

Chapter 3

Digital Data

So far, chapters 1 and 2 have dealt with audio and video signals, respectively. Both of these have dealt with analog waveforms. In this chapter, we will discuss digital signals in general terms, but will not discuss digital *circuits* — if you are not familiar with the circuitry used for digital electronics, or need a refresher, you may wish to look at Appendix C.

Digital vs. Analog

Think about the speedometer in your car. Some cars have an analog speedometer, which has a round dial with a pointer which swings around to point to the speed. More ritzy cars may have a digital speedometer with a numeric display which shows the speed as a number like 45 or 53 miles per hour (mph).

Suppose you're moving at 53.7 miles per hour. With a digital speedometer, you have no way of knowing your exact speed — depending on its design, it will read either 53 or 54 (and possibly even flip back and forth between the two values), and you can't really tell that you're going at 53.7 mph.

With an analog speedometer, you may at the very least be able to tell whether you're closer to 53 or 54.

DETOUR 

And if (1) it is accurate, and (2) you don't mind taking your eyes off the road for a while so you can examine

 **END OF DETOUR**

the reading carefully, you may even be able to get closer than that.

Warning from our legal department — don't try this. It may be dangerous to your health!

There, in essence, you have the difference between an analog display and a digital display. The resolution of the digital display is limited by the number of digits — with a two-digit display, 53 or 54 is as close as you get. The resolution of the analog display is much better, if you take the time and care to read it properly.

Note that resolution and accuracy are not the same. Car speedometers are notoriously inaccurate — they may read 53 mph, for example, when you are really moving at only 49 miles per hour. But if you speed up by $\frac{1}{2}$ mile per hour, the pointer in the analog display will nudge up a little higher, whereas the digital may not go to the next reading. That is resolution — it lets you “resolve” or tell the difference between closely spaced values.

Now imagine that your car is actually a race car, and you want to build a telemetry system which will let your pit crew read out your speed around the track in *real time* — that is, while you are racing around the oval. If you have an analog speedometer, then the easiest way is to send an analog signal — one which will be proportional to the position of the pointer on your dial. For example, you could attach the pointer to a potentiometer, which would output a voltage proportional to your speed. This voltage could then be sent via a radio transmitter to the pit crew. The voltage would vary up and down in step with your speed — it would be an analog.

With a digital speedometer, on the other hand, it would be easiest to send the digits themselves through the radio link with some kind of code.

In one case, you would be sending an analog signal through an analog radio or *analog channel*; in the other, you would be sending digital information through a *digital channel*. The word *channel* describes the communications link itself.

There are, however, two other possible (though more complex) approaches. You could send the analog voltage from your analog speedometer through a digital channel by converting it with an analog-to-digital converter, or you could send the digits from your digital speedometer through an analog channel by converting the digital into an analog signal.

We will see such applications in future chapters, but for now let's stick to the basics. What are some of the advantages in analog vs. digital information?

- We have already mentioned the resolution. The resolution of a digital system is limited to the number of digits used for each reading. At first, that seems like a disadvantage, but keep in mind that it is always possible to increase the number of digits. For example, that car speedometer could have a six-digit display, and read out your speed as 53.7237 mph. Never mind the accuracy (it would be very expensive to measure the speed that accurately); it could still resolve the difference between 53.7237 mph and 53.7236 mph. No analog speedometer could hope to do that!
- Think also of what happens in the presence of noise. For example, as you drive, both the dashboard and your eyes are bouncing up and down. You may have a hard time seeing just where the analog speedometer's pointer is, whereas you can still tell the difference between 53 and 54 on a digital speedometer.

The presence of noise is a powerful factor in deciding whether analog or digital information is best. Suppose you have an analog system where you send a voltage that represents a speed. For example, 53 mph might be

sent as 5.3 volts, whereas 54 mph is sent as 5.4 volts. If there is ½ volt of noise (or ½ volt of voltage drop somewhere along the way), 5.3 volts going in might come out as 4.8 volts; your pit crew might think you are going at 48 mph when your true speed is 53.

