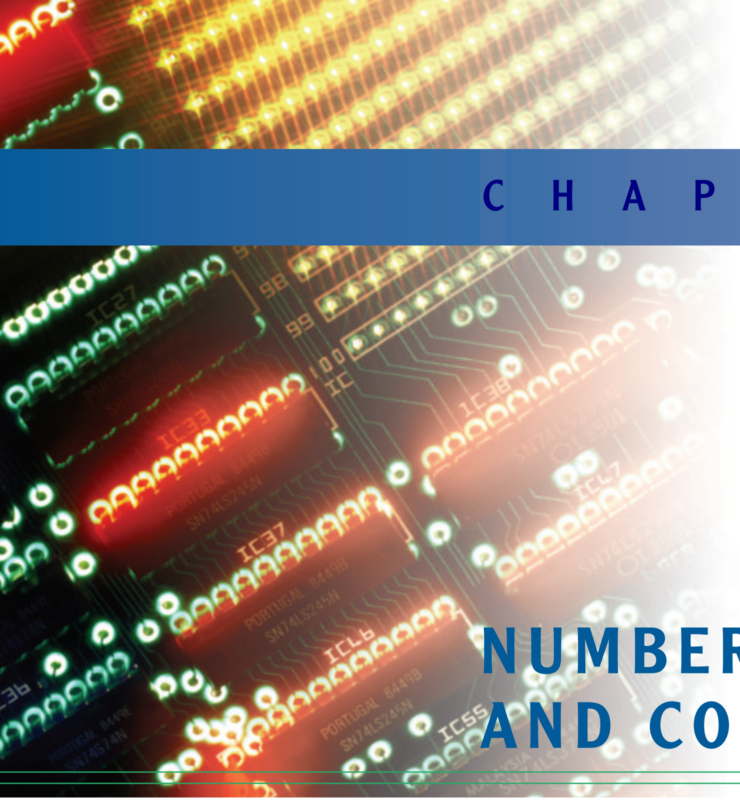
CHAPTER 1.2



NUMBER SYSTEMS AND CODES

INTRODUCTION

The binary number system is the most important one in digital systems, but several others are also important. The decimal system is important because it is universally used to represent quantities outside a digital system. This means that there will be situations where decimal values must be converted to binary values before they are entered into the digital system. For example, when you punch a decimal number into your hand calculator (or computer), the circuitry inside the machine converts the decimal number to a binary value.

Likewise, there will be situations where the binary values at the outputs of a digital system must be converted to decimal values for presentation to the outside world. For example, your calculator (or computer) uses binary numbers to calculate answers to a problem and then converts the answers to decimal digits before displaying them.

As you will see, it is not easy to simply look at a large binary number and convert it to its equivalent decimal value. It is very tedious to enter a long sequence of 1s and 0s on a keypad, or to write large binary numbers on a piece of paper. It is especially difficult to try to convey a binary quantity while speaking to someone. The hexadecimal (base-16) number system has become a very standard way of communicating numeric values in digital systems. The great advantage is that hexadecimal numbers can be converted easily to and from binary.

Other methods of representing decimal quantities with binary-encoded digits have been devised that are not truly number systems but offer the ease of conversion between the binary code and the decimal number system. This is referred to as binary-coded decimal.

2-1 BINARY-TO-DECIMAL CONVERSIONS

As explained in Chapter 1, the binary number system is a positional system where each binary digit (bit) carries a certain weight based on its position relative to the LSB. Any binary number can be converted to its decimal equivalent simply by summing together the weights of the various positions in the binary number that contain a 1. To illustrate, let's change 11011_2 to its decimal equivalent.

$$\begin{array}{ccccccccccl} 1 & & 1 & & 0 & & 1 & & 1_2 & & \\ 2^4 & + & 2^3 & + & 0 & + & 2^1 & + & 2^0 & = & 16 + 8 + 2 + 1 \\ & & & & & & & & & = & 27_{10} \end{array}$$

Let's try another example with a greater number of bits:

$$\begin{array}{ccccccccccccccccl} 1 & & 0 & & 1 & & 1 & & 0 & & 1 & & 0 & & 1_2 & = & \\ 2^7 & + & 0 & + & 2^5 & + & 2^4 & + & 0 & + & 2^2 & + & 0 & + & 2^0 & = & 181_{10} \end{array}$$

Note that the procedure is to find the weights (i.e., powers of 2) for each bit position that contains a 1, and then to add them up. Also note that the MSB has a weight of 2^7 even though it is the eighth bit; this is because the LSB is the first bit and has a weight of 2^0.

