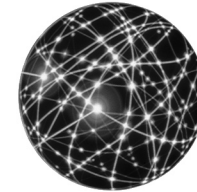


CHAPTER 3

Error Detection and Correction



OBJECTIVES

After reading this chapter, the student will be able to:

- determine how errors in asynchronous digital data transmissions are detected using parity.
- determine how errors in asynchronous digital data transmissions are corrected using LRC/VRC.
- determine how errors in synchronous digital data transmissions are detected using checksum and CRC.
- determine how errors in synchronous digital data transmissions are corrected using Hamming Code.
- determine how errors are corrected using automatic request for retransmission.

OUTLINE

- 3.1 Introduction
- 3.2 Asynchronous Data Error Methods
- 3.3 Synchronous Data Error Methods
- 3.4 Error-Testing Equipment

3.1 INTRODUCTION

The occurrence of a data bit error in a serial stream of digital data is an infrequent occurrence. Even less frequent is the experience of numerous errors within the transmission of a single message. Usually if a number of errors occur then it can be presumed that either a significant interference occurred affecting the transmission line or that there is a major failure in the communications path. Largely because of the extremely low bit-error rates in data transmissions, most error detection methods and algorithms are designed to address the detection or correction of a single bit error. However, as we shall soon see, many of these methods will also detect multiple errors. Error correction, though, will remain a one-bit error concern.

Section 3.1 Review Questions

1. Why are most error detection and correction methods designed for single-bit detection or correction?
2. What two conclusions are drawn when multiple errors in a transmission are detected?

parity
error-detection (parity) and
error-correction (VRC)
techniques based on the odd or
even count of logic 1's in a
transmitted character.

3.2 ASYNCHRONOUS DATA ERROR METHODS

Probably the most common and oldest method of error detection is the use of **parity**. While parity is used in both asynchronous and synchronous data streams, it seems to find greater use in low-speed asynchronous transmission applications, however, its use is not exclusive to this.

Parity Error Detection

Parity works by adding an additional bit to each character word transmitted. The state of this bit is determined by a combination of factors, the first of which is the type of parity system employed. The two types are even and odd parity. The second factor is the number of logic 1 bits in the data character. In an even parity system, the parity bit is set to a low state if the number of logic 1s in the data word is even. If the count is odd, then the parity bit is set high. For an odd parity system, the state of the parity bit is reversed. For an odd count, the bit is set low, and for an even count, it is set high.

EXAMPLE 3-1

What is the state of the parity bit for both an odd and an even parity system for the extended ASCII character B?

SOLUTION

The extended ASCII character B has a bit pattern of 01000010 (42 H). The number of logic 1s in that pattern is two, which is an even count. For an even parity system, the parity bit would be set low and for an odd parity system, it would be set high.

To detect data errors, each character word that is sent has a parity bit computed for it and appended after the last bit of each character is sent as illustrated in Figure 3-1. At the receiving site, parity bits are recalculated for each received character. The parity bits sent with each character are compared to the parity bits the receiver computes. If their states do not match, then an error has occurred. If the states do match, then the character *may* be errorfree.

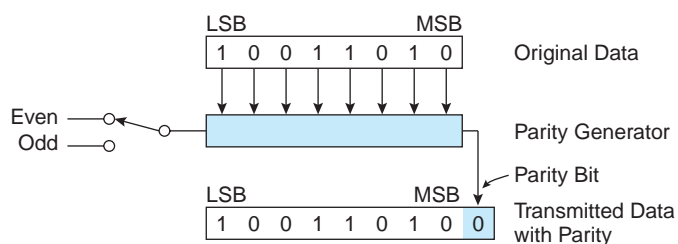


FIGURE 3-1 Appending Parity Bit

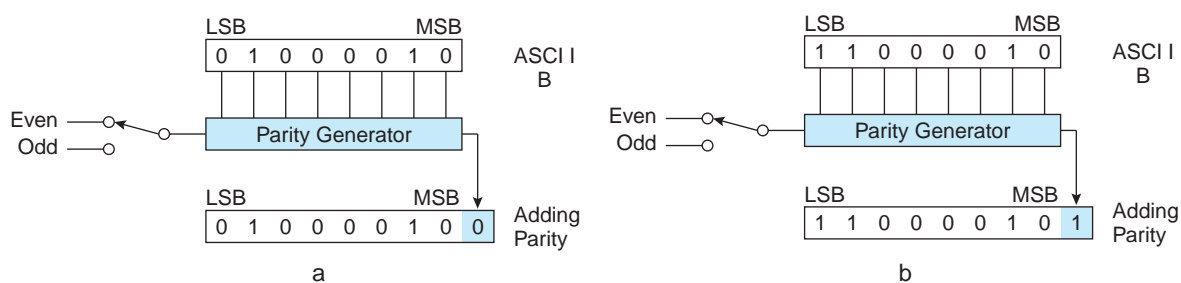


FIGURE 3-2 Even Parity for ASCII B (a), Parity for Bad ASCII B (b)

EXAMPLE 3-2

The ASCII character B is transmitted with an even-parity bit appended to it. Illustrate how the receiver would detect an error.

SOLUTION

As shown in Figure 3-2a, the state of the even-parity bit for the ASCII B is low, so the complete data stream for the character sent, starting with the least significant bit (LSB) is: 010000100. Notice there are now nine bits—eight bits for the extended ASCII character B and one for the parity bit. The breakdown of the data stream is:

LSB	MSB Parity
0 1 0 0 0 0 1 0 0	

Suppose that the LSB becomes corrupted during transmission. The receiver receives the character as: 110000100. When the receiver computes a parity bit for the character data, it results in a high state of the parity, bit as shown in Figure 3-2b. This is compared with the transmitted parity, which is a low state. Since they do not agree, the receiver determines that an error has occurred. Note that the receiver cannot determine which bit is bad, only that one of them is wrong.

A match between transmitted parity and receiver-calculated parity does not guarantee that the data has not been corrupted. Indeed, if an even number of errors occur in a

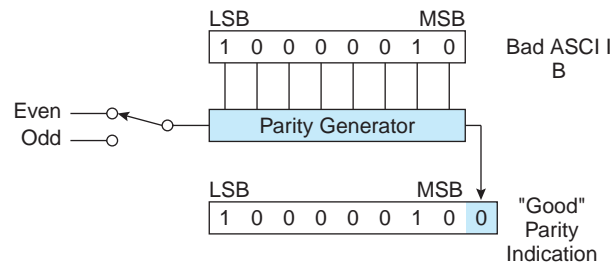


FIGURE 3-3 Multiple Errors in ASCII B

single character, then the parity for the corrupted data will be the same state as the good data. For instance, suppose the two lowest bits in the character B were bad as seen in Figure 3-3. The total number of ones in the data stream would still be an even count and the parity bit calculated at the receiver would be a low state and would match the one transmitted. This does not present a major problem, since the occurrence of two errors in an eight-bit character is excessive and usually indicates a major problem in the system. Such a problem would cause errors to occur in other characters and one of them would eventually be detected. Since the occurrence of errors is extremely low, parity is successful in detecting more than 95% of the errors that occur.

Parity Generation

The hardware circuit used to generate the state of the parity bit is composed of a number of exclusive OR gates as shown in Figure 3-4. To understand how the exclusive OR gate can be used to determine the even or odd count of the data bits, we first must examine its truth table.

