2

CODING TECHNIQUES

2.1 INTRODUCTION

Many errors in a computer system are committed at the bit or byte level when information is either transmitted along communication lines from one computer to another or else within a computer from the memory to the microprocessor or from microprocessor to input/output device. Such transfers are generally made over high-speed internal buses or sometimes over networks. The simplest technique to protect against such errors is the use of error-detecting and error-correcting codes. These codes are discussed in this chapter in this context. In Section 3.9, we see that error-correcting codes are also used in some versions of RAID memory storage devices.

The reader should be familiar with the material in Appendix A and Sections B1–B4 before studying the material of this chapter. It is suggested that this material be reviewed briefly or studied along with this chapter, depending on the reader's background.

The word *code* has many meanings. Messages are commonly coded and decoded to provide secret communication [Clark, 1977; Kahn, 1967], a practice that technically is known as cryptography. The municipal rules governing the construction of buildings are called building codes. Computer scientists refer to individual programs and collections of programs as software, but many physicists and engineers refer to them as computer codes. When information in one system (numbers, alphabet, etc.) is represented by another system, we call that other system a code for the first. Examples are the use of binary numbers to represent numbers or the use of the ASCII code to represent the letters, numerals, punctuation, and various control keys on a computer keyboard (see

Table C.1 in Appendix C for more information). The types of codes that we discuss in this chapter are *error-detecting* and *-correcting codes*. The principle that underlies error-detecting and -correcting codes is the addition of specially computed redundant bits to a transmitted message along with added checks on the bits of the received message. These procedures allow the detection and sometimes the correction of a modest number of errors that occur during transmission.

The computation associated with generating the redundant bits is called *coding*; that associated with detection or correction is called *decoding*. The use of the words *message*, *transmitted*, and *received* in the preceding paragraph reveals the origins of error codes. They were developed along with the mathematical theory of information largely from the work of C. Shannon [1948], who mentioned the codes developed by Hamming [1950] in his original article. (For a summary of the theory of information and the work of the early pioneers in coding theory, see J. R. Pierce [1980, pp. 159–163].) The preceding use of the term *transmitted bits* implies that coding theory is to be applied to digital signal transmission (or a digital model of analog signal transmission), in which the signals are generally pulse trains representing various sequences of 0s and 1s. Thus these theories seem to apply to the field of communications; however, they also describe information transmission in a computer system. Clearly they apply to the signals that link computers connected by modems and telephone lines or local area networks (LANs) composed of transceivers, as well as coaxial wire and fiber-optic cables or wide area networks (WANs) linking computers in distant cities. A standard model of computer architecture views the central processing unit (CPU), the address and memory buses, the input/output (I/O) devices, and the memory devices (integrated circuit memory chips, disks, and tapes) as digital signal (computer word) transmission, storage, manipulation, generation, and display devices. From this perspective, it is easy to see how error-detecting and -correcting codes are used in the design of modems, memory stems, disk controllers (optical, hard, or floppy), keyboards, and printers.

The difference between error detection and error correction is based on the use of redundant information. It can be illustrated by the following electronic mail message:

> Meet me in Manhattan at the information desk at Senn Station on July 43. I will arrive at 12 noon on the train from Philadelphia.

Clearly we can *detect an error* in the date, for extra information about the calendar tells us that there is no date of July 43. Most likely the digit should be a 1 or a 2, but we can't tell; thus the error can't be corrected without further information. However, just a bit of extra knowledge about New York City railroad stations tells us that trains from Philadelphia arrive at Penn (Pennsylvania) Station in New York City, not the Grand Central Terminal or the PATH Terminal. Thus, Senn is not only *detected* as an error, but is also *corrected* to Penn. Note

that in all cases, error detection and correction required additional (redundant) information. We discuss both error-detecting and error-correcting codes in the sections that follow. We could of course send return mail to request a retransmission of the e-mail message (again, redundant information is obtained) to resolve the obvious transmission or typing errors.

In the preceding paragraph we discussed retransmission as a means of correcting errors in an e-mail message. The errors were detected by a redundant source and our knowledge of calendars and New York City railroad stations. In general, with pulse trains we have no knowledge of "the right answer." Thus if we use the simple brute force redundancy technique of transmitting each pulse sequence twice, we can compare them to detect errors. (For the moment, we are ignoring the rare situation in which both messages are identically corrupted and have the same wrong sequence.) We can, of course, transmit three times, compare to detect errors, and select the pair of identical messages to provide error correction, but we are again ignoring the possibility of identical errors during two transmissions. These brute force methods are inefficient, as they require many redundant bits. In this chapter, we show that in some cases the addition of a single redundant bit will greatly improve error-detection capabilities. Also, the efficient technique for obtaining error correction by adding more than one redundant bit are discussed. The method based on triple or N copies of a message are covered in Chapter 4. The coding schemes discussed so far rely on short "noise pulses," which generally corrupt only one transmitted bit. This is generally a good assumption for computer memory and address buses and transmission lines; however, disk memories often have sequences of errors that extend over several bits, or *burst errors*, and different coding schemes are required.

The measure of performance we use in the case of an error-detecting code is the *probability of an undetected error*, P_{ue}, which we of course wish to minimize. In the case of an error-correcting code, we use the *probability of transmitted error*, P_e, as a measure of performance, or the *reliability*, R, (*probability of success*), which is $(1 - P_e)$. Of course, many of the more sophisticated coding techniques are now feasible because advanced integrated circuits (logic and memory) have made the costs of implementation (dollars, volume, weight, and power) modest.

The type of code used in the design of digital devices or systems largely depends on the types of errors that occur, the amount of redundancy that is cost-effective, and the ease of building coding and decoding circuitry. The source of errors in computer systems can be traced to a number of causes, including the following:

1. Component failure
2. Damage to equipment
3. "Cross-talk" on wires
4. Lightning disturbances

5. Power disturbances
6. Radiation effects
7. Electromagnetic fields
8. Various kinds of electrical noise

Note that we can roughly classify sources 1, 2, and 3 as causes that are internal to the equipment; sources 4, 6, and 7 as generally external causes; and sources 5 and 6 as either internal or external. Classifying the source of the disturbance is only useful in minimizing its strength, decreasing its frequency of occurrence, or changing its other characteristics to make it less disturbing to the equipment. The focus of this text is what to do to protect against these effects and how the effects can compromise performance and operation, assuming that they have occurred. The reader may comment that many of these error sources are rather rare; however, our desire for ultrareliable, long-life systems makes it important to consider even rare phenomena.

The various types of interference that one can experience in practice can be illustrated by the following two examples taken from the aircraft field. Modern aircraft are crammed full of digital and analog electronic equipment that are generally referred to as avionics. Several recent instances of military crashes and civilian troubles have been noted in modern electronically controlled aircraft. These are believed to be caused by various forms of electromagnetic interference, such as passenger devices (e.g., cellular telephones); "cross-talk" between various onboard systems; external signals (e.g., Voice of America Transmitters and Military Radar); lightning; and equipment malfunction [Shooman, 1993]. The systems affected include the following: autopilot, engine controls, communication, navigation, and various instrumentation. Also, a previous study by Cockpit (the pilot association of Germany) [Taylor, 1988, pp. 285–287] concluded that the number of soft fails (probably from alpha particles and cosmic rays affecting memory chips) increased in modern aircraft. See Table 2.1 for additional information.

TABLE 2.1 Increase of Soft Fails with Airplane Generation

Airplane Type	Altitude (1,000s feet)				Total Reports	No. of Aircraft	Soft Fails per a/c
	Ground-5	5–20	20–30	30+			
B707	2	0	0	2	4	14	0.29
B727/737	11	7	2	4	24	39/28	0.36
B747	11	0	1	6	18	10	1.80
DC10	21	5	0	29	55	13	4.23
A300	96	12	6	17	131	10	13.10

Source: [Taylor, 1988].

It is not clear how the number of flight hours varied among the different airplane types, what the computer memory sizes were for each of the aircraft, and the severity level of the fails. It would be interesting to compare this data to that observed in the operation of the most advanced versions of B747 and A320 aircraft, as well as other more recent designs.

There has been much work done on coding theory since 1950 [Rao, 1989]. This chapter presents a modest sampling of theory as it applies to fault-tolerant systems.

2.2 BASIC PRINCIPLES

Coding theory can be developed in terms of the mathematical structure of *groups, subgroups, rings, fields, vector spaces, subspaces, polynomial algebra*, and *Galois fields* [Rao, 1989, Chapter 2]. Another simple yet effective development of the theory based on algebra and logic is used in this text [Arazi, 1988].

2.2.1 Code Distance

We will deal with strings of binary digits (0 or 1), which are of specified length and called the following synonymous terms: binary *block*, binary *vector*, binary *word*, or just *code word*. Suppose that we are dealing with a 3-bit message (b_1, b_2, b_3) represented by the bits x_1, x_2, x_3. We can speak of the eight combinations of these bits—see Table 2.2(a)—as the code words. In this case they are assigned according to the sequence of binary numbers. The *distance* of a code is the *minimum* number of bits by which any one code word differs from another. For example, the first and second code words in Table 2.2(a) differ only in the right-most digit and have a distance of 1, whereas the first and the last code words differ in all 3 digits and have a distance of 3. The total number of comparisons needed to check all of the word pairs for the minimum code distance is the number of combinations of 8 items taken 2 at a time $\binom{8}{2}$, which is equal to 8!/2!6! = 28.

A simpler way of visualizing the distance is to use the "cube method" of displaying switching functions. A cube is drawn in three-dimensional space (x, y, z), and a main diagonal goes from $x = y = z = 0$ to $x = y = z = 1$. The distance is the number of cube edges between any two code words that represent the vertices of the cube. Thus, the distance between 000 and 001 is a single cube edge, but the distance between 000 and 111 is 3 since 3 edges must be traversed to get between the two vertices. (In honor of one of the pioneers of coding theory, the code distance is generally called the *Hamming distance*.) Suppose that noise changes a single bit of a code word from 0 to 1 or 1 to 0. The first code word in Table 2.2(a) would be changed to the second, third, or fifth, depending on which bit was corrupted. Thus there is no way to detect a single-bit error (or a multibit error), since any change in a code word transforms it

TABLE 2.2 Examples of 3- and 4-Bit Code Words

(a) 3-Bit Code Words			(b) 4-Bit Code Words: 3 Original Bits plus Added Even-Parity (Legal Code Words)				(c) Illegal Code Words for the Even-Parity Code of (b)			
x_1 b_1	x_2 b_2	x_3 b_3	x_1 p_1	x_2 b_1	x_3 b_2	x_4 b_3	x_1 p_1	x_2 b_1	x_3 b_2	x_4 b_3
0	0	0	0	0	0	0	1	0	0	0
0	0	1	1	0	0	1	0	0	0	1
0	1	0	1	0	1	0	0	0	1	0
0	1	1	0	0	1	1	1	0	1	1
1	0	0	1	1	0	0	0	1	0	0
1	0	1	0	1	0	1	1	1	0	1
1	1	0	0	1	1	0	1	1	1	0
1	1	1	1	1	1	1	0	1	1	1

into another legal code word. One can create error-detecting ability in a code by adding *check bits*, also called *parity bits*, to a code.

The simplest coding scheme is to add one redundant bit. In Table 2.2(b), a single check bit (parity bit p_1) is added to the 3-bit code words b_1, b_2, and b_3 of Table 2.2(a), creating the eight new code words shown. The scheme used to assign values to the parity bit is the coding rule; in this case, p_1 is chosen so that the number of one bits in each word is an even number. Such a code is called an *even-parity* code, and the words in Table 2.1(b) become legal code words and those in Table 2.1(c) become illegal code words. Clearly we could have made the number of one bits in each word an odd number, resulting in an *odd-parity* code, and so the words in Table 2.1(c) would become the legal ones and those in 2.1(b) become illegal.

2.2.2 Check-Bit Generation and Error Detection

The code generation rule (even parity) used to generate the parity bit in Table 2.2(b) will now be used to design a parity-bit generator circuit. We begin with a Karnaugh map for the switching function p_1 (b_1, b_2, and b_3) where the parity bit is a function of the three code bits as given in Fig. 2.1(a). The resulting Karnaugh map is given in this figure. The top left cell in the map corresponds to $p_1 = 0$ when b_1, b_2, and $b_3 = 000$, whereas the top right cell represents $p_1 = 1$ when b_1, b_2, and $b_3 = 001$. These two cells represent the first two rows of Table 2.2(b); the other cells in the map represent the other six rows in the table. Since none of the ones in the Karnaugh map touch, no simplification is possible, and there are four minterms in the circuit, each generated by the four gates shown in the circuit. The OR gate "collects" these minterms, generating a parity check bit p_1 whenever a sequence of pulses b_1, b_2, and b_3 occurs.

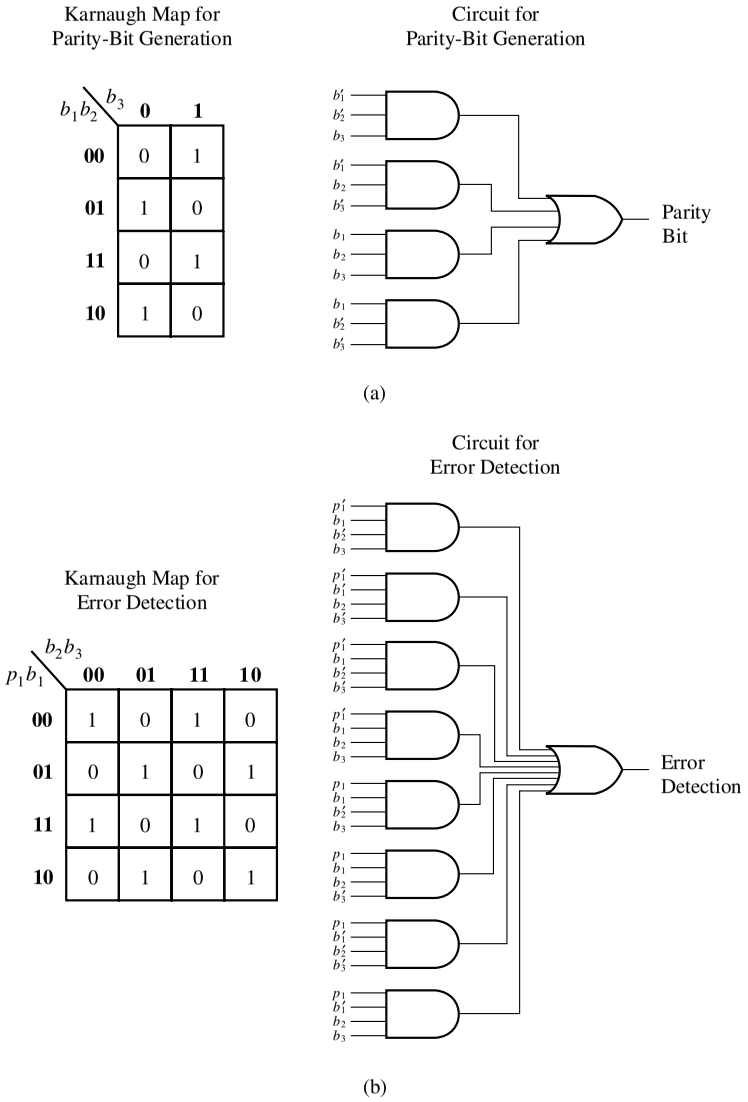


Figure 2.1 Elementary parity-bit coding and decoding circuits. (a) Generation of an even-parity bit for a 3-bit code word. (b) Detection of an error for an even-parity-bit code for a 3-bit code word.

