modified version of Fig. 12.4-6*a*, with N subframes, 6 control bits per subframe, and $6 \times N \times n$ message and stuff bits per subframe. Obtain an expression for s in terms of n, and explain why this scheme would be unsatisfactory.

12.4-6 Suppose both the input and output clock rates of the AT & T M12 multiplexer drift by a maximum factor of $1 \pm \delta$ relative to their nominal rates. Use worst-case analysis to determine the upper limit on δ.

12.4-7 A certain multiplexer combines data from several 110-bps asynchronous terminals for synchronous transmission at 4800 bps. The multiplexer inserts a synchronizing character after every 19 data characters, and it requires 2% of the output for control and stuff characters. How many terminals can be accommodated along with three 1200-bps data signals?

12.4-8 A 2400-bps synchronous signal is to be generated by multiplexing asynchronous data from one 1200-bps terminal, two 300-bps terminals, and several 110-bps terminals. The multiplexer inserts a synchronizing character after every 49 data characters, and it requires 3% of the output for control and stuff characters. Find the maximum allowed number of 110-bps terminals. Then draw a multiplexer diagram like Fig. 12.4-8 using a processor with eight input ports.

12.4-9 A 3600-bps synchronous signal is to be generated by multiplexing asynchronous data from one 1200-bps terminal, two 600-bps terminals, one 300-bps terminal, and several 150-bps terminals. The multiplexer inserts a synchronizing character after every 49 data characters, and it requires 2% of the output for control and stuff characters. Find the maximum allowed number of 150-bps terminals. Then draw a multiplexer diagram like Fig. 12.4-8 using a processor with six input ports.

CHAPTER

THIRTEEN

ERROR-CONTROL CODING

Transmission errors in digital communication depend on the signal-to-noise ratio. If a particular system has a fixed value of S/N and the error rate is unacceptably high, then some other means of improving reliability must be sought. Error-control coding often provides the best solution.

Error-control coding involves systematic addition of extra digits to the transmitted message. These extra *check digits* convey no information by themselves, but they make it possible to detect or correct errors in the regenerated message digits. In principle, information theory holds out the promise of nearly errorless transmission, as will be discussed in Chap. 15. In practice, we seek some compromise between conflicting considerations of reliability, efficiency, and equipment complexity. A multitude of error-control codes have therefore been devised to suit various applications.

This chapter starts with an overview of error-control coding, emphasizing the distinction between *error detection* and *error correction* and systems that employ these strategies. Subsequent sections describe the two major types of code implementations, *block codes* and *convolutional codes*. We'll stick entirely to binary coding, and we'll omit formal mathematical analysis. Detailed treatments of error-control coding are provided by the references cited in the supplementary reading list.

13.1 ERROR DETECTION AND CORRECTION

Coding for error detection, without correction, is simpler than error-correction coding. When a two-way channel exists between source and destination, the receiver can request retransmission of information containing detected errors.

This error-control strategy, called *automatic repeat request* (ARQ), particularly suits data communication systems such as computer networks. However, when retransmission is impossible or impractical, error control must take the form of *forward error correction* (FEC) using an error-correcting code. Both strategies will be examined here, after an introduction to simple but illustrative coding techniques.

Repetition and Parity-Check Codes

When you try to talk to someone across a noisy room, you may need to repeat yourself to be understood. A brute-force approach to binary communication over a noisy channel likewise employs *repetition*, so each message bit is represented by a *codeword* consisting of n identical bits. Any transmission error in a received codeword alters the repetition pattern by changing a 1 to a 0 or vice versa.

If transmission errors occur randomly and independently with probability $P_e = \alpha$, then the binomial frequency function from Eq. (1), Sect. 4.4, gives the probability of i errors in an n-bit codeword as

$$P(i, n) = \binom{n}{i} \alpha^i (1 - \alpha)^{n-i} \approx \binom{n}{i} \alpha^i \qquad \alpha \ll 1 \tag{1a}$$

where

$$\binom{n}{i} = \frac{n!}{i!(n-i)!} = \frac{n(n-1)\cdots(n-i+1)}{i!} \tag{1b}$$

We'll proceed on the assumption that $\alpha \ll 1$—which does not necessarily imply reliable transmission since $\alpha = 0.1$ satisfies our condition but would be an unacceptable error probability for digital communication. Repetition codes improve reliability when α is sufficiently small that $P(i + 1, n) \ll P(i, n)$ and, consequently, several errors per word are much less likely than a few errors per word.

Consider, for instance, a triple-repetition code with codewords 000 and 111. All other received words, such as 001 or 101, clearly indicate the presence of errors. Depending on the decoding scheme, this code can *detect* or *correct* erroneous words. For error detection without correction, we say that any word other than 000 or 111 is a detected error. Single and double errors in a word are thereby detected, but triple errors result in an undetected *word error* with probability

$$P_{we} = P(3, 3) = \alpha^3$$

For error correction, we use majority-rule decoding based on the assumption that at least two of the three bits are correct. Thus, 001 and 101 are decoded as 000 and 111, respectively. This rule corrects words with single errors, but double or triple errors result in a decoding error with probability

$$P_{we} = P(2, 3) + P(3, 3) = 3\alpha^2 - 2\alpha^3$$

Since $P_e = \alpha$ would be the error probability without coding, we see that either decoding scheme for the triple-repetition code greatly improves reliability if, say, $\alpha \leq 0.01$. However, this improvement is gained at the cost of reducing the message bit rate by a factor of 1/3.