A and B are used to designate the two inputs to an exclusive OR gate and Y is its output state. An exclusive OR is a device whose output is low if the inputs are the same (or an even number of ones) and a high if they are opposite (or an odd number of ones). The exclusive OR gate can be used for a number of functions, including binary addition and subtraction, controlled inversion, and bit comparison.

The output of an exclusive OR for binary addition is used to represent the sum of its two inputs. Notice that $0 + 0 = 0$, $0 + 1 = 1$, $1 + 0 = 1$, and $1 + 1 = 0$. Of course, the last addition, in true binary form is 10, but since there is no way for a single exclusive OR gate to show the carry over, only the 0 will show on the truth table.

Subtraction can be viewed in a similar manner. $0 - 0 = 0$, $0 - 1 = 1$ (after borrowing), $1 - 0 = 1$, and $1 - 1 = 0$. Once again, the exclusive OR indicates the difference result as long as we ignore the need to borrow to subtract 1 from 0.

As a bit comparator, the exclusive OR's output (Y) is low when both inputs (A and B) are the same logic level and high when they are different.

When considering the exclusive OR for inverting a bit under controlled conditions, in Table 3-1, treat input A as a control input and input B as a data bit input. Notice that

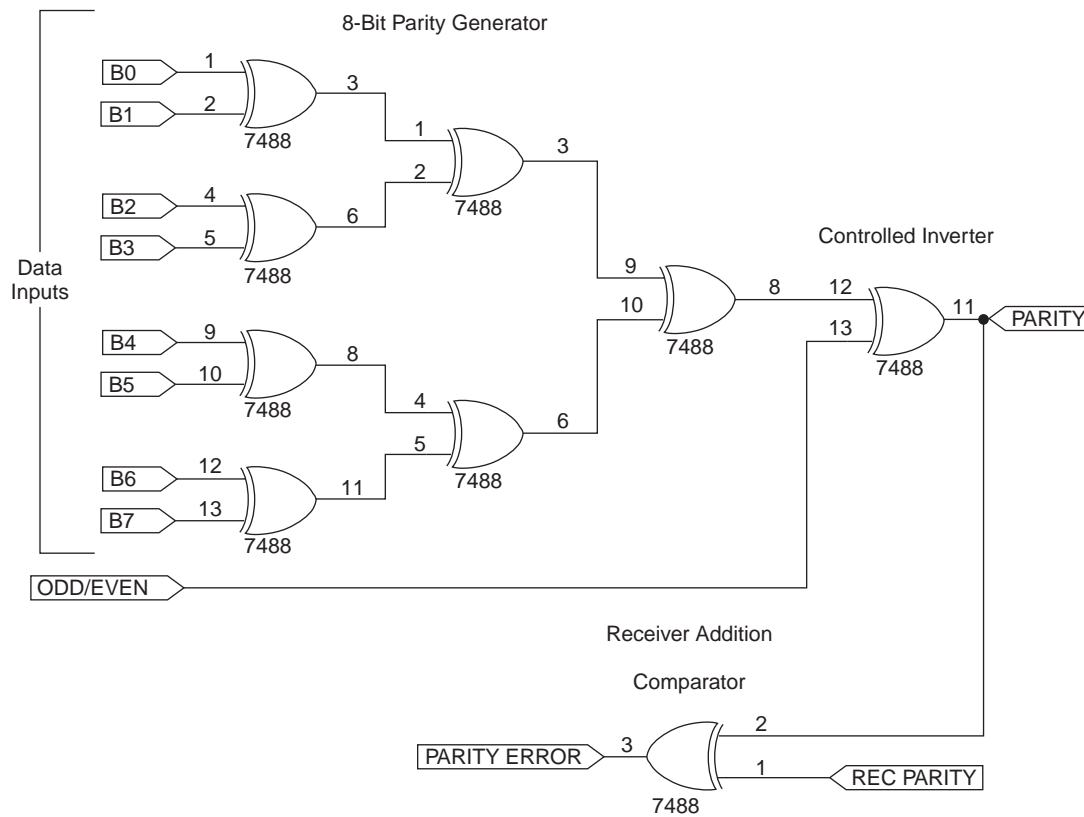


FIGURE 3-4 Parity Generator

when the control input A is low (0), the output Y is the same state as the data input B . When the control input A is high (1), output Y is the opposite state of input B .

In this chapter, we will use all these functions of the exclusive OR, starting with the parity-bit generator and checker.

In Figure 3-4, each of the bits in the data-word inputs are summed together using the exclusive OR gates. The final sum's LSB result will be the even parity state. Since we need to be able to select either an even-parity or odd-parity system application, there has to be a way to produce one or the other. What is most interesting for the generation of a parity bit, is that when the LSB of a binary sum is low, the sum is even and when the sum is odd, the LSB is high. This defines an even parity result directly from the output of the exclusive OR summers. To make the output reflect an odd-parity system, it is only a matter of inverting that output to get the odd-parity state. However, we want to do this when an odd-parity system is desired. As long as an even-parity system is used, we do not want to invert the output. The solution to this problem is to use another exclusive OR as a controlled inverter at the output of the generator circuit as shown in Figure 3-4.

The final step, this time at the receiver, is to compare the parity sent with the one the receiver computes. In the receiver, another parity generator is employed to generate

TABLE 3-1**Exclusive OR Truth Table**

A	B	$Y = A + B \text{ or } A - B$
0	0	0
0	1	1
1	0	1
1	1	0

Exclusive OR Truth Table Used for Sum or Difference Discarding Carry Out or Borrow

Control	Data Input	Data Output (Y)
0	0	0 Non-Inverted Output
0	1	1
1	0	1 Inverted Output
1	1	0

Exclusive OR Truth Table Controlled Inverter Application

A	B	Comparison Results (Y)
0	0	0 – Same Logic Level
0	1	1 – Different
1	0	1 – Different
1	1	0 – Same Logic Level

Exclusive OR as Digital Comparator

a received parity bit. This bit is compared to the transmitted parity bit to determine if an error has occurred.

The parity circuit is identical at both the transmitter and receiver sites with the exception of the use of the comparator. The transmitter does not require its use while the receiver does. Parity generators are produced as medium-scale integrated circuits (MSIC), like the Signetics 8262 shown in Figure 3-5. This chip uses two NOR gates on the output to provide a means for disabling parity using the inhibit input for systems that do not use parity for error detection. By using two NOR gates both odd- and even-parity states are available on separate output pins. Comparisons by the receiver using this chip require an external exclusive OR for the comparator function.