The addition of the parity bit creates a set of legal and illegal words; thus we can detect an error if we check for legal or illegal words. In Fig. 2.1(b) the Karnaugh map displays ones for legal code words and zeroes for illegal code words. Again, there is no simplification since all the minterms are separated, so the error detector circuit can be composed by generating all the illegal word minterms (indicated by zeroes) in Fig. 2.1(b) using eight AND gates followed by an 8-input OR gate as shown in the figure. The circuits derived in Fig. 2.1 can be simplified by using exclusive or (EXOR) gates (as shown in the next section); however, we have demonstrated in Fig. 2.1 how check bits can be generated and how errors can be detected. Note that parity checking will detect errors that occur in either the message bits or the parity bit.

2.3 PARITY-BIT CODES

2.3.1 Applications

Three important applications of parity-bit error-checking codes are as follows:

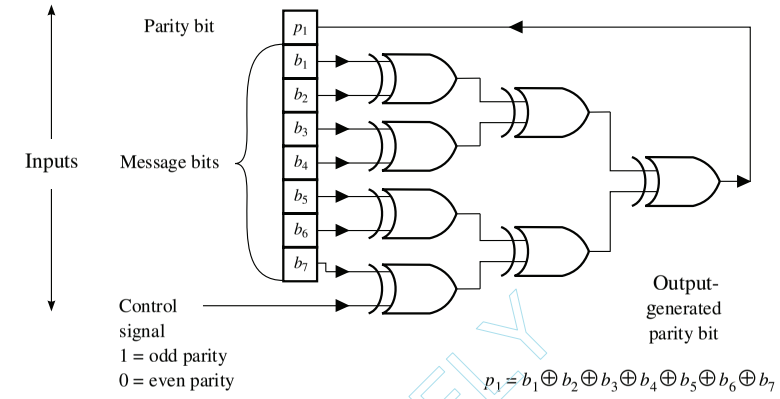
1. The transmission of characters over telephone lines (or optical, microwave, radio, or satellite links). The best known application is the use of a modem to allow computers to communicate over telephone lines.
2. The transmission of data to and from electronic memory (memory read and write operations).
3. The exchange of data between units within a computer via various data and control buses.

Specific implementation details may differ among these three applications, but the basic concepts and circuitry are very similar. We will discuss the first application and use it as an illustration of the basic concepts.

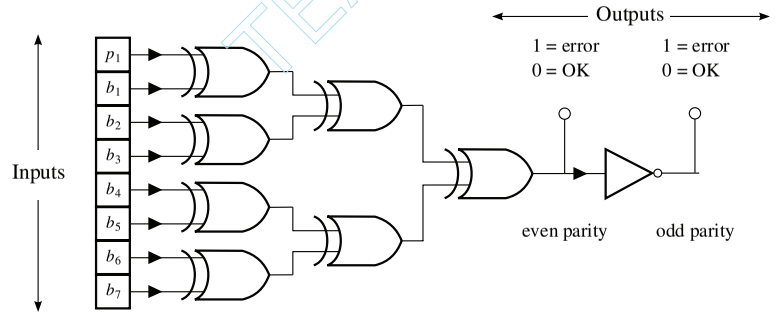
2.3.2 Use of Exclusive OR Gates

This section will discuss how an additional bit can be added to a byte for error detection. It is common to represent alphanumeric characters in the input and output phases of computation by a single byte. The ASCII code is almost universally used. One technique uses the entire byte to represent $2^8 = 256$ possible characters (the extended character set that is used on IBM personal computers, containing some Greek letters, language accent marks, graphic characters, and so forth, as well as an additional ninth parity bit. The other approach limits the character set to 128, which can be expressed by seven bits, and uses the eighth bit for parity.

Suppose we wish to build a parity-bit generator and code checker for the case of seven message bits and one parity bit. Identifying the minterms will reveal a generalization of the checkerboard diagram similar to that given in the



(a) Parity-Bit Encoder (generator)



(b) Parity-Bit Decoder (checker)

Figure 2.2 Parity-bit encoder and decoder for a transmitted byte: (a) A 7-bit parity encoder (generator); (b) an 8-bit parity decoder (checker).

Karnaugh maps of Fig. 2.1. Such checkerboard patterns indicate that EXOR gates can be used to simplify the circuit. A circuit using EXOR gates for parity-bit generation and for checking of an 8-bit byte is given in Fig. 2.2. Note that the circuit in Fig. 2.2(a) contains a control input that allows one to easily switch from even to odd parity. Similarly, the addition of the NOT gate (inverter) at the output of the checking circuit allows one to use either even or odd parity.

Most modems have these refinements, and a switch chooses either even or odd parity.

2.3.3 Reduction in Undetected Errors

The purpose of parity-bit checking is to detect errors. The extent to which such errors are detected is a measure of the success of the code, whereas the probability of not detecting an error, P_{ue}, is a measure of failure. In this section we analyze how parity-bit coding decreases P_{ue}. We include in this analysis the reliability of the parity-bit coding and decoding circuit by analyzing the reliability of a standard IC parity code generator/checker. We model the failure of the IC chip in a simple manner by assuming that it fails to detect errors, and we ignore the possibility that errors are detected when they are not present.

Let us consider the addition of a ninth parity bit to an 8-bit message byte. The parity bit adjusts the number of ones in the word to an even (odd) number and is computed by a parity-bit generator circuit that calculates the EXOR function of the 8 message bits. Similarly, an EXOR-detecting circuit is used to check for transmission errors. If 1, 3, 5, 7, or 9 errors are found in the received word, the parity is violated, and the checking circuit will detect an error. This can lead to several consequences, including "flagging" the error byte and retransmission of the byte until no errors are detected. The probability of interest is the probability of an undetected error, P'_{ue}, which is the probability of 2, 4, 6, or 8 errors, since these combinations do not violate the parity check. These probabilities can be calculated by simply using the binomial distribution (see Appendix A5.3). The probability of r failures in n occurrences with failure probability q is given by the binomial probability $B(r:n,q)$. Specifically, $n = 9$ (the number of bits) and $q =$ the probability of an error per transmitted bit; thus

General:

$$B(r:9,q) = \binom{9}{r} q^r (1-q)^{9-r} \tag{2.1}$$

Two errors:

$$B(2:9,q) = \binom{9}{2} q^2 (1-q)^{9-2} \tag{2.2}$$

Four errors:

$$B(4:9,q) = \binom{9}{4} q^4 (1-q)^{9-4} \tag{2.3}$$

and so on.

For q, relatively small (10^{-4}), it is easy to see that Eq. (2.3) is much smaller than Eq. (2.2); thus only Eq. (2.2) needs to be considered (probabilities for r = 4, 6, and 8 are negligible), and the probability of an undetected error with parity-bit coding becomes

$$P'_{ue} = B(2:9, q) = 36q^2(1-q)^7 \tag{2.4}$$

We wish to compare this with the probabilty of an undetected error for an 8-bit transmission without any checking. With no checking, all errors are undetected; thus we must compute $B(1:8, q) + \cdots + B(8:8, q)$, but it is easier to compute

$$\begin{aligned} P_{ue} &= 1 - P(0 \text{ errors}) = 1 - B(0:8, q) = 1 - \binom{8}{0} q^0(1-q)^{8-0} \\ &= 1 - (1-q)^8 \end{aligned} \tag{2.5}$$

Note that our convention is to use P_{ue} for the case of no checking, and P'_{ue} for the case of checking.

The ratio of Eqs. (2.5) and (2.4) yields the improvement ratio due to the parity-bit coding as follows:

$$P_{ue}/P'_{ue} = [1 - (1-q)^8]/[36q^2(1-q)^7] \tag{2.6}$$

For small q we can simplify Eq. (2.6) by replacing $(1 \pm q)^n$ by $1 \pm nq$ and $[1/(1-q)]$ by $1 + q$, which yields

$$P_{ue}/P'_{ue} = [2(1+7q)/9q] \tag{2.7}$$

The parameter q, the probability of failure per bit transmitted, is quoted as 10^{-4} in Hill and Peterson [1981]. The failure probability q was 10^{-5} or 10^{-6} in the 1960s and '70s; now, it may be as low as 10^{-7} for the best telephone lines [Rubin, 1990]. Equation (2.7) is evaluated for the range of q values; the results appear in Table 2.3 and in Fig. 2.3.

The improvement ratio is quite significant, and the overhead—adding 1 parity bit out of 8 message bits—is only 12.5%, which is quite modest. This probably explains why a parity-bit code is so frequently used.

In the above analysis we assumed that the coder and decoder are perfect. We now examine the validity of that assumption by modeling the reliability of the coder and decoder. One could use a design similar to that of Fig. 2.2; however, it is more realistic to assume that we are using a commercial circuit device: the SN74180, a 9-bit odd/even parity generator/checker (see Texas Instruments [1988]), or the newer 74LS280 [Motorola, 1992]. The SN74180 has an equivalent circuit (see Fig. 2.4), which has 14 gates and inverters, whereas the pin-compatible 74LS280 with improved performance has 46 gates and inverters in its equivalent circuit. Current prices of the SN74180 and the similar 74LS280 ICs are about 10–75 cents each, depending on logic family and order quantity. We will use two such devices since the same chip can be used as a coder and a decoder (generator/checker). The logic diagram of this device is shown in Fig. 2.4.

TABLE 2.3 Evaluation of the Reduction in Undetected Errors from Parity-Bit Coding: Eq. (2.7)

Bit Error Probability, q	Improvement Ratio: P_{ue}/P'_{ue}
10^{-4}	2.223×10^3
10^{-5}	2.222×10^4
10^{-6}	2.222×10^5
10^{-7}	2.222×10^6
10^{-8}	2.222×10^7

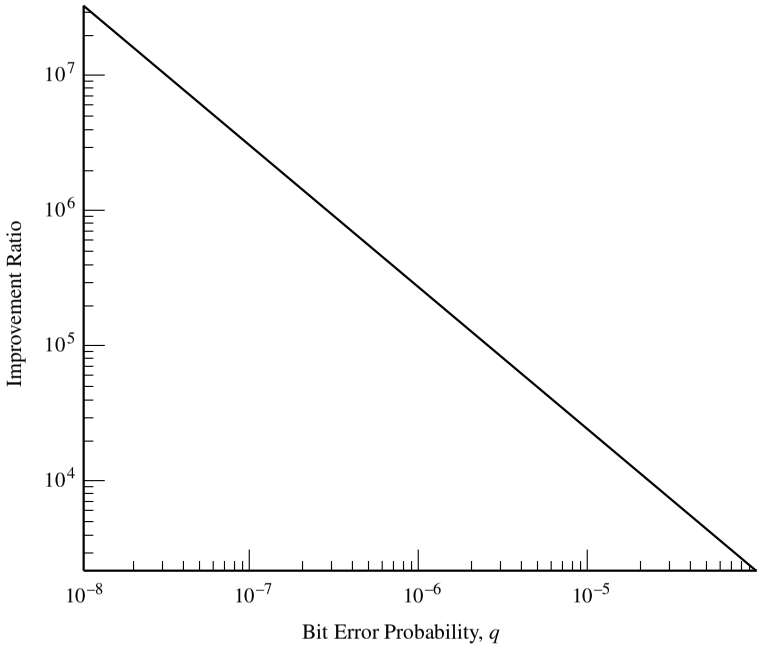


Figure 2.3 Improvement ratio of undetected error probability from parity-bit coding.

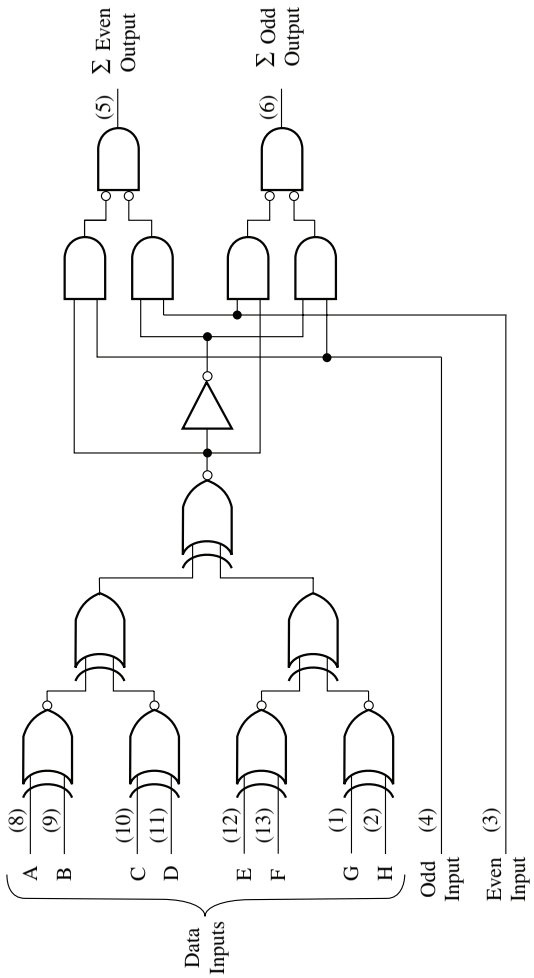


Figure 2.4 Logic diagram for SN74180 [Texas Instruments, 1988, used with permission].