REVIEW QUESTIONS

1. Convert 100011011011_2 to its decimal equivalent.
2. What is the weight of the MSB of a 16-bit number?

2-2 DECIMAL-TO-BINARY CONVERSIONS

There are two ways to convert a decimal *whole* number to its equivalent binary-system representation. The first method is the reverse of the process described in Section 2-1. The decimal number is simply expressed as a sum of powers of 2, and then 1s and 0s are written in the appropriate bit positions. To illustrate:

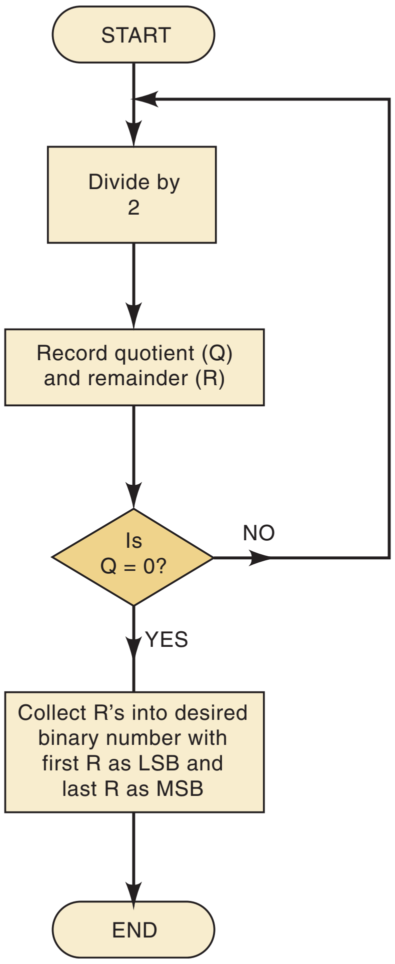
$$\begin{array}{llccccccccccc} 45_{10} = 32 + 8 + 4 + 1 & = & 2^5 & + & 0 & + & 2^3 & + & 2^2 & + & 0 & + & 2^0 \\ & = & 1 & & 0 & & 1 & & 1 & & 0 & & 1_2 \end{array}$$

Note that a 0 is placed in the 2^1 and 2^4 positions, since all positions must be accounted for. Another example is the following:

$$\begin{array}{llccccccccccccc} 76_{10} = 64 + 8 + 4 & = & 2^6 & + & 0 & + & 0 & + & 2^3 & + & 2^2 & + & 0 & + & 0 \\ & = & 1 & & 0 & & 0 & & 1 & & 1 & & 0 & & 0_2 \end{array}$$

Repeated Division

Another method for converting decimal integers uses repeated division by 2. The conversion, illustrated below for 25_{10}, requires repeatedly dividing the decimal number by 2 and writing down the remainder after each division until a quotient of 0 is obtained. Note that the binary result is obtained by writing the first remainder as the LSB and the last remainder as the MSB. This process, diagrammed in the flowchart of Figure 2-1, can also be used to convert from decimal to any other number system, as we shall see.



START
Divide by 2
Record quotient (Q) and remainder (R)
Is Q = 0?
NO
YES
Collect R's into desired binary number with first R as LSB and last R as MSB
END

$$\frac{25}{2} = \boxed{12} + \text{remainder of } 1 \quad \text{LSB}$$

$$\frac{\boxed{12}}{2} = 6 + \text{remainder of } 0$$

$$\frac{6}{2} = 3 + \text{remainder of } 0$$

$$\frac{3}{2} = 1 + \text{remainder of } 1$$

$$\frac{1}{2} = 0 + \text{remainder of } 1 \quad \text{MSB}$$

$$25_{10} = \boxed{1\ \ 1\ \ 0\ \ 0\ \ 1_2}$$

FIGURE 2-1 Flowchart for repeated-division method of decimal-to-binary conversion of integers. The same process can be used to convert a decimal integer to any other number system.

EXAMPLE 2-1

Convert 37_{10} to binary. Try to do it on your own before you look at the solution.

