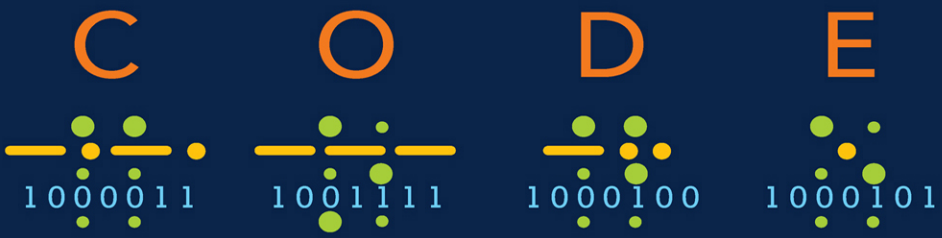
The Hidden Language of Computer Hardware and Software



SECOND EDITION

CHARLES PETZOLD

The Hidden Language of
Computer Hardware and Software

C O D E

SECOND EDITION

CHARLES PETZOLD

Code: The Hidden Language of Computer Hardware and Software: Second Edition
Published with the authorization of Microsoft Corporation by: Pearson Education, Inc.

ISBN-13: 978-0-13-790910-0
ISBN-10: 0-13-790910-1

Library of Congress Control Number: 2022939292

ScoutAutomatedPrintCode

Contents

About the Author

Charles Petzold is also the author of *The Annotated Turing: A Guided Tour through Alan Turing's Historic Paper on Computability and the Turing Machine* (Wiley, 2008). He wrote a bunch of other books too, but they're mostly about programming applications for Microsoft Windows, and they're all obsolete now. He lives in New York City with his wife, historian and novelist Deirdre Sinnott, and two cats named Honey and Heidi. His website is www.charlespetzold.com.

This page intentionally left blank

Preface to the Second Edition

The first edition of this book was published in September 1999. With much delight I realized that I had finally written a book that would never need revising! This was in stark contrast to my first book, which was about programming applications for Microsoft Windows. That one had already gone through five editions in just ten years. My second book on the OS/2 Presentation Manager (the what?) became obsolete much more quickly. But *Code*, I was certain, would last forever.

My original idea with *Code* was to start with very simple concepts but slowly build to a very deep understanding of the workings of digital computers. Through this steady progression up the hill of knowledge, I would employ a minimum of metaphors, analogies, and silly illustrations, and instead use the language and symbols of the actual engineers who design and build computers. I also had a very clever trick up my sleeve: I would use ancient technologies to demonstrate universal principles under the assumption that these ancient technologies were already quite old and would never get older. It was as if I were writing a book about the internal combustion engine but based on the Ford Model T.

I still think that my approach was sound, but I was wrong in some of the details. As the years went by, the book started to show its age. Some of the cultural references became stale. Phones and fingers supplemented keyboards and mice. The internet certainly existed in 1999, but it was nothing like what it eventually became. Unicode—the text encoding that allows a uniform representation of all the world's languages as well as emojis—got less than a page in the first edition. And JavaScript, the programming language that has become pervasive on the web, wasn't mentioned at all.

Those problems would probably have been easy to fix, but there existed another aspect of the first edition that continued to bother me. I wanted to show the workings of an actual CPU—the central processing unit that

forms the brain, heart, and soul of a computer—but the first edition didn't quite make it. I felt that I had gotten close to this crucial breakthrough but then I had given up. Readers didn't seem to complain, but to me it was a glaring flaw.

That deficiency has been corrected in this second edition. That's why it's some 70 pages longer. Yes, it's a longer journey, but if you come along with me through the pages of this second edition, we shall dive much deeper into the internals of the CPU. Whether this will be a more pleasurable experience for you or not, I do not know. If you feel like you're going to drown, please come up for air. But if you make it through Chapter 24, you should feel quite proud, and you'll be pleased to know that the remainder of the book is a breeze.

The Companion Website

The first edition of *Code* used the color red in circuit diagrams to indicate the flow of electricity. The second edition does that as well, but the workings of these circuits are now also illustrated in a more graphically interactive way on a new website called CodeHiddenLanguage.com.

You'll be reminded of this website occasionally throughout the pages of this book, but we're also using a special icon, which you'll see in the margin of this paragraph. Hereafter, whenever you see that icon—usually accompanying a circuit diagram—you can explore the workings of the circuit on the website. (For those who crave the technical background, I programmed these web graphics in JavaScript using the HTML5 `canvas` element.)

The CodeHiddenLanguage.com website is entirely free to use. There is no paywall, and the only advertisement you'll see is for the book itself. In a few of the examples, the website uses cookies, but only to allow you to store some information on your computer. The website doesn't track you or do anything evil.

I will also be using the website for clarifications or corrections of material in the book.

The People Responsible

The name of one of the people responsible for this book is on the cover; some others are no less indispensable but appear inside on the copyright and colophon pages.

In particular, I want to call out Executive Editor Haze Humbert, who approached me about the possibility of a second edition uncannily at precisely the right moment that I was ready to do it. I commenced work in January 2021, and she skillfully guided us through the ordeal, even as the book went several months past its deadline and when I needed some reassurance that I hadn't completely jumped the shark.

The project editor for the first edition was Kathleen Atkins, who also understood what I was trying to do and provided many pleasant hours of collaboration. My agent at that time was Claudette Moore, who also saw the value of such a book and convinced Microsoft Press to publish it.

The technical editor for the first edition was Jim Fuchs, who I remember catching a lot of embarrassing errors. For the second edition, technical reviewers Mark Seemann and Larry O'Brien also caught a few flubs and helped me make these pages better than they would have been otherwise.

I thought that I had figured out the difference between "compose" and "comprise" decades ago, but apparently I have not. Correcting errors like that was the invaluable contribution of copy editor Scout Festa. I have always relied on the kindness of copyeditors, who too often remain anonymous strangers but who battle indefatigably against imprecision and abuse of language.

Any errors that remain in this book are solely my responsibility.

I want to again thank my beta readers of the first edition: Sheryl Canter, Jan Eastlund, the late Peter Goldeman, Lynn Magalska, and Deirdre Sinnott (who later became my wife).

The numerous illustrations in the first edition were the work of the late Joel Panchot, who I understand was deservedly proud of his work on this book. Many of his illustrations remain, but the need for additional circuit diagrams inclined me to redo all the circuits for the sake of consistency. (More technical background: These illustrations were generated by a program I wrote in C# using the SkiaSharp graphics library to generate Scalable Vector Graphics files. Under the direction of Senior Content Producer Tracey Croom, the SVG files were converted into Encapsulated PostScript for setting up the pages using Adobe InDesign.)

And Finally

I want to dedicate this book to the two most important women in my life.

My mother battled adversities that would have destroyed a lesser person. She provided a strong direction to my life without ever holding me back. We celebrated her 95th (and final) birthday during the writing of this book.

My wife, Deirdre Sinnott, has been essential and continues to make me proud of her achievements, her support, and her love.

And to the readers of the first edition, whose kind feedback has been extraordinarily gratifying.

Charles Petzold
May 9, 2022

Pearson's Commitment to Diversity, Equity, and Inclusion

Pearson is dedicated to creating bias-free content that reflects the diversity of all learners. We embrace the many dimensions of diversity, including but not limited to race, ethnicity, gender, socioeconomic status, ability, age, sexual orientation, and religious or political beliefs.

Education is a powerful force for equity and change in our world. It has the potential to deliver opportunities that improve lives and enable economic mobility. As we work with authors to create content for every product and service, we acknowledge our responsibility to demonstrate inclusivity and incorporate diverse scholarship so that everyone can achieve their potential through learning. As the world's leading learning company, we have a duty to help drive change and live up to our purpose to help more people create a better life for themselves and to create a better world.