Noise affects a digital system differently. Suppose 1 volt represents the digit 1, and 2 volts represents a 2, and so on (No, this is not the way it is done.) Then ½ volt of noise is unlikely to change one digit into another. In other words, a small amount of noise has virtually no effect. A large amount of noise, on the other hand, can make huge differences.

You may have noticed this with a cordless or cellular phone. An analog cordless or cellular phone will “degrade gracefully” as you get to the edge of its coverage area. As you get to the edge, things get gradually worse and worse, until you decide that it is just not worth continuing. A digital phone, on the other hand, will work perfectly up to some point, and then suddenly become tremendously noisy or distorted, or simply quit.

The trick in making digital transmission work is to make the difference between digits very large, so that even a large amount of noise will be unlikely to change one digit into another. The best way to do that is to have only two digits (0 and 1), so that it is easy to spot the difference between them, and difficult to mistake one for the other. In other words, binary numbers.

Binary Numbers

In personal communications, we humans generally use the decimal number system. In this system, we have the ten different digits 0 through 9. The word *decimal* comes from the Latin word for “ten”. Both computers and communications equipment, on the other hand, use the *binary* number system (the prefix “bi” comes from “two”). The most important reason is that these numbers are less likely to be misread as errors — how often have you had to look closely at a page to tell the difference between a 6 and an 8, for example? That would not be as likely to happen if you were to use only zeroes and ones.

So let’s think binary: Suppose someone asked you to count the beans in a jar, but specified that you are not allowed to use any number that has a 2, 3, 4, 5, 6, 7, 8, or 9 in it; only numbers with a 0 or a 1 are allowed. How would you count?

Like any good computer person, you’d start with zero, and count

0
1

But now you realize you’re not allowed to use anything from 2 through 9, and so you skip ahead to

10
11

Now you’re stuck again. You can’t write down 12, 13, or even 20 or 30 or 90, because they contain forbidden digits, so you skip ahead to

100
101

Now you again have to skip ahead to

110
111

and now you must skip a whole series of numbers until you get to

1000
1001

and so on.

Congratulations. In writing down the numbers

0
1
10
11
100
101
110
111
1000
1001

you have just invented the *binary number system*. Each of the counts in this table corresponds to one of the numbers of our decimal number system, as shown in Table 3-1.

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

The decimal number system has ten different digits, whereas the binary number system has only two. These two *binary digits* are called *bits*.

Binary numbers are used because, with just two different digits, it’s less likely that they will be confused. For example, in a typical digital circuit, the 0 bit might be something “near” 0 volts, while the 1 bit might be something “near” 3 or 5 volts. As long as the 0 voltage

doesn't get too big, or the 1 doesn't get too small, circuitry can still reliably tell the difference between them. If you tried to represent decimal digits with ten different voltages, it might be too difficult to tell the difference between one digit and another if the voltage changed a bit as it travels from one place to another.

Although the numbers in Table 3-1 are all different lengths, note that the value of a number doesn't change if you put extra zeroes in front of it. In decimal, for example, 7, 007, and even 00000007 all have the same value. In computers, binary numbers are often stored in groups of eight bits, called a *byte*. Thus the binary equivalent of a 7 would usually appear as 00000111, rather than just 111.

Looking at Table 3-1 again, we note that, if you were limited to one-bit-long numbers, you could only express two different numbers: 0 and 1. If you used two-bit-long numbers, then you could express four different numbers: 00, 01, 10, and 11, which correspond to the decimal numbers 0, 1, 2, and 3. Likewise, a three-bit number can have eight different values, which correspond to the decimal numbers 0 through 7. We can generalize this rule as follows: a number with n bits can have 2^n different values, corresponding to the decimal numbers from 0 through $2^n - 1$. For example, an eight-bit byte can have 2^8 or 256 different values, which correspond to the decimal numbers from 0 through 255.