Solution

$$\frac{37}{2} = \boxed{18}.5 \longrightarrow \text{remainder of 1 (LSB)}$$

$$\frac{\boxed{18}}{2} = 9.0 \longrightarrow \quad 0$$

$$\frac{9}{2} = 4.5 \longrightarrow \quad 1$$

$$\frac{4}{2} = 2.0 \longrightarrow \quad 0$$

$$\frac{2}{2} = 1.0 \longrightarrow \quad 0$$

$$\frac{1}{2} = 0.5 \longrightarrow \quad 1 \text{ (MSB)}$$

Thus, $37^{10} = \mathbf{100101_2}$.

Counting Range

Using N bits, we can represent decimal numbers ranging from 0 to $2^N - 1$, a total of 2^N different numbers.

EXAMPLE 2-2

(a) What is the total range of decimal values that can be represented in eight bits?

(b) How many bits are needed to represent decimal values ranging from 0 to 12,500?

Solution

(a) Here we have $N = 8$. Thus, we can represent decimal numbers from 0 to $2^8 - 1 =$ **255.** We can verify this by checking to see that 11111111_2 converts to 255_{10}.

(b) With 13 bits, we can count from decimal 0 to $2^{13} - 1 = 8191$. With 14 bits, we can count from 0 to $2^{14} - 1 = 16{,}383$. Clearly, 13 bits aren't enough, but 14 bits will get us up beyond 12,500. Thus, the required number of bits is **14**.

2-3 HEXADECIMAL NUMBER SYSTEM

The **hexadecimal number system** uses base 16. Thus, it has 16 possible digit symbols. It uses the digits 0 through 9 plus the letters A, B, C, D, E, and F as the 16 digit symbols. The digit positions are weighted as powers of 16 as shown below, rather than as powers of 10 as in the decimal system.

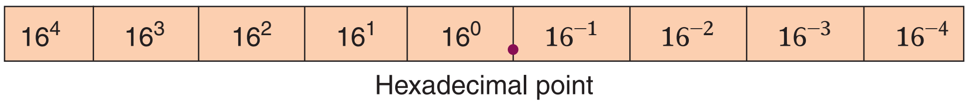


TABLE 2-1

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

Hex-to-Decimal Conversion

A hex number can be converted to its decimal equivalent by using the fact that each hex digit position has a weight that is a power of 16. The LSD has a weight of $16^0 = 1$; the next higher digit position has a weight of $16^1 = 16$; the next has a weight of $16^2 = 256$; and so on. The conversion process is demonstrated in the examples below.

Decimal-to-Hex Conversion

Recall that we did decimal-to-binary conversion using repeated division by 2. Likewise, decimal-to-hex conversion can be done using repeated division by 16 (Figure 2-1). The following example contains two illustrations of this conversion.

EXAMPLE 2-3

(a) Convert 423_{10} to hex.

Solution

$$\frac{423}{16} = 26 + \text{remainder of } 7$$

$$\frac{26}{16} = 1 + \text{remainder of } 10$$

$$\frac{1}{16} = 0 + \text{remainder of } 1$$

$$423_{10} = 1A7_{16}$$

(b) Convert 214_{10} to hex.

Solution

$$\frac{214}{16} = 13 + \text{remainder of } 6$$

$$\frac{13}{16} = 0 + \text{remainder of } 13$$

$$214_{10} = D6_{16}$$

Binary-to-Hex Conversion

Conversion from binary to hex is just the reverse of the process above. The binary number is grouped into groups of *four* bits, and each group is converted to its equivalent hex digit. Zeros (shown shaded) are added, as needed, to complete a four-bit group.

$$1\,1\,1\,0\,1\,0\,0\,1\,1\,0_2 = \underbrace{0\;0\;1\;1}_{3}\;\underbrace{1\;0\;1\;0}_{A}\;\underbrace{0\;1\;1}_{6} \quad = 3A6_{16}$$

To perform these conversions between hex and binary, it is necessary to know the four-bit binary numbers (0000 through 1111) and their equivalent hex digits. Once these are mastered, the conversions can be performed quickly without the need for any calculations. This is why hex is so useful in representing large binary numbers.

For practice, verify that $101011111_2 = 15F_{16}$.