Our ambition is to purposefully contribute to a world where:

- Everyone has an equitable and lifelong opportunity to succeed through learning.
- Our educational products and services are inclusive and represent the rich diversity of learners.
- Our educational content accurately reflects the histories and experiences of the learners we serve.
- Our educational content prompts deeper discussions with learners and motivates them to expand their own learning (and worldview).

While we work hard to present unbiased content, we want to hear from you about any concerns or needs with this Pearson product so that we can investigate and address them.

- Please contact us with concerns about any potential bias at https://www.pearson.com/report-bias.html.

Chapter Six

Logic with Switches

What is truth? Aristotle thought that logic had something to do with it. The collection of his teachings known as the *Organon* (which dates from the fourth century BCE) is the earliest extensive writing on the subject of logic. To the ancient Greeks, logic was a means of analyzing language in the search for truth and thus was considered a form of philosophy. The basis of Aristotle's logic was the *syllogism*. The most famous syllogism (which isn't actually found in the works of Aristotle) is

All men are mortal;
Socrates is a man;
Hence, Socrates is mortal.

In a syllogism, two premises are assumed to be correct, and from these a conclusion is deduced.

The mortality of Socrates might seem straightforward enough, but there are many varieties of syllogisms. For example, consider the following two premises, proposed by the 19th-century mathematician Charles Dodgson (also known as Lewis Carroll):

All philosophers are logical;
An illogical man is always obstinate.

The conclusion—*Some obstinate persons are not philosophers*—isn't obvious at all. Notice the unexpected and disturbing appearance of the word *some*.

For over two thousand years, mathematicians wrestled with Aristotle's logic, attempting to corral it using mathematical symbols and operators.

Prior to the 19th century, the only person to come close was Gottfried Wilhelm von Leibniz (1648–1716), who dabbled with logic early in life but then went on to other interests (such as independently inventing calculus at the same time as Isaac Newton).

And then came George Boole.

Science & Society Picture Library/Getty Images

George Boole was born in England in 1815 into a world where the odds were certainly stacked against him. Because he was the son of a shoemaker and a former maid, Britain's rigid class structure would normally have prevented Boole from achieving anything much different from his ancestors. But aided by an inquisitive mind and his helpful father (who had strong interests in science, mathematics, and literature), young George gave himself the type of education that was normally the privilege of upper-class boys; his studies included Latin, Greek, and mathematics. As a result of his early papers on mathematics, in 1849 Boole was appointed the first Professor of Mathematics at Queen's College, Cork, in Ireland.

Several mathematicians in the mid-1800s had been working on a mathematical definition of logic (most notably Augustus De Morgan), but it was Boole who had the real conceptual breakthrough, first in the short book *The Mathematical Analysis of Logic, Being an Essay Towards a Calculus of Deductive Reasoning* (1847) and then in a much longer and more ambitious text, *An Investigation of the Laws of Thought on Which Are Founded the Mathematical Theories of Logic and Probabilities* (1854), more conveniently referred to as *The Laws of Thought*. Boole died in 1864, at the age of 49, after hurrying to class in the rain and contracting pneumonia.

The title of Boole's 1854 book suggests an ambitious motivation: Boole believed that the human brain uses logic to think, so if we were to find a way to represent logic with mathematics, we would also have a mathematical description of how the brain works. But Boole's mathematics can be studied without necessarily buying in to his neuropsychology.

Boole invented a whole different kind of algebra that was eventually called Boolean algebra to distinguish it from conventional algebra.

In conventional algebra, letters are often used to stand for numbers. These are called *operands*, and they are combined in various ways with *operators*, most often + and ×. For example:

$$A = 3 \times (B + 5)$$

When we do conventional algebra, we follow certain rules. These rules have probably become so ingrained in our practice that we no longer think of them as rules and might even forget their names. But rules indeed underlie all the workings of any form of mathematics.

The first rule is that addition and multiplication are *commutative*. That means we can switch around the symbols on each side of the operators:

$$A + B = B + A$$
$$A \times B = B \times A$$

By contrast, subtraction and division are *not* commutative.

Addition and multiplication are also *associative*, that is

$$A + (B + C) = (A + B) + C$$
$$A \times (B \times C) = (A \times B) \times C$$

And finally, multiplication is said to be *distributive* over addition:

$$A \times (B + C) = (A \times B) + (A \times C)$$

Another characteristic of conventional algebra is that it always deals with numbers, such as pounds of tofu or numbers of ducks or distances that a train travels or the seconds of a day.

It was Boole's genius to make algebra more abstract by divorcing it from concepts of number. In Boolean algebra, the operands refer not to numbers but instead to *classes*. A class is simply a group of things, similar to what in later times came to be known as a *set*.

Let's talk about cats. Cats can be either male or female. For convenience, we can use the letter M to refer to the class of male cats and F to refer to the class of female cats. Keep in mind that these two symbols do *not* represent numbers of cats. The number of male and female cats can change by the minute as new cats are born and old cats (regrettably) pass away. The letters stand for classes of cats—cats with specific characteristics. Instead of referring to male cats, we can just say "M."

We can also use other letters to represent the color of the cats. For example, T can refer to the class of tan cats, B can be the class of black cats, W the class of white cats, and O the class of cats of all "other" colors—all cats not in the class T, B, or W.

Finally (at least as far as this example goes), cats can be either neutered or unneutered. Let's use the letter N to refer to the class of neutered cats and U for the class of unneutered cats.

In conventional (numeric) algebra, the operators + and × are used to indicate addition and multiplication. In Boolean algebra, the same + and × symbols are used, and here's where things might get confusing. Everybody knows how to add and multiply numbers in conventional algebra, but how do we add and multiply *classes?*

Well, we don't actually add and multiply in Boolean algebra. Instead, the + and × symbols mean something else entirely.

The + symbol in Boolean algebra means a *union* of two classes. A union of two classes is everything in the first class combined with everything in the second class. For example, B + W represents the class of all cats that are either black or white.

The × symbol in Boolean algebra means an *intersection* of two classes. An intersection of two classes is everything that is in *both* the first class *and* the second class. For example, F × T represents the class of all cats that are both female and tan. As in conventional algebra, we can write F × T as F·T or simply FT (which is what Boole preferred). You can think of the two letters as two adjectives strung together: "female tan" cats.

To avoid confusion between conventional algebra and Boolean algebra, sometimes the symbols ∪ and ∩ are used for union and intersection instead of + and ×. But part of Boole's liberating influence on mathematics was to make the use of familiar operators more abstract, so I've decided to stick with his decision not to introduce new symbols into his algebra.

The commutative, associative, and distributive rules all hold for Boolean algebra. What's more, in Boolean algebra the + operator is distributive over the × operator. This isn't true of conventional algebra:

$$W + (B \times F) = (W + B) \times (W + F)$$

The union of white cats and black female cats is the same as the intersection of two unions: the union of white cats and black cats, and the union of white cats and female cats. This is somewhat difficult to grasp, but it works.

Three more symbols are necessary to complete Boolean algebra. Two of these symbols might look like numbers, but they're really not because they're treated a little differently than numbers. The symbol 1 in Boolean algebra means "the universe"—that is, everything we're talking about. In this example, the symbol 1 means "the class of all cats." Thus,

$$M + F = 1$$

This means that the union of male cats and female cats is the class of all cats. Similarly, the union of tan cats and black cats and white cats and other colored cats is also the class of all cats:

$$T + B + W + O = 1$$

And you achieve the class of all cats this way, too:

$$N + U = 1$$

The 1 symbol can be used with a minus sign to indicate the universe *excluding* something. For example,

$$1 - M$$

is the class of all cats except the male cats. The universe excluding all male cats is the same as the class of female cats:

$$1 - M = F$$

The third symbol that we need is the 0 (zero), and in Boolean algebra the 0 means an empty class—a class of nothing. The empty class results when we take an intersection of two mutually exclusive classes—for example, cats that are both male and female:

$$F \times M = 0$$

Notice that the 1 and 0 symbols sometimes work the same way in Boolean algebra as in conventional algebra. For example, the intersection of all cats and female cats is the class of female cats:

$$1 \times F = F$$

The intersection of no cats and female cats is the class of no cats:

$$0 \times F = 0$$

The union of no cats and all female cats is the class of female cats:

$$0 + F = F$$

But sometimes the result doesn't look the same as in conventional algebra. For example, the union of all cats and female cats is the class of all cats:

$$1 + F = 1$$

This doesn't make much sense in conventional algebra.

