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 Nine

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.)

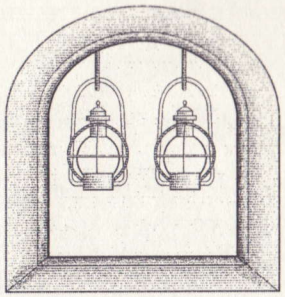
"Listen, my children, and you shall hear / Of the midnight ride of Paul Revere," wrote Henry Wadsworth Longfellow, and while he might not have been historically accurate when describing how Paul Revere alerted the American colonies that the British had invaded, he did provide a thought-provoking example 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..."

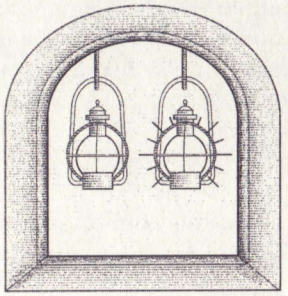
To summarize, Paul Revere's friend has two lanterns. If the British are invading by land, he will put just one lantern in the church tower. If the British are coming by sea, he will put both lanterns in the church tower.

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.

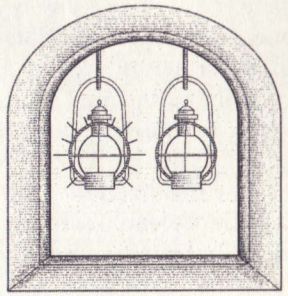
Let's assume that the two lanterns are actually permanent fixtures in the church tower. Normally they aren't lit:



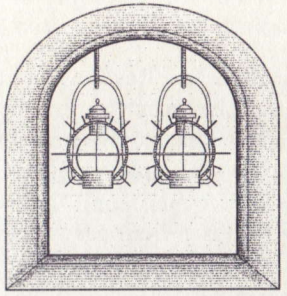
This means that the British aren't yet invading. If one of the lanterns is lit,



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

$$2^7 = 128$$

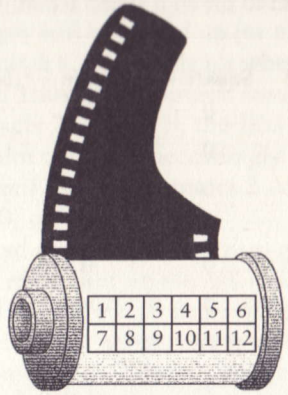
is equivalent to this statement:

$$\log_2 128 = 7$$

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 ¾-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:

10100011010110001001100100011010001101000110101010111001011001101101100100111011001101000100101

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:

Left-Side Codes

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

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:

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 ¾-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:

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:

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:

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	·−·		

Here's the table converted to bits:

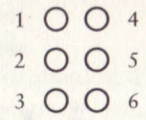
A	101100	J	101101101100	S	1010100
B	1101010100	K	110101100	T	1100
C	11010110100	L	1011010100	U	10101100
D	11010100	M	1101100	V	1010101100
E	100	N	110100	W	101101100
F	1010110100	O	1101101100	X	11010101100
G	110110100	P	10110110100	Y	110101101100
H	101010100	Q	110110101100	Z	11011010100
I	10100	R	10110100		

Notice that all the codes begin with a 1 bit and end with a pair of 0 bits. The pair of 0 bits represents the pause between letters in the same word. The code for the space between words is another pair of 0 bits. So the Morse code for "hi there" is normally given as

but Morse code using bits can look like the cross section of the UPC code:

```
1010101001010000110010101010010010111010010000
```


In terms of bits, Braille is much simpler than Morse code. Braille is a 6-bit code. Each character is represented by an array of six dots, and each of the six dots can be either raised or not raised. As I explained in Chapter 3, the dots are commonly numbered 1 through 6:



The word "code" (for example) is represented by the Braille symbols:

If a raised dot is 1 and a flat dot is 0, each of the characters in Braille can be represented by a 6-bit binary number. The four Braille symbols for the letters in the word "code" are then simply:

100100 101010 100110 100010

where the leftmost bit corresponds to the 1 position in the grid, and the rightmost bit corresponds to the 6 position.

As we shall see later in this book, bits can represent words, pictures, sounds, music, and movies as well as product codes, film speeds, movie ratings, an invasion of the British army, and the intentions of one's beloved. But most fundamentally, bits are numbers. All that needs to be done when bits represent other information is to count the number of possibilities. This determines the number of bits that are needed so that each possibility can be assigned a number.

Bits also play a part in *logic*, that strange blend of philosophy and mathematics for which a primary goal is to determine whether certain statements are true or false. True and false can also be 1 and 0.