Counting in Hexadecimal

When counting in hex, each digit position can be incremented (increased by 1) from 0 to F. Once a digit position reaches the value F, it is reset to 0, and the next digit position is incremented. This is illustrated in the following hex counting sequences:

(a) 38, 39, 3A, 3B, 3C, 3D, 3E, 3F, 40, 41, 42
(b) 6F8, 6F9, 6FA, 6FB, 6FC, 6FD, 6FE, 6FF, 700

Note that when there is a 9 in a digit position, it becomes an A when it is incremented.

With N hex digit positions, we can count from decimal 0 to $16^N - 1$, for a total of 16^N different values. For example, with three hex digits, we can count from 000_{16} to FFF_{16}, which is 0_{10} to 4095_{10}, for a total of $4096 = 16^3$ different values.

Usefulness of Hex

Hex is often used in a digital system as sort of a "shorthand" way to repre-sent strings of bits.

Hex is simply used as a convenience for thehumans involved. You should memorize the 4-bit binary pattern for eachhexadecimal digit. Only then will you realize the usefulness of this tool indigital systems.

EXAMPLE 2-4

Convert decimal 378 to a 16-bit binary number by first converting to hexadecimal.

Solution

$$\frac{378}{16} = 23 + \text{remainder of } 10_{10} = A_{16}$$

$$\frac{23}{16} = 1 + \text{remainder of } 7$$

$$\frac{1}{16} = 0 + \text{remainder of } 1$$

Thus, $378_{10} = 17A_{16}$. This hex value can be converted easily to binary 000101111010. Finally, we can express 378_{10} as a 16-bit number by adding four leading 0s:

$$378_{10} = 0000 \quad 0001 \quad 0111 \quad 1010_2$$

EXAMPLE 2-5

Convert $B2F_{16}$ to decimal.

Solution

$$\begin{aligned} B2F_{16} &= B \times 16^2 + 2 \times 16^1 + F \times 16^0 \\ &= 11 \times 256 + 2 \times 16 + 15 \\ &= 2863_{10} \end{aligned}$$

Summary of Conversions

1. When converting from binary [or hex] to decimal, use the method of taking the weighted sum of each digit position.
2. When converting from decimal to binary [or hex], use the method of repeatedly dividing by 2 [or 16] and collecting remainders (Figure 2-1).
3. When converting from binary to hex, group the bits in groups of four, and convert each group into the correct hex digit.
4. When converting from hex to binary, convert each digit into its four-bit equivalent.

Binary-Coded-Decimal Code

When a decimal number is represented by its equivalent binary number, we call it **straight binary coding**.

If *each* digit of a decimal number is represented by its binary equivalent, the result is a code called **binary-coded-decimal** (hereafter abbreviated BCD). Since a decimal digit can be as large as 9, four bits are required to code each digit (the binary code for 9 is 1001).

To illustrate the BCD code, take a decimal number such as 874. Each *digit* is changed to its binary equivalent as follows:

$$\begin{array}{cccc} 8 & 7 & 4 & \text{(decimal)} \\ \downarrow & \downarrow & \downarrow & \\ 1000 & 0111 & 0100 & \text{(BCD)} \end{array}$$

As another example, let us change 943 to its BCD-code representation:

$$\begin{array}{cccc} 9 & 4 & 3 & \text{(decimal)} \\ \downarrow & \downarrow & \downarrow & \\ 1001 & 0100 & 0011 & \text{(BCD)} \end{array}$$

Once again, each decimal digit is changed to its straight binary equivalent. Note that four bits are *always* used for each digit.

EXAMPLE 2-6

Convert 0110100000111001 (BCD) to its decimal equivalent.

Solution

Divide the BCD number into four-bit groups and convert each to decimal.

$$\underbrace{0110}_{6}\ \underbrace{1000}_{8}\ \underbrace{0011}_{3}\ \underbrace{1001}_{9}$$

EXAMPLE 2-7

Convert the BCD number 011111000001 to its decimal equivalent.