Because F is the class of all female cats, and (1 – F) is the class of all cats that aren't female, the union of these two classes is 1:

$$F + (1 - F) = 1$$

And the intersection of the two classes is 0:

$$F \times (1 - F) = 0$$

Historically, this formulation represents an important concept in logic: It's called the *law of contradiction*, and it indicates that something can't be both itself and the opposite of itself.

Where Boolean algebra really looks different from conventional algebra is in a statement like this:

$$F \times F = F$$

The statement makes perfect sense in Boolean algebra: The intersection of female cats and female cats is still the class of female cats. But it sure wouldn't look quite right if F referred to a number. Boole considered

$$X^2 = X$$

to be the single statement that differentiates his algebra from conventional algebra. Another Boolean statement that looks funny in terms of conventional algebra is this:

$$F + F = F$$

The union of female cats and female cats is still the class of female cats.

Boolean algebra provides a mathematical method for solving the syllogisms of Aristotle. Let's look at the first two-thirds of that famous syllogism again, but now using gender-neutral language:

All persons are mortal;
Socrates is a person.

We'll use P to represent the class of all persons, M to represent the class of mortal things, and S to represent the class of Socrates. What does it mean to say that "all persons are mortal"? It means that the intersection of the class of all persons and the class of all mortal things is the class of all persons:

$$P \times M = P$$

It would be wrong to say that $P \times M = M$, because the class of all mortal things includes cats, dogs, and elm trees.

Saying, "Socrates is a person" means that the intersection of the class containing Socrates (a very small class) and the class of all persons (a much larger class) is the class containing Socrates:

$$S \times P = S$$

Because we know from the first equation that P equals $(P \times M)$, we can substitute that into the second equation:

$$S \times (P \times M) = S$$

By the associative law, this is the same as

$$(S \times P) \times M = S$$

But we already know that $(S \times P)$ equals S, so we can simplify by using this substitution:

$$S \times M = S$$

And now we're finished. This formula tells us that the intersection of Socrates and the class of all mortal things is S, which means that Socrates is mortal. If we found instead that $(S \times M)$ equaled 0, we'd conclude that Socrates wasn't mortal. If we found that $(S \times M)$ equaled M, the conclusion would have to be that all mortals were Socrates!

Using Boolean algebra might seem like overkill for proving this obvious fact (particularly considering that Socrates demonstrated his mortality 2400 years ago), but Boolean algebra can also be used to determine whether something satisfies a certain set of criteria.

Perhaps one day you walk into a pet shop and say to the salesperson, "I want a male cat, neutered, either white or tan; or a female cat, neutered, any color but white; or I'll take any cat you have as long as it's black." And the salesperson says to you, "So you want a cat from the class of cats represented by the following expression:

$$(M \times N \times (W + T)) + (F \times N \times (1 - W)) + B$$

Right?" And you say, "Yes! Exactly!"

In verifying that the salesperson is correct, you might want to represent the concepts of union and intersection using the words OR and AND. I'm capitalizing these words because the words normally represent concepts in English, but they can also represent operations in Boolean algebra. When you form a union of two classes, you're actually accepting things from the first class OR the second class. And when you form an intersection, you're accepting only those things in both the first class AND the second class. In addition, you can use the word NOT wherever you see a 1 followed by a minus sign. In summary,

- + (a union) can also mean OR.
- × (an intersection) can also mean AND.
- 1 – (the universe without something) means NOT.

So the expression can also be written like this:

(M AND N AND (W OR T)) OR (F AND N AND (NOT W)) OR B

This is very nearly what you said. Notice how the parentheses clarify your intentions. You want a cat from one of three classes:

(M AND N AND (W OR T))
OR
(F AND N AND (NOT W))
OR
B

With this formula written down, the salesperson can perform something called a *Boolean test*. This involves another variation of Boolean algebra, where the letters refer to *properties* or *characteristics* or *attributes* of cats, and they can be assigned the numbers 0 or 1. The numeral 1 means Yes, True, this particular cat satisfies these criteria, while the numeral 0 means No, False, this cat doesn't satisfy these criteria.

First the salesperson brings out an unneutered tan male. Here's the expression of acceptable cats:

$$(M \times N \times (W + T)) + (F \times N \times (1 - W)) + B$$

And here's how it looks with 0s and 1s substituted:

$$(1 \times 0 \times (0 + 1)) + (0 \times 0 \times (1 - 0)) + 0$$

Notice that the only symbols assigned 1s are M and T because the cat is male and tan.

What we must do now is simplify this expression. If it simplifies to 1, the cat satisfies your criteria; if it simplifies to 0, the cat doesn't. While we're simplifying the expression, keep in mind that we're not really adding and multiplying, although generally we can pretend that we are. Most of the same rules apply when + means OR and × means AND. (Sometimes in modern texts the symbols ∧ and ∨ are used for AND and OR instead of × and +. But here's where the + and × signs perhaps ease the job, because the rules are similar to conventional algebra.)

When the × sign means AND, the possible results are

$$0 \times 0 = 0$$
$$0 \times 1 = 0$$
$$1 \times 0 = 0$$
$$1 \times 1 = 1$$

In other words, the result is 1 only if both the left operand AND the right operand are 1. This operation works exactly the same way as regular multiplication, and it can be summarized in a little table. The operation is shown in the upper-left corner, and the possible combinations of operators are shown in the top row and the left column:

AND	**0**	**1**
0	0	0
1	0	1

When the + sign means OR, the possible results are

$$0 + 0 = 0$$
$$0 + 1 = 1$$
$$1 + 0 = 1$$
$$1 + 1 = 1$$

The result is 1 if either the left operand OR the right operand is 1. This operation produces results very similar to those of regular addition, except

that in this case 1 + 1 equals 1. (If a cat is tan or if a cat is tan means that it's tan.) The OR operation can be summarized in another little table:

OR	0	1
0	0	1
1	1	1

We're ready to use these tables to calculate the result of the expression

$$(1 \times 0 \times 1) + (0 \times 0 \times 1) + 0 = 0 + 0 + 0 = 0$$

The result 0 means No, False, this kitty won't do.

Next the salesperson brings out a neutered white female. The original expression was

$$(M \times N \times (W + T)) + (F \times N \times (1 - W)) + B$$

Substitute the 0s and 1s again:

$$(0 \times 1 \times (1 + 0)) + (1 \times 1 \times (1 - 1)) + 0$$

And simplify it:

$$(0 \times 1 \times 1) + (1 \times 1 \times 0) + 0 = 0 + 0 + 0 = 0$$

And another poor kitten must be rejected.

Next the salesperson brings out a neutered gray female. (Gray qualifies as an "other" color—not white or black or tan.) Here's the expression:

$$(0 \times 1 \times (0 + 0)) + (1 \times 1 \times (1 - 0)) + 0$$

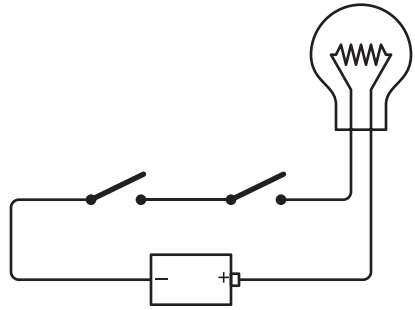
Now simplify it:

$$(0 \times 1 \times 0) + (1 \times 1 \times 1) + 0 = 0 + 1 + 0 = 1$$

The final result 1 means Yes, True, a kitten has found a home. (And it was the cutest one too!)