2.3.4 Effect of Coder–Decoder Failures

An approximate model for IC reliability is given in Appendix B3.3, Fig. B7. The model assumes the failure rate of an integrated circuit is proportional to the square root of the number of gates, g, in the equivalent logic model. Thus the failure rate per million hours is given as $\lambda_b = C(g)^{1/2}$, where C was computed from 1985 IC failure-rate data as 0.004. We can use this model to estimate the failure rate and subsequently the reliability of an IC parity generator checker. In the equivalent gate model for the SN74180 given in Fig. 2.4, there are 5 EXNOR, 2 EXOR, 1 NOT, 4 AND, and 2 NOR gates. Note that the output gates (5) and (6) are NOR rather than OR gates. Sometimes for good and proper reasons integrated circuit designers use equivalent logic using different gates. Assuming the 2 EXOR and 5 EXNOR gates use about 1.5 times as many transistors to realize their function as the other gates, we consider them as equivalent to 10.5 gates. Thus we have 17.5 equivalent gates and $\lambda_b = 0.004(17.5)^{1/2}$ failures per million hours = 1.67×10^{-8} failures per hour.

In formulating a reliability model for a parity-bit coder–decoder scheme, we must consider two modes of failure for the coded word: A, where the coder and decoder do not fail but the number of bit errors is an even number equal to 2 or more; and B, where the coder or decoder chip fails. We ignore chip failure modes, which sometimes give correct results. The probability of undetected error with the coding scheme is given by

$$P'_{ue} = P(A + B) = P(A) + P(B) \tag{2.8}$$

In Eq. (2.8), the chip failure rates are per hour; thus we write Eq. (2.8) as

$$\begin{aligned} P'_{ue} = {} & P[\text{no coder or decoder failure during 1 byte transmission}] \\ & \times P[\text{2 or more errors}] \\ & + P[\text{coder or decoder failure during 1 byte transmission}] \end{aligned} \tag{2.9}$$

If we let B be the bit transmission rate per second, then the number of seconds to transmit a bit is $1/B$. Since a byte plus parity is 9 bits, it will take $9/B$ seconds to transmit and $9/3{,}600B$ hours to transmit the 9 bits.

If we assume a constant failure rate λ_b for the coder and decoder, the reliability of a coder–decoder pair is $e^{-2\lambda_b t}$ and the probability of coder or decoder failure is $(1 - e^{-2\lambda_b t})$. The probability of 2 or more errors per hour is given by Eq. (2.4); thus Eq. (2.9) becomes

$$P'_{ue} = e^{-2\lambda_b t} \times 36q^2(1 - q)^7 + (1 - e^{-2\lambda_b t}) \tag{2.10}$$

where

$$t = 9/3{,}600B \tag{2.11}$$

TABLE 2.4 The Reduction in Undetected Errors from Parity-Rate Coding Including the Effect of Coder–Decoder Failures

Bit Error Probability q	Improvement Ratio: P_{ue}/P'_{ue} for Several Transmission Rates			
	300 Bits/Sec	1,200 Bits/Sec	9,600 Bits/Sec	56,000 Bits/Sec
10^{-4}	2.223×10^3	2.223×10^3	2.223×10^3	2.223×10^3
10^{-5}	2.222×10^4	2.222×10^4	2.222×10^4	2.222×10^4
10^{-6}	2.228×10^5	2.218×10^5	2.222×10^5	2.222×10^5
10^{-7}	1.254×10^6	1.962×10^6	2.170×10^6	2.213×10^6
5×10^{-8}	1.087×10^6	2.507×10^6	4.053×10^6	4.372×10^6
10^{-8}	2.841×10^5	1.093×10^6	6.505×10^6	1.577×10^7

The undetected error probability with no coding is given by Eq. (2.5) and is independent of time

$$P_{ue} = 1 - (1 - q)^8 \tag{2.12}$$

Clearly if the failure rate is small or the bit rate B is large, $e^{-2\lambda_b t} \approx 1$, the failure probabilities of the coder–decoder chips are insignificant, and the ratio of Eq. (2.12) and Eq. (2.10) will reduce to Eq. (2.7) for high bit rates B. If we are using a parity code for memory bit checking, the bit rate will be essentially the memory cycle time if we assume that a long succession of memory operations and the effect of chip failures are negligible. However, in the case of parity-bit coding in a modem, the baud rate will be lower and chip failures can be significant, especially in the case where q is small. The ratio of Eq. (2.12) to Eq. (2.10) is evaluated in Table 2.4 (and plotted in Fig. 2.5) for typical modem bit rates B = 300, 1,200, 9,600, and 56,000. Note that the chip failure rate is insignificant for $q = 10^{-4}$, 10^{-5}, and 10^{-6}; however, it does make a difference for $q = 10^{-7}$ and 10^{-8}. If the bit rate B is infinite, the effect of chip failure disappears, and we can view Table 2.3 as depicting this case.

2.4 HAMMING CODES

2.4.1 Introduction

In this section, we develop a class of codes created by Richard Hamming [1950], for whom they are named. These codes will employ c check bits to detect more than a single error in a coded word, and if enough check bits are used, some of these errors can be corrected. The relationships among the number of check bits and the number of errors that can be detected and corrected are developed in the following section. It will not be surprising that the case in which $c = 1$ results in a code that can detect single errors but cannot correct errors; this is the parity-bit code that we had just discussed.

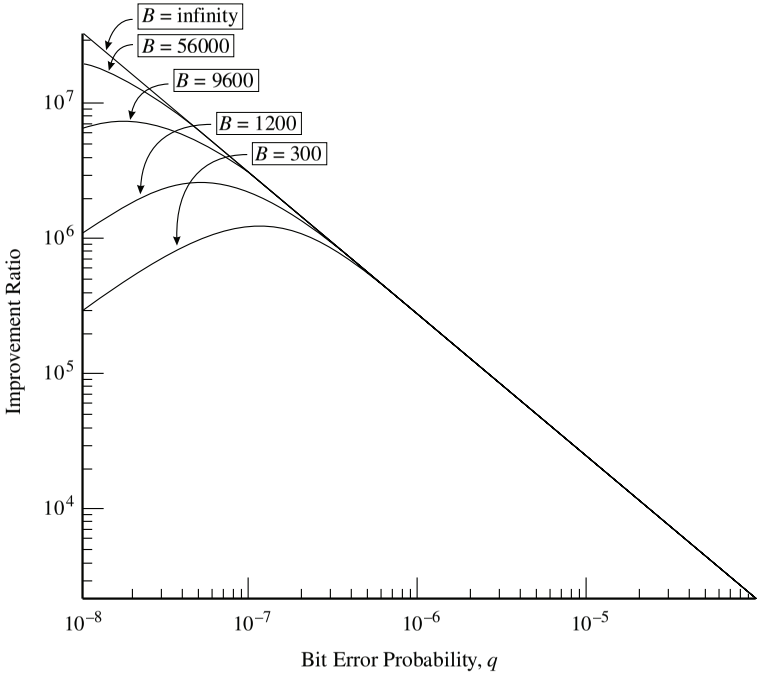


Figure 2.5 Improvement ratio of undetected error probability from parity-bit coding (including the possibility of coder–decoder failure). B is the transmission rate in bits per second.

2.4.2 Error-Detection and -Correction Capabilities

We defined the concept of Hamming distance of a code in the previous section. Now, we establish the error-detecting and -correcting abilities of a code based on its Hamming distance. The following results apply to *linear codes*, in which the difference and sum between any two code words (addition and subtraction of their binary representations) is also a code word. Most of this chapter will deal with linear codes. The following notations are used in this chapter:

d = the Hamming distance of a code (2.13)

D = the number of errors that a code can detect (2.14a)

C = the number of errors that a code can correct (2.14b)

n = the total number of bits in the coded word (2.15a)

$$m = \text{the number of message or information bits} \tag{2.15b}$$

$$c = \text{the number of check (parity) bits} \tag{2.15c}$$

where d, D, C, n, m, and c are all integers ≥ 0.

As we said previously, the model we will use is one in which the check bits are added to the message bits by the coder. The message is then "transmitted," and the decoder checks for any detectable errors. If there are enough check bits, and if the circuit is so designed, some of the errors are corrected. Initially, one can view the error-detection process as a check of each received word to see if the word belongs to the illegal set of words. Any set of errors that convert a legal code word into an illegal one are detected by this process, whereas errors that change a legal code word into another legal code word are not detected. To detect D errors, the Hamming distance must be at least one larger than D.

$$d \geq D + 1 \tag{2.16}$$

This relationship must be so because a single error in a code word produces a new word that is a distance of one from the transmitted word. However, if the code has a basic distance of one, this error results in a new word that belongs to the legal set of code words. Thus for this single error to be detectable, the code must have a basic distance of two so that the new word produced by the error does not belong to the legal set and therefore must correspond to the detectable illegal set. Similarly, we could argue that a code that can detect two errors must have a Hamming distance of three. By using induction, one establishes that Eq. (2.16) is true.

We now discuss the process of error correction. First, we note that to correct an error we must be able to detect that an error has occurred. Suppose we consider the parity-bit code of Table 2.2. From Eq. (2.16) we know that $d \geq 2$ for error detection; in fact, $d = 2$ for the parity-bit code, which means that we have a set of legal code words that are separated by a Hamming distance of at least two. A single bit error creates an illegal code word that is a distance of one from *more than 1* legal code word; thus we cannot correct the error by seeking the closest legal code word. For example, consider the legal code word 0000 in Table 2.2(b). Suppose that the last bit is changed to a one yielding 0001, which is the second illegal code word in Table 2.2(c). Unfortunately, the distance from that illegal word to each of the eight legal code words is 1, 1, 3, 1, 3, 1, 3, and 3 (respectively). Thus there is a four-way tie for the closest legal code word. Obviously we need a larger Hamming distance for error correction. Consider the number line representing the distance between any 2 legal code words for the case of $d = 3$ shown in Fig. 2.6(a). In this case, if there is 1 error, we move 1 unit to the right from word a toward word b. We are still 2 units away from word b and at least that far away from any other word, so we can recognize word a as the closest and select it as the correct word. We can generalize this principle by examining Fig. 2.6(b). If there are C errors to correct, we have moved a distance of C away from code word a; to have this

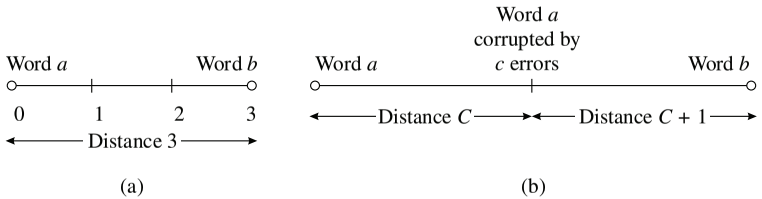


Figure 2.6 Number lines representing the distances between two legal code words.

word closer than any other word, we must have at least a distance of $C + 1$ from the erroneous code word to the nearest other legal code word so we can correct the errors. This gives rise to the formula for the number of errors that can be corrected with a Hamming distance of d, as follows:

$$d \geq 2C + 1 \tag{2.17}$$

Inspecting Eqs. (2.16) and (2.17) shows that for the same value of d,

$$D \geq C \tag{2.18}$$

We can combine Eqs. (2.17) and (2.18) by rewriting Eq. (2.17) as

$$d \geq C + C + 1 \tag{2.19}$$

If we use the smallest value of D from Eq. (2.18), that is, $D = C$, and substitute for one of the Cs in Eq. (2.19), we obtain

$$d \geq D + C + 1 \tag{2.20}$$

which summarizes and combines Eqs. (2.16) to (2.18).

One can develop the entire class of Hamming codes by solving Eq. (2.20), remembering that $D \geq C$ and that d, D, and C are integers ≥ 0. For $d = 1$, $D = C = 0$—no code is possible; if $d = 2$, $D = 1$, $C = 0$—we have the parity bit code. The class of codes governed by Eq. (2.20) is given in Table 2.5.

The most popular codes are the parity code; the $d = 3$, $D = C = 1$ code—generally called a single error-correcting and single error-detecting (SECSED) code; and the $d = 4$, $D = 2$, $C = 1$ code—generally called a single error-correcting and double error-detecting (SECDED) code.

2.4.3 The Hamming SECSED Code

The Hamming SECSED code has a distance of 3, and corrects and detects 1 error. It can also be used as a double error-detecting code (DED).

Consider a Hamming SECSED code with 4 message bits (b_1, b_2, b_3, and b_4) and 3 check bits (c_1, c_2, and c_3) that are computed from the message bits by equations integral to the code design. Thus we are dealing with a 7-bit word. A brute

TABLE 2.5 Relationships Among *d*, *D*, and *C*

d	*D*	*C*	Type of Code
1	0	0	No code possible
2	1	0	Parity bit
3	1	1	Single error detecting; single error correcting
3	2	0	Double error detecting; zero error correcting
4	3	0	Triple error detecting; zero error correcting
4	2	1	Double error detecting; single error correcting
5	4	0	Quadruple error detecting; zero error correcting
5	3	1	Triple error detecting; single error correcting
5	2	2	Double error detecting; double error correcting
6	5	0	Quintuple error detecting; zero error correcting
6	4	1	Quadruple error detecting; single error correcting
6	3	2	Triple error detecting; double error correcting
etc.			

force detection–correction algorithm would be to compare the coded word in question with all the $2^7 = 128$ code words. No error is detected if the coded word matched any of the $2^4 = 16$ legal combinations of message bits. No detected errors means either that none have occurred or that too many errors have occurred (the code is not powerful enough to detect so many errors). If we detect an error, we compute the distance between the illegal code word and the 16 legal code words and effect error correction by choosing the code word that is closest. Of course, this can be done in one step by computing the distance between the coded word and all 16 legal code words. If one distance is 0, no errors are detected; otherwise the minimum distance points to the corrected word.

The information in Table 2.5 just tells us the possibilities in constructing a code; it does not tell us how to construct the code. Hamming [1950] devised a scheme for coding and decoding a SECSED code in his original work. Check bits are interspersed in the code word in bit positions that correspond to powers of 2. Word positions that are not occupied by check bits are filled with message bits. The length of the coded word is n bits composed of c check bits added to m message bits. The common notation is to denote the code word (also called *binary word, binary block*, or *binary vector*) as (n, m). As an example, consider a (7, 4) code word. The 3 check bits and 4 message bits are located as shown in Table 2.6.