EXAMPLE 3-3

In the received asynchronous ASCII data stream using two-stop bits, shown below, there is an error. First, determine which parity system, odd or even, is being used. Then determine which character is bad. Lastly decode the message using the ASCII

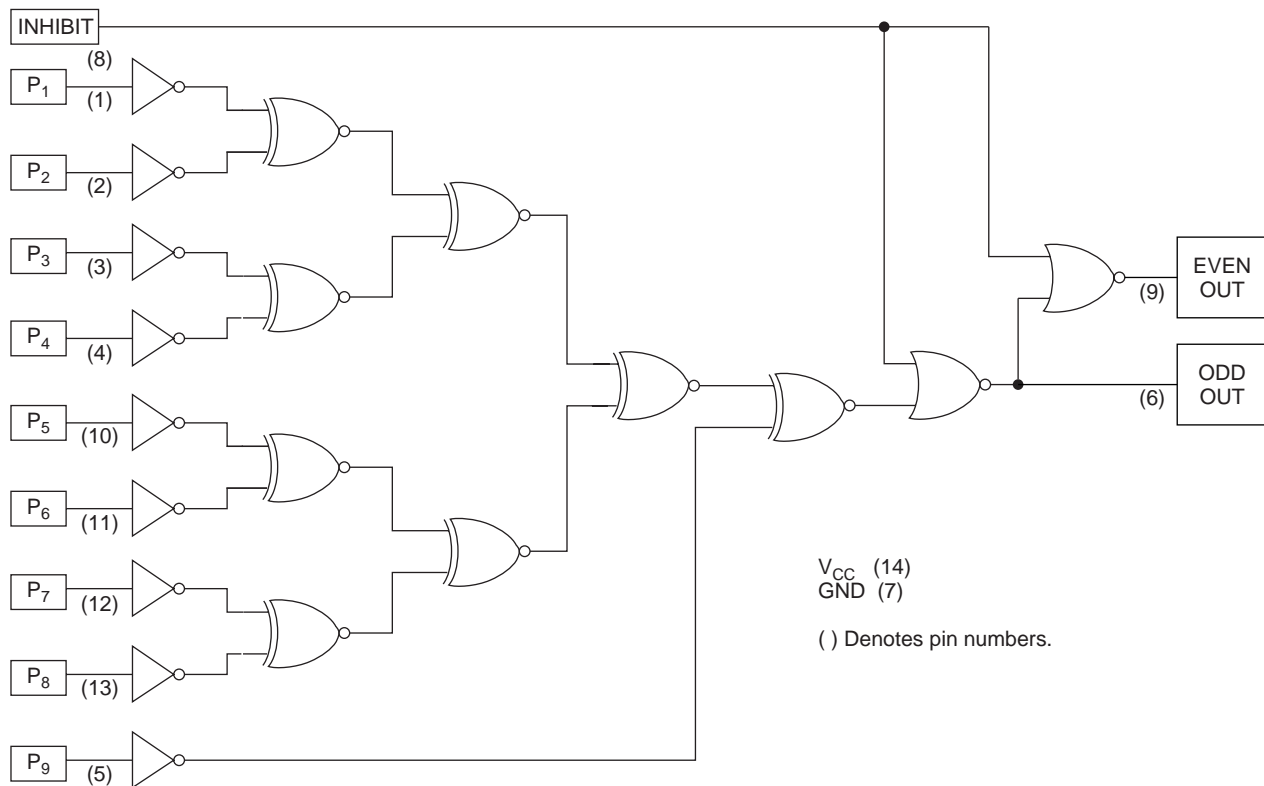


FIGURE 3-5 IC Parity Generator

chart from Chapter 1. From this verify that the bad character is indeed bad. What was the intended message? (Spaces between each 4-bits are included for readability)

```
0000 1001 0110 1010 0110 1100 0010 1101 1000 0011 1111 0100
0010 0010 011
```

SOLUTION

Each character consists of seven ASCII bits, a start bit and two-stop bits plus the parity bit. The start bit is the leftmost bit of each character, followed by the LSB data bit through the MSB bit, then the parity bit, and finally, the two-stop bits. This means that each character contains eleven total bits. So step one is to define each character:

```
0000 1001 011 0101 0011 011 0000 1011 011
0000 0111 111 0100 0010 011
```

Next strip out the start and stop bits:

```
0001 0010 1010 0110 0001 0110 0000 1111 1000 0100
```

Now count the ones in the first seven bits of each word and determine the even- and odd-parity states for each. From this we can surmise that even-parity was used—all the characters except the middle one satisfy even-parity results. This means that the middle character probably has the error in it.

By using the ASCII chart in Chapter 1, the message spells out “Hehp!” This message should probably be “Help!” verifying that the middle character is incorrect.

ARQ

To correct errors using parity, the receiving station can only request that the message containing the error be retransmitted. A system that is capable of requesting retransmission of a bad message automatically in response to detecting an error has *Automatic Request for Retransmission* or *Automatic Repeat Request (ARQ)* processing within its communications software package. ARQ was originally designed to be used with a special type of character code that used a seven-bit character size. The uniqueness of that code was that each character code contained three bits that were high and four that were low. If any character received is detected with more or less than three high bits in it, it is flagged as bad and the receive station automatically requests that the character be retransmitted. The automatic request process has been incorporated into other error detection software such as those which respond to parity errors with any available character code.

Not all systems using parity have ARQ functions included in their programs. Some use a parity error flag contained within a status register, which, when read by an application program, causes a message to be sent to a terminal to inform a user that a data error has occurred. It is then up to that user to determine if it is necessary to request the message be retransmitted. The advantage of doing this is that some errors are less critical than others and do not require taking up communication time for a retransmission of the data. For instance, an error in a plain text message may be obvious enough that the user can easily determine the correct character in the text. In that case, there is no need to have the text resent. The drawback to this method is that the display of the message must be interrupted to inform the user that an error has occurred and that it does require some action on the user's part.

Data Correction Using LRC/VRC

As described in the preceding section, parity is primarily used for detecting errors in a serial data character. A bad parity match indicates a logic error has occurred in one of the character's data bits. The use of parity called a **vertical redundancy check (VRC)** can be extended to allow single-bit error correction to take place in a received data stream. By having the ability to correct an error, a receiver would not require a message to be retransmitted, but could do the correction itself. The trade-off in using an error-correction scheme is that an additional character has to be sent with the message and additional software and/or hardware must be used to create and interpret that character. For asynchronous data transmission, that character is known as the **longitudinal redundancy check (LRC)** character.

vertical redundancy check (VRC)

longitudinal redundancy check (LRC)
error-correction method that uses parity and bit summing.

Using a VRC/LRC system, the message is sent with each character containing the regular even-parity bit known as the VRC bit. As with error-detection schemes, any mismatch between transmitted and received VRCs indicates that the character contains a bad data bit. In order to correct the bad bit, what is left to be done is to determine which of the character's bits is the bad one. This is where LRC comes in. It is used to create a cross-matrix type of configuration where the VRC bit denotes the row (character) and the LRC, the column (bit position) of the message's bad bit. At the sending site, each of the data bits of each character is exclusive ORed with the bits of all the other data bits. This is best illustrated by example.

EXAMPLE 3-4

Determine the states of the LRC bits for the asynchronous ASCII message "Help!"

SOLUTION

The first step in understanding the process is to list each of the message's characters with their ASCII code and even VRC parity bit:

LSB						MSB	VRC	CHARACTER
0	0	0	1	0	0	1	0	H
1	0	1	0	0	1	1	0	e
0	0	1	1	0	1	1	0	l
0	0	0	0	1	1	1	1	p
1	0	0	0	0	1	0	0	!