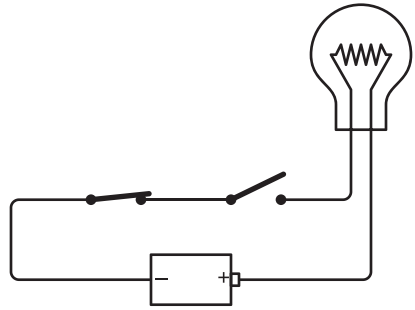
Later that evening, when the kitten is curled up sleeping in your lap, you wonder whether you could have wired some switches and a lightbulb to help you determine whether particular kittens satisfied your criteria. (Yes, you are a strange kid.) Little do you realize that you're about to make a crucial conceptual breakthrough. You're about to perform some experiments that will unite the algebra of George Boole with electrical circuitry and thus make possible the design and construction of digital computers. But don't let that intimidate you.

To begin your experiment, you connect a lightbulb and battery as you would normally, but you use two switches instead of one:

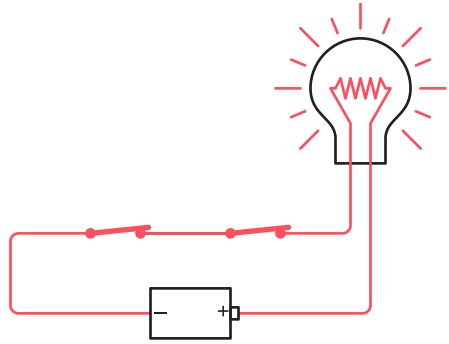


The world icon in the outer margin indicates that an interactive version of the circuit is available on the website CodeHiddenLanguage.com.

Switches connected in this way—one right after the other—are said to be wired *in series*. If you close the left switch, nothing happens:



Similarly, if you leave the left switch open and close the right switch, nothing happens. The lightbulb lights up only if both the left switch and the right switch are closed, as shown here:



The key word here is *and*. Both the left switch *and* the right switch must be closed for the current to flow through the circuit.

This circuit is performing a little exercise in logic. In effect, the lightbulb is answering the question "Are both switches closed?" We can summarize the workings of this circuit in the following table:

Left Switch	Right Switch	Lightbulb
Open	Open	Not lit
Open	Closed	Not lit
Closed	Open	Not lit
Closed	Closed	Lit

If you think of the switches and the lightbulb as Boolean operators, then these states can be assigned numbers of 0 and 1. A 0 can mean "switch is open" and a 1 can mean "switch is closed." A lightbulb has two states; a 0 can mean "lightbulb is not lit" and a 1 can mean "lightbulb is lit." Now let's simply rewrite the table:

Left Switch	Right Switch	Lightbulb
0	0	0
0	1	0
1	0	0
1	1	1

Notice that if we swap the left switch and the right switch, the results are the same. We really don't have to identify which switch is which. So the table can be rewritten to resemble the AND and OR tables that were shown earlier:

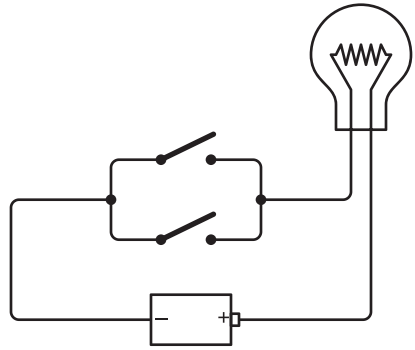
Switches in Series	0	1
0	0	0
1	0	1

And indeed, this is the *same* as the AND table. Check it out:

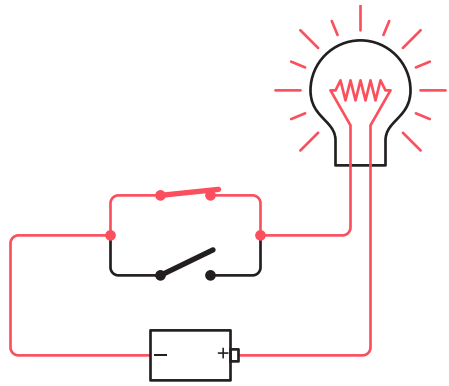
AND	0	1
0	0	0
1	0	1

This simple circuit is actually performing an AND operation in Boolean algebra.

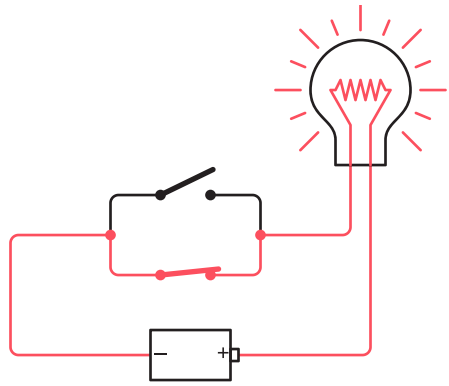
Now try connecting the two switches a little differently:



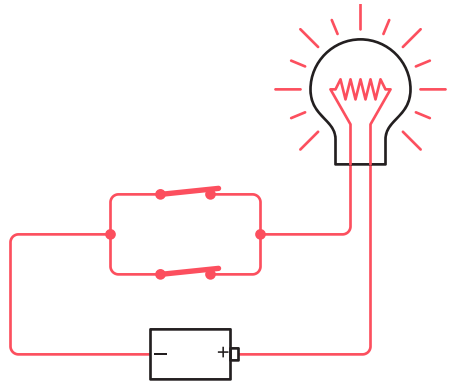
These switches are said to be connected *in parallel*. The difference between this and the preceding connection is that this lightbulb will light if you close the top switch:



or close the bottom switch:



or close both switches:



The lightbulb lights if the top switch *or* the bottom switch is closed. The key word here is *or*.

Again, the circuit is performing an exercise in logic. The lightbulb answers the question "Is either switch closed?" The following table summarizes how this circuit works:

Left Switch	Right Switch	Lightbulb
Open	Open	Not lit
Open	Closed	Lit
Closed	Open	Lit
Closed	Closed	Lit

Again, using 0 to mean an open switch or an unlit lightbulb and 1 to mean a closed switch or a lit lightbulb, this table can be rewritten this way:

Left Switch	Right Switch	Lightbulb
0	0	0
0	1	1
1	0	1
1	1	1

Again, it doesn't matter if the two switches are swapped, so the table can also be rewritten like this:

Switches in Parallel	0	1
0	0	1
1	1	1

And you've already guessed that this is the same as the Boolean OR:

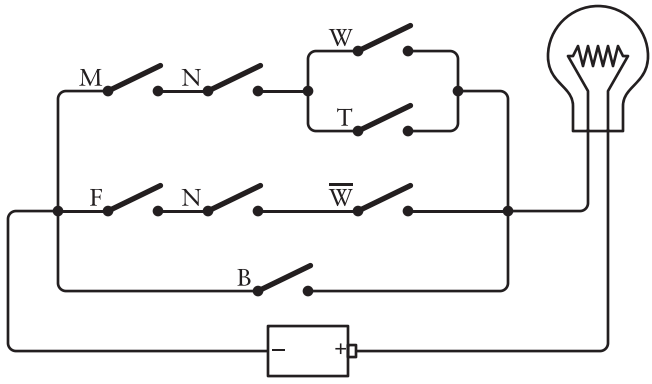
OR	**0**	**1**
0	0	1
1	1	1

This means that two switches in parallel are performing the equivalent of a Boolean OR operation.