Number conversions

Suppose you are given the binary number 01000110; what does it equal in decimal? One way to answer this question would be to enlarge Table 3-1. For example, the last line of the table tells us that binary 1010 equals the decimal number 10. The next line would then say that binary 1011 is decimal 11; the following line would say that binary 1100 is decimal 12, and so on. But it would take us quite a while until we got to the line that says that binary 01000110 is the decimal number 70. Fortunately, there is a shortcut method.

Consider the decimal number 358 or "three hundred fifty eight." We can write this out as

$$3 \text{ hundreds} + 5 \text{ tens} + 8 \text{ ones}$$

Note that 358 is very different from 538 or 853; the position of each digit is significant. The 8 on the right end, for example, is the units digit; the digit to its left is the tens, the one to its left is the hundreds, and so on.

To clearly show the importance of position, we could write the 358 as

$$(3 \times 10^2) + (5 \times 10^1) + (8 \times 10^0)$$

The powers of ten clearly show that the rightmost digit is the 10^0 or units digit (since a number to the 0 power equals 1), the next left digit is the 10^1 or tens digit, and the left digit is the 10^2 or hundreds digit. If our number was longer, the next digit to the left would be the 10^3 or thousands digit, and so on.

Binary numbers could be written in exactly the same way, except that we use powers of 2 rather than powers of 10. For example, the binary number 1001 (which Table 3-1 tells us is a decimal 9) could be written as

$$(1 \times 2^3) + (0 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$$

This clearly adds up to $8 + 0 + 0 + 1$, or 9. Here we see that the rightmost digit is the 2^0 or units digit, the next left digit has a value of 2^1 or 2 (except that it is multiplied by 0, so it adds nothing to the result), and so on.

So back to our original question — what is the binary 01000110 equal to? We could write it out like this:

$$(0 \times 2^7) + (1 \times 2^6) + (0 \times 2^5) + (0 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (0 \times 2^0)$$

which simplifies to

$$(1 \times 2^6) + (1 \times 2^2) + (1 \times 2^1) = 64 + 4 + 2 = 70$$

Hexadecimal numbers

For a computer, telling the difference between the number 0010011010011101 and 0010010010011101 is relatively easy, but for us humans it is not. As a crutch, humans often use the hexadecimal number system (sometimes just called *hex*) to help them read binary numbers. (Hexa means six, so hexa-decimal means 6 and 10, or sixteen.)

With hexadecimal numbers, long binary numbers like 0010011010011101 are generally split into groups of four bits, like this: 0010 0110 1001 1101, and then each group of four bits is replaced by its equivalent from Table 3-2.

Binary	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A

Binary	Hexadecimal
1011	B
1100	C
1101	D
1110	E
1111	F

Thus the number 0010 0110 1001 1101 becomes 269D. The number 0010010010011101, on the other hand, becomes 0010 0100 1001 1101 or 249D. We humans then find it easy to tell the difference.

It is important to remember that hexadecimal numbers are simply a crutch to make our job easier — they are not actually used within a computer or digital system, which is purely binary inside. Even when you type a hex number like 4F on a keyboard, the computer will quickly convert it into its internal binary 01001111.

Computer Codes

A typical byte, such as the number 01000110 in a computer's memory, could represent one of several things:

- A number, or part of a number, used in some computation. For example, the 01000110 could just mean 70, or it might be part of a longer number, such as 011110100100011000011101
- A coded instruction, or part of an instruction, telling a computer what to do. For example, in some particular computer, the number 01000110 might be a code that means “add two numbers”.
- A numeric code for a letter, number, or punctuation mark, usually coded in ASCII. These are often called *alphanumeric characters*, or just *characters*. The 01000110, for example, could stand for the letter F.

Since the same byte could stand for several different things, the sender and the receiver clearly must have some way of knowing which is which; this is done by examining the context.