Next, for each vertical column, find the LRC bit by applying the exclusive OR function. To make this process easier, you can consider the results of the exclusive OR process as being low or zero (0), if the number of ones (1) are even, and one (1) if the count is odd. For instance, in the LSB column, there are two 1's, so the LRC bit for that column is a 0. And for the rest:

LSB						MSB	VRC	CHARACTER
0	0	0	1	0	0	1	0	H
1	0	1	0	0	1	1	0	e
0	0	1	1	0	1	1	0	l
0	0	0	0	1	1	1	1	p
1	0	0	0	0	1	0	0	!
0	0	0	0	1	0	0	1	LRC

When the message is transmitted, the LRC character is sent following the last character of the message. The receiver reads in the message and duplicates the process, including the LRC character in the exclusive OR process. If there were no errors, then the VRC's would all match and the resulting LRC would be 0.

EXAMPLE 3-5

Show how a good message would produce an LRC of 0 at the receiver.

SOLUTION

Repeat the process as before, but include the LRC character this time. Note that the number of 1s in each column are always even if there are no errors present:

LSB						MSB	VRC	CHARACTER
0	0	0	1	0	0	1	0	H
1	0	1	0	0	1	1	0	e
0	0	1	1	0	1	1	0	l
0	0	0	0	1	1	1	1	p
1	0	0	0	0	1	0	0	!
0	0	0	0	1	0	0	1	LRC
0	0	0	0	0	0	0	0	Receiver LRC

So far, the LRC/VRC combination seems to aid in detecting errors. If an error occurs, one of the VRCs won't match and the LRC would not be zero. The question is how is it used to pinpoint the bit in the bad character. Again the best way to see how this works is to continue our example.

EXAMPLE 3-6

Illustrate how LRC/VRC is used to correct a bad bit.

SOLUTION

We will use the same message, but by placing an error in the received data would cause the l character to print as an h. You can compare the data with the good example to satisfy yourself as to which bit is bad and confirm that the LRC process does indeed pick out the same bit.

LSB						MSB	VRC	CHARACTER
0	0	0	1	0	0	1	0	H
1	0	1	0	0	1	1	0	e
0	0	0	1	0	1	1	0	h
0	0	0	0	1	1	1	1	p
1	0	0	0	0	1	0	0	!
0	0	0	0	1	0	0	1	LRC
0	0	1	0	0	0	0	1	Received LRC

The first thing that we note is that the VRC for the character *h* will not match the one the receiver will generate, which will be a 1 since the number of ones in *h* are odd and the number of ones in *l* was even. The received LRC will have two indications that something is no longer right. Its own VRC bit will not match the

transmitted VRC bit and, secondly, its value is not zero. The state of the VRC doesn't matter since any non-zero result will indicate that there is a problem. This is how the system uses the receive LRC to determine which bit is bad: The receiver already knows which is the bad character by checking the VRC (parity) bits. By inspecting the receive LRC, the answer comes jumping off the page. The one bit in the LRC that is not a zero is in the same bit position as the bad bit in the *h/l* character! Now that the bad character and the bad bit position are discovered, the receiver needs only to invert that bad bit to make the character correct.

We can illustrate the matrix concept behind LRC/VRC by circling the bad VRC and LRC bit and seeing where they intersect in the message matrix as shown for our example here:

LSB						MSB	VRC	CHARACTER		
0	0	0	1	0	0	1	0	H		
1	0	1	0	0	1	1	0	e		
0	0	0	1	0	1	1	1	h/l		
0	0	0	0	1	1	1	1	p		
1	0	0	0	0	1	0	0	!		
0	0	0	0	1	0	0	1	LRC		
		0	0	1	0	0	0	0	1	Received LRC

forward error correction (FEC)
method of correction that occurs as messages are received.

This error-correction method and others which are similar, are known as **forward error correction (FEC)** because errors are corrected as the message is received. There is no requirement to retransmit the message as long as the errors remain infrequent. If more than one error occurs in a message, then more than one LRC and one VRC bit will be bad and there is no way to determine which LRC bit goes with which VRC character. In this case, the excessive number of bit errors is indicative of a severe or catastrophic condition. Once the cause of the problem is resolved, the message will have to be retransmitted in its entirety.

Section 3.2 Review Questions

1. What is the state of the parity bit using an odd-parity system for the extended ASCII character M?
2. Why doesn't a good match between transmitted and received parity bits guarantee that the character is good?
3. What is the chief advantage of the ARQ function?

4. List the functions that an exclusive OR logic gate can be used for.
5. Which type of parity system is used for the VRC bits in the LRC/VRC error-correcting scheme?
6. How many errors in a message can be corrected using LRC?
7. What does an incorrect VRC bit indicate?
8. What does a one in a bit position in a LRC character indicate?

3.3 SYNCHRONOUS DATA ERROR METHODS

overhead

Any non-data bits or characters sent with a transmission.

Synchronous data are transmitted at higher data rates in as an efficient manner as possible. Start-and-stop framing and parity bits are omitted from the data stream to reduce overhead. **Overhead** is defined as any bits sent that do not contain actual data information. This includes framing bits, preambles, error-detection characters, or bits, etc. It should be noted that in some synchronous data systems, parity is occasionally employed for error detection. Most high-speed synchronous transmissions, however, do not follow this practice. The reason is that most errors in high-speed transmissions occur in bursts, which could render parity-error detection less effective. These error bursts result from some external interference or other effect on the line that causes several bits to be corrupted at once. Single-bit errors occur less frequently. Because of this, and the desire to reduce the overhead in synchronous transmissions, error-detection methods have evolved to detect single and multiple errors within a data stream.

Synchronous error detection works by creating an additional character to be sent with the data stream. At the receive site, the process is duplicated and the two error-detection characters are compared similarly to comparing two parity bits. If the characters match then the data received has no errors. If they do not match, an error has occurred and the message has to be retransmitted. Note that one major difference between using error-detection characters versus single-parity bits, is that if the transmitted and received characters match, then the data is good. Using parity, matching parity bits does not guarantee that the character received was good. The computation of error characters is carried on quickly to support the higher data rates of transmission.

CRC

cyclic redundancy check (CRC)

error-detection method that uses a pseudo-division-process.

One of the most frequently used error-detection methods for synchronous data transmissions is **cyclic redundancy check (CRC)** developed by IBM. This method uses a pseudo-binary-division process to create the error or CRC character, which is appended to the end of the message. The hardware circuitry that generates the CRC character at the transmitter is duplicated at the receiver. This circuitry is incorporated into the transmit-and-receive shift registers that send and receive the actual message. We will begin by ex-

ploring the method used to create and check the CRC character and then view the circuitry that performs the operation.

In the original specification for CRC, IBM specified a 16-bit CRC character designated as CRC-16. The process uses a constant “divisor” to perform the “division” process, which appears in binary for CRC-16 as:

1000 1000 0001 0000 1

Again, spaces are used for clarity. In actuality, there are no spaces. Now that we have a “divisor” we need something for it to be divided into. This is the message. The process is begun by adding 16 zeros (one less than the number of bits in the “divisor”) after the last bit of the message. These 16-bits will eventually be replaced by the CRC character. The “divisor” is then exclusive ORed with the seventeen most significant bits of the message. Enough additional message bits are appended to the result of the exclusive OR process to fill out 17-bit positions starting with the first logic 1 of the exclusive OR result. The process is repeated until the last bits of the message (including the added zeros) are used. The result from the last exclusive OR process becomes the CRC-character that replaces the 16-zero bits originally added to the message (leading zeros are added as needed). This process is best viewed by example, which will use a smaller CRC “divisor” to shorten the process.