When you originally entered the pet shop, you told the salesperson, "I want a male cat, neutered, either white or tan; or a female cat, neutered, any color but white; or I'll take any cat you have as long as it's black," and the salesperson developed this expression:

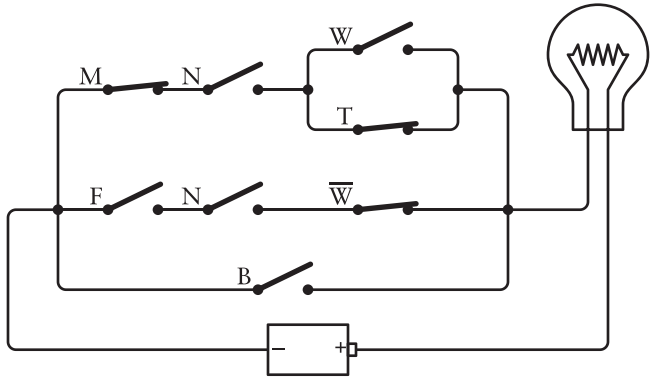
$$(M \times N \times (W + T)) + (F \times N \times (1 - W)) + B$$

Now that you know that two switches wired in series perform a logical AND (which is represented by a × sign) and two switches in parallel perform a logical OR (which is represented by the + sign), you can wire up eight switches like so:

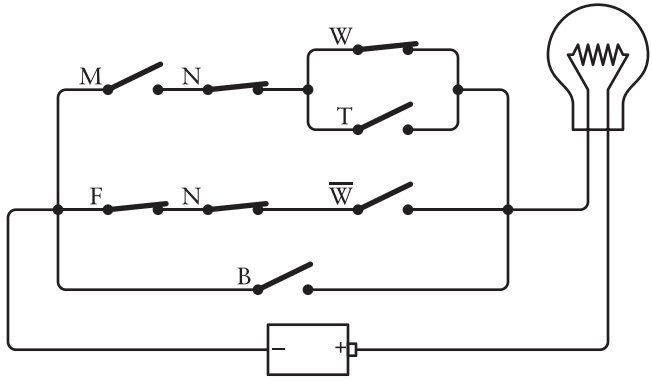


Each switch in this circuit is labeled with a letter—the same letters as in the Boolean expression. $\overline{W}$ means NOT W and is an alternative way to write 1 – W. Indeed, if you go through the wiring diagram from left to right starting at the top and moving from top to bottom, you'll encounter the letters in the same order in which they appear in the expression. Each × sign in the expression corresponds to a point in the circuit where two switches (or groups of switches) are connected in series. Each + sign in the expression corresponds to a place in the circuit where two switches (or groups of switches) are connected in parallel.

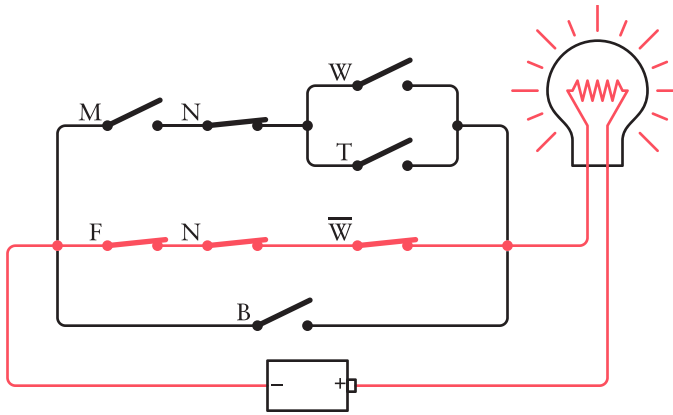
As you'll recall, the salesperson first brought out an unneutered tan male. Close the appropriate switches:



Although the M, T, and NOT $\overline{W}$ switches are closed, we don't have a complete circuit to light up the lightbulb. Next the salesperson brought out a neutered white female:



Again, the right switches aren't closed to complete a circuit. But finally, the salesperson brought out a neutered gray female:



And that's enough to complete the circuit, light up the lightbulb, and indicate that the kitten satisfies all your criteria.

George Boole never wired such a circuit. He never had the thrill of seeing a Boolean expression realized in switches, wires, and lightbulbs. One obstacle, of course, was that the incandescent lightbulb wasn't invented until 15 years after Boole's death. But the telegraph had been invented ten years before the publication of Boole's *The Laws of Thought*, and an important part of the telegraph system was a simple device that could perform operations of logic with much more agility than mere switches could.

Chapter Eleven

Bit by Bit by Bit

A story dating from at least the 1950s tells of a man traveling home after a stint in a distant prison. He doesn't know if he'll be welcomed back, so he requests a sign in the form of some cloth tied around a branch of a tree. In one version of the story, the man is traveling by train to his family, and he hopes to see a white ribbon on an apple tree. In another, he's traveling by bus to his wife, and he's looking for a yellow handkerchief on an oak tree. In both versions of the story, the man arrives to see the tree covered with hundreds of these banners, leaving no doubt of his welcome.

The story was popularized in 1973 with the hit song "Tie a Yellow Ribbon Round the Ole Oak Tree," and since then, displaying a yellow ribbon has also become a custom when family members or loved ones are away at war.

The man who requested that yellow ribbon 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, all the man really wanted was a simple yes or no. He wanted a yellow ribbon 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. Equally effective as the yellow ribbon (but perhaps more awkward to put into song lyrics) would be a traffic sign in the front yard: perhaps "Merge" or "Wrong Way."

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

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 you discovered in the two previous chapters, there's nothing 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).

There's nothing special about the decimal number system, but there *is* something special about binary, because binary is 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, and we can't do much of anything with just that.

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 one binary digit is a very small quantity indeed.

Sometimes when a word is invented, it also assumes a new meaning. That's certainly true in this case. Beyond the *binary digits* used by dolphins for counting, the bit has come to be regarded in the computer age 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, even if that information is as important as the yellow ribbon. 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.

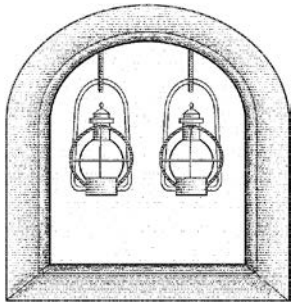
"Listen, my children, and you shall hear / Of the midnight ride of Paul Revere," wrote Henry Wadsworth Longfellow, and while he might not have been historically accurate when describing how Paul Revere alerted the American colonies that the British had invaded, he did provide a thought-provoking example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

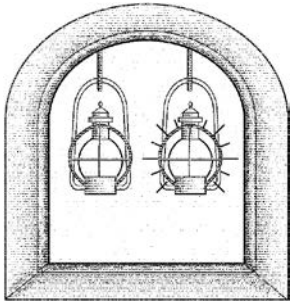
To summarize, Paul Revere's friend has two lanterns. If the British are invading by land, he will put just one lantern in the church tower. If the British are coming by sea, he will put both lanterns in the church tower.

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 circumstance will be conveyed by the *absence* of lanterns in the church tower.

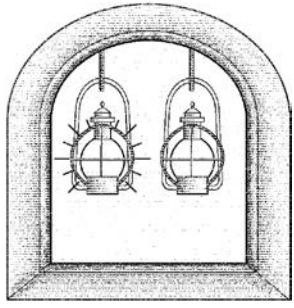
Let's assume that the two lanterns are actually permanent fixtures in the church tower. Normally they aren't lit:



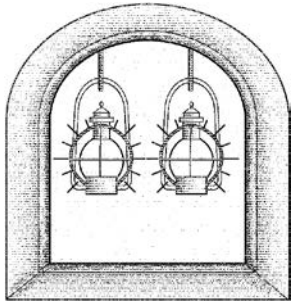
This means that the British aren't yet invading. If one of the lanterns is lit,



or



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



the British are coming by sea.

Each lantern is a bit and can be represented by a 0 or 1. The story of the yellow ribbon demonstrates 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 sufficed. 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 allow 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 offset the effect of *noise*. The word *noise* is used in communication theory to refer to anything that interferes with communication. A bad mobile connection is an obvious example of noise that interferes with a phone communication. Communication over the phone is usually successful 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*. 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 in which 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 not, 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.