Solution

$$\underbrace{0111}_{7}\ \underset{\downarrow}{1100}\ \underbrace{0001}_{1}$$

The forbidden code group indicates an error in the BCD number

2-5 THE GRAY CODE

Digital systems operate at very fast speeds and respond to changes that occur in the digital inputs. Just as in life, when multiple input conditions are changing at the same time, the situation can be misinterpreted and cause an erroneous reaction. When you look at the bits in a binary count sequence, it is clear that there are often several bits that must change states at the same time. For example, consider when the three-bit binary number for 3 changes to 4: all three bits must change state.

In order to reduce the likelihood of a digital circuit misinterpreting a changing input, the **Gray code** has been developed as a way to represent a sequence of numbers. The unique aspect of the Gray code is that only one bit ever changes between two successive numbers in the sequence. Table 2-2 shows the translation between three-bit binary and Gray code values. To convert binary to Gray, simply start on the most significant bit and use it as the Gray MSB as shown in Figure 2-2(a). Now compare the MSB binary with the next binary bit (B1). If they are the same, then G1 = 0. If they are different, then G1 = 1. G0 can be found by comparing B1 with B0.

TABLE 2-2 Three-bit binary and Gray code equivalents.

B_2	B_1	B_0	G_2	G_1	G_0
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	1
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	1	1	1
1	1	0	1	0	1
1	1	1	1	0	0

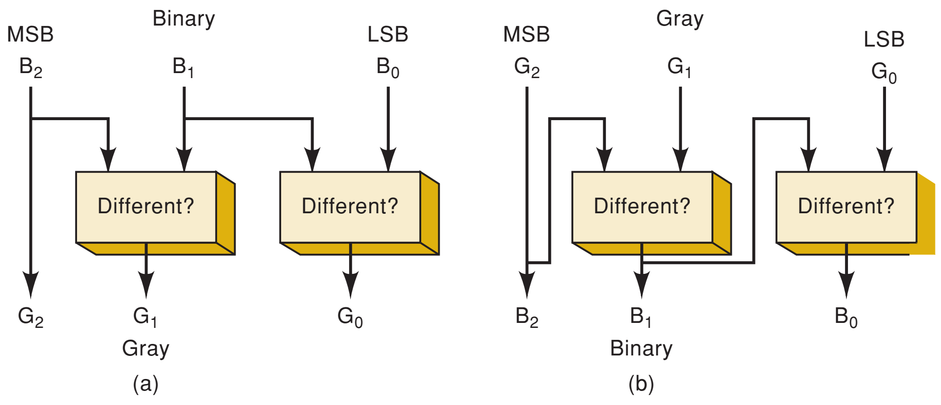


FIGURE 2-2 Converting (a) binary to Gray and (b) Gray to binary.

Refer the textbook for application of Gray code in shaft position encoders

2-6 PUTTING IT ALL TOGETHER

Table 2-3 gives the representation of the decimal numbers 1 through 15 in the binary and hex number systems and also in the BCD and Gray codes. Examine it carefully and make sure you understand how it was obtained. Especially note how the BCD representation always uses four bits for each decimal digit.

TABLE 2-3

Decimal	Binary	Hexadecimal	BCD	GRAY
0	0	0	0000	0000
1	1	1	0001	0001
2	10	2	0010	0011
3	11	3	0011	0010
4	100	4	0100	0110
5	101	5	0101	0111
6	110	6	0110	0101
7	111	7	0111	0100
8	1000	8	1000	1100
9	1001	9	1001	1101
10	1010	A	0001 0000	1111
11	1011	B	0001 0001	1110
12	1100	C	0001 0010	1010
13	1101	D	0001 0011	1011
14	1110	E	0001 0100	1001
15	1111	F	0001 0101	1000

2-7 THE BYTE, NIBBLE, AND WORD

Bytes

Most microcomputers handle and store binary data and information in groups of eight bits, so a special name is given to a string of eight bits: it is called a **byte.** A byte always consists of eight bits, and it can represent any of numerous types of data or information. The following examples will illustrate.

EXAMPLE 2-8

How many bytes are in a 32-bit string (a string of 32 bits)?

Solution

32/8 = 4, so there are **four** bytes in a 32-bit string.

EXAMPLE 2-9

What is the largest decimal value that can be represented in binary using two bytes?

Solution

Two bytes is 16 bits, so the largest binary value will be equivalent to decimal $2^{16} - 1 = 65{,}535$.

Nibbles

Binary numbers are often broken down into groups of four bits known as nibble.