EXAMPLE 3-7

Compute the CRC-4 character for the following message using a “divisor” constant of 10011:

1100 0110 1011 01

SOLUTION

CRC-4 is used for illustration purposes since an example using CRC-16 looks cumbersome on paper and is difficult to follow. However, the principle is the same. Notice that the “divisor” is 5-bits, one more than the number indicated by the CRC type (CRC-4). The same was true for CRC-16, which had a 17-bit “divisor.” We start the process by adding four zeros to the data stream and removing the spaces we have been using for convenience:

110001101011010000

Next, set up the problem to appear as a division problem:

10011 $\overline{)110001101011010000}$

Start the “division” process by exclusive OR the “divisor” with the first five bits of the message:

10011 $\overline{)110001101011010000}$
 $\underline{10011}$
 1011

Now bring “down” one bit so that the result of the exclusive OR process is filled out to the “divisor” size and repeat the process:

$$\begin{array}{r}
 10011 \quad \overline{)110001101011010000} \\
 \underline{10011} \\
 10111 \\
 \underline{10011} \\
 100
 \end{array}$$

Continue with the process until all of the bits in the message plus the added four zeros are used up:

$$\begin{array}{r}
 10011 \quad \overline{)110001101011010000} \\
 \underline{10011} \\
 10111 \\
 \underline{10011} \\
 10010 \\
 \underline{10011} \\
 11011 \\
 \underline{10011} \\
 10000 \\
 \underline{10011} \\
 11100 \\
 \underline{10011} \\
 11110 \\
 \underline{10011} \\
 11010 \\
 \underline{10011} \\
 1001 = \text{CRC character}
 \end{array}$$

The CRC character is appended onto the end of the message and transmitted. At the receiver, the process is repeated, except that there are no zeros added to the message. Instead, the CRC character fills up those positions. If the result of the process at the receiver produces zero then no errors occurred. If any bit or combination of bits are wrong, then the receiver will yield a non-zero result.

EXAMPLE 3-8

Demonstrate how a receiver detects a good message and one with several errors in it.

SOLUTION

Redo the process of EXAMPLE 3-7, but this time use the CRC character in place of the extra zeros:

```

10011  )110001101011011001
          10011
          10111
          10011
          10010
          10011
          11011
          10011
          10000
          10011
          11110
          10011
          11010
          10011
          10011
          10011
          0000 = CRC character is zero

```

The changes in the bits brought down are highlighted. Notice how they produce different results from EXAMPLE 3-7. This eventually results in a CRC of 0000 if everything is correct. Now let's suppose there are three errors in the message that will also be highlighted. Follow the problem through to see how the CRC will be non-zero:

```

                                bad bits
10011  )1100011110111011001
          10011
          10111
          10011
          10011
          10011
          011101
          10011
          11101
          10011
          11100
          10011
          11110
          10011
          11011
          10011
          1000 = CRC character is non-zero

```

NOTE

Given the small size (CRC-4) of this example, there could easily be error combinations that would produce a zero CRC result. This is the reason that most CRC systems today are either CRC-16, CRC-32 or CRC-64.

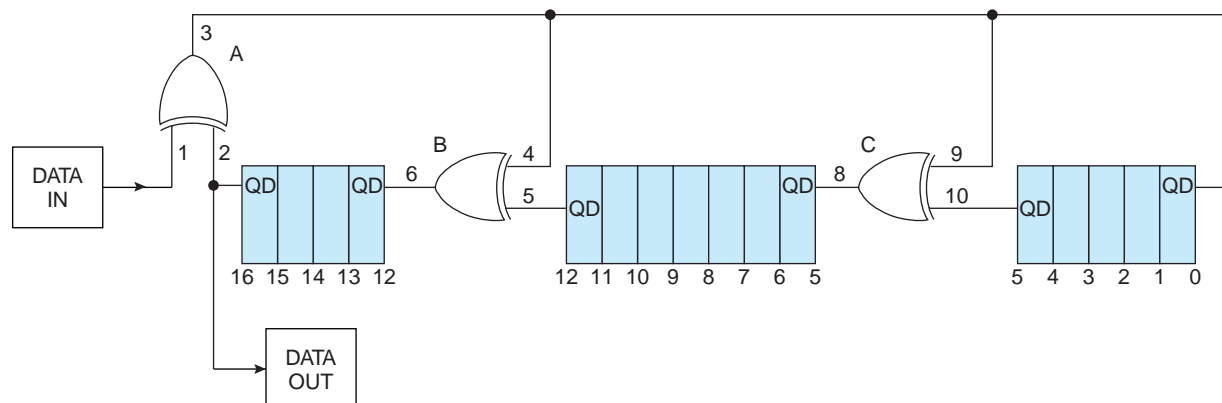


FIGURE 3-6 CRC-16 Block Diagram

A block diagram of the shift register circuit that implements CRC-16 is shown in Figure 3-6. Each rectangle represents a data flip-flop with the D-input on the right and the Q output on the left of the block. The numbering on the bottom of the flip-flop register reflects a design process used to develop the circuit. We shall see how that process works shortly. Exclusive OR gates are placed in between certain sections of the shift register determined by the “divisor.” To facilitate the design of the CRC-16 or any other CRC circuit, a form of quadratic expression is used to represent the “divisor.” This expression is developed by using the powers of 2 for each bit position of the “divisor” that contains a logic 1. For CRC-16, those positions are:

$$\begin{array}{cccc} 16 & 12 & 5 & 0 \\ 10001000000100001 \end{array}$$

which, as a quadratic expression, is written as:

$$G(X) = X^{16} + X^{12} + X^5 + 1 \quad \text{note: } X^0 = 1$$

The flip-flops in the register are numbered by similar bit position numbers. Exclusive OR gates are inserted at the indicated bit positions in the quadratic expression. The Q output of the last flip-flop drives one input of an exclusive OR gate.

The other input is supplied from the serial data stream to be transmitted or received. The Q output of the flip-flop numbered 12 drives the input of another exclusive OR gate as does the one numbered 5. The second inputs of these gates are driven by the output of the exclusive OR connect to bit 16 and the data input. This same line is returned to the input of flip-flop number 0. The output data stream is taken from the last flip-flop (number 15/16). To see how this design is developed, let’s do one for the CRC-4 circuit used in Example 3-7 and 3-8.

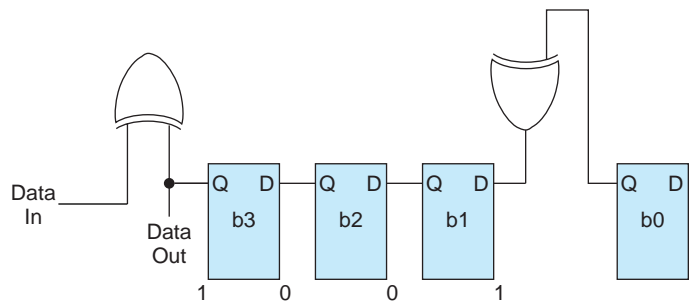


FIGURE 3-7 Exclusive OR Placements

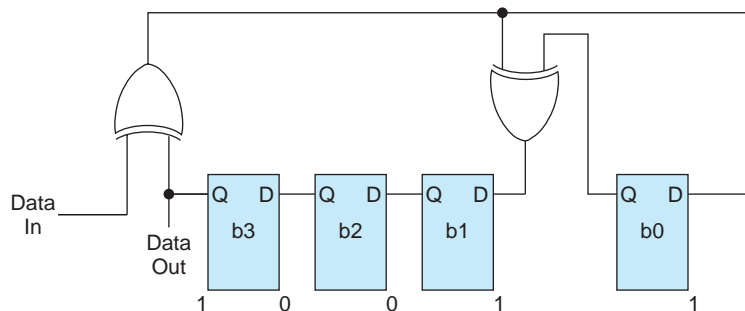


FIGURE 3-8 CRC-4 Circuit

EXAMPLE 3-9

Develop the block diagram for the CRC-4 circuit used in Examples 3-7 and 3-8.