ASCII

ASCII stands for the American Standard Code for Information Interchange, a common code which is used to convert alphanumeric characters into bits so they can be stored in a computer or transmitted through a communications network. There are other codes as well, but ASCII has become almost universal.

ASCII actually consists of seven bits, not eight. But because most computers today store numbers in groups of eight bits (bytes), an extra eighth bit is therefore often put in front to stretch the number to a full 8 bits. Table 3-3 shows the normal 7-bit ASCII code.

For example, the number 01000110 consists of an extra 0, followed by the seven bits 1000110. Separating these seven bits into the first three bits (100) and the last four bits (0110), we go to the 100 column and the 0110 row and find the letter F. So an F would be coded as 01000110, whereas the lower case f would be 01100110.

Table 3-3 consists of six columns of “printable” characters — characters which actually print on a page or display on the screen (although there are two that really don't: code 0100000 which is the blank space used between words, and 1111111 which is a delete character originally used to allow you to sort of “cross out” a mistake.

The left two columns, columns 000 and 001, hold the *control characters*. These do not print; rather, they are used to control the transmission of text. Some of these are seldom used; others are very common. For example, CR is the *carriage return* or enter key; LF is a *line feed* which tells a printer to go to the next line; BS is the *backspace* key, and ESC is the *escape* key.

The control characters are *usually* generated by using the CTRL key on a keyboard, which forces the

1 st 3 →	000	001	010	011	100	101	110	111
last 4 ↓								
0000	Null	DLE	space	0	@	P	'	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	\$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	Bell	ETB	'	7	G	W	g	w
1000	BS	CAN	(8	H	X	h	x
1001	Tab	EM)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[k	{
1100	FF	FS	,	<	L	\	l	
1101	CR	GS	-	=	M]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL
	Control characters		Printable characters					

first bit of the ASCII code to change to a 0. For example, the M key generates the code 1001101; holding the CTRL key while pressing the M changes the first bit from 1 to 0, and thus generates the 0001101 code, which is a CR. (We say *usually*, because many programs can choose to interpret the CTRL key in other ways.)

The Eighth Bit

The ASCII code is a seven-bit code, whereas most computer memory or data transmission deals with groups of eight bits, so we must insert an extra eighth bit to fill the empty space. So what does the eighth bit do? This extra bit can be treated in several ways.

- It can be left unused. But since you can't just leave the bit empty, you must put something into it. Hence you could routinely just force it to be a 0 or a 1. In this case, the lower-case letter *a*, for example, could be stored as 01100001 in some computers, and 11100001 in others.
- It can be used to increase the number of available characters. For instance, in most personal computers, this extra bit is a 0 for all normal ASCII characters, but becomes a 1 for special characters. For example, 01100001 in such a computer is still an *a*, but 11100001 is used to store the greek letter β . This allows the computer to use an extended character set which includes symbols like \pm , π , $\sqrt{\quad}$, \geq , and $^\circ$, which are not in the regular ASCII code. This code is sometimes called *extended ASCII*.
- It can be used for *parity* error checking. This is not usually done in small computers, where errors tend to be rare because signals travel short distances or because other methods are used to check and correct errors. But parity checking is more common when data travels a long distance.

Parity

Parity comes in two types — *even parity* and *odd parity*. In even parity, the eighth bit is chosen so that the total number of ones in the byte is even; in odd parity it is made odd. For instance, the ASCII code for *a* is 1100001, which has an odd number of ones. For even parity, an extra 1 would be added so that the total number of ones in 11100001 is even; in odd parity the extra bit would be 0 to keep an odd number of ones in 01100001. Another example is the capital letter A, whose ASCII code is 1000001. This would become 01000001 in even parity, and 11000001 in odd parity.

If you've ever seen abbreviations like 8N1 or 7E1, now you can understand what this means:

- 8N1 means eight data bits with No parity bit, and one stop pulse (we will explain the stop pulse in a later chapter.)
- 7E1 means seven data bits plus an Even parity bit, and one stop pulse.