EXAMPLE 2-11

How many nibbles are in a byte?

Solution 2

EXAMPLE 2-12

What is the hex value of the least significant nibble of the binary number 1001 0101?

Solution

1001 0101 The least significant nibble is 0101 = 5.

Words

Bits, nibbles, and bytes are terms that represent a fixed number of binary digits. As systems have grown over the years, their capacity (appetite?) for handling binary data has also grown. A **word** is a group of bits that repre-sents a certain unit of information. The size of the word depends on the size of the data pathway in the system that uses the information. The **word size** can be defined as the number of bits in the binary word that a digital system operates on.

For example, the personal computer on your desk can handle eight bytes at a time, so it has a word size of 64 bits.

2-8 ALPHANUMERIC CODES

In addition to numerical data, a computer must be able to handle nonnu-merical information. In other words, a computer should recognize codes that represent letters of the alphabet, punctuation marks, and other special char-acters as well as numbers. These codes are called **alphanumeric codes**. A complete alphanumeric code would include the 26 lowercase letters, 26 up-percase letters, 10 numeric digits, 7 punctuation marks, and anywhere from 20 to 40 other characters, such as +, /, #, %, *, and so on. We can say that an alphanumeric code represents all of the various characters and functions that are found on a computer keyboard.

ASCII Code

The most widely used alphanumeric code is the **American Standard Code for Information Interchange (ASCII)**. The ASCII (pronounced "askee") code is a seven-bit code, and so it has $2^7 = 128$ possible code groups.

TABLE 2-4 Standard ASCII codes.

Character	HEX	Decimal	Character	HEX	Decimal	Character	HEX	Decimal	Character	HEX	Decimal
NUL (null)	0	0	Space	20	32	@	40	64	`	60	96
Start Heading	1	1	!	21	33	A	41	65	a	61	97
Start Text	2	2	“	22	34	B	42	66	b	62	98
End Text	3	3	#	23	35	C	43	67	c	63	99
End Transmit.	4	4	$	24	36	D	44	68	d	64	100
Enquiry	5	5	%	25	37	E	45	69	e	65	101
Acknowlege	6	6	&	26	38	F	46	70	f	66	102
Bell	7	7	`	27	39	G	47	71	g	67	103
Backspace	8	8	(	28	40	H	48	72	h	68	104
Horiz. Tab	9	9	)	29	41	I	49	73	i	69	105
Line Feed	A	10	*	2A	42	J	4A	74	j	6A	106
Vert. Tab	B	11	+	2B	43	K	4B	75	k	6B	107
Form Feed	C	12	,	2C	44	L	4C	76	l	6C	108
Carriage Return	D	13	-	2D	45	M	4D	77	m	6D	109
Shift Out	E	14	.	2E	46	N	4E	78	n	6E	110
Shift In	F	15	/	2F	47	O	4F	79	o	6F	111
Data Link Esc	10	16	0	30	48	P	50	80	p	70	112
Direct Control 1	11	17	1	31	49	Q	51	81	q	71	113
Direct Control 2	12	18	2	32	50	R	52	82	r	72	114
Direct Control 3	13	19	3	33	51	S	53	83	s	73	115
Direct Control 4	14	20	4	34	52	T	54	84	t	74	116
Negative ACK	15	21	5	35	53	U	55	85	u	75	117
Synch Idle	16	22	6	36	54	V	56	86	v	76	118
End Trans Block	17	23	7	37	55	W	57	87	w	77	119
Cancel	18	24	8	38	56	X	58	88	x	78	120
End of Medium	19	25	9	39	57	Y	59	89	y	79	121
Substitue	1A	26	:	3A	58	Z	5A	90	z	7A	122
Escape	1B	27	;	3B	59	[	5B	91	{	7B	123
Form separator	1C	28	<	3C	60	\	5C	92	|	7C	124
Group separator	1D	29	=	3D	61	]	5D	93	}	7D	125
Record Separator	1E	30	>	3E	62	^	5E	94	~	7E	126
Unit Separator	1F	31	?	3F	63	_	5F	95	Delete	7F	127

EXAMPLE 2-14

An operator is typing in a C language program at the keyboard of a certain microcomputer. The computer converts each keystroke into its ASCII code and stores the code as a byte in memory. Determine the binary strings that will be entered into memory when the operator types in the following C statement:

```
if (x>3)
```