TABLE 2.6 Bit Positions for Hamming SECSED ($d = 3$) Code

Bit positions	x_1	x_2	x_3	x_4	x_5	x_6	x_7
Check bits	c_1	c_2	—	c_3	—	—	—
Message bits	—	—	b_1	—	b_2	b_3	b_4

TABLE 2.7 Relationships Among *n*, *c*, and *m* for a SECSED Hamming Code

Length, *n*	Check Bits, *c*	Message Bits, *m*
1	1	0
2	2	0
3	2	1
4	3	1
5	3	2
6	3	3
7	3	4
8	4	4
9	4	5
10	4	6
11	4	7
12	4	8
13	4	9
14	4	10
15	4	11
16	5	11
etc.		

In the code shown, the 3 check bits are sufficient for codes with 1 to 4 message bits. If there were another message bit, it would occupy position x_9, and position x_8 would be occupied by a fourth check bit. In general, c check bits will cover a maximum of $(2^c - 1)$ word bits or $2^c \geq n + 1$. Since $n = c + m$, we can write

$$2^c \geq [c + m + 1] \tag{2.21}$$

where the notation $[c + m + 1]$ means the smallest integer value of c that satisfies the relationship. One can solve Eq. (2.21) by assuming a value of n and computing the number of message bits that the various values of c can check. (See Table 2.7.)

If we examine the entry in Table 2.7 for a message that is 1 byte long, $m = 8$, we see that 4 check bits are needed and the total word length is 12 bits. Thus we can say that the ratio c/m is a measure of the code overhead, which in this case is 50%. The overhead for common computer word lengths, m, is given in Table 2.8.

Clearly the overhead approaches 10% for long word lengths. Of course, one should remember that these codes are competing for efficiency with the parity-bit code, in which 1 check bit represents only a 1.6% overhead for a 64-bit word length.

We now return to our (7, 4) SECSED code example to explain how the check bits are generated. Hamming developed a much more ingenious and

TABLE 2.8 Overhead for Various Word Lengths (m) for a Hamming SECSED Code

Code Length, n	Word (Message) Length, m	Number of Check Bits, c	Overhead $(c/m) \times 100\%$
12	8	4	50
21	16	5	31
38	32	6	19
54	48	6	13
71	64	7	11

efficient design and method for detection and correction. The Hamming code positions for the check and message bits are given in Table 2.6, which yields the code word $c_1c_2b_1c_3b_2b_3b_4$. The check bits are calculated by computing the exclusive, or $\oplus$, of 3 appropriate message bits as shown in the following equations:

$$c_1 = b_1 \oplus b_2 \oplus b_4 \tag{2.22a}$$
$$c_2 = b_1 \oplus b_3 \oplus b_4 \tag{2.22b}$$
$$c_3 = b_2 \oplus b_3 \oplus b_4 \tag{2.22c}$$

Such a choice of check bits forms an obvious pattern if we write the 3 check equations below the word we are checking, as is shown in Table 2.9. Each parity bit and message bit present in Eqs. (2.22a–c) is indicated by a "1" in the respective rows (all other positions are 0). If we read down in each column, the last 3 bits are the binary number corresponding to the bit position in the word.

Clearly, the binary number pattern gives us a design procedure for constructing parity check equations for distance 3 codes of other word lengths. Reading across rows 3–5 of Table 2.9, we see that the check bit with a 1 is on the left side of the equation and all other bits appear as $\oplus$ on the right-hand side.

As an example, consider that the message bits $b_1b_2b_3b_4$ are 1010, in which case the check bits are

TABLE 2.9 Pattern of Parity Check Bits for a Hamming (7, 4) SECSED Code

Bit positions in word	x_1	x_2	x_3	x_4	x_5	x_6	x_7
Code word	c_1	c_2	b_1	c_3	b_2	b_3	b_4
Check bit c_1	1	0	1	0	1	0	1
Check bit c_2	0	1	1	0	0	1	1
Check bit c_3	0	0	0	1	1	1	1

$$c_1 = 1 \oplus 0 \oplus 0 = 1 \tag{2.23a}$$
$$c_2 = 1 \oplus 1 \oplus 0 = 0 \tag{2.23b}$$
$$c_3 = 0 \oplus 1 \oplus 0 = 1 \tag{2.23c}$$

and the code word is $c_1c_2b_1c_3b_2b_3b_4 = 1011010$.

To check the transmitted word, we recalculate the check bits using Eqs. (2.22a–c) and obtain c_1', c_2', and c_3'. The old and the new parity check bits are compared, and any disagreement indicates an error. Depending on which check bits disagree, we can determine which message bit is in error. Hamming devised an ingenious way to make this check, which we illustrate by example.

Suppose that bit 3 of the message we have been discussing changes from a "1" to a "0" because of a noise pulse. Our code word then becomes $c_1c_2b_1c_3b_2b_3b_4 = 1011000$. Then, application of Eqs. (2.22a–c) yields c_3', c_2', and $c_1' = 110$ for the new check bits. Disagreement of the check bits in the message with the newly calculated check bits indicates that an error has been detected. To locate the error, we calculate error-address bits, $e_3e_2e_1$, as follows:

$$e_1 = c_1 \oplus c_1' = 1 \oplus 1 = 0 \tag{2.24a}$$
$$e_2 = c_2 \oplus c_2' = 0 \oplus 1 = 1 \tag{2.24b}$$
$$e_3 = c_3 \oplus c_3' = 1 \oplus 0 = 1 \tag{2.24c}$$

The binary address of the error bit is given by $e_3e_2e_1$, which in our example is 110 or 6. Thus we have detected correctly that the sixth position, b_3, is in error. If the address of the error bit is 000, it indicates that no error has occurred; thus calculation of $e_3e_2e_1$ can serve as our means of error detection and correction. To correct a bit that is in error once we know its location, we replace the bit with its complement.

The generation and checking operations described above can be derived in terms of a parity code matrix (essentially the last three rows of Table 2.9), a column vector that is the coded word, and a row vector called the *syndrome*, which is $e_3e_2e_1$ that we called the binary address of the error bit. If no errors occur, the syndrome is zero. If a single error occurs, the syndrome gives the correct address of the erroneous bit. If a double error occurs, the syndrome is nonzero, indicating an error; however, the address of the erroneous bit is incorrect. In the case of triple errors, the syndrome is zero and the errors are not detected. For a further discussion of the matrix representation of Hamming codes, the reader is referred to Siewiorek [1992].

2.4.4 The Hamming SECDED Code

The SECDED code is a distance 4 code that can be viewed as a distance 3 code with one additional check bit. It can also be a triple error-detecting code (TED). It is easy to design such a code by first designing a SECSED code and

TABLE 2.10 Interpretation of Syndrome for a Hamming (8, 4) SECDED Code

e_1	e_2	e_3	e_4	Interpretation
0	0	0	0	No errors
a_1	a_2	a_3	1	One error, $a_1a_2a_3$
a_1	a_2	a_3	0	Two errors, $a_1a_2a_3$, not 000
0	0	0	1	Three errors
0	0	0	0	Four errors

then adding an appended check bit, which is a parity bit over all the other message and check bits. An even-parity code is traditionally used; however, if the digital electronics generating the code word have a failure mode in which the chip is burned out and all bits are 0, it will not be detected by an even-parity scheme. Thus odd parity is preferred for such a case. We expand on the (7, 4) SECSED example of the previous section and affix an additional check bit (c_4) and an additional syndrome bit (e_4) to obtain a SECDED code.

$$c_4 = c_1 \oplus c_2 \oplus b_1 \oplus c_3 \oplus b_2 \oplus b_3 \oplus b_4 \tag{2.25}$$

$$e_4 = c_4 \oplus c_4' \tag{2.26}$$

The new coded word is $c_1c_2b_1c_3b_2b_3b_4c_4$. The syndrome is interpreted as given in Table 2.10.

Table 2.8 can be modified for a SECDED code by adding 1 to the code length column and 1 to the check bits column. The overhead values become 63%, 38%, 22%, 15%, and 13%.

2.4.5 Reduction in Undetected Errors

The probability of an undetected error for a SECSED code depends on the error-correction philosophy. Either a nonzero syndrome can be viewed as a single error—and the error-correction circuitry is enabled—or it can be viewed as detection of a double error. Since the next section will treat uncorrected error probabilities, we assume in this section that the nonzero syndrome condition for a SECSED code means that we are detecting 1 or 2 errors. (Some people would call this simply a distance 3 double error-detecting, or DED, code.) In such a case, the error detection fails if 3 or more errors occur. We discuss these probability computations by using the example of a code for a 1-byte message, where $m = 8$ and $c = 4$ (see Table 2.8). If we assume that the dominant term in this computation is the probability of 3 errors, then we can see Eq. (2.1) and write

$$P_{ue}' = B(3:12) = 220q^3(1-q)^9 \tag{2.27}$$

TABLE 2.11 Evaluation of the Reduction in Undetected Errors for a Hamming SECSED Code: Eq. (2.25)

Bit Error Probability, q	Improvement Ratio: P_{ue}/P_{ue}'
10^{-4}	3.640×10^{6}
10^{-5}	3.637×10^{8}
10^{-6}	3.636×10^{10}
10^{-7}	3.636×10^{12}
10^{-8}	3.636×10^{14}

Following simplifications similar to those used to derive Eq. (2.7), the undetected error ratio becomes

$$P_{ue}/P_{ue}' = 2(1+9q)/55q^2 \tag{2.28}$$

This ratio is evaluated in Table 2.11.

2.4.6 Effect of Coder–Decoder Failures

Clearly, the error improvement ratios in Table 2.11 are much larger than those in Table 2.3. We now must include the probability of the generator/checker circuitry failing. This should be a more significant effect than in the case of the parity-bit code for two reasons. First, the undetected error probabilities are much smaller with the SECSED code, and second, the generator/checker will be more complex. A practical circuit for checking a (7, 4) SECSED code is given in Wakerly [p. 298, 1990] and is reproduced in Fig. 2.7. For the reader who is not experienced in digital circuitry, some explanation is in order. The three 74LS280 ICs (U_1, U_2, and U_3) are similar to the SN74180 shown in Fig. 2.4. Substituting Eq. (2.22a) into Eq. (2.24a) shows that the syndrome bit e_1 is dependent on the $\oplus$ of c_1, b_1, b_2, and b_4, and from Table 2.6 we see that these are bit positions x_1, x_3, x_5, and x_7, which correspond to the inputs to U_1. Similarly, U_2 and U_3 compute e_2 and e_3. The decoder U_4 (see Appendix C6.3) activates one of its 8 outputs, which is the address of the error bit. The 8 output gates (U_5 and U_6) are exclusive or gates (see Appendix C; only 7 are used). The output of the U_4 selects the erroneous bit from the bus DU(1–7), complements it (performing a correction), and passes through the other 6 bits unchanged. Actually the outputs DU(1–7) are all complements of the desired values; however, this is simply corrected by a group of inverters at the output or inversion of the next stage of digital logic. For a check-bit generator, we can use three 74LS280 chips to generate e_1, e_2, and e_3.

We can compute the reliability of the generator/checker circuitry by again using the IC failure rate model of Section B3.3, $\lambda_b = 0.004\sqrt{g}$. We assume

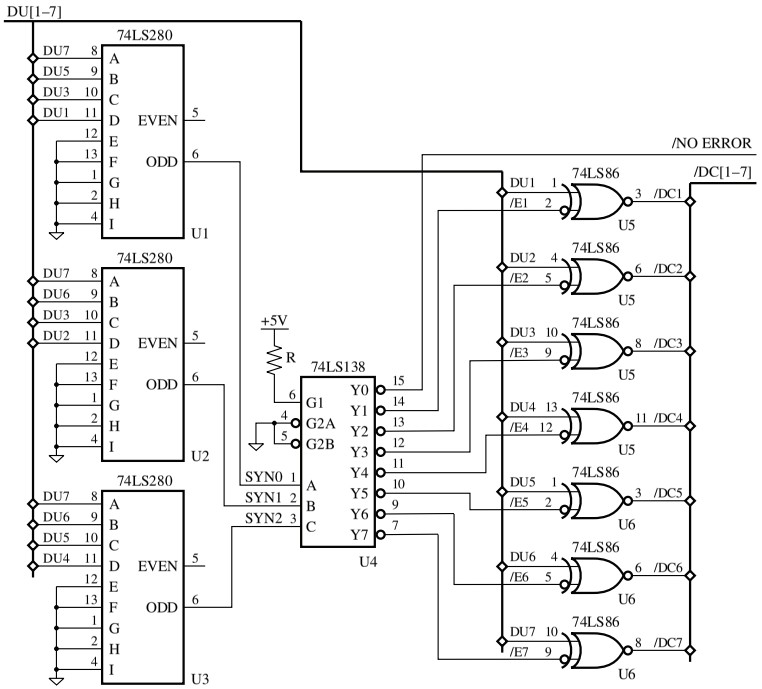


Figure 2.7 Error-correcting circuit for a Hamming (7, 4) SECSED code [Reprinted by permission of Pearson Education, Inc., Upper Saddle River, NJ 07458; from Wakerly, 2000, p. 298].

that any failure in the IC causes system failure, so the reliability diagram is a series structure and the failure rates add. The computation is detailed in Table 2.12. (See also Fig. 2.7.)

Thus the failure rate for the coder plus decoder is $\lambda = 13.58 \times 10^{-8}$, which is about four times as large as that for the parity bit case ($2 \times 1.67 \times 10^{-8}$) that was calculated previously.

We now incorporate the possibility of generator/checker failure and how it affects the error-correction performance in the same manner as we did with the parity-bit code in Eqs. (2.8)–(2.11). From Table 2.8 we see that a 1-byte (8-bit) message requires 4 check bits; thus the SECSED code is (12, 8). The example developed in Table 2.12 and Fig. 2.7 was for a (7, 4) code, but we can easily modify these results for the (12, 8) code we have chosen to discuss. First, let us consider the code generator. The 74LS280 chips are designed to generate parity check bits for up to an 8-bit word, so they still suffice; however, we now

TABLE 2.12 Computation of Failure Rates for a (7, 4) SECSED Hamming Generator/Checker Circuitry

IC	Function	Gates,[a] g	$\lambda_b = 0.004\sqrt{g} \times 10^{-6}$	Number in Circuit	Failure Rate/hr
74LS280	Parity-bit generator	17.5	1.67×10^{-8}	3 in generator	5.01×10^{-8}
74LS280	Parity-bit generator	17.5	1.67×10^{-8}	3 in checker	5.01×10^{-8}
74LS138	Decoder	16.0	1.60×10^{-8}	1 in checker	1.60×10^{-8}
74LS86	EXOR package	6.0	9.80×10^{-9}	2 in checker	1.96×10^{-8}
				Total	13.58×10^{-8}

[a] Using 1.5 gates for each EXOR and ENOR gate.

need to generate 4 check bits, so a total of 4 will be required. In the case of the checker (see Fig. 2.7), we will also require four 74LS280 chips to generate the y-syndrome bits. Instead of a 3-to-8 decoder we will need a 4-to-16 decoder for the next stage, which can be implemented by using two 74LS138 chips and the appropriate connections at the enable inputs (G1, G2A, and G2B), as explained in Appendix C6.3. The output stage composed of 74LS86 chips will not be required if we are only considering error detection, since the nonerror output is sufficient for this. Thus we can modify Table 2.12 to compute the failure rate that is shown in Table 2.13. Note that one could argue that since we are only computing the error-detection probabilities, the decoders and output correction EXOR gates are not needed, and only an OR gate with the syndrome inputs is needed to detect a 0000 syndrome that indicates no errors.