For over a decade toward the end of the 20th century, the film critics Gene Siskel and Robert Ebert demonstrated a use of bits in the TV program they hosted, called *At the Movies*. After delivering their more detailed movie reviews they would issue a final verdict with a thumbs-up or a thumbs-down.

If those two thumbs are bits, they can represent four possibilities:

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 back in the day of *At the Movies*, if your friend asked you, "What was the verdict from Siskel and Ebert about that new 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," or if you converted to quaternary, "Two." 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 while watching *At the Movies* you maintained a list of the films and how Siskel and Ebert voted with their thumbs, you could have added 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 three-digit binary numbers. As you discovered toward the end of the previous chapter, these three-digit binary numbers can also be expressed as octal numbers 0 through 7.

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 the combinations of three digits 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 seven-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 math you need is 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

$$2^7 = 128$$

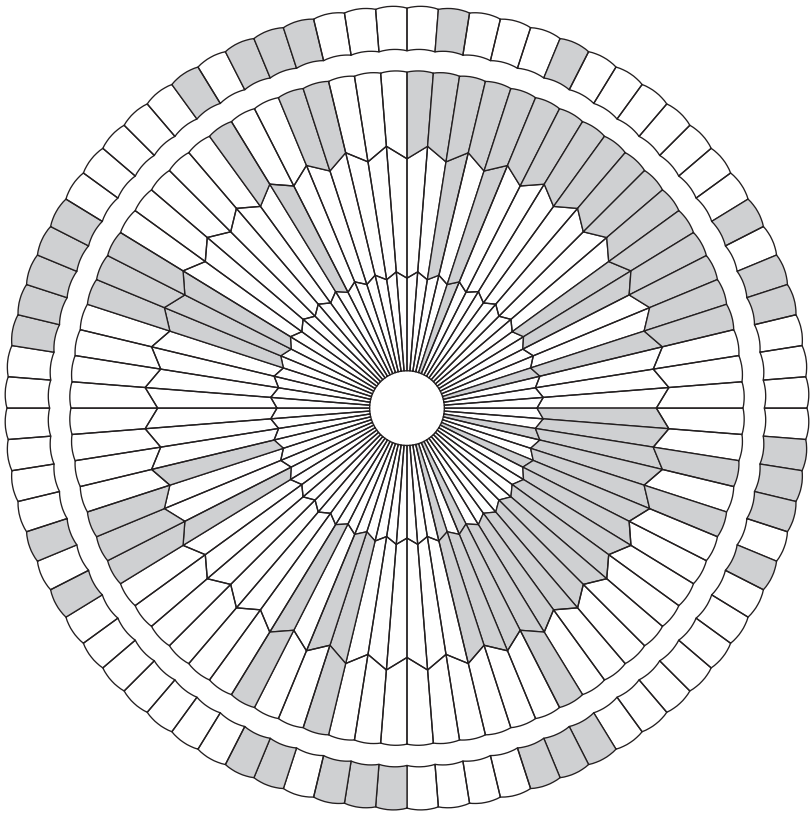
is equivalent to this statement:

$$\log_2 128 = 7$$

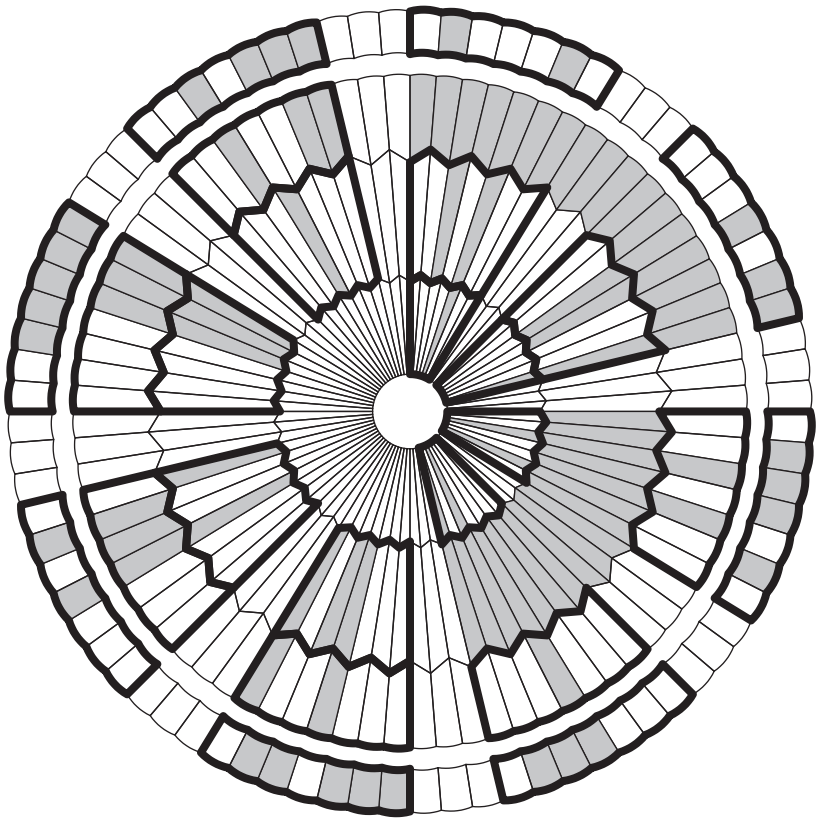
So if the base-two logarithm of 128 is 7 and the base-two logarithm of 256 is 8, then what's the base-two logarithm of numbers in between 128 and 256—for example, 200? It's actually about 7.64, but we really don't have to know that. If we needed to represent 200 different things with bits, we'd need 8 bits, just as when Paul Revere needed two lanterns to convey one of three possibilities. Going strictly by the mathematics, the number of bits required for Paul Revere's three possibilities is the base-two logarithm of 3, or about 1.6, but in a practical sense, he needed 2.

Bits are often hidden from casual observation deep within our electronic appliances. We can't see the bits encoded inside our computers, or streaming through the wires of our networks, or in the electromagnetic waves surrounding Wi-Fi hubs and cell towers. But sometimes the bits are in clear view.

Such was the case on February 18, 2021, when the *Perseverance* rover landed on Mars. The parachute seen in a photograph from the rover was assembled from 320 orange and white strips of fabric arranged in four concentric circles:

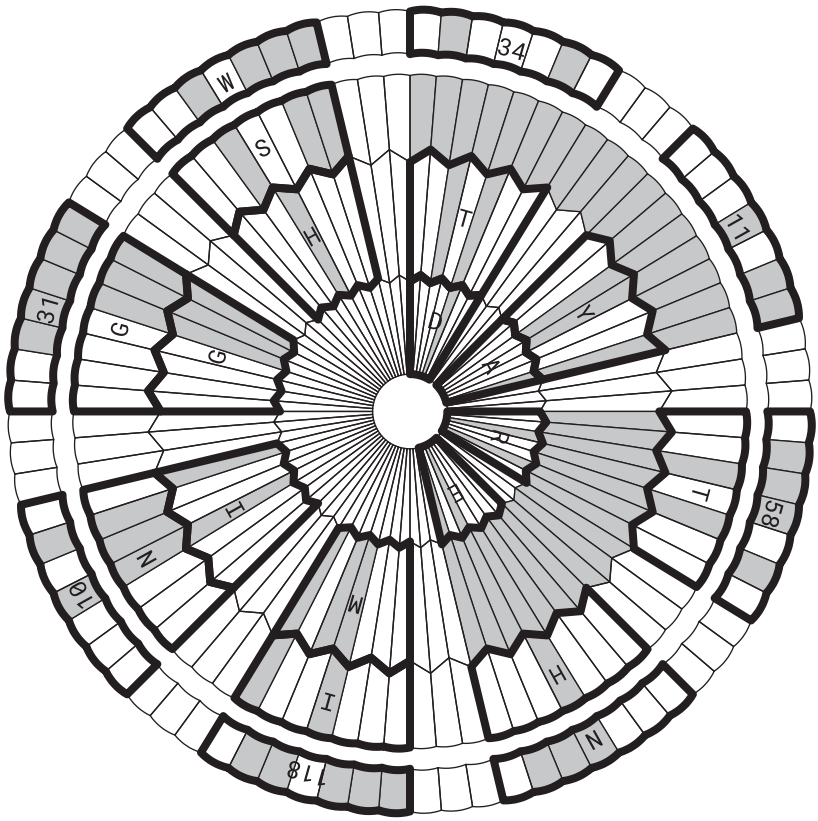


It didn't take long for Twitter users to decode the pattern. The key is to divide the strips of fabric into groups of seven containing both orange and white. These groups of seven strips are always separated by three white strips. The areas consisting of consecutive orange strips are ignored. In this diagram, each group of seven strips is surrounded by a heavy black line:



Each of these groups is a binary number with a white strip representing 0 and an orange strip representing 1. Right above the inner circle is the first group. Going clockwise, these seven strips encode the binary number 0000100, or decimal 4. The 4th letter of the alphabet is D. The next one going clockwise is 0000001, or decimal 1. That's an A. Next is 0010010, or decimal 18. The 18th letter of the alphabet is R. Next is 00000101, or decimal 5, which is an E. The first word is DARE.

Now jump to the next outer level. The bits are 0001101, or decimal 13, the letter M. When you finish, you'll spell out three words, a phrase that originated with Teddy Roosevelt and that has become the unofficial motto of the NASA Jet Propulsion Laboratory.



Around the outer circle are some encoded numbers as well, revealing the latitude and longitude of the Jet Propulsion Laboratory: 34°11′58″N 118°10′31″W. With the simple coding system used here, there's nothing that distinguishes letters and numbers. The numbers 10 and 11 that are part of the geographic coordinates could be the letters J and K. Only the context tells us that they're numbers.

Perhaps the most common visual display of binary digits is the ubiquitous Universal Product Code (UPC), that little barcode symbol that appears on virtually every packaged item that we purchase. The UPC is one of dozens of barcodes used for various purposes. If you have the printed version of this book, you'll see on the back cover another type of barcode that encodes the book's International Standard Book Number, or ISBN.

Although the UPC inspired some paranoia when it was first introduced, it's really an innocent little thing, invented for the purpose of automating retail checkout and inventory, which it does fairly successfully. Prior to the UPC, it wasn't possible for supermarket registers to provide an itemized sales receipt. Now it's commonplace.

Of interest to us here is that the UPC is a binary code, although it might not seem like one at first. It might be interesting 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 10¾-ounce can of Campbell's Chicken Noodle Soup:

That same UPC appeared in the first edition of this book. It hasn't changed in over 20 years!

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, or four times the width of the thinnest bar. Similarly, the wider gaps between the bars are two, three, or 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 barcode symbol isn't exactly what the scanner "sees" at the checkout counter. The scanner doesn't try to interpret the numbers printed 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? In fact, the original invention of scannable barcodes was partially inspired by Morse code.

As the computer scans this information from left to right, it assigns a 1 bit to the first black bar it encounters and a 0 bit to the next white gap. The subsequent gaps and bars are read as a series of 1, 2, 3, or 4 bits in a row, depending on the width of the gap or the bar. The correspondence of the scanned barcode 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-had 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. You'll see shortly how each of these is a code for a numeric digit 0 through 9. 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. This 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 six 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.

Examine these codes more closely, and you'll discover that they all have 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 opposites or *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 decimal digits encoded in the 10¾-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: 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 the numbers, and moreover, we haven't come close to revealing any secret information. Yet there isn't anything left in the UPC to decode. Those 30 vertical lines resolve to just 12 digits.

Of the 12 decimal digits, the first (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. You can try it out: 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. 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 right sides 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

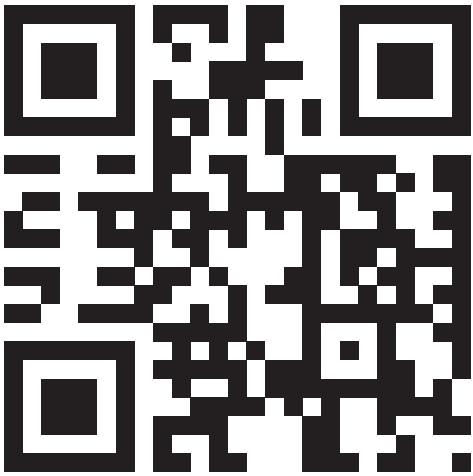
These 7-bit codes are all different from the codes read when the UPC is scanned from left to right. There's no ambiguity.

One way to cram more information in a scannable code is to move to two dimensions. Instead of a string of thick and thin bars and spaces, create a grid of black and white squares.

The most common two-dimensional barcode is probably the Quick Response (QR) code, first developed in Japan in 1994 and now used for a variety of purposes.

Creating your own QR code is free and easy. Several websites exist for that very purpose. Software is also readily available that can scan and decode QR codes through a camera on a mobile device. Dedicated QR scanners are available for industrial purposes, such as tracking shipments or taking inventory in warehouses.

Here's a QR code that encodes the URL of the website for this book, CodeHiddenLanguage.com:

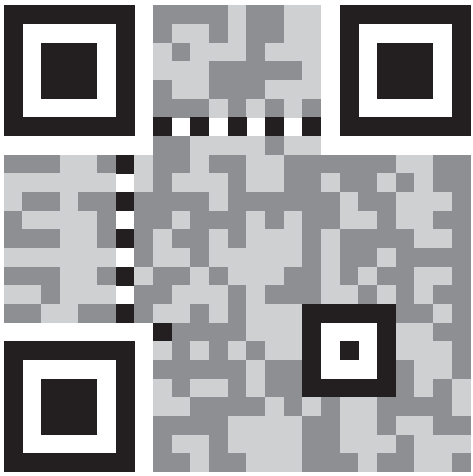


If you have an app on your mobile device that can read QR codes, you can point it at that image and go to the website.

QR codes consist of a grid of squares that are called *modules* in the official QR specification. This particular QR code has 25 modules horizontally and vertically, which is a size called Version 2. Forty different sizes of QR codes are supported; Version 40 has 177 modules horizontally and vertically.

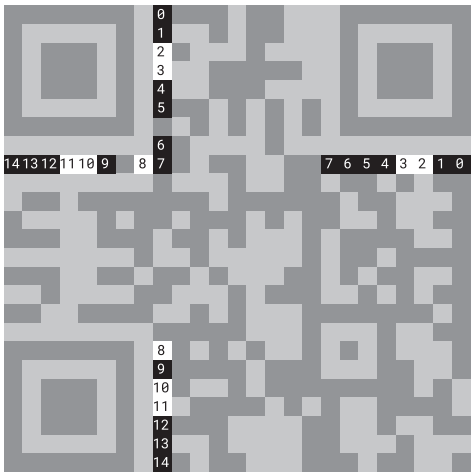
If each little block is interpreted as a bit—0 for white and 1 for black—a grid of this size potentially encodes 25 times 25, or 625 bits. But the real storage capability is about a third of that. Much of the information is devoted to a mathematically complex and sophisticated scheme of error correction. This protects the QR code from tampering and can also aid in recovering data that might be missing from a damaged code. I will not be discussing QR code error correction.