More efficient codes are based on the notion of *parity*. The parity of a binary word is said to be even when the word contains an even number of 1s, while odd parity means an odd number of 1s. The codewords for an *error-detecting parity-check code* are constructed with $n - 1$ message bits and one check bit chosen such that all codewords have the same parity. With $n = 3$ and even parity, the valid codewords are 000, 011, 101, and 110, the last bit in each word being the parity check. When a received word has odd parity, 001 for instance, we immediately know that it contains a transmission error—or three errors or, in general, an odd number of errors. Error correction is not possible because we don't know where the errors fall within the word. Furthermore, an even number of errors preserves valid parity and goes unnoticed.

Under the condition $\alpha \ll 1$, double errors occur far more often than four or more errors per word. Hence, the probability of an undetected error in an n-bit parity-check codeword is

$$P_{we} \approx P(2, n) \approx \frac{n(n-1)}{2} \alpha^2 \tag{2}$$

For comparison purposes, *uncoded* transmission of words containing $n - 1$ message bits would have

$$P_{uwe} = 1 - P(0, n - 1) \approx (n - 1)\alpha$$

Thus, if $n = 10$ and $\alpha = 10^{-3}$, then $P_{uwe} \approx 10^{-2}$ whereas coding yields $P_{we} \approx 5 \times 10^{-5}$ with a rate reduction of just 9/10. These numbers help explain the popularity of parity checking for error detection in computer systems.

As an example of parity checking for error correction, Fig. 13.1-1 illustrates an error-correcting scheme in which the codeword is formed by arranging k message bits in a square array whose rows and columns are checked by $2\sqrt{k}$ parity bits. A transmission error in one message bit causes a row and column

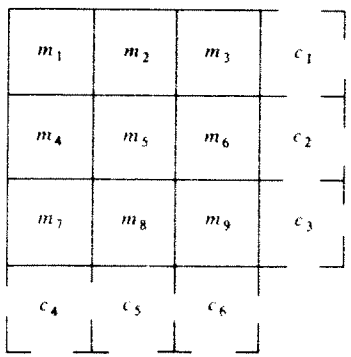


Figure 13.1-1 Square array for error correction by parity checking.

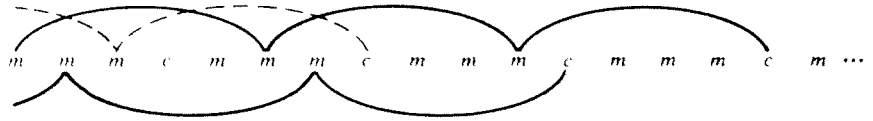


Figure 13.1-2 Interleaved check bits for error control with burst errors.

parity failure with the error at the intersection, so single errors can be corrected. This code also detects double errors.

Throughout the foregoing discussion we've assumed that transmission errors appear randomly and independently in a codeword. This assumption holds for errors caused by white noise or filtered white noise. But *impulse noise* produced by lightning and switching transients causes errors to occur in *bursts* that span several successive bits. Burst errors also appear when radio-transmission systems suffer from rapid fading. Such multiple errors wreak havoc on the performance of conventional codes and must be combated by special techniques. Parity checking controls burst errors if the check bits are *interleaved* so that the checked bits are widely spaced, as represented in Fig. 13.1-2 where a curved line connects the message bits and check bit in one parity word.

Code Vectors and Hamming Distance

Rather than continuing a piecemeal survey of particular codes, we now introduce a more general approach in terms of code *vectors*. An arbitrary n-bit codeword can be visualized in an n-dimensional space as a vector whose elements or coordinates equal the bits in the codeword. We thus write the codeword 101 in row-vector notation as $X = (1\ 0\ 1)$. Figure 13.1-3 portrays all possible 3-bit codewords as dots corresponding to the vector tips in a three-dimensional space. The solid dots in part (*a*) represent the triple-repetition code, while those in part (*b*) represent a parity-check code.

Notice that the triple-repetition code vectors have greater separation than the parity-check code vectors. This separation, measured in terms of the

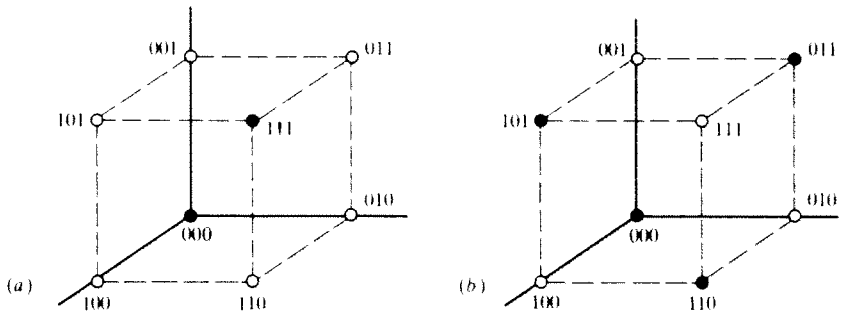


Figure 13.1-3 Vectors representing 3-bit codewords. (*a*) Triple-repetition code; (*b*) parity-check code.

Hamming distance, has direct bearing on the error-control power of a code. The Hamming distance $d(X, Y)$ between two vectors X and Y is defined to equal the number of different elements. For instance, if $X = (1\ 0\ 1)$ and $Y = (1\ 1\ 0)$ then $d(X, Y) = 2$ because the second and third elements are different.

The *minimum distance* $d_{\min}$ of a particular code is the smallest Hamming distance between valid code vectors. Consequently, error detection is always possible when the number of transmission errors in a codeword is less than $d_{\min}$ so the erroneous word is not a valid vector. Conversely, when the number of errors equals or exceeds $d_{\min}$, the erroneous word may correspond to another valid vector and the errors cannot be detected.