Using the information in Table 2.13 and Eq. (2.27), we obtain an expression similar to Eq. (2.10), as follows:

$$P'_{ue} = e^{-\lambda t}220q^3(1-q)^9 + (1 - e^{-\lambda t}) \tag{2.29}$$

where λ is 19.50×10^{-8} failures per hour and t is $12/3600B$.

We formulate the improvement ratio by dividing Eq. (2.29) by Eq. (2.12); the ratio is given in Table 2.14 and is plotted in Fig. 2.8. The data presented in Table 2.11 is also plotted in Fig. 2.8 and represents the line labeled $B \to \infty$, which represents the case for a nonfailing generator/checker.

2.4.7 How Coder–Decoder Failures Affect SECSED Codes

Because the Hamming SECSED code results in a lower value for undetected errors than the parity-bit code, the effect of chip failures is even more pronounced. Of course the coding is still a big improvement, but not as much as one would predict. In fact, by comparing Figs. 2.8 and 2.5 we see that for B =300, the parity-bit scheme is superior to the SECSED scheme for values of q less than about 2×10^{-7}; for B =1,200, the parity-bit scheme is superior to the SECSED scheme for values of q less than about 10^{-7}. The general conclusion is that for more complex error detection schemes, one should evaluate the effects of generator/checker failures, since these may be of considerable importance for small values of q. (Chip-specific failure rates may be required.)

More generally, we should compute whether generator/checker failures significantly affect the code performance for the given values of q and B. If such failures are significant, we can consider the following alternatives:

1. Consider a simpler coding scheme if q is very small and B is low.
2. Consider other coding schemes if they use simpler generator/checker circutry.
3. Use other digital logic designs that utilize fewer but larger chips. Since the failure rate is proportional to $\sqrt{g}$, larger-scale integration improves reliability.

TABLE 2.13 Computation of Failure Rates for a (12, 8) DED Hamming Generator/Checker Circuitry

IC	Function	Gates,[a] g	$\lambda_b = 0.004\sqrt{g} \times 10^{-6}$	Number in Circuit	Failure Rate/hr
74LS280	Parity-bit generator	17.5	1.67×10^{-8}	4 in generator	6.68×10^{-8}
74LS280	Parity-bit generator	17.5	1.67×10^{-8}	4 in checker	6.68×10^{-8}
74LS138	Decoder	16.0	1.60×10^{-8}	2 in checker	3.20×10^{-8}
74LS86	EXOR package	6.0	0.98×10^{-8}	3 in checker	2.94×10^{-8}
				Total	19.50×10^{-8}

[a] Using 1.5 gates for each EXOR and ENOR gate.

TABLE 2.14 The Reduction in Undetected Errors from a Hamming (12, 8) DED Code Including the Effect of Coder–Decoder Failures

Bit Error Probability q	Improvement Ratio: P_{ue}/P'_{ue} for Several Transmission Rates			
	300 Bits/Sec	1,200 Bits/Sec	9,600 Bits/Sec	56,000 Bits/Sec
10^{-4}	3.608×10^6	3.629×10^6	3.637×10^6	3.638×10^6
10^{-5}	3.88×10^7	1.176×10^8	2.883×10^8	3.480×10^8
10^{-6}	4.34×10^6	1.738×10^7	1.386×10^8	7.939×10^8
10^{-7}	4.35×10^5	1.739×10^6	1.391×10^7	8.116×10^7
10^{-8}	4.35×10^4	1.739×10^5	1.391×10^6	8.116×10^6

4. Seek to lower IC failure rates via improved derating, burn-in, use of high reliability ICs, and so forth.
5. Seek fault-tolerant or redundant schemes for code generator and code checker circuitry.

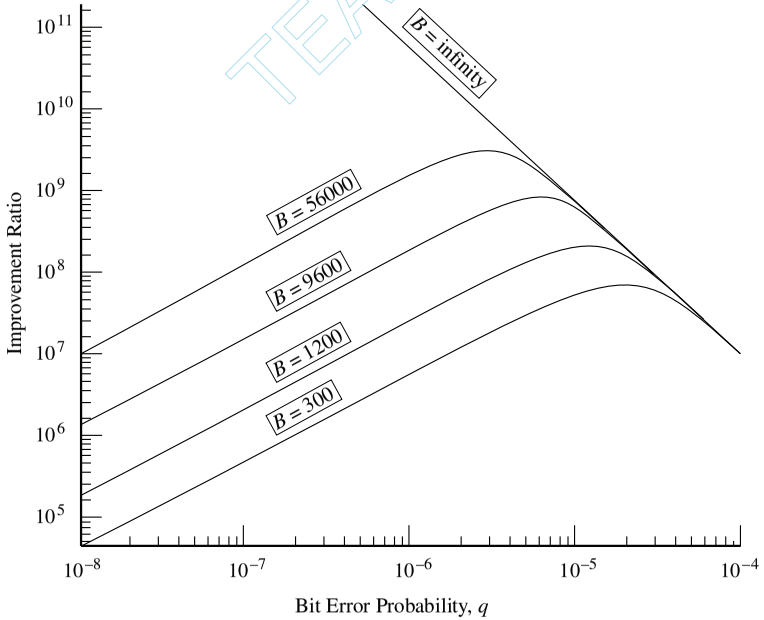


Figure 2.8 Improvement ratio of undetected error probability from a SECSED code, including the possibility of coder–decoder failure. B is the transmission rate in bits per second.

2.5 ERROR-DETECTION AND RETRANSMISSION CODES

2.5.1 Introduction

We have discussed both error detection and correction in the previous sections of this chapter. However, performance metrics (the probabilities of undetected errors) have been discussed only for error detection. In this section, we introduce metrics for evaluating the error-correction performance of various codes. In discussing the applications for parity and Hamming codes, we have focused on information transmission as a typical application. Clearly, the implementations and metrics we have developed apply equally well to memory scheme protection, cache checking, bus-transmission checks, and so forth. Thus, when we again use a data-transmission data application to discuss error correction, the results will also apply to the other application.

The Hamming error-correcting codes provide a direct means of error correction; however, if our transmission channel allows communication in both directions (bidirectional), there is another possibility. If we detect an error, we can send control signals back to the source to ask for retransmission of the erroneous byte, work, or code block. In general, the appropriate measure of error correction is the reliability (probability of no error).

2.5.2 Reliability of a SECSED Code

To discuss the reliability of transmission, we again focus on 1 transmitted byte and compute the reliability with and without error correction. The reliability of a single transmitted byte without any error correction is just the probability of no errors occurring, which was calculated as the second term in Eq. (2.5).

$$R = (1 - q)^8 \tag{2.30}$$

In the case of a SECSED code (12, 8), single errors are corrected; thus the reliability is given by

$$R = P(\text{no errors} + 1 \text{ error}) \tag{2.31}$$

and since these are mutually exclusive events,

$$R = P(\text{no errors}) + P(1 \text{ error}) \tag{2.32}$$

the binomial distribution yields

$$R' = (1 - q)^{12} + 12q(1 - q)^{11} = (1 - q)^{11}(1 + 11q) \tag{2.33}$$

Clearly, $R' \geq R$; however, for small values of q, both are very close to 1, and it is easier to compare the unreliability $U = 1 - R$. Thus a measure of the improvement of a SECSED code is given by

TABLE 2.15 Evaluation of the Reduction in Unreliability for a Hamming SECSED Code: Eq. (2.35)

Bit Error Probability, q	Improvement Ratio: $\frac{1-U}{1-U'}$
10^{-4}	6.61×10^2
10^{-5}	6.61×10^3
10^{-6}	6.61×10^4
10^{-7}	6.61×10^5
10^{-8}	6.61×10^6

$$(1-U)/(1-U') = [1-(1-q)^8]/[1-(1-q)^{11}(1+11q)] \tag{2.34}$$

and approximating this for small q yields

$$(1-U)/(1-U') = 8/121q \tag{2.35}$$

which is evaluated for typical values of q in Table 2.15.

The foregoing evaluations neglected the probability of IC generator and checker failure. However, the analysis can be broadened to include these effects as was done in the preceding sections.

2.5.3 Reliability of a Retransmitted Code

If it is possible to retransmit a code block after an error has been detected, one can improve the reliability of the transmission. In such a case, the reliability expression becomes

$$R' = P(\text{no error} + \text{detected error and no error on retransmisson}) \tag{2.36}$$

and since these are mutually exclusive events and independent events,

$$R' = P(\text{no error}) + P(\text{detected error}) \times P(\text{no error on retransmission}) \tag{2.37}$$

Since the error probabilities on initial transmission and on retransmission are the same, we obtain

$$R' = P(\text{no error})[1 + P(\text{detected error})] \tag{2.38}$$

For the case of a parity-bit code, we transmit 9 bits; the probability of detecting an error is approximately the probability of 1 error. Substitution in Eq. (2.38) yields

$$R' = (1-q)^9[1 + 9q(1-q)^8] \tag{2.39}$$

Comparing the ratio of unreliabilities yields

$$(1-U)/(1-U') = [1-(1-q)^8]/[1-[(1-q)^9[1+9q(1-q)^8]]] \tag{2.40}$$

and simplification for small q yields

$$(1-U)/(1-U') = 8q/[9q^2 - 828q^3] \tag{2.41}$$

Similarly, we can use a Hamming distance 3 code (12, 8) to detect up to 2 errors and retransmit. In this case, the probability of detecting an error is approximately the probability of 1 or 2 errors. Substitution in Eq. (2.38) yields

$$R' = (1-q)^{12}[1 + (12q(1-q)^{11} + 66q^2(1-q)^{10})] \tag{2.42}$$

and the unreliability ratio becomes

$$(1-U)/(1-U') = [1-(1-q)^8]/[1-[(1-q)^{12}[1+(12q(1-q)^{11} + 66q^2(1-q)^{10}]]] \tag{2.43}$$

and simplification for small q yields

$$(1-U)/(1-U') = 8q/[78q^2 - 66q^3] \tag{2.44}$$

Equations (2.41) and (2.44) are evaluated in Table 2.16 for typical values of q. Comparison of Tables 2.15 and 2.16 shows that both retransmit schemes are superior to the error correction of a SECSED code, and that the parity-bit retransmit scheme is the best. However, retransmit has at least a 100% overhead penalty, and Table 2.8 shows typical SECSED overheads of 11–50%.

TABLE 2.16 Evaluation of the Improvement in Reliability by Code Retransmission for Parity and Hamming $d = 3$ Code

Bit Error Probability, q	Parity-Bit Retransmission $(1-U)/(1-U')$: Eq. (2.41)	Hamming $d = 3$ Retransmission $(1-U)/(1-U')$: Eq. (2.44)
10^{-4}	8.97×10^3	1.026×10^3
10^{-5}	8.90×10^4	1.026×10^4
10^{-6}	8.89×10^5	1.026×10^5
10^{-7}	8.89×10^6	1.026×10^6
10^{-8}	8.89×10^7	1.026×10^7

The foregoing evaluations neglected the probability of IC generator and checker failure as well as the circuitry involved in controlling retransmission. However, the analysis can be broadened to include these effects, and a more detailed comparison can be made.

2.6 BURST ERROR-CORRECTION CODES

2.6.1 Introduction

The codes previously discussed have all been based on the assumption that the probability that bit b_i is corrupted by an error is largely independent of whether bit b_{i-1} is correct or is in error. Furthermore, the probability of a single bit error, q, is relatively small; thus the probability of more than one error in a word is quite small. In the case of a burst error, the probability that bit b_i is corrupted by an error is much larger if bit b_{i-1} is incorrect than if bit b_{i-1} is correct. In other words, the errors commonly come in bursts rather than singly. One class of applications that are subject to burst errors are rotational magnetic and optical storage devices (e.g., music CDs, CD-ROMs, and hard and floppy disk drives). Magnetic tape used for pictures, sound, or data is also affected by burst errors.

Examples of the patterns of typical burst errors are given in the four 12-bit messages (m_1–m_4) shown in the forthcoming equations. The common notation is used where b represents a correct message bit and x represents an erroneous message bit. (For the purpose of identification, assume that the bits are numbered 1–12 from left to right.)

$$m_1 = bbbxxbxbbbbb \tag{2.45a}$$
$$m_2 = bxbxxbbbbbbb \tag{2.45b}$$
$$m_3 = bbbbxbxbbbbb \tag{2.45c}$$
$$m_4 = bxxbbbbbbbbb \tag{2.45d}$$

Messages 1 and 2 each have 3 errors that extend over 4 bits (e.g., in m_1 the error bits are in positions 4, 5, and 7); we would refer to them as bursts of length 4. In message 3, the burst is of length 3; in message 4, the burst is of length 2. In general, we call the burst length t. The burst length is really a matter of definition; for example, one could interpret messages 1 and 2 as 2 bursts—one of length 1 and one of length 2. In practice, this causes no confusion, for t is a parameter of a burst code and is fixed in the initial design of the code. Thus if t is chosen as length 4, all 4 of the messages would have 1 burst. If t is chosen as length 3, messages 1 and 2 would have two bursts, and messages 3 and 4 would have 1 burst.

Most burst error codes are more complex than the Hamming codes that were just discussed; thus the remainder of this chapter will present a succinct introduction to the basis of such codes and will briefly introduce one of the most popular burst codes: the Reed–Solomon code [Golumb, 1986].