Error detection and error correction

The idea behind parity is that every character sent has a specific 0 or 1 bit in that eighth or parity position, determined by the rest of the bits. If any one of the bits in that group is somehow changed due to an error in transmission, the number of ones will add up to the wrong number (it will either change from odd to even, or from even to odd), and the receiver will detect that an error has occurred. But note that if two bits (or any even number of bits) get changed, the error can't be detected. For instance, if the 01100001 for an odd-parity *a* gets changed to 01100111, a two-bit change, the number of ones is still odd, but the *a* was changed to a *g* without the receiver being able to detect a parity error.

Detecting errors with a parity bit was more useful years ago than it is now. Back when data transmission was very slow, a click or other short noise pulse in the system might corrupt just one bit; parity checking was then likely to catch the error. With modern high-speed data transmission, most problems are likely to corrupt more than one bit. In that case, using just one parity bit, the chance of an error occurring but the parity still accidentally being correct is 50%; hence the chance of catching an error is only 50%, or 1 out of 2.

Detecting, and even possibly correcting, errors in digital data is important, because an error in even just one bit can mean a huge mistake. Hence the parity bit is a useful step, but not enough. We will discuss these better methods later, but for now let us simply understand that

- Methods do exist for detecting errors, and
- Once you detect an error, it is also possible to correct that error, either through some complex method (like fortune-telling?) or simply by asking the sender to repeat the message

Hence it is possible to send a digital message from one place to another with absolutely no errors, even over long distances. This is certainly something that could not be done with analog transmission. Let's apply this knowledge to the telephone network.

In the early days of telephones, all connections were analog. When you placed a long-distance call from, for example, New York to California, your voice would be carried in analog form (as a voltage or current wave-

form) from coast to coast. Since there is a lot of wire needed to connect the east coast to the west, the signal would gradually get corrupted by noise and get weaker. Hence, every few miles, amplifiers called *repeaters* would be needed to get the signal strength back up to a reasonable level. These amplifiers would amplify your voice, but would also amplify any noise or distortion that had become mixed with the signal. By the time it arrived at the west coast, all this noise and distortion had accumulated to give a fairly noisy signal.

This changed when the telephone network was digitized. As soon as your voice signal arrives at the local telephone company building (and sometimes even sooner), it is digitized into a digital signal. That digital signal is then sent on its way. As before, every now and then that signal has to be amplified because it loses signal strength. But instead of a repeater amplifier, the telephone carriers use a *regenerator*. This is a device which receives the ones and zeroes, separates them from any noise that had been added along the way (and note — it is possible to do so without any errors), and then recreates a new signal (regenerates) with new bits. In other words, instead of simply amplifying the old signal, it generates a completely new signal without any noise. In this way, the digital signal representing your voice can arrive at the west coast with exactly the same bits as when it left your neighborhood. It is then converted back to an analog voice signal and sent to your friend on the west coast. It does not matter whether you are 3000 feet apart, or 3000 miles — the voice quality is the same.

Conclusion

This chapter has really dealt with two ideas: that there are analog signals and digital signals, and also that there are analog channels and digital channels to send them through. These can be combined in four ways:

You can send

- analog signals through an analog channel
- digital signals through a digital channel
- analog signals through a digital channel (by converting them)
- digital signals through an analog channel (also by converting them)

Traditional communications, up through the 1960's or so, dealt with analog information like voice or pictures, traveling through analog channels. The digital approach did not come until later, with the development of solid state devices, integrated circuits, and microprocessors.

Digital channels have the advantage of being able to move information great distances without error. This is clearly an advantage for digital information like com-

puter files. But it is even an advantage for analog information like telephone calls, which can stand some errors like noise and distortion, but which can be improved by using digital methods.

Since much of electronic communications was developed in the analog days, and still forms the basis of much of today's technology, we will explain analog methods first in this book. Purely digital methods will come later.