where it's supposed to be, it won't acknowledge that it has interpreted the UPC. This center guard pattern is one of several precautions against a code that has been tampered with or badly printed.

The center guard pattern is followed by another six groups of 7 bits each, which are then followed by a right-hand guard pattern, which is always 101. As I'll explain later, the presence of a guard pattern at the end allows the UPC code to be scanned backward (that is, right to left) as well as forward.

So the entire UPC encodes 12 numeric digits. The left side of the UPC encodes 6 digits, each requiring 7 bits. You can use the following table to decode these bits:

Table 9-1. Left-Side Codes

0001101 = 0	0110001 = 5
<hr/>	
0011001 = 1	0101111 = 6
<hr/>	
0010011 = 2	0111011 = 7
<hr/>	
0111101 = 3	0110111 = 8
<hr/>	
0100011 = 4	0001011 = 9
<hr/>	

Notice that each 7-bit code begins with a 0 and ends with a 1. If the scanner encounters a 7-bit code on the left side that begins with a 1 or ends with a 0, it knows either that it hasn't correctly read the UPC code or that the code has been tampered with. Notice also that each code has only two groups of consecutive 1 bits. This implies that each digit corresponds to two vertical bars in the UPC code.

You'll see that each code in this table has an odd number of 1 bits. This is

another form of error and consistency checking known as *parity*. A group of bits has *even parity* if it has an even number of 1 bits and *odd parity* if it has an odd number of 1 bits. Thus, all of these codes have odd parity.

To interpret the six 7-bit codes on the right side of the UPC, use the following table:

Table 9-2. Right-Side Codes

1110010 = 0	1001110 = 5
1100110 = 1	1010000 = 6
1101100 = 2	1000100 = 7
1000010 = 3	1001000 = 8
1011100 = 4	1110100 = 9

These codes are the complements of the earlier codes: Wherever a 0 appeared is now a 1, and vice versa. These codes always begin with a 1 and end with a 0. In addition, they have an even number of 1 bits, which is even parity.

So now we're equipped to decipher the UPC. Using the two preceding tables, we can determine that the 12 digits encoded in the 10 3/4-ounce can of Campbell's Chicken Noodle Soup are

0 51000 01251 7

This is *very* disappointing. As you can see, these are precisely the same numbers that are conveniently printed at the bottom of the UPC. (This makes a lot of sense because if the scanner can't read the code for some reason, the person at the register can manually enter the numbers. Indeed, you've undoubtedly seen this happen.) We didn't have to go through all that work to decode them, and moreover, we haven't come close to decoding any secret information. Yet there isn't anything left in the UPC to decode. Those 30 vertical lines resolve to just 12 digits.

The first digit (a 0 in this case) is known as the *number system character*. A 0 means that this is a regular UPC code. If the UPC appeared on variable-weight grocery items such as meat or produce, the code would be a 2. Coupons are coded with a 5.

The next five digits make up the manufacturer code. In this case, 51000 is the code for the Campbell Soup Company. All Campbell products have this code. The five digits that follow (01251) are the code for a particular product of that company, in this case, the code for a 10 ¾-ounce can of chicken noodle soup. This product code has meaning only when combined with the manufacturer's code. Another company's chicken noodle soup might have a different product code, and a product code of 01251 might mean something totally different from another manufacturer.

Contrary to popular belief, the UPC doesn't include the price of the item. That information has to be retrieved from the computer that the store uses in conjunction with the checkout scanners.

The final digit (a 7 in this case) is called the *modulo check character*. This character enables yet another form of error checking. To examine how this works, let's assign each of the first 11 digits (0 51000 01251 in our example) a letter:

A BCDEF GHIJK

Now calculate the following:

$$3 \times (A + C + E + G + I + K) + (B + D + F + H + J)$$

and subtract that from the next highest multiple of 10. That's called the *modulo check character*. In the case of Campbell's Chicken Noodle Soup, we have

$$3 \times (0 + 1 + 0 + 0 + 2 + 1) + (5 + 0 + 0 + 1 + 5) = 3 \times 4 + 11 = 23$$

The next highest multiple of 10 is 30, so

$$30 - 23 = 7$$

and that's the modulo check character printed and encoded in the UPC.

This is a form of redundancy. If the computer controlling the scanner doesn't calculate the same modulo check character as the one encoded in the UPC, the computer won't accept the UPC as valid.

Normally, only 4 bits would be required to specify a decimal digit from 0 through 9. The UPC uses 7 bits per digit. Overall, the UPC uses 95 bits to encode only 11 useful decimal digits. Actually, the UPC includes blank space (equivalent to nine 0 bits) at both the left and the right side of the guard pattern. That means the entire UPC requires 113 bits to encode 11 decimal digits, or over 10 bits per decimal digit!

Part of this overkill is necessary for error checking, as we've seen. A product code such as this wouldn't be very useful if it could be easily altered by a customer wielding a felt-tip pen.

The UPC also benefits by being readable in both directions. If the first digits that the scanning device decodes have even parity (that is, an even number of 1 bits in each 7-bit code), the scanner knows that it's interpreting the UPC code from right to left. The computer system then uses this table to decode the right-side digits:

Table 9-3. Right-Side Codes in Reverse

0100111 = 0	0111001 = 5
0110011 = 1	0000101 = 6
0011011 = 2	0010001 = 7
0100001 = 3	0001001 = 8
0011101 = 4	0010111 = 9

and this table for the left-side digits:

Table 9-4. Left-Side Codes in Reverse

1011000 = 0	1000110 = 5
1001100 = 1	1111010 = 6
1100100 = 2	1101110 = 7
1011110 = 3	1110110 = 8
1100010 = 4	1101000 = 9

These 7-bit codes are all different from the codes read when the UPC is scanned from left to right. There's no ambiguity.

We began looking at codes in this book with Morse code, composed of dots, dashes, and pauses between the dots and dashes. Morse code doesn't immediately seem like it's equivalent to zeros and ones, yet it is.

Recall the rules of Morse code: A dash is three times as long as a dot. The dots and dashes of a single letter are separated by a pause the length of a dot. Letters within a word are separated by pauses equal in length to a dash. Words are separated by pauses equal in length to two dashes.

Just to simplify this analysis a bit, let's assume that a dash is twice the length of a dot rather than three times. That means that a dot can be a 1 bit and a dash can be two 1 bits. Pauses are 0 bits.

Here's the basic table of Morse code from [Chapter 2](#):