2.6.2 Error Detection

We begin by giving an example of a burst error-detection code [Arazi, 1988]. Consider a 12-bit-long code word (also called a *code block* or *code vector*, V), which includes both message and check bits as follows:

$$V = (x_1x_2x_3x_4x_5x_6x_7x_8x_9x_{10}x_{11}x_{12}) \tag{2.46}$$

Let us choose to deal with bursts of length $t = 4$. Equations for calculating the check bits in terms of the message bits can be developed by writing a set of equations in which the bits are separated by t positions. Thus for $t = 4$, each equation contains every fourth bit.

$$x_1 \oplus x_5 \oplus x_9 = 0 \tag{2.47a}$$
$$x_2 \oplus x_6 \oplus x_{10} = 0 \tag{2.47b}$$
$$x_3 \oplus x_7 \oplus x_{11} = 0 \tag{2.47c}$$
$$x_4 \oplus x_8 \oplus x_{12} = 0 \tag{2.47d}$$

Each bit appears in only one equation. Assume there is either 0 or only 1 burst in the code vector (multiple bursts in a single word are excluded). Thus each time there is 1 erroneous bit, one of the four equations will equal 1 rather than 0, indicating a single error. To illustrate this, suppose x_2 is an error bit. Since we are assuming a burst length of 4 and at most 1 burst per code vector, the only other *possible* erroneous bits are x_3, x_4, and x_5. (At this point, we don't know if 0, 1, 2, or 3 errors occur in bits 3–5.) Examining Eq. (2.47b), we see that it is not possible for x_6 or x_{10} to be erroneous bits, so it is not possible for 2 errors to cancel out in evaluating Eq. (2.47b). In fact, if we analyze the set of Eqs. (2.47a–d), we see that the number of nonzero equations in the set is equal to the number of bit errors in the burst.

Since there are 4 check equations, we need 4 check bits; any set of 4 bits in the vector can be chosen as check bits, provided that 1 bit is chosen from each equation (2.47a–d). For clarity, it probably makes sense to choose the 4 check bits as the first or last 4 bits in the vector; such a choice in any type of code is referred to as a *systematic code*. Suppose we choose the first 4 bits. We then obtain a (12, 8) systematic burst code of length 4, where c_i stands for a check bit and b_i a message bit.

$$V = (c_1c_2c_3c_4b_1b_2b_3b_4b_5b_6b_7b_8) \tag{2.48}$$

A moment's reflection shows that we have now maneuvered Eqs. (2.47a–d) so that with cs and bs substituted for the xs, we obtain

$$c_1 \oplus b_1 \oplus b_5 = 0 \tag{2.49a}$$
$$c_2 \oplus b_2 \oplus b_6 = 0 \tag{2.49b}$$
$$c_3 \oplus b_3 \oplus b_7 = 0 \tag{2.49c}$$
$$c_4 \oplus b_4 \oplus b_8 = 0 \tag{2.49d}$$

which can be used to compute the check bits. These equations are therefore the basis of the check-bit generator, which can be done with 74180 or 74280 IC chips.

The same set of equations form the basis of the error-checking circuitry. Based on the fact that the number of nonzero equations in the set of Eqs. (2.47a–d) is equal to the number of bit errors in the burst, we can modify Eqs. (2.47a–c) so that they explicitly yield bits of a syndrome vector, $e_1e_2e_3e_4$.

$$e_1 = x_1 \oplus x_5 \oplus x_9 \tag{2.50a}$$
$$e_2 = x_2 \oplus x_6 \oplus x_{10} \tag{2.50b}$$
$$e_3 = x_3 \oplus x_7 \oplus x_{11} \tag{2.50c}$$
$$e_4 = x_4 \oplus x_8 \oplus x_{12} \tag{2.50d}$$

The nonerror condition occurs when all the syndrome bits are 0. In general, the number of errors detected is the arithmetic sum: $e_1 + e_2 + e_3 + e_4$. Note that because we originally chose $t = 4$ in this design, no more than 4 errors can be detected. Again, the checker can be done with 74180 or 74280 IC chips. Alternatively, one can use individual gates. To generate the check bits, 4 EXOR gates are sufficient; 8 EXOR gates and an output OR gate are sufficient for error checking (cf. Fig. 2.2). However, if one wishes to determine how many errors have occurred, the output OR gate in the checker can be replaced by a few half-adders or full-adders to compute the arithmetic sum: $e_1 + e_2 + e_3 + e_4$.

We can now state some properties of burst codes that were illustrated by the above discussion. The reader is referred to the references for proof [Arazi, 1988].

Properties of Burst Codes

1. For a burst length of t, t check bits are needed for error detection. (Note: this is independent of the message length m.)
2. For m message bits and a burst length of t, the code word length $n = m + t$.
3. There are t check-bit equations:
   (a) The first check-bit equation starts with bit 1 and contains all the bits that are $t + 1$, $2t + 1$, … $kt + 1$ (where $kt + 1 \leq n$).
   (b) The second check-bit equation starts with bit 2 and contains all the bits that are $t + 2$, $2t + 2$, … $kt + 2$ (where $kt + 2 \leq n$).
   . . . . . .
   (t) The t'th check-bit equation starts with bit t and contains all the bits that are $2t$, $3t$, … kt (where $kt \leq n$).

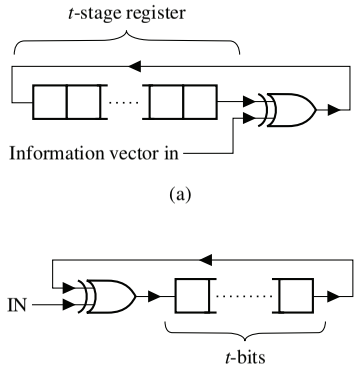


Figure 2.9 Burst error-detection circuitry using an LFSR: (a) encoder; (b) decoder. [Reprinted by permission of MIT Press, Cambridge, MA 02142; from Arazi, 1988, p. 108.]

4. The EXOR of all the bits in $3a$ should = 0 and similarly for properties $3b$, … $3t$.
5. The word length n need not be an integer multiple of t, but for practicality, we assume that it is. If necessary, the word can be padded with additional dummy bits to achieve this.
6. Generation and checking for a burst error code (as well as other codes) can be realized by a linear feedback shift register (LFSR). (See Fig. 2.9.)
7. In general, the LFSR has a delay of $t \times$ the shift time.
8. The generating and checking for a burst error code can be realized by an EXOR tree circuit (cf. Fig. 2.2), in which the number of stages is $\leq \log_2(t)$ and the delay is $\leq \log_2(t) \times$ the EXOR gate-switching time.

These properties are explored further in the problems at the end of this chapter. To summarize, in this section we have developed the basic equations for burst error-detection codes and have shown that the check-bit generator and checker circuitry can be implemented with EXOR trees, parity-bit chips, or LFSRs. In general, the LFSR implementation requires less hardware, but the delay time is linear in the burst length t. In the case of EXOR trees, there is more hardware needed; however, the time delay is less, for it increases proportionally to the log of t. In either case, for the modest size $t = 4$ or 5, the differences in time delay and hardware are not that significant. Both designs should be attempted, and a choice should be made.

The case of burst error *correction* is more difficult. It is discussed in the next section.

2.6.3 Error Correction

We now state some additional properties of burst codes that will lead us to an error-correction procedure. In general, these are properties associated with a shifting of the error syndrome of a burst code and an ancient theorem of number theory related to the mod function. The theorem from number theory is called the *Chinese Remainder Theorem* [Rosen, 1991, p. 134] and was first given as a puzzle by the first-century Chinese mathematician Sun-Tsu. It will turn out that the method of error correction will depend on first locating a region in the code word of t consecutive bits that contains the start of the error burst, followed by pinpointing which of these t bits is the start of the burst. The methodology is illustrated by applying the principles to the example given in Eq. (2.46). For a development of the theory and proofs, the reader is referred to Arazi [1988] and Rosen [1991].

The error syndrome can be viewed as a cyclic shift of the burst error pattern. For example, if we assume a single burst and $t = 4$, then substitution of error pattern for $x_1x_2x_3x_4$ into Eqs. (2.50a–d) will yield a particular syndrome pattern. To compute what the syndrome would be, we note that if $x_1x_2x_3x_4$ = **bbbb**, all the bits are correct and the syndrome must be 0000. If bit 1 is in error (either changed from a correct 1 to an erroneous 0 or from a correct 0 to an erroneous 1), then Eq. (4.50a) will yield a 1 for e_1 (since there is only 1 burst, bits x_5–x_{12} must be all valid bs). Suppose the error pattern is $x_1x_2x_3x_4$ = **xbxx**, then all other bits in the 12-bit vector are b and substitution into Eqs. (2.50a–d) yields

$$e_1 = \mathbf{x} \oplus x_5 \oplus x_9 = 1 \tag{2.51a}$$
$$e_2 = \mathbf{b} \oplus x_6 \oplus x_{10} = 0 \tag{2.51b}$$
$$e_3 = \mathbf{x} \oplus x_7 \oplus x_{11} = 1 \tag{2.51c}$$
$$e_4 = \mathbf{x} \oplus x_8 \oplus x_{12} = 1 \tag{2.51d}$$

which is a syndrome pattern $e_1e_2e_3e_4$ = 1011. Similarly, error pattern $x_4x_5x_6x_7$ = **xbxx**, where all other bits are b, yields syndrome equations as follows:

$$e_1 = x_1 \oplus \mathbf{b} \oplus x_9 = 0 \tag{2.52a}$$
$$e_2 = x_2 \oplus \mathbf{x} \oplus x_{10} = 1 \tag{2.52b}$$
$$e_3 = x_3 \oplus \mathbf{x} \oplus x_{11} = 1 \tag{2.52c}$$
$$e_4 = \mathbf{x} \oplus x_8 \oplus x_{12} = 1 \tag{2.52d}$$

which is a syndrome pattern $e_1e_2e_3e_4$ = 0111. We can view 0111 as a pattern that can be transformed into 1011 by cyclic-shifting *left* (end-around-rotation left) three times. We will show in the following material that the same syndrome is obtained by shifting the code vector *right* four times.

We begin marking a burst error pattern with the first erroneous bit in the word; thus burst error patterns always start with an **x**. Since the burst is t bits long, the syndrome equations (2.50a–d) include bits that differ by t positions. Therefore, if we shift the burst error pattern in the code vector by t positions to the right, the burst error pattern generates the same syndrome. There can be at most u placements of the burst pattern in a code vector that results in the same syndrome; if the code vector is n bits long, u is the largest integer such that $ut \le n$. Without loss of generality, we can always pad the message bits with dummy bits such that $ut = n$. We define the *mod* function x mod y as the remainder that is obtained when we divide the integer x by the integer y. Thus, if $ut = n$, we can then say that n mod $u = 0$. These relationships will soon be used to devise an algorithm for burst error correction.

The location of the start of the burst error pattern in a word is related to the amount of shift (end-around and cyclic) of the pattern that is observed in the syndrome. We can illustrate this relationship by using the burst pattern **xbxx** as an example, where **xbxx** is denoted by 1011: meaning incorrect, correct, incorrect, incorrect. In Table 2.17, we illustrate the relationship between the start of the error burst and the rotational shift (end-around shift) in the detected error syndrome. We begin by renumbering the code vector, Eq. (2.46), so it starts with bit 0:

$$V = (x_0x_1x_2x_3x_4x_5x_6x_7x_8x_9x_{10}x_{11}) \tag{2.53}$$

A study of Table 2.17 shows that the number of syndrome shifts is related to the bit number by (bit number) mod 4. For example, if the burst starts with bit no. 3, we have 3 mod 4 (which is 3), so the syndrome is the error pattern shifted 3 places to the right. If we want to recover the syndrome, we shift 3 places to the left. In the case of a burst starting with bit no. 4, 4 mod 4 is 0, so the syndrome pattern and the burst pattern agree.

Thus, if we know the position in the code word at which the burst starts (defined as x), and if the burst length is t, then we can obtain the burst pattern by shifting the syndrome x mod t places to the left. Knowing the starting position of the burst (x) and the burst pattern, we can correct any erroneous bits. Thus our task is now to find x.

The procedure for solving for x depends on the Chinese Remainder Theorem, a previously mentioned mathematical theorem in number theory. This theorem states that if p and q are relatively prime numbers (meaning their only common factor is 1), and if $0 \le x \le (pq - 1)$, then knowledge of x mod p and x mod q allows us to solve for x. We already have one equation: x mod t; to generate another equation, we define $u = 2t - 1$ and calculate x from x mod u [Arazi, 1988]. Note that t and $2t - 1$ are relatively prime since if a number divides t, it also divides $2t$ but not $2t - 1$. Also, we must show that $0 \le x \le (tu - 1)$; however, we already showed that $tu \le n$. Substitution yields $0 \le x \le (n - 1)$, which must be true since the latest bit position to start a burst error (x) for a burst of length t is $n - t < n - 1$.

The above relationships show that it is possible to solve for the beginning

TABLE 2.17 Relationship Between Start of the Error Burst and the Syndrome for the 12-Bit-Long Code Given in Eq. (2.49)

Burst Start Bit No.	Code Vector Positions												Error Syndrome				To Recover Syndrome Shift
	0	1	2	3	4	5	6	7	8	9	10	11	e_1	e_2	e_3	e_4	
0	x	b	x	x	b	b	b	b	b	b	b	b	1	0	1	1	0
1	b	x	b	x	x	b	b	b	b	b	b	b	1	1	0	1	1 left
2	b	b	x	b	x	x	b	b	b	b	b	b	1	1	1	0	2 left
3	b	b	b	x	b	x	x	b	b	b	b	b	0	1	1	1	3 left
4	b	b	b	b	x	b	x	x	b	b	b	b	1	0	1	1	0
5	b	b	b	b	b	x	b	x	x	b	b	b	1	1	0	1	1 left
6	b	b	b	b	b	b	x	b	x	x	b	b	1	1	1	0	2 left
7	b	b	b	b	b	b	b	x	b	x	x	b	0	1	1	1	3 left
8	b	b	b	b	b	b	b	b	x	b	x	x	1	0	1	1	0

of the burst error x and the burst error pattern. Given this information, by simply complementing the incorrect bits, error correction is performed. The remainder of this section details how we set up equations to calculate the check bits (generator) and to calculate the burst pattern and location (checker); this is done by means of an illustrative example. One circuit implementation using shift registers is discussed as well.

The number of check bits is equal to $u + t$, and since $u = 2c - 1$ and $n = ut$, the number of message bits is determined. We formulate check bit equations in a manner analogous to that used in error checking.