SOLUTION

First create the quadratic expression from the “divisor”:

$$\begin{array}{r} 4 \quad 1 \quad 0 \\ 1 \quad 0 \quad 0 \quad 1 \quad 1 = X^4 + X^1 + 1 \end{array}$$

Since this is a CRC-4 error-detection circuit, there will be four flip flops in the circuit. An exclusive OR is connected to the last flip and between the first and second one as seen in Figure 3-7.

To complete the circuit, connect the data in to the other input of the exclusive OR connected to the last flip-flop. Take its output and connect it to the other exclusive OR and the D input of the first flip-flop. There you have it—one CRC-4 circuit as shown in Figure 3-8.

When using the circuit, it is first reset to all zeros (in effect adding zeros to the message). The input data stream is fed into the circuit and shifted through. The output is sent to the transmitting circuit. After the message is fully shifted through the register, its contents will contain the CRC character, which is then shifted out attaching itself to the end of the message. As it is shifted out, zeros are shifted in, initializing the register for its next use.

The receiver directs the incoming data stream to the data input of its CRC shift register and the process is repeated. The last set of bits received and shifted through the register is the transmitted CRC character. If all is well, then the register will contain all zeros after everything has been shifted in and through the circuit.

To reduce possible confusion between shifting in zeros to clear the register and detecting a CRC result of zero, some CRC systems are set up so that the receiver's "divisor" is the complement of the transmitters. The result of this setup is that when the receiver is finished, instead of having all zeros in the CRC register if the message is good, it will have all ones.

Checksum Error Detection

checksum
error-detection process that uses the sum of the data stream in bytes.

Another method of error detection uses a process known as **checksum** to generate an error-detection character. The character results from summing all the bytes of a message together, discarding and carry-over from the addition. Again, the process is repeated at the receiver and the two checksums are compared. A match between receiver checksum and transmitted checksum indicates good data. A mismatch indicates an error has occurred.

This method, like CRC, is capable of detecting single or multiple errors in the message. The major advantage of checksum is that it is simple to implement in either hardware or software. The drawback to checksum is that, unless you use a fairly large checksum (16- or 32-bit instead of 8-bit), there are several data-bit patterns that could produce the same checksum result, thereby decreasing its effectiveness. It is possible that if enough errors occur in a message that a checksum could be produced that would be the same as a good message. This is why both checksum and CRC error-detection methods do not catch 100% of the errors that *could* occur, they both come pretty close.

EXAMPLE 3-10

What is the checksum value for the extended ASCII message "Help!"?

SOLUTION

The checksum value is found by adding up the bytes representing the Help! characters:

01001000	H
01100101	e
01101100	l
01110000	p
<u>00100001</u>	!
00010000	Checksum

The hardware solution relies once more on exclusive OR gates, which perform binary-bit addition. Each 8-bits of data are exclusive ORed with the accumulated total of all previous 8-bit groups. The final accumulated total is the checksum character.

Error Correction

Error detection is an acceptable method of handling data errors in lan-based networks because retransmission of most messages result in a short delay and a little extra use of bandwidth resources. Imagine a satellite orbiting around Jupiter or Saturn, transmitting critical visual data as binary stream information. The time it takes for those transmissions to reach Earth is measured in hours. During this time, the satellite has adjusted its orbit and is soaring across new territory and sending additional data. Correcting errors in these messages cannot be done by retransmission. A request for that retransmission takes as long to get to the satellite as the original message took to get to Earth. Then consider the time it would take to retransmit the message. What would the satellite do with new data, reach it while it tries to handle the retransmitting of old data? The memory needed to hold the old data in case it would need to be resent is astronomical to say the least. Instead, an error-correcting method such as the Hamming code is used so that errors can be corrected as they are detected.

Hamming Code

For synchronous data streams, a error-correcting process called **Hamming code** is commonly used. This method is fairly complex from the standpoint of creating and interpreting the error bits. It is implemented in software algorithms and relies on a lot of preliminary conditions agreed upon by the sender and receiver.

Error bits, called Hamming bits, are inserted into the message at random locations. It is believed that the randomness of their locations reduces the statistical odds that these Hamming bits themselves would be in error. This is based on a mathematical assumption that because there are so many more messages bits compared to Hamming bits, that there is a greater chance for a message bit to be in error than for a Hamming bit to be wrong. Another school of thought disputes this, claiming that each and every bit in the message, including the Hamming bits, has the same chance of being corrupted as any other bit. Be that as it may, Hamming bits are inserted into the data stream randomly. The only crucial point in the selection of their locations is that both the sender and receiver are aware of where they actually are.

The first step in the process is to determine how many Hamming bits (**H**) are to be inserted between the message (**M**) bits. Then their actual placement is selected. The number of bits in the message (**M**) are counted and used to solve the following equation to determine the number of Hamming (**H**) bits:

$$2^H \geq M + H - 1 \quad (3-1)$$

Hamming code

error-correction method based on the number of logic 1 states in a message.

Once the number of Hamming bits is determined, the actual placement of the bits into the message is performed. It is important to note that despite the random nature of the Hamming bit placements, the exact same placements must be known and used by both the transmitter and the receiver. This is necessary so that the receiver can remove the Hamming bits from the message sent by the transmitter and compare them with a similar set of bits generated at the receiver.

EXAMPLE 3-11

How many Hamming bits are required when using the Hamming code with the extended ASCII synchronous message “Help!” ?

SOLUTION

The total number of bits in the message is:

$$M = 8\text{-bits/character} \times 5 \text{ characters} = 40 \text{ bits}$$

This number is used in Equation 3-1 to determine the number of Hamming bits:

$$2^H \geq 40 + H + 1$$

The closest value to try is 6 bits for H , since $2^6 = 64$, which is greater than $40 + 6 + 1 = 47$. This satisfies the equation.

Once the Hamming bits are inserted into their positions within the message, their states (high or low) need to be determined. Starting with the least significant bit (LSB) as bit 1, the binary equivalent of each message-bit position with a high (1) state is exclusive ORed with every other bit position containing a 1. The result of the exclusive OR process is the states of the Hamming bits. Once again, as with previous error detection- and correction-processes, it is best to view how the Hamming code works by using an example.

EXAMPLE 3-12

Determine the states of the six Hamming bits inserted into the message “Help!” at every other bit position starting with the LSB.