Mostly obviously, the QR code also contains several fixed patterns that assist the QR scanner in properly orienting the grid. In the following image, the fixed patterns are shown in black and white, and everything else is shown in gray:



The three large squares at the corners are known as *finder patterns*; the smaller square toward the lower right is known as an *alignment pattern*. These assist the QR code reader in properly orienting the code and compensating for any distortion. The horizontal and vertical sequences of alternating black and white cells near the top and at the left are called *timing patterns* and are used for determining the number of cells in the QR code. In addition, the QR code must be entirely surrounded by a quiet zone, which is a white border four times as wide as a cell.

Programs that create a QR code have several options, including different systems of error correction. Information required for a QR code reader to perform this error correction (and other tasks) is encoded in 15 bits called *format information*. These 15 bits appear twice in the QR code. Here are those 15 bits labeled 0 through 14 on the right and bottom of the upper-left finder pattern, and repeated below the upper-right finder pattern and to the right of the lower-left finder pattern:



Bits are sometimes labeled with numbers like this to indicate how they constitute a longer value. The bit labeled 0 is the least significant bit and appears at the far right of the number. The bit labeled 14 is the most significant bit and appears at the left. If white cells are 0 bits and black cells are 1 bits, here is that complete 15-bit number:

111001011110011

Why is bit 0 the least significant bit? Because it occupies the position in the full number corresponding to 2 to the zero power. (See the top of page 109 if you need a reminder of how bits compose a number.)

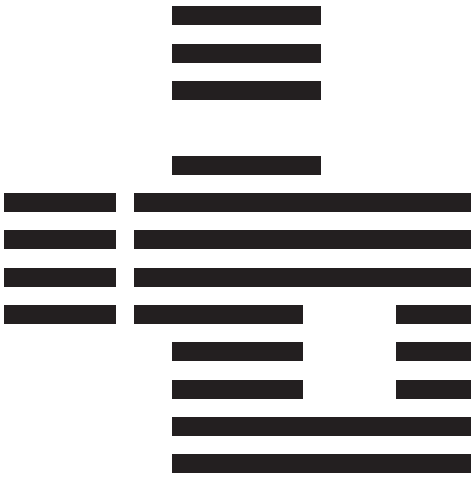
The actual numeric value of this 15-bit number is not important, because it consolidates three pieces of information. The two most significant bits indicate one of four error-correction levels. The ten least significant bits specify a 10-bit BCH code used for error correction. (BCH stands for the inventors of this type of code: Bose, Chaudhuri, and Hocquenghem. But I promised I wouldn't discuss the QR code error correction!)

In between the 2-bit error-correction level and the 10-bit BCH code are three bits that are *not* used for error correction. I've highlighted those three bits in bold:

111**001**011110011

It turns out that QR code readers work best when there are approximately an equal number of black and white squares. With some encoded information, this will not be the case. The program that creates the QR code is responsible for selecting a *mask pattern* that evens out the number of black and white squares. This mask pattern is applied to the QR code to flip selected cells from white or black, or black to white, and hence the bits that they represent from 0 to 1 and from 1 to 0.

The documentation of the QR code defines eight different mask patterns that can be specified by the eight 3-bit sequences 000, 001, 010, 011, 100, 101, 110, and 111. The value in the QR code that we're examining is 100, and that corresponds to a mask pattern consisting of a series of horizontal lines alternating every other row:



Every cell in the original QR code that corresponds to a white area in this mask remains unchanged. Every cell that corresponds to a black area must be flipped from white to black, or from black to white. Notice that the mask avoids altering the fixed areas and the QR information area. Here's what happens when this mask is applied to the original QR code:

The mask doesn't change the fixed and information areas. Otherwise, if you compare this image with the original QR code, you'll see that the top row is reversed in color, the second row is the same, the third row is reversed, and so on.

Now we're ready to start digging into the actual data. Begin with the four bits in the lower-right corner. In the following image, those cells are numbered 0 through 3, where 3 is the most significant bit and 0 is the least significant bit:

These four bits are known as the *data type* indicator, and they indicate what kind of data is encoded in the QR code. Here are a few of the possible values:

Date-Type Indicator	Meaning
0001	Numbers only
0010	Uppercase letters and numbers
0100	Text encoded as 8-bit values
1000	Japanese kanji

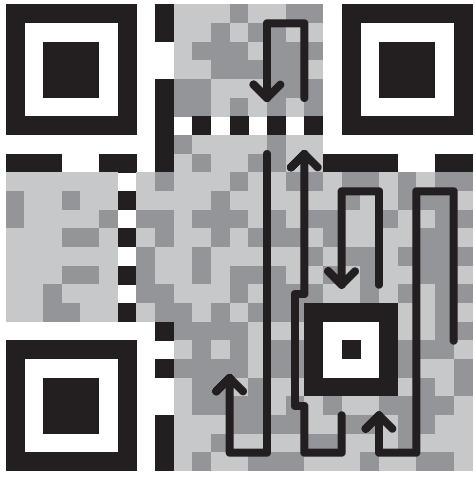
The value for this QR code is 0100, meaning that the data consists of 8-bit values that encode text.

The next item is stored in the eight cells above the data type indicator. These eight bits are numbered 0 through 7 in this illustration:

= 00011010 or 26 decimal

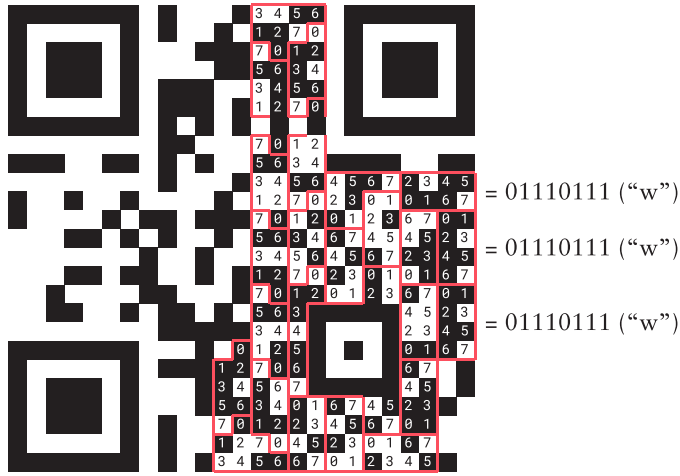
This value is 00011010, which is 26 in decimal. That's the number of characters encoded in the QR code.

The order of these characters is systematic but weird. The characters begin right above the character count. Each character usually—though not always—occupies an area that is two cells wide and four cells tall, and the characters wind through the grid like this:



Not all characters occupy areas that are two cells wide and four cells tall. Fortunately, the official QR specification is quite precise about how the bits are oriented when the area is not rectangular. In this next image,

the cells for each of the 26 characters are outlined in red, and the cells are numbered 0 through 7, where 0 denotes the least significant bit and 7 the most significant bit:



The QR specification indicates that text is encoded in the QR code using 8-bit values defined in a standard known as ISO/IEC 8859. That’s a fancy term for a variation of the American Standard Code for Information Interchange (ASCII), which I’ll be discussing in more detail in Chapter 13.

The first character is 01110111, which is the ASCII code for *w*. The next character up is the same. The next character extends to the left, but it is also another *w*. Now proceed down the next two pairs of columns. The next character is 00101110, which is the period, then 01000011, the uppercase C followed by 01101111, *o*. The next character straddles the next pair of rows. It’s 01100100: *d*. The next character begins below the alignment pattern and continues above it. The ASCII code is 01100101, which is *e*. Continue in this way to spell out www.CodeHiddenLanguage.com.

That’s it. Most of what’s left in the QR code is devoted to error correction.

Codes such as the UPC and QR certainly look forbidding at first glance, and people might be forgiven for assuming that they encode secret (and perhaps devious) information. But in order for these codes to be widely used, they must be well documented and publicly available. The more that they’re used, the more potentially valuable they become as another extension of our vast array of communication media.

Bits are everywhere, but toward the end of my discussion of the QR code, I referred to “8-bit values.” There’s a special word for 8-bit values. You may have heard of it.

Index

D

N

O

P