The following example illustrates how the two sets of check bits are generated, how one formulates and solves for x mod u and x mod t to solve for x, and how the burst error pattern is determined. In our example, we let $t = 3$ and calculate $u = 2t - 1 = 2 \times 3 - 1 = 5$. In this case, the word length $n = u \times t = 5 \times 3 = 15$. The code vector is given by

$$V = (x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13} x_{14}) \tag{2.54}$$

The $t + u$ check equations are generated from a set of u equations that form the auxiliary syndrome. For our example, the $u = 5$ auxiliary syndrome equations are:

$$s_0 = x_0 \oplus x_5 \oplus x_{10} \tag{2.55a}$$
$$s_1 = x_1 \oplus x_6 \oplus x_{11} \tag{2.55b}$$
$$s_2 = x_2 \oplus x_7 \oplus x_{12} \tag{2.55c}$$
$$s_3 = x_3 \oplus x_8 \oplus x_{13} \tag{2.55d}$$
$$s_4 = x_4 \oplus x_9 \oplus x_{14} \tag{2.55e}$$

and the set of $t = 3$ equations that form the syndrome are

$$e_1 = x_0 \oplus x_3 \oplus x_6 \oplus x_9 \oplus x_{12} \tag{2.56a}$$
$$e_2 = x_1 \oplus x_4 \oplus x_7 \oplus x_{10} \oplus x_{13} \tag{2.56b}$$
$$e_3 = x_2 \oplus x_5 \oplus x_8 \oplus x_{11} \oplus x_{14} \tag{2.56c}$$

If we want a systematic code, we can place the 8 check bits at the beginning or the end of the word. Let us assume that they go at the end (x_7–x_{14}) and that these check bits c_0–c_7 are calculated from Eqs. (2.55a–e) and (2.56a–c). The first 7 bits (x_0–x_6) are message bits, and the transmitted word is

$$V = (b_0 b_1 b_2 b_3 b_4 b_5 b_6 c_0 c_1 c_2 c_3 c_4 c_5 c_6 c_7) \tag{2.57}$$

As an example, let us assume that the message bits b_0–b_6 are 1011010. Substitution of these values in Eqs. (2.55a–e) and (2.56a–c) *that must initially be 0* yields a set of equations that can be solved for the values of c_0–c_7. One can show by substitution that the values c_0–c_7 = 10000010 satisfy the equations.

(Shortly, we will describe code generation circuitry that can solve for the check bits in a straightforward manner.) Thus the transmitted word is

$$V_t = (b_0 b_1 b_2 b_3 b_4 b_5 b_6) = 1011010 \quad \text{for the message part} \quad (2.58a)$$

$$V_t = (c_0 c_1 c_2 c_3 c_4 c_5 c_6 c_7) = 10000010 \quad \text{for the check part} \quad (2.58b)$$

Let us assume that the received word is

$$V_r = (101101000100010) \quad (2.59)$$

We now begin the error-recovery procedure by calculating the auxiliary syndrome by substitution of Eq. (2.59) in Eqs. (2.55a–e) yielding

$$s_0 = 1 \oplus 1 \oplus 0 = 0 \quad (2.60a)$$

$$s_1 = 0 \oplus 0 \oplus 0 = 0 \quad (2.60b)$$

$$s_2 = 1 \oplus 0 \oplus 1 = 1 \quad (2.60c)$$

$$s_3 = 1 \oplus 0 \oplus 0 = 0 \quad (2.60d)$$

$$s_4 = 0 \oplus 1 \oplus 0 = 1 \quad (2.60e)$$

The fact that the auxiliary syndrome is not all 0's indicates that 1 or more errors have occurred. In fact, since two equations are nonzero, there are two errors. Furthermore, it can be shown that the burst error pattern associated with the auxiliary syndrome must always start with an x and all bits $> t$ must be valid bits. Thus, the burst error pattern (since $t = 3$) must be $x??bb = 1??00$. This means the auxiliary syndrome pattern should start with a 1 and end in two 0's. The unique solution is that the auxiliary syndrome pattern must be shifted to the left two places yielding 10100 so that the first bit is 1 and the last two bits are 0. In addition, we deduce that the real syndrome (and the burst pattern) is 101. Similarly, Eqs. (2.56a–c) yield

$$e_0 = 1 \oplus 1 \oplus 0 \oplus 1 \oplus 0 = 1 \quad (2.61a)$$

$$e_1 = 0 \oplus 0 \oplus 0 \oplus 0 \oplus 1 = 1 \quad (2.61b)$$

$$e_2 = 1 \oplus 1 \oplus 0 \oplus 0 \oplus 0 = 0 \quad (2.61c)$$

Thus, to get the known syndrome—found from Eqs. (2.61a–c)—to be 101, we must shift the real syndrome left one place. Based on these shift results, our two mod equations become

$$\text{for } u: \quad x \bmod u = x \bmod 5 = 2 \quad (2.62a)$$

$$\text{for } t: \quad x \bmod t = x \bmod 3 = 1 \quad (2.62b)$$

We now know the burst pattern 101 and have two equations (2.62a, b) that can be solved for the start of the burst pattern given by x. Substitution of trial values into Eq. (2.62a) yields $x = 2$, which satisfies (2.62a) but not (2.62b). The

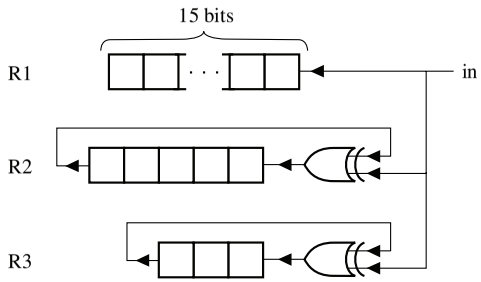


Figure 2.10 Basic error decoder for $u = 5$ and $t = 3$ burst code based on three shift registers. (Additional circuitry is needed for a complete decoder.) The input (IN) is a train of shift pulses. [Reprinted by permission of MIT Press, Cambridge, MA 02142; from Arazi, 1988, p. 123.]

next value that satisfies Eq. (2.62a) is $x = 7$, and since this value also satisfies Eq. (2.62b), it is a solution. We conclude that the burst error started at position $x = 7$ (the eighth bit, since the count starts with 0) and that is was xbx, so the eighth and tenth bits must be complemented. Thus the received and corrected versions of the code vector are

$$V_r = (101101000100010) \quad (2.63a)$$

$$\Updownarrow$$

$$V_c = (101101010000010) \quad (2.63b)$$

Note that Eqs. (2.63a, b) agrees with Eqs. (2.58a, b).

One practical decoder implementation for the $u = 5$ and $t = 3$ code discussed above is based on three shift registers (R1, R2, and R3) shown in Fig. 2.10. Such a circuit is said to employ linear feedback shift registers (LFSR).

Initially, R1 is loaded with the received code vector, R2 is loaded with the auxiliary syndrome calculated from EXOR trees or parity-bit chips that implement Eqs. (2.60a–e), and R3 is loaded with the syndrome calculated from EXOR trees or parity-bit chips that implement Eqs. (2.61a–c). Using our previous example, R1 is loaded with Eqs. (2.58a, b), R2 with 00101, and R3 with 110. R2 and R3 are shifted left until the left 3 bits of R2 agree with R3, and the leftmost bit is a 1. A count of the number of left shifts yields the start position of the burst error (x), and the contents of R3 is the burst pattern. Circuitry to complement the appropriate bits results in error correction. In the circuit shown, when the error pattern is recovered in R3, R1 has the burst error in the left 3 bits of the register. If correction is to be performed by shifting, the leftmost 3 bits in R1 and R3 can be EXORed and restored in R1. This would assume that the bits shifted out of R1 go to a storage register or are circulated back to R1 and, after error detection, the bits in the repaired word are shifted to their proper position. For more details, see Arazi [1988].

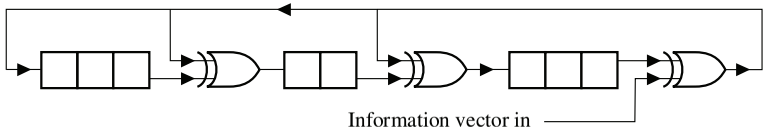


Figure 2.11 Basic encoder circuit for $u = 5$ and $t = 3$ burst code based on three shift registers. (Additional circuitry is needed for a complete decoder.) The input (IN) is the information vector (message). [Reprinted by permission of MIT Press, Cambridge, MA 02142; from Arazi, 1988, p. 125.]

One can also generate the check bits (encoder) by using LFSRs. One such circuit for our code example is given in Fig. 2.11. For more details, see Arazi [1988].

2.7 REED–SOLOMON CODES

2.7.1 Introduction

One technique to mitigate against burst errors is to simply interleave data so that a burst does not affect more than a few consecutive data bits at a time. A more efficient approach is to use codes that are designed to detect and correct burst errors. One of the most popular types of error-correcting codes is the Reed–Solomon (RS) code. This code is useful for correcting both random and burst errors, but it is especially popular in burst error situations and is often used with other codes in a convolutional code (see Section 2.8).

2.7.2 Block Structure

The RS code is a block-type code and operates on multiple rather than individual bits. Data is processed in a batch called a block instead of continuously. Each block is composed of n symbols, each of which has m bits. The block length $n = 2^m - 1$ symbols. A message is k symbols long, and n–k additional check symbols are added to allow error correction of up to t error symbols. Block length and symbol sizes can be adjusted to accommodate a wide range of message sizes. For an RS code, one can show that

$$(n - k) = 2t \qquad \text{for } n\text{–}k \text{ even} \tag{2.64a}$$

$$(n - k) = 2t + 1 \qquad \text{for } n\text{–}k \text{ odd} \tag{2.64b}$$

$$\text{minimum distance} = d_{\min} = 2t + 1 \text{ symbols} \tag{2.64c}$$

As a typical example [AHA Applications Note], we will assume $n = 255$ and $m = 8$ (a symbol is 1 byte long). Thus from Eq. (2.64a), if we wish to correct up to 10 errors, then $t = 10$ and $(n - k) = 20$. We therefore have 235 message symbols and 20 check symbols. The code rate (efficiency) of the code is given by k/n, which is $(235/255) = 0.92$ or 92%.

2.7.3 Interleaving

Interleaving is a technique that can be used with RS and other block codes to improve performance. Individual bits are shifted to spread them over several code blocks. The effect is to spread out long bursts so that error correction can occur even for code bursts that are longer than t bits. After the message is received, the bits are deinterleaved.

2.7.4 Improvement from the RS Code

We can calculate the improvement from the RS code in a manner similar to that which was used in the Hamming code. Now, the P_{ue} is the probability of an undetected error in a code block and P_{se} is the probability of a symbol error. Since the code can correct up to t errors, the block error probability is that of having more than t symbol errors in a block, which can be written as

$$P_{ue} = 1 - \sum_{i=0}^{t} \binom{n}{i} (P_{se})^i (1 - P_{se})^{n-i} \tag{2.65}$$

If we didn't have an RS code, any error in a code block would be uncorrectable, and the probability is given as

$$P_{ue} = 1 - (1 - P_{se})^n \tag{2.66}$$

One can plot a set of curves to illustrate the error-correcting performance of the code. A graph of Eq. (2.65) appears in Fig. 2.12 for the example in our discussion. Figure 2.12 is similar to Figs. 2.5 and 2.8 except that the x-axis is plotted in opposite order and the y-axis has not been normalized by dividing by Eq. (2.66). Reading from the curve, we see for the case where $t = 5$ and $P_{se} = 10^{-3}$:

$$P_{ue} = 3 \times 10^{-7} \tag{2.67}$$

2.7.5 Effect of RS Coder–Decoder Failures

We can use Eqs. (2.8) and (2.9) to evaluate the effect of coder–decoder failures. However, instead of computing per byte of transmission, we compute per block of transmission. Thus, by analogy with Eqs. (2.10) and (2.11), for our example we have

$$P_{ue} = e^{-2\lambda_b t} \times 3 \times 10^{-7} + (1 - e^{-2\lambda_b t}) \tag{2.68}$$

where

$$t = 8 \times 255/3,600B \tag{2.69}$$

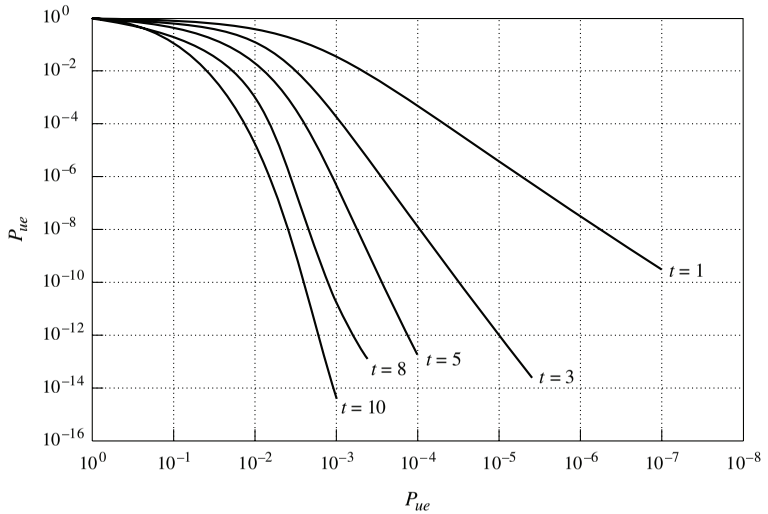


Figure 2.12 Probability of an uncorrected error in a block of 255 1-byte symbols with 235 message symbols, 20 check symbols, and an error-correcting capacity of up to 10 errors versus the probability of symbol error [AHA Applications Note, used with permission].

We can compute when P_{ue} is composed of equal values for code failures and chip failures by equating the first and second terms of Eq. (2.68). Substituting a typical value of $B = 19{,}200$, we find that this occurs when the chip failure rate is equal to about 5.04×10^{-3} failures per hour. Using our model, the chip failure rate $= 0.004\sqrt{g}\ 10^{-6}$, which is equivalent to $g = 1.6 \times 10^{12}$—a very unlikely value. However, if we assume that $P_{se} = 10^{-4}$, then from Fig. 2.12 we see that $P_{ue} = 3 \times 10^{-13}$ and for $B = 19{,}200$ that the effects are equal if the chip failure is equal to about 5.08×10^{-9}. Substitution into our chip failure rate model shows that this occurs when $g \approx 2$. Thus coder–decoder failures predominate for the second case.