SOLUTION

In the last example, we determined that six Hamming bits were required for the “Help!” message. For simplicity, we shall insert the Hamming bits a little less randomly:

H	e	l	p	!
0100100001	10010101	10110001	11000000	1H0H0H0H0H1H

Starting from the LSB on the right, the first 1 is encountered in bit position 2, the next in position 12 and so forth:

Bit Position	Equivalent Binary
2	0 0 0 0 1 0
12	0 0 1 1 0 0
19	0 1 0 0 1 1
20	0 1 0 1 0 0
21	0 1 0 1 0 1
25	0 1 1 0 0 1
26	0 1 1 0 1 0
28	0 1 1 1 0 0
29	0 1 1 1 0 1
31	0 1 1 1 1 1
33	1 0 0 0 0 1
36	1 0 0 1 0 0
37	1 0 0 1 0 1
42	1 0 1 0 1 0
45	<u>1 0 1 1 0 1</u>
H	= 1 0 0 1 1 0

All these binary values are exclusive ORed together—an odd number of ones produces a 1, and an even count, a 0—to create the Hamming bits values. These values are substituted for the H-bits in the message. The entire thing is then transmitted and the process repeated at the receiver:

0100100001100101011011000111000000110000010110

If the message was received without any errors, then the Hamming-bit states produced at the receiver will match the ones sent. If an error in one bit did occur during transmission, then the difference between the transmitted Hamming bits and the receiver results will be the bit position of the bad bit. This bit is then inverted to its correct state.

The limitation imposed by the Hamming code is twofold. First, it works only for single-bit errors, and secondly, if one of the Hamming bits becomes corrupted, then the receiver will actually invert a correct bit and place an error in the message stream.

EXAMPLE 3-13

Demonstrate how the Hamming code is used to correct a single-bit error in the data stream.

SOLUTION

During the transmission of the message, bit 19 experiences a noise spike that causes it to be received as a 0 instead of 1. The receiver goes through the

process of determining the states of the Hamming code, resulting in this calculation:

2	0 0 0 0 1 0
12	0 0 1 1 0 0
20	0 1 0 1 0 0
21	0 1 0 1 0 1
25	0 1 1 0 0 1
26	0 1 1 0 1 0
28	0 1 1 1 0 0
29	0 1 1 1 0 1
31	0 1 1 1 1 1
33	1 0 0 0 0 1
36	1 0 0 1 0 0
37	1 0 0 1 0 1
42	1 0 1 0 1 0
45	<u>1 0 1 1 0 1</u>
H =	1 1 0 1 0 1

Notice that bit 19 is not included in the list since it was received as a low-state instead of a high-state. Now we compare the Hamming code transmitted to this one the receiver just derived:

Transmitted code:	1 0 0 1 1 0
Receiver code:	<u>1 1 0 1 0 1</u>
	0 1 0 0 1 1 = bit 19

There is no “black magic” mystery to why the Hamming code works. The originally transmitted codes are formulated by adding binary bits together (the exclusive OR process), ignoring carries. A similar process occurs at the receiver. If a bit has changed, then the two sums will be different and the difference between them will be the bit position number that was not added at either the transmitter or the receiver. By comparing the two Hamming codes using exclusive OR gates, the numbers are effectively being subtracted from one another (another function of the exclusive OR gate) and the difference is the bad bit position.

Section 3.3 Review Questions

1. Why is CRC-16 preferred to parity for error detection in synchronous data systems?
2. Why are larger CRC “divisors” preferred over shorter ones?
3. What is the 16-bit checksum value for the extended ASCII message “That’s a 10-4”? Do not forget space characters!
4. Why are larger checksums preferred over shorter ones?

5. How many Hamming bits are required for the extended ASCII message “Now is the time for all good men. . .”? Do not forget to count all the space and . characters!
6. What happens if a Hamming bit is corrupted during data transfer?
7. What is the stated purpose of placing Hamming bits randomly throughout a message?

3.4 ERROR TESTING EQUIPMENT

There are many different types of equipment used for testing the effect of errors on a data communication link. Two types of equipment used to check for error rate occurrences in communications systems are the **bit-error rate tester (BERT)** and the **error-free second (EFS) test box**.

bit-error rate tester (BERT)

instrument for testing errors in a bit stream.

error free seconds (EFS)

The number of seconds data is transmitted without errors.

bit-error rate (BER)

Measure of the number of errors in a stream of data.

Bit-Error Rate

Bit-error Rate (BER) is the measure of the occurrence of an incorrect bit in a stream of data. It is classified as one error in so many bits transmitted. For example, one bit in 10^6 bits is a bit-error rate of one in a million bits transmitted. Frequently, this specification is shortened to a bit-error rate of 10^{-6} since the one-bit error is understood. The exponent is negated, indicating the infrequency of the error occurrence.

Another measure of bit-error rate is by percentage and is calculated using this formula:

$$\text{bit - error rate} = \frac{\text{number of bad bits}}{\text{total number of bits sent}} \times 100\% \quad (3-2)$$

Bit-error-rate testers are available that can test a data link for a number of different types of error occurrences. Chief among these is the bit-error rate, but also included are parity- and framing-error testing. The tester can be used singularly by tapping into the line between the terminal and the modem as shown in Figure 3-9. In this placement, the tester can be used to monitor the line or to inject test data sequences into the line.

Generally, the receiving end is terminated in a **loop-back** arrangement at some point. Loop-backs take the received data and return them to the sending station. This is accomplished by connecting the transmit and receive data lines together at the originating point (point A in Figure 3-9) to test the local sending loop; the remote end of the telephone line (point B) to test the telephone line connection; or point C, for testing the secondary's receive modem.

A known pattern of data is generated by the bit-error tester and sent down the line. One common and familiar pattern is “Quick brown fox jumps over the lazy dog’s back,”

loop-back

Circuit that returns transmitted data to the source for the purposes of testing the line.

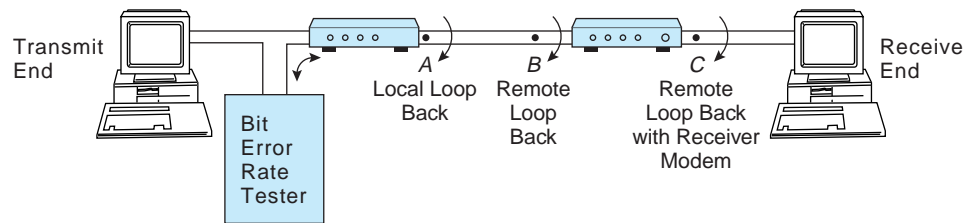


FIGURE 3-9 Line Test Using a Bit Error Rate Tester

which contains all the letters of the alphabet. Other patterns include alternating ones and zeros and repeating single characters. When the data reaches the loop-back, they are echoed back to the sender. The bit-error tester monitors the returned data and checks for errors. A counter in the tester keeps track of how many errors occurred to determine the bit-error rate. Most bit-error testers are capable of operating at a wide range of data rates and can be used to test asynchronous and synchronous data systems.

Error-Free Seconds

Another measure of bit errors called *error-free seconds (EFS)*, is used as a measure of error occurrences for data transmissions from 2.4 Kbps to 2.5 Mbps. Instead of measuring bit errors occurring in a specified number of transmitted bits, EFS is a measure of the number of seconds of transmission time that contain at least one error. An error-free second percentage is computed using this formula:

$$\text{error-free seconds} = 100\% - \frac{\text{seconds with an error}}{\text{total transmission seconds}} \times 100\% \quad (3-3)$$

Error-free second testers are used in the same manner as bit-error rate testers and serve a similar purpose for digital data systems.