Solution

Locate each character (including the space) in Table 2-4 and record its ASCII code.

i	69	0110	1001
f	66	0110	0110
space	20	0010	0000
(	28	0010	1000
x	78	0111	1000
>	3E	0011	1110
3	33	0011	0011
)	29	0010	1001

Note that a 0 was added to the leftmost bit of each ASCII code because the codes must be stored as bytes (eight bits). This adding of an extra bit is called *padding with 0s.*

2-9 PARITY METHOD FOR ERROR DETECTION

Whenever information is transmitted from one device (the transmitter) to another device (the receiver), there is a possibility that errors can occur such that the receiver does not receive the identical information that was sent by the transmitter.

The transmitter sends a relatively noise-free serial digital signal over a signal line to a receiver. However, by the time the signal reaches the receiver, it contains a certain degree of noise superimposed on the original signal. Occasionally, the noise is large enough in amplitude that it will alter the logic level of the signal, as it does at point x. When this occurs, the receiver may incorrectly interpret that bit as a logic 1, which is not what the transmitter has sent.

One of the simplest and most widely used schemes for error detection is the **parity method**.

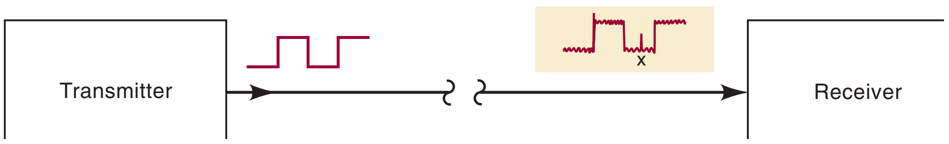


FIGURE 2-4 Example of noise causing an error in the transmission of digital data.

Parity Bit

A **parity bit** is an extra bit that is attached to a code group that is being transferred from one location to another. The parity bit is made either 0 or 1, depending on the number of 1s that are contained in the code group. Two different methods are used.

In the *even-parity* method, the value of the parity bit is chosen so that the total number of 1s in the code group (including the parity bit) is an *even* number. For example, suppose that the group is 1000011. This is the ASCII character "C." The code group has *three* 1s. Therefore, we will add a parity bit of 1 to make the total number of 1s an even number. The *new* code group, *including the parity bit,* thus becomes

$$\underset{\uparrow}{1}\ \ 1\ 0\ 0\ 0\ 0\ 1\ 1$$

added parity bit*

If the code group contains an even number of 1s to begin with, the parity bit is given a value of 0. For example, if the code group were 1000001 (the ASCII code for "A"), the assigned parity bit would be 0, so that the new code, *including the parity bit,* would be 01000001.

The *odd-parity* method is used in exactly the same way except that the parity bit is chosen so the total number of 1s (including the parity bit) is an *odd* number. For example, for the code group 1000001, the assigned parity bit would be a 1. For the code group 1000011, the parity bit would be a 0.

Regardless of whether even parity or odd parity is used, the parity bit becomes an actual part of the code word. For example, adding a parity bit to the seven-bit ASCII code produces an eight-bit code. Thus, the parity bit is treated just like any other bit in the code.

The parity bit is issued to detect any *single-bit* errors that occur during the transmission of a code from one location to another. For example, suppose that the character "A" is being transmitted and *odd* parity is being used. The transmitted code would be

1 1 0 0 0 0 0 1

When the receiver circuit receives this code, it will check that the code contains an odd number of 1s (including the parity bit). If so, the receiver will assume that the code has been correctly received. Now, suppose that because of some noise or malfunction the receiver actually receives the following code:

1 1 0 0 0 0 0 0

The receiver will find that this code has an *even* number of 1s. This tells the receiver that there must be an error in the code because presumably the transmitter and receiver have agreed to use odd parity. There is no way, however, that the receiver can tell which bit is in error because it does not know what the code is supposed to be.

It should be apparent that this parity method would not work if *two* bits were in error, because two errors would not change the "oddness" or "evenness" of the number of 1s in the code. In practice, the parity method is used only in situations where the probability of a single error is very low and the probability of double errors is essentially zero.

*The parity bit can be placed at either end of the code group, but it is usually placed to the left of the MSB.