Another approach to investigating the impact of chip failures is to use manufacturers' data on RS coder–decoder failures. Some data exists [AHA Reliability Report, 1995] that is derived from accelerated tests. To collect enough failure data for low-failure-rate components, an accelerated life test—the Arrhenius Relationship—is used to scale back the failure rates to normal operating temperatures (70–85°C). The resulting failure rates range from 50 to 700×10^{-9} failures per hour, which certainly exceeds the just-calculated significant failure rate threshold of 5.08×10^{-9}, which was the value calculated for 19,200 baud and a block error of 10^{-4}. (Note: using the gate model, we calculate $\lambda = 700 \times 10^{-9}$ as equivalent to about 30,000 gates.) Clearly we conclude that the chip failures will predominate for some common ranges of the system parameters.

2.8 OTHER CODES

There are many other types of error codes. We will briefly discuss the special features of the more popular codes and refer the reader to the references for additional details.

1. *Burst error codes.* All the foregoing codes assume that errors occur infrequently and are independent, generally corrupting a single bit or a few bits in a word. In some cases, errors may occur in bursts. If a study of the failure modes of the device or medium we wish to protect by our coding indicates that errors occur in bursts to affect all the bits in a word, other coding techniques are required. The reader should consult the references for works that discuss Binary Block codes, *m*-out-of-*n* codes, Berger codes, Cyclic codes, and Reed–Solomon codes [Pradhan, 1986, 1993].

   BCH codes. This is a code that was independently discovered by Bose, Chaudhury, and Hocquenghem. (Reed–Solomon codes are a subclass of BCH codes.) These codes can be viewed as extensions of Hamming codes, which are easier to design and implement for a large number of correctable errors.

   Concatenated codes. This refers to the process of lumping more than one code word together to reduce the overhead—generally less for long code words (cf., Table 2.8). Disadvantages include higher error probability (since check bits cover several words), more complexity and depth, and a delay for associated decoding trees.

   Convolutional codes. Sometimes, codes are "nested"; for example, information can be coded by an inner code, and the resulting alphabet of legal code words can be treated as a "symbol" subjected to an outer code. An example might be the use of a Hamming SECSED code as the inner code word and a Reed–Solomon code as an outer code scheme.

   Check sum. The sum of all the numbers in a block of words is added, modulo 2, and the block and the sum are transmitted. The words in the received block are added again and the check sum is recomputed and checked with the transmitted sum. This is an error-detecting code.

   Duplication. One can always transmit the result twice and check the two copies. Although this may be inefficient, it is the only technique in some cases: for example, if we wish to check logical operations, AND, OR, and NAND.

   Fire code. An interleaved code for burst errors. The similar Reed–

Solomon code is now more popular since it is somewhat more efficient.

Hamming codes. Other codes in the family use more error-correcting and -detecting bits, thereby achieving higher levels of fault tolerance.

IC chip parity. Can be one bit per word, one bit per byte, or interlaced parity where b bits are watched over by i check bits. Thus each check bit "watches over" b/i bits.

Interleaving. One approach to dealing with burst codes is to disassemble codes into a number of words, then reassemble them so that one bit is chosen from each word. For example, one could take 8 bytes and interleave (also called interlace) the bits so that a new byte is constructed from all the first bits of the original 8 bytes, another is constructed from all the second bits, and so forth. In this example, as long as the burst length is less than 8 bits and we have only one burst per 8 bytes, we are guaranteed that each new word can contain at most one error.

Residue m *codes.* This is used for checking certain arithmetic operations, such as addition, multiplication, and shifting. One computes the code bits (residue, R) that are concatenated (|, i.e., appended) to the message N to from $N|R$. The residue is the remainder left when N/m. After transmission or computation, the new message bits N' are divided by m to form R'. Disagreement of R and R' indicates an error.

Viterby decoding. A decoding algorithm for error correction of a Reed–Solomon or other convolutional code based on enumerating all the legal code words and choosing the one closest to the received words. For medium-sized search spaces, an organized search resembling a branching tree was devised by Viterbi in 1967; it is often used to shorten the search. Forney recognized in 1968 that such trees are repetitive, so he devised an improvement that led to a diagram looking like a "lattice" used for supporting plants and trees.

REFERENCES

AHA Reliability Report No. 4011. Reed–Solomon Coder/Decoder. Advanced Hardware Architectures, Inc., Pullman, WA, 1995.

AHA Applications Note. Primer: Reed–Solomon Error Correction Codes (ECC). Advanced Hardware Architectures, Pullman, WA, Inc.

Arazi, B. *A Commonsense Approach to the Theory of Error Correcting Codes.* MIT Press, Cambridge, MA, 1988.

Forney, G. D. Jr. *Concatenated Codes.* MIT Press Research Monograph, no. 37. MIT Press, Cambridge, MA, 1966.

Golomb, S. W. Optical Disk Error Correction. *Byte* (May 1986): 203–210.

Gravano, S. *Introduction to Error Control Codes.* Oxford University Press, New York, 2000.

Hamming, R. W. Error Detecting and Correcting Codes. *Bell System Technical Journal* 29 (April 1950): 147–160.

Houghton, A. D. *The Engineer's Error Coding Handbook.* Chapman and Hall, New York, 1997.

Johnson, B. W. *Design and Analysis of Fault Tolerant Digital Systems.* Addison-Wesley, Reading, MA, 1989.

Johnson, B. W. *Design and Analysis of Fault Tolerant Digital Systems*, 2d ed. Addison-Wesley, Reading, MA, 1994.

Jones, G. A., and J. M. Jones. *Information and Coding Theory.* Springer-Verlag, New York, 2000.

Lala, P. K. *Fault Tolerant and Fault Testable Hardware Design.* Prentice-Hall, Englewood Cliffs, NJ, 1985.

Lala, P. K. *Self-Checking and Fault-Tolerant Digital Design.* Academic Press, San Diego, CA, 2000.

Lee, C. *Error-Control Block Codes for Communications Engineers.* Telecommunication Library, Artech House, Norwood, MA, 2000.

Peterson, W. W. *Error-Correcting Codes.* MIT Press (Cambridge, MA) and Wiley (New York), 1961.

Peterson, W. W., and E. J. Weldon Jr. *Error Correcting Codes*, 2d ed. MIT Press, Cambridge, MA, 1972.

Pless, V. *Introduction to the Theory of Error-Correcting Codes.* Wiley, New York, 1998.

Pradhan, D. K. *Fault-Tolerant Computing Theory and Technique*, vol. I. Prentice-Hall, Englewood Cliffs, NJ, 1986.

Pradhan, D. K. *Fault-Tolerant Computing Theory and Technique*, vol. II. Prentice-Hall, Englewood Cliffs, NJ, 1993.

Rao, T. R. N., and E. Fujiwara. *Error-Control Coding for Computer Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1989.

Shooman, M. L. *Probabilistic Reliability: An Engineering Approach.* McGraw-Hill, New York, 1968.

Shooman, M. L. *Probabilistic Reliability: An Engineering Approach*, 2d ed. Krieger, Melbourne, FL, 1990.

Shooman, M. L. The Reliability of Error-Correcting Code Implementations. *Proceedings Annual Reliability and Maintainability Symposium*, Las Vegas, NV, January 22–25, 1996.

Shooman, M. L., and F. A. Cassara. The Reliability of Error-Correcting Codes on Wireless Information Networks. *International Journal of Reliability, Quality, and Safety Engineering*, special issue on Reliability of Wireless Communication Systems, 1996.

Siewiorek, D. P., and F. S. Swarz. *The Theory and Practice of Reliable System Design.* The Digital Press, Bedford, MA, 1982.

Siewiorek, D. P., and R. S. Swarz. *Reliable Computer Systems Design and Evaluation*, 2d ed. The Digital Press, Bedford, MA, 1992.

Siewiorek, D. P., and R. S. Swarz. *Reliable Computer Systems Design and Evaluation*, 3d ed. A. K. Peters, www.akpeters.com, 1998.

Spencer, J. L. The Highs and Lows of Reliability Predictions. *Proceedings Annual Reliability and Maintainability Symposium*, 1986. IEEE, New York, NY, pp. 156–162.

Stapper, C. H. et al. High-Reliability Fault-Tolerant 16-M Bit Memory Chip. *Proceedings Annual Reliability and Maintainability Symposium*, January 1991. IEEE, New York, NY, pp. 48–56.

Taylor, L. *Air Travel How Safe Is It?* BSP Professional Books, Cambridge, MA, 1988.

Texas Instruments, *TTL Logic Data Book*. 1988, pp. 2-597–2-599.

Wakerly, J. F. *Digital Design Principles and Practice*. Prentice-Hall, Englewood Cliffs, NJ, 1994.

Wakerly, J. F. *Digital Design Principles and Practice*, 3d. ed. Prentice-Hall, Englewood Cliffs, NJ, 2000.

Wells, R. B. *Applied Coding and Information Theory for Engineers*. Prentice-Hall, Englewood Cliffs, NJ, 1998.

Wicker, S. B., and V. K. Bhargava. *Reed–Solomon Codes and their Applications*. IEEE Press, New York, 1994.

Wiggert, D. *Codes for Error Control and Synchronization*. Communications and Defense Library, Artech House, Norwood, MA, 1998.

Wolf, J. J., M. L. Shooman, and R. R. Boorstyn. Algebraic Coding and Digital Redundancy. *IEEE Transactions on Reliability* R-18, 3 (August 1969): 91–107.

PROBLEMS

2.1. Find a recent edition of *Jane's all the World's Aircraft* in a technical or public library. Examine the data given in Table 2.1 for soft failures for the 6 aircraft types listed. From the book, determine the approximate number of electronic systems (aircraft avionics) for each of the aircraft that are computer-controlled (digital rather than analog). You may have to do some intelligent estimation to determine this number. One section in the book gives information on the avionics systems installed. Also, it may help to know that the U.S. companies (all mergers) that provide most of the avionics systems are Bendix/King/Allied, Sperry/Honeywell, and Collins/Rockwell. (Hint: You may have to visit the Web sites of the aircraft manufacturers or the avionics suppliers for more details.

(a) Plot the number of soft fails per aircraft versus the number of avionics systems on board. Comment on the results.

(b) It would be better to plot soft fails per aircraft versus the number of words of main memory for the avionics systems on board. Do you have any ideas on how you could obtain such data?

2.2. Compute the minimum code distance for all the code words given in Table 2.2.

(a) Compute for column (a) and comment.

(b) Compute for column (b) and comment.

(c) Compute for column (c) and comment.

2.3. Repeat the parity-bit coder and decoder designs given in Fig. 2.1 for an 8-bit word with 7 message bits and 1 parity bit. Does this approach to design of a coder and decoder present any difficulties?

2.4. Compare the design of problem 2.3 with that given in Fig. 2.2 on the basis of ease of design, complexity, practicality, delay time (assume all gates have a delay of D), and number of gates.

2.5. Compare the results of problem 2.4 with the circuit of Fig. 2.4.

2.6. Compute the binomial probabilities $B(r:8,q)$ for $r =$ 1 to 8.

(a) Does the sum of these probabilities check with Eq. (2.5)?

(b) Show for what values of q the term $B(1:8,q)$ dominates all the error-occurrence probabilities.

2.7. Find a copy of the latest military failure-rate manual (MIL-HDBK-217) and plot the data on Fig. B7 of Appendix B. Does it agree? Can you find any other IC failure-rate information? (Hint: The telecommunication industry and the various national telephone companies maintain large failure-rate databases. Also, the *Annual Reliability and Maintainability Symposium* from the IEEE regularly publishes papers with failure-rate data.) Does this data agree with the other results? What advances have been made in the last decade or so?

2.8. Assume that a 10% reduction in the probability of undetected error from coder and decoder failures is acceptable.

(a) Compute the value of B at which a 10% reduction occurs for fixed values of q.

(b) Plot the results of part (a) and interpret.

2.9. Check the results given in Table 2.5. How is the distance d related to the number of check bits? Explain.

2.10. Check the values given in Tables 2.7 and 2.8.

2.11. The Hamming SECSED code with 4 message bits and 3 check bits is used in the text as an example (Section 2.4.3). It was stated that we could use a brute force technique of checking all the legal or illegal code words for error detection, as was done for the parity-bit code in Fig. 2.1.

(a) List all the legal and illegal code words for this example and show that the code distance is 3.

(b) Design an error-detector circuit using minimized two-level logic (cf. Fig. 2.1).

2.12. Design a check bit generating circuit for problem 2.11 using Eqs. (2.22a–c) and EXOR gates.

2.13. One technique for error correction is to pick the nearest code word as the correct word once an error has been detected.

(a) Devise a software algorithm that can be used to program a microprocessor to perform such error correction.

(b) Devise a hardware design that performs the error correction by choosing the closest word.

(c) Compare complexity and speed of designs (a) and (b).

2.14. An error-correcting circuit for a Hamming (7, 4) SECSED is given in Fig. 2.7. How would you generate the check bits that are defined in Eqs. (2.22a–c)? Is there a better way than that suggested in problem 2.12?

2.15. Compare the designs of problems 2.11, 2.12, and 2.13 with Hamming's technique in problem 2.14.

2.16. Give a complete design for the code generator and checker for a Hamming (12, 8) SECSED code following the approach of Fig. 2.7.

2.17. Repeat problem 2.16 for a SECDED code.

2.18. Repeat problem 2.8 for the design referred to in Table 2.14.

2.19. Retransmission as described in Section 2.5 tends to decrease the effective baud rate (B) of the transmission. Compare the unreliability and the effective baud rate for the following designs:

(a) Transmit each word twice and retransmit when they disagree.

(b) Transmit each word three times and use the majority of the three values to determine the output.

(c) Use a parity-bit code and only retransmit when the code detects an error.

(d) Use a Hamming SECSED code and only retransmit when the code detects an error.

2.20. Add the probabilities of generator and checker failure for the reliability examples given in Section 2.5.3.

2.21. Assume we are dealing with a burst code design for error detection with a word length of 12 bits and a maximum burst length of 4, as noted in Eqs. (2.46)–(2.50). Assume the code vector $V(x_1, x_2, \ldots, x_{12}) = V(c_1c_2c_3c_410100011)$.

(a) Compute $c_1c_2c_3c_4$.

(b) Assume no errors and show how the syndrome works.

(c) Assume one error in bit c_2 and show how the syndrome works.

(d) Assume one error in bit x_9; then show how the syndrome works.

(e) Assume two errors in bits x_8 and x_9; then show how the syndrome works.

(f) Assume three errors in bits x_8, x_9, and x_{10}; then show how the syndrome works.