EXAMPLE 3-14

Contrast the specifications, bit-error rate, and error-free seconds for a system that experienced five errors in 25 Mbytes of data transmitted in 20 seconds. Two errors occurred in the seventh second of transmission, one error in the twelfth second and the last two errors in the last second of transmission.

SOLUTION

The bit-error rate is found by using Equation 3-2:

$$\text{bit-error rate} = (5 \times 100\%) / (25 M \times 8) = 2.5 \times 10^{-6}\%$$

While error-free seconds uses Equation 3-3:

$$\text{error-free seconds} = 100\% - (3 \text{ error seconds}) / (20) \times 100\% = 85\%$$

This all means that the system experiences 1 error in every 2.5 million bits and is error-free 85% of the time.

Section 3.4 Review Questions

1. What is the bit-error rate for a system that experiences 3 bit errors in a transmission of 512 K bytes of data?
2. What is the error-free second percentage of a system that experiences three seconds with errors in a total transmission time of two hours?

SUMMARY

Error detection and correction methods are necessary to assure the integrity of the data sent from one location to another. The types of methods used support both asynchronous- and synchronous-type data streams. Asynchronous error detection is facilitated by the use of a parity bit with each character of data sent. Error correction for asynchronous data utilizes the LRC/VRC method, which duplicates the parity process (VRC) and examines each character by bit position (LRC). Synchronous data streams apply CRC or checksum for error detection and the Hamming code for error correction. Table 3-2 summarizes the error methods discussed in this chapter and supplies a quick comparison reference for them.

TABLE 3-2

Error Methods Summary

Error Method	Data Type	Detection Corrections	Number of Errors Detectable
Parity	Asynchronous	Detection	One per Character
LRC/VRC	Asynchronous	Correction	One per Message
Checksum	Either	Detection	Unlimited
CRC	Synchronous	Detection	Unlimited
Hamming Code	Synchronous	Correction	One per Message

Error Method	Overhead
Parity	One Bit Added per Character
LRC/VRC	One Bit per Character Plus LRC Character
Checksum	Checksum Character at End of Message
CRC	CRC Bytes at End of Message
Hamming Code	Hamming Bits Inserted into Data Stream

QUESTIONS

Section 3.2

1. Which error-detection method is used most frequently with asynchronous data streams?
2. How many errors can parity reliably detect in a single character?
3. Which type of logic gate is used to generate parity bits?
4. List the uses of an exclusive OR gate.
5. What is one drawback when using parity for error detection?
6. Define overhead in terms of serial data messages.
7. How does ARQ facilitate the correction of messages with errors in them?
8. What is the advantage of using ARQ for error correction?
9. What is the prime limitation of the LRC error-correction method?
10. What is meant by forward error correction?
11. How is the LRC character generated?
12. What does a bad VRC match signify?
13. How does the LRC pinpoint the bad bit in a character?
14. What is the hexadecimal value of the LRC character for the following message using regular (7-bit) ASCII:

Our Last Date

15. What is the hexadecimal value of the LRC character for the following message using extended (8-bit) ASCII:

The Yellow Brick Road

Section 3.3

16. Using the CRC-4 “divisor” used in this chapter, what is the CRC-4 character for this regular ASCII message:

This 1
17. Using the CRC-4 “divisor” used in this chapter, what is the CRC-4 character for this regular ASCII message:

Lefty
18. Draw the CRC-6 block diagram for the circuit using this “divisor”: 1000101
19. Draw the CRC-6 block diagram for the circuit using this “divisor”:

$$X^6 + X^5 + X^1 + 1$$
20. What is the checksum character, in hexadecimal, for the following synchronous message using extended ASCII?

Come to our aid now!
21. What is the checksum character, in hexadecimal, for the following synchronous message using extended ASCII?

mmiller@devry-phx.edu
22. How many Hamming bits are used for the message in question 7?
23. How many Hamming bits are used for the message in question 8?
24. What is the value of the Hamming bits, placed starting with the least significant bit position and in every third place after that for the extended ASCII message “See us”?

(Hamming placement example:H b3 b2 H b1 b0 H)

25. What is the value of the Hamming bits, placed starting with the least significant bit position and in every other place after that for the extended ASCII message “DeVry”?
(Hamming placement example:H b3 H b2 H b1 H b0 H)

Section 3.4

26. Select the error-detection method you would choose for each of the conditions below. Support your selections.
- Direct memory transfer from a hard disk to a computer’s RAM memory.
 - Communication link between your home computer and an associate’s computer via modems and the telephone lines.
 - Messages between workstations on a production line.
 - Communications between bank automatic teller machines and a central computer.
 - Surface photographs of Venus sent to Earth by an Explorer satellite.
 - Computerized accounting network.
27. List the two main test functions of bit-error rate testers.
28. Contrast the functions of bit-error rate and error-free second specifications.

DESIGN PROBLEMS

1. Design, construct, and verify the operation of a parity generator circuit. The circuit allows selection of even or odd parity and indicates when a parity error occurs. Another option for the circuit is to operate with a choice of 7- or 8-bit data inputs.

2. Design, construct, and verify the operation of a CRC-16 generating circuit. The “divisor” to be used is the one discussed in this chapter. The requirements for the circuit are as follows:

- While the CRC-16 bytes are being formed, the original bits are shifted through unchanged.
- After the last data bit is shifted out, the CRC-16 is shifted out from the circuit.
- Before the data are shifted through and after the CRC-16 is shifted out, the data line is to be in an idle line 1 condition.

Essentially, the design of the CRC circuit follows the process shown in the text for CRC-4. However, this design project is extended to include the logic circuitry in conjunction with the CRC-16 circuit to meet the requirements above.

ANSWERS TO REVIEW QUESTIONS

Section 3.1

- Occurrences of bit errors are infrequent, so error detection and correction schemes are designed to handle these infrequent errors.
- A significant interference or major system failure has occurred.

Section 3.2

1. one
2. If an even number of errors occurred within a character, the parity state would be the same as that for a good character.
3. Retransmission of a message when an error is sensed is requested automatically without intervention from the user.
4. Add, subtract, and compare a single bit. Controlled inverter.
5. Even
6. One
7. A bad character
8. The corresponding bad bit in a character.

Section 3.3

1. CRC can detect multiple errors dependably, parity cannot.
2. Larger CRC “divisors” reduce the likelihood that two data streams could produce the same CRC character.
3.

T	0101	0100
h	0110	1000
a	0110	0001
t	0111	0100
'	0010	1100
s	0111	0011
space	0010	0000
a	0110	0001
space	0010	0000
1	0011	0001
0	0011	0000
-	0010	1101
4	<u>0011</u>	<u>0100</u>
	0000	1111 = checksum
4. Reduces the chance that two different messages would produce the same checksum.
5. $M = 35 \times 8 = 280$ bits
 $2^H \geq 280 + H + 1$
 $2^9 = 512 \geq 281 + 9 \geq 290$
9 Hamming bits
6. The receiver could invert a good bit making it bad.
7. Statistically reduces the chance that a Hamming bit is corrupted.

Section 3.4

1. .585 m% or $585 \times 10^{-6}\%$
2. 99.96%