Further reasoning along this line leads to the following distance requirements for various degrees of error control capability:

Detect up to ℓ errors per word	$d_{\min} \geq \ell + 1$	(3*a*)
Correct up to t errors per word	$d_{\min} \geq 2t + 1$	(3*b*)
Correct up to t errors and detect $\ell > t$ errors per word	$d_{\min} \geq t + \ell + 1$	(3*c*)

By way of example, we see from Fig. 13.1-3 that the triple-repetition code has $d_{\min} = 3$. Hence, this code could be used to detect $\ell \leq 3 - 1 = 2$ errors per word or to correct $t \leq (3 - 1)/2 = 1$ error per word — in agreement with our previous observations. A more powerful code with $d_{\min} = 7$ could correct triple errors or it could correct double errors and detect quadruple errors.

The power of a code obviously depends on the number of bits added to each codeword for error-control purposes. In particular, suppose that the codewords consist of $k < n$ message bits and $n - k$ parity bits checking the message bits. This structure is known as an (n, k) *block code*. The minimum distance of an (n, k) block code is upper-bounded by

$$d_{\min} \leq n - k + 1 \tag{4}$$

and the code's efficiency is measured by the *code rate*

$$R_c \triangleq k/n \tag{5}$$

Regrettably, the upper bound in Eq. (4) is realized only by repetition codes, which have $k = 1$ and very inefficient code rate $R_c = 1/n$. Considerable effort has thus been devoted to the search for powerful and reasonably efficient codes, a topic we'll return to in the next section.

FEC Systems

Now we're prepared to examine the forward error correction system diagrammed in Fig. 13.1-4. Message bits come from an information source at rate r_b. The encoder takes blocks of k message bits and constructs an (n, k) block code with

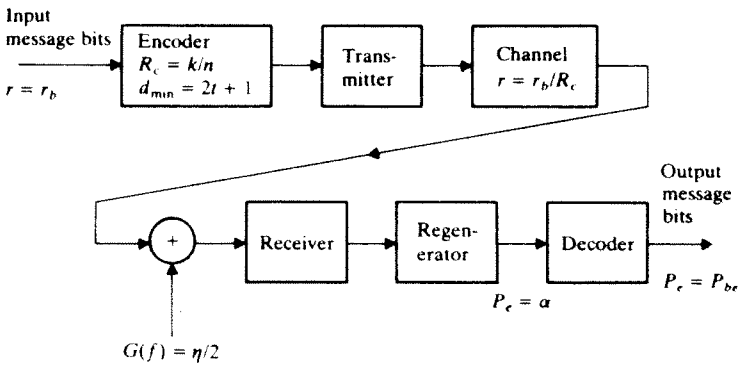


Figure 13.1-4 FEC system.

code rate $R_c = k/n < 1$. The bit rate on the channel therefore must be greater than r_b, namely

$$r = (n/k)r_b = r_b/R_c \tag{6}$$

The code has $d_{\min} = 2t + 1 \leq n - k + 1$, and the decoder operates strictly in an error-correction mode.

We'll investigate the performance of this FEC system when additive white noise causes random errors with probability $\alpha \ll 1$. The value of α depends, of course, on the signal energy and noise density at the receiver. If E_b represents the average energy per *message* bit, then the average energy per *code* bit is $R_c E_b$ and the ratio of bit energy to noise density is

$$\gamma_c \triangleq R_c E_b/\eta = R_c \gamma_b \tag{7}$$

where $\gamma_b = E_b/\eta$. Our performance criterion will be the probability of output message-bit errors, denoted by P_{be} to distinguish it from the word error probability P_{we}.

The code always corrects up to t errors per word and some patterns of more than t errors may also be correctable, depending upon the specific code vectors. Thus, the probability of a decoding word error is upper-bounded by

$$P_{we} \leq \sum_{i=t+1}^{n} P(i, n)$$

For a rough but reasonable performance estimate, we'll take the approximation

$$P_{we} \approx P(t + 1, n) \approx \binom{n}{t+1} \alpha^{t+1} \tag{8}$$

which means that an uncorrected word typically has $t + 1$ bit errors. On the average, there will be $(k/n)(t + 1)$ message-bit errors per uncorrected word, the remaining errors being in check bits. When Nk bits are transmitted in $N \gg 1$ words, the expected total number of erroneous message bits at the output is $(k/n)(t + 1)NP_{we}$. Hence,

$$P_{be} = \frac{t+1}{n} P_{we} \approx \binom{n-1}{t} \alpha^{t+1} \tag{9}$$

in which we have used Eq. (1*b*) to combine $(t + 1)/n$ with the binomial coefficient.

If the noise has a gaussian distribution and the transmission system has been optimized (i.e., polar signaling and matched filtering), then the transmission error probability is given by Eq. (16), Sect. 11.2, as

$$\begin{aligned} \alpha &= Q(\sqrt{2\gamma_c}) = Q(\sqrt{2R_c \gamma_b}) \\ &\approx (4\pi R_c \gamma_b)^{-1/2} e^{-R_c \gamma_b} \qquad R_c \gamma_b \geq 5 \end{aligned} \tag{10}$$

The gaussian tail approximation invoked here follows from Eq. (10), Sect. 4.4, and is consistent with the assumption that $\alpha \ll 1$. Thus, our final result for the output error probability of the FEC system becomes

$$\begin{aligned} P_{be} &= \binom{n-1}{t} [Q(\sqrt{2R_c \gamma_b})]^{t+1} \\ &\approx \binom{n-1}{t} (4\pi R_c \gamma_b)^{-(t+1)/2} e^{-(t+1)R_c \gamma_b} \end{aligned} \tag{11}$$

Uncoded transmission on the same channel would have

$$P_{ube} = Q(\sqrt{2\gamma_b}) \approx (4\pi\gamma_b)^{-1/2} e^{-\gamma_b} \tag{12}$$

since the signaling rate can be decreased from r_b/R_c to r_b.

A comparison of Eqs. (11) and (12) brings out the importance of the code parameters $t = (d_{\min} - 1)/2$ and $R_c = k/n$. The added complexity of an FEC system is justified provided that t and R_c yield a value of P_{be} significantly less than P_{ube}. The exponential approximations show that this essentially requires $(t + 1)R_c > 1$. Hence, a code that only corrects single or double errors should have a relatively high code rate, while more powerful codes may succeed despite lower code rates. The channel parameter γ_b also enters into the comparison, as demonstrated by the following example.

Example 13.1-1 Suppose we have a (15, 11) block code with $d_{\min} = 3$, so $t = 1$ and $R_c = 11/15$. An FEC system using this code would have $\alpha = Q[\sqrt{(22/15)\gamma_b}]$ and $P_{be} = 14\alpha^2$, whereas uncoded transmission on the same channel would yield $P_{ube} = Q(\sqrt{2\gamma_b})$. These three probabilities are plotted versus γ_b in dB in Fig. 13.1-5. If $\gamma_b > 8$ dB, we see that coding decreases the error probability by at least an order of magnitude compared to uncoded transmission. At $\gamma_b = 10$ dB, for instance, uncoded transmission yields $P_{ube} \approx 4 \times 10^{-6}$ whereas the FEC system has $P_{be} \approx 10^{-7}$ even though the higher channel bit rate increases the transmission error probability to $\alpha \approx 6 \times 10^{-5}$.

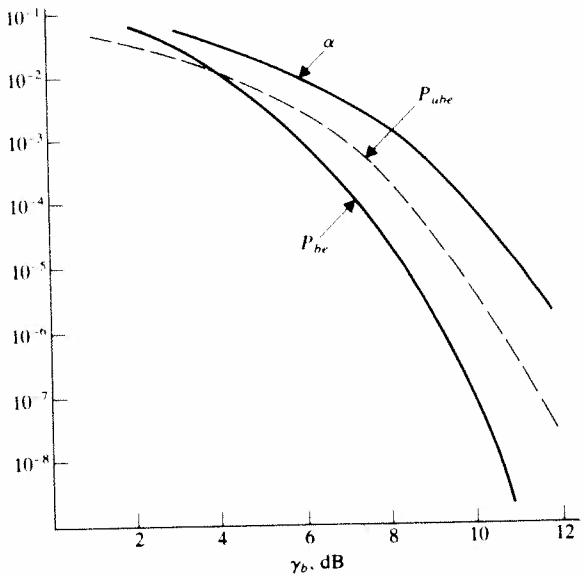


Figure 13.1-5 Curves of error probabilities in Example 13.1-1.

If $\gamma_b < 8$ dB, however, coding does not significantly improve reliability and actually makes matters worse when $\gamma_b < 4$ dB. Furthermore, an uncoded system could achieve better reliability than the FEC system simply by increasing the signal-to-noise ratio about 1.5 dB. Hence, this particular code doesn't save much signal power, but it would be effective if γ_b has a fixed value in the vicinity of 8–10 dB.

Exercise 13.1-1 Suppose the system in Example 13.1-1 is operated at $\gamma_b \approx 8$ dB so $\alpha = 0.001$. Evaluate $P(i, n)$ for $i = 0$, 1, 2, and 3. Do your results support the approximation in Eq. (8)?

ARQ Systems

The automatic-repeat-request strategy for error control is based on error detection and retransmission rather than forward error correction. Consequently, ARQ systems differ from FEC systems in three important respects. First, an (n, k) block code designed for error detection generally requires fewer check bits and has a higher k/n ratio than a code designed for error correction. Second, an ARQ system needs a return transmission path and additional hardware in order to implement repeat transmission of codewords with detected errors. Third, the forward transmission bit rate must make allowance for repeated word transmissions. The net impact of these differences becomes clearer after we describe the operation of the ARQ system represented by Fig. 13.1-6.

Each codeword constructed by the encoder is stored temporarily and transmitted to the destination where the decoder looks for errors. The decoder issues a *positive acknowledgement* (ACK) if no errors are detected, or a *negative acknowledgement* (NAK) if errors are detected. A negative acknowledgement causes the input controller to retransmit the appropriate word from those stored by the input buffer. A particular word may be transmitted just once, or it may be transmitted two or more times, depending on the occurrence of transmission errors. The function of the output controller and buffer is to assemble the output bit stream from the codewords that have been accepted by the decoder.

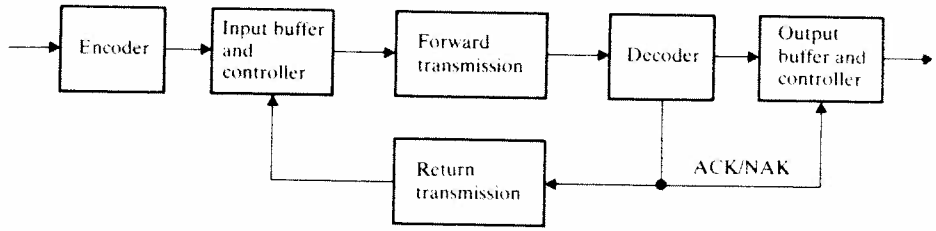


Figure 13.1-6 ARQ system.

Compared to forward transmission, return transmission of the ACK/NAK signal involves a low bit rate and we can reasonably assume a negligible error probability on the return path. Under this condition, all codewords with detected errors are retransmitted as many times as necessary, so the only output errors appear in words with undetected errors. For an (n, k) block code with $d_{\min} = \ell + 1$, the corresponding output error probabilities are

$$P_{we} = \sum_{i=\ell+1}^{n} P(i, n) \approx P(\ell + 1, n) \approx \binom{n}{\ell + 1} \alpha^{\ell+1} \tag{13}$$

$$P_{be} = \frac{\ell + 1}{n} P_{we} \approx \binom{n-1}{\ell} \alpha^{\ell+1} \tag{14}$$

which are identical to the FEC expressions, Eqs. (8) and (9), with ℓ in place of t. Since the decoder accepts words that have either no errors or undetectable errors, the *word retransmission probability* is given by

$$p = 1 - [P(0, n) + P_{we}]$$

But a good error-detecting code should yield $P_{we} \ll P(0, n)$. Hence,

$$p \approx 1 - P(0, n) = 1 - (1 - \alpha)^n \approx n\alpha \tag{15}$$

where we've used the approximation $(1 - \alpha)^n \approx 1 - n\alpha$ based on $n\alpha \ll 1$.

As for the retransmission process itself, there are three basic ARQ schemes illustrated by the timing diagrams in Fig. 13.1-7. The asterisk marks words received with detected errors which must be retransmitted. The *stop-and-wait* scheme in part *a* requires the transmitter to stop after every word and wait for acknowledgment from the receiver. Just one word needs to be stored by the input buffer, but the transmission time delay t_d in each direction results in an *idle time* of duration $D \geq 2t_d$ between words. Idle time is eliminated by the *go-back-N* scheme in part *b* where codewords are transmitted continuously. When the receiver sends a NAK signal, the transmitter goes back N words in the buffer and

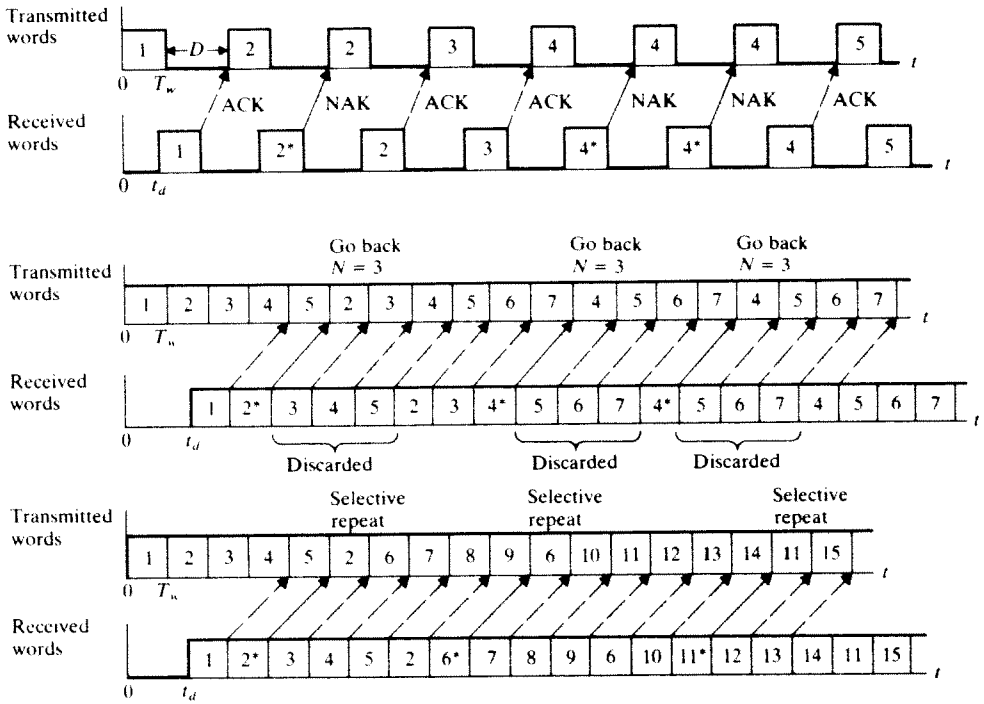


Figure 13.1-7 ARQ schemes. (*a*) Stop-and-wait; (*b*) go-back-N; (*c*) selective-repeat.

retransmits starting from that point. The receiver discards the $N - 1$ intervening words, correct or not, in order to preserve proper sequence. The *selective-repeat* scheme in part *c* puts the burden of sequencing on the output controller and buffer, so that only words with detected errors need to be retransmitted.

Clearly, a selective-repeat ARQ system has the highest *throughput efficiency*. To set this on a quantitative footing, we observe that the total number of transmissions of a given word is a discrete random variable m governed by the event probabilities $P(m = 1) = 1 - p$, $P(m = 2) = p(1 - p)$, etc. The average number of transmitted words per accepted word is then

$$\bar{m} = 1(1 - p) + 2p(1 - p) + 3p^2(1 - p) + \cdots$$

$$= (1 - p)(1 + 2p + 3p^2 + \cdots) = \frac{1}{1 - p} \tag{16}$$

since $1 + 2p + 3p^2 + \cdots = (1 - p)^{-2}$. On the average, the system must transmit $n\bar{m}$ bits for every k message bits, so the throughput efficiency is

$$R_c' = \frac{k}{n\bar{m}} = \frac{k}{n}(1 - p) \tag{17}$$

in which $p \approx n\alpha$, from Eq. (15).

We use the symbol R_c' here to reflect the fact that the forward-transmission bit rate r and the message bit rate r_b are related by

$$r = r_b/R_c'$$

comparable to the relationship $r = r_b/R_c$ in an FEC system. Thus, when the noise has a gaussian distribution, the transmission error probability α is calculated from Eq. (10) using R_c' instead of $R_c = k/n$. Furthermore, if $p \ll 1$, then $R_c' \approx k/n$. But an error-detecting code has a larger k/n ratio than an error-correcting code of equivalent error-control power. Under these conditions, the more elaborate hardware needed for selective-repeat ARQ may pay off in terms of better performance than an FEC system would yield on the same channel.

The expression for $\bar{m}$ in Eq. (16) also applies to a stop-and-wait ARQ system. However, the idle time reduces efficiency by the factor $T_w/(T_w + D)$ where $D \geq 2t_d$ is the round-trip delay and T_w is the word duration given by $T_w = n/r \leq k/r_b$. Hence,

$$R_c' = \frac{k}{n}\frac{1 - p}{1 + (D/T_w)} \leq \frac{k}{n}\frac{1 - p}{1 + (2t_d r_b/k)} \tag{18}$$

in which the upper bound comes from writing $D/T_w \geq 2t_d r_b/k$.

A go-back-N ARQ system has no idle time, but N words must be retransmitted for each word with detected errors. Consequently, we find that

$$\bar{m} = 1 + \frac{Np}{1 - p} \tag{19}$$

and

$$R_c' = \frac{k}{n}\frac{1 - p}{1 - p + Np} \leq \frac{k}{n}\frac{1 - p}{1 - p + (2t_d r_b/k)p} \tag{20}$$

where the upper bound reflects the fact that $N \geq 2t_d/T_w$.

Unlike selective-repeat ARQ, the throughput efficiency of the stop-and-wait and go-back-N schemes depends on the round-trip delay. Equations (18) and (20) reveal that both of these schemes have reasonable efficiency if the delay and bit rate are such that $2t_d r_b \ll k$. However, stop-and-wait ARQ has very low efficiency when $2t_d r_b \geq k$, whereas the go-back-N scheme may still be satisfactory provided that the retransmission probability p is small enough.

Finally, we should at least describe the concept of *hybrid* ARQ systems. These systems consist of an FEC subsystem within the ARQ framework, thereby combining desirable properties of both error-control strategies. For instance, a hybrid ARC system might employ a block code with $d_{\min} = t + \ell + 1$, so the decoder can correct up to t errors per word and detect but not correct words with $\ell > t$ errors. Error correction reduces the number of words that must be retransmitted, thereby increasing the throughput without sacrificing the higher reliability of ARQ.

Example 13.1-2 Suppose a selective-repeat ARQ system uses a simple parity-check code with $k = 9$, $n = 10$, and $\ell = 1$. The transmission channel is corrupted by gaussian noise and we seek the value of γ_b needed to get $P_{be} \approx 10^{-5}$. Equation (14) yields the required transmission error probability $\alpha = (10^{-5}/9)^{1/2} \approx 1.1 \times 10^{-3}$, and the corresponding word retransmission probability in Eq. (15) is $p \approx 10\alpha \ll 1$. Hence, the throughput efficiency will be $R_c' \approx k/n = 0.9$, from Eq. (17). Since $\alpha = Q(\sqrt{2R_c'\gamma_b})$ we call upon the plot of Q in Table T.6 to obtain our final result $\gamma_b \approx 3.1^2/1.8 = 5.3$ or 7.3 dB. As a comparison, Fig. 13.1-5 shows that uncoded transmission would have $P_{ube} \approx 6 \times 10^{-4}$ if $\gamma_b = 7.3$ dB and requires $\gamma_b \approx 9.6$ dB to get $P_{ube} = 10^{-5}$. The ARQ system thus achieves a power saving of about 2.3 dB.

Exercise 13.1-2 Assume that the system in Example 13.1-2 has $r_b = 50$ kbps and $t_d = 0.2$ ms. By calculating R_c' show that the go-back-N scheme would be acceptable but not the stop-and-wait scheme when channel limitations require $r \leq 100$ kbps.

13.2 LINEAR BLOCK CODES

This section describes the structure, properties, and implementation of block codes. We start with a matrix representation of the encoding process that generates the check bits for a given block of message bits. Then we use the matrix representation to investigate decoding methods for error detection and correction. The section closes with a brief introduction to the important class of cyclic block codes.

Matrix Representation of Block Codes

An (n, k) block code consists of n-bit vectors, each vector corresponding to a unique block of $k < n$ message bits. Since there are 2^k different k-bit message blocks and 2^n possible n-bit vectors, the fundamental strategy of block coding is to choose the 2^k code vectors such that the minimum distance is as large as possible. But the code should also have some structure that facilitates the encoding and decoding processes. We'll therefore focus on the class of *systematic linear* block codes.

Let an arbitrary code vector be represented by

$$X = (x_1 \quad x_2 \quad \cdots \quad x_n)$$

where the elements $x_1, x_2, \ldots,$ are, of course, binary digits. A code is *linear* if it includes the all-zero vector and if the sum of any two code vectors produces another vector in the code. The *sum* of two vectors, say X and Z, is defined as

$$X + Z \triangleq (x_1 \oplus z_1 \quad x_2 \oplus z_2 \quad \cdots \quad x_n \oplus z_n) \tag{1}$$

in which the elements are combined according to the rules of mod-2 addition given in Eq. (2), Sect. 11.4.

As a consequence of linearity, we can determine a code's minimum distance by the following argument. Let the number of nonzero elements of a vector X be symbolized by $w(X)$, called the vector *weight*. The Hamming distance between any two code vectors X and Z is then

$$d(X, Z) = w(X + Z)$$

since $x_1 \oplus z_1 = 1$ if $x_1 \neq z_1$, etc. The distance between X and Z therefore equals the weight of another code vector $X + Z$. But if $Z = (0 \ 0 \ \cdots \ 0)$ then $X + Z = X$; hence,

$$d_{\min} = [w(X)]_{\min} \qquad X \neq (0 \quad 0 \quad \cdots \quad 0) \tag{2}$$

In other words, the minimum distance of a linear block code equals the *smallest nonzero vector weight*.

A *systematic* block code consists of vectors whose first k elements (or last k elements) are identical to the message bits, the remaining $n - k$ elements being check bits. A code vector then takes the form

$$X = (m_1 \quad m_2 \quad \cdots \quad m_k \quad c_1 \quad c_2 \quad \cdots \quad c_q) \tag{3a}$$

where

$$q = n - k \tag{3b}$$

For convenience, we'll also write code vectors in the partitioned notation

$$X = (M \quad | \quad C)$$

in which M is a k-bit message vector and C is a q-bit check vector. Partitioned notation lends itself to the matrix representation of block codes.

Given a message vector M, the corresponding code vector X for a systematic linear (n, k) block code can be obtained by a matrix multiplication

$$X = MG \tag{4}$$

The matrix G is a $k \times n$ *generator matrix* having the general structure

$$G \triangleq [I_k \quad | \quad P] \tag{5a}$$

where I_k is the $k \times k$ *identity matrix* and P is a $k \times q$ submatrix of binary digits represented by

$$P = \begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1q} \\ p_{21} & p_{22} & \cdots & p_{2q} \\ \vdots & \vdots & & \vdots \\ p_{k1} & p_{k2} & \cdots & p_{kq} \end{bmatrix} \tag{5b}$$

The identity matrix in G simply reproduces the message vector for the first k elements of X, while the submatrix P generates the check vector via

$$C = MP \tag{6a}$$

This binary matrix multiplication follows the usual rules with mod-2 addition instead of conventional addition. Hence, the jth element of C is computed using the jth column of P, and

$$c_j = m_1 p_{1j} \oplus m_2 p_{2j} \oplus \cdots \oplus m_k p_{kj} \tag{6b}$$

for $j = 1, 2, \ldots, q$. All of these matrix operations are less formidable than they appear because every element equals either 0 or 1.

The matrix representation of a block code provides a compact analytical vehicle and, moreover, leads to hardware implementations of the encoder and decoder. But it does not tell us how to pick the elements of the P submatrix to achieve specified code parameters such as $d_{\min}$ and R_c. Consequently, good codes are discovered with the help of considerable inspiration and perspiration, guided by mathematical analysis. In fact, Hamming (1950) devised the first popular block codes several years before the underlying theory was formalized by Slepian (1956).

Example 13.2-1 Hamming codes A *Hamming code* is an (n, k) linear block code with $q \geq 3$ check bits and

$$n = 2^q - 1 \qquad k = n - q \tag{7a}$$

The code rate is

$$R_c = \frac{k}{n} = 1 - \frac{q}{2^q - 1} \tag{7b}$$

and thus $R_c \approx 1$ if $q \gg 1$. Independent of q, the minimum distance is fixed at

$$d_{\min} = 3 \tag{7c}$$

so a Hamming code can be used for single-error correction or double-error detection. To construct a systematic Hamming code, you simply let the k rows of the P submatrix consist of all q-bit words with two or more 1s, arranged in any order.

For example, consider a systematic Hamming code with $q = 3$, so $n = 2^3 - 1 = 7$ and $k = 7 - 3 = 4$. According to the previously stated rule, an appropriate generator matrix is

$$G = \left[\begin{array}{cccc|ccc} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{array}\right]$$

The last three columns constitute the P submatrix whose rows include all 3-bit words that have two or more 1s. Given a block of message bits $M =$

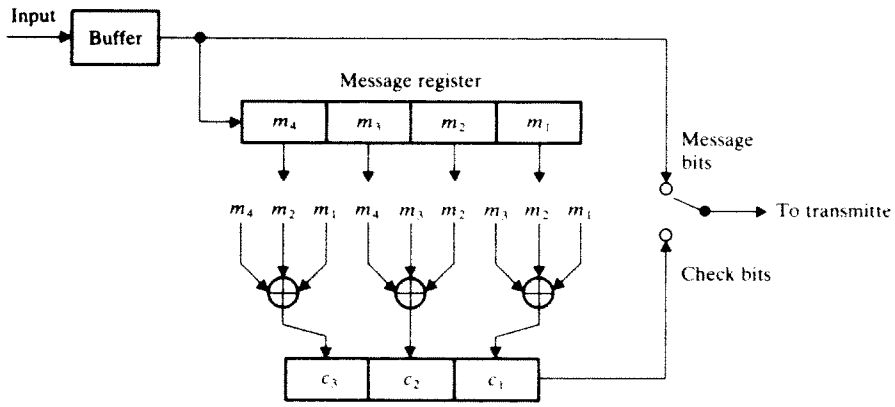


Figure 13.2-1 Encoder for (7, 4) Hamming code.

$(m_1\ m_2\ m_3\ m_4)$, the check bits are determined from the set of equations

$$c_1 = m_1 \oplus m_2 \oplus m_3 \oplus 0$$
$$c_2 = 0 \oplus m_2 \oplus m_3 \oplus m_4$$
$$c_3 = m_1 \oplus m_2 \oplus 0 \oplus m_4$$

These check-bit equations are obtained by substituting the elements of P into Eq. (6).

Figure 13.2-1 depicts an encoder that carries out the check-bit calculations for this (7, 4) Hamming code. Each block of message bits going to the transmitter is also loaded into a message register. The cells of the message register are connected to exclusive-OR gates whose outputs equal the check bits. The check bits are stored in another register and shifted out to the transmitter after the message bits. An input buffer holds the next block of message bits while the check bits are shifted out. The cycle then repeats with the next block of message bits.

Table 13.2-1 lists the resulting $2^4 = 16$ codewords and their weights. The smallest nonzero weight equals 3, confirming that $d_{\min} = 3$.

Table 13.2-1 Codewords for the (7, 4) Hamming code

M	C	$w(X)$	M	C	$w(X)$
0 0 0 0	0 0 0	0	1 0 0 0	1 0 1	3
0 0 0 1	0 1 1	3	1 0 0 1	1 1 0	4
0 0 1 0	1 1 0	3	1 0 1 0	0 1 1	4
0 0 1 1	1 0 1	4	1 0 1 1	0 0 0	3
0 1 0 0	1 1 1	4	1 1 0 0	0 1 0	3
0 1 0 1	1 0 0	3	1 1 0 1	0 0 1	4
0 1 1 0	0 0 1	3	1 1 1 0	1 0 0	4
0 1 1 1	0 1 0	4	1 1 1 1	1 1 1	7

Exercise 13.2-1 Consider a systematic (6, 3) block code generated by the submatrix

$$P = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

Write the check-bit equations and tabulate the codewords and their weights to show that $d_{\min} = 3$.

Syndrome Decoding

Now let Y stand for the received vector when a particular code vector X has been transmitted. Any transmission errors will result in $Y \neq X$. The decoder detects or corrects errors in Y using stored information about the code.

A direct way of performing error detection would be to compare Y with every vector in the code. This method requires storing all 2^k code vectors at the receiver and performing up to 2^k comparisons. But efficient codes generally have large values of k, which implies rather extensive and expensive decoding hardware. As an example, you need $q \geq 5$ to get $R_c \geq 0.8$ with a Hamming code; then $n \geq 31$, $k \geq 26$, and the receiver must store a total of $n \times 2^k > 10^9$ bits!

More practical decoding methods for codes with large k involve parity-check information derived from the code's P submatrix. Associated with any systematic linear (n, k) block code is a $q \times n$ matrix H called the *parity-check matrix*. This matrix is defined by

$$H^T \triangleq \begin{bmatrix} P \\ \hline I_q \end{bmatrix} \tag{8}$$

where H^T denotes the *transpose* of H and I_q is the $q \times q$ identity matrix. Relative to error detection, the parity-check matrix has the crucial property

$$XH^T = (0 \quad 0 \quad \cdots \quad 0) \tag{9}$$

provided that X belongs to the set of code vectors. However, when Y is not a code vector, the product YH^T contains at least one nonzero element.

Therefore, given H^T and a received vector Y, *error detection* can be based on

$$S = YH^T \tag{10}$$

a q-bit vector called the *syndrome*. If all elements of S equal zero, then either Y equals the transmitted vector X and there are no transmission errors, or Y equals some other code vector and the transmission errors are *undetectable*. Otherwise, errors are indicated by the presence of nonzero elements in S. Thus, a decoder for error detection simply takes the form of a syndrome calculator. A comparison of Eqs. (10) and (6) shows that the hardware needed is essentially the same as the encoding circuit.

Error correction necessarily entails more circuitry but it, too, can be based on the syndrome. We develop the decoding method by introducing an n-bit *error vector* E whose nonzero elements mark the positions of transmission errors in Y. For instance, if $X = (1\ 0\ 1\ 1\ 0)$ and $Y = (1\ 0\ 0\ 1\ 1)$ then $E = (0\ 0\ 1\ 0\ 1)$. In general,

$$Y = X + E \tag{11a}$$

and, conversely,

$$X = Y + E \tag{11b}$$

since a second error in the same bit location would cancel the original error. Substituting $Y = X + E$ into $S = YH^T$ and invoking Eq. (9), we obtain

$$S = (X + E)H^T = XH^T + EH^T = EH^T \tag{12}$$

which reveals that the syndrome depends entirely on the error pattern, not the specific transmitted vector.

However, there are only 2^q different syndromes generated by the 2^n possible n-bit error vectors, including the no-error case. Consequently, a given syndrome does not uniquely determine E. Or, putting this another way, we can correct just $2^q - 1$ patterns with one or more errors, and the remaining patterns are *uncorrectable*. We should therefore design the decoder to correct the $2^q - 1$ most likely error patterns—namely those patterns with the *fewest* errors, since single errors are more probable than double errors, and so forth. This strategy, known as *maximum-likelihood decoding*, is optimum in the sense that it minimizes the word-error probability. Maximum-likelihood decoding corresponds to choosing the code vector that has the smallest Hamming distance from the received vector.

To carry out maximum-likelihood decoding, you must first compute the syndromes generated by the $2^q - 1$ most probable error vectors. The *table-lookup decoder* diagrammed in Fig. 13.2-2 then operates as follows. The decoder calculates S from the received vector Y and looks up the assumed error vector $\hat{E}$ stored in the table. The sum $Y + \hat{E}$ generated by exclusive-OR gates finally constitutes

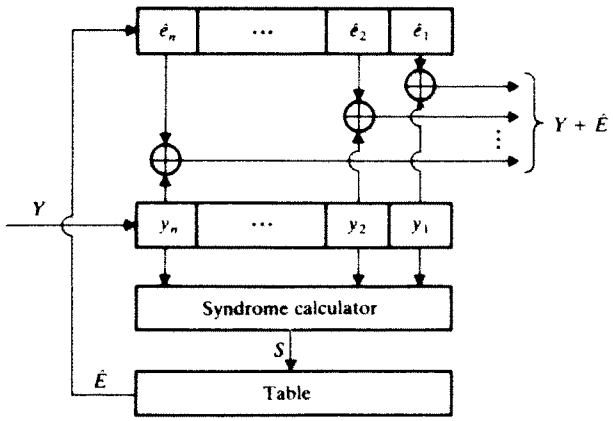


Figure 13.2-2 Table-lookup decoder.

the decoded word. If there are no errors, or if the errors are uncorrectable, then $S = (0\ 0\ \cdots\ 0)$ so $Y + \hat{E} = Y$. The check bits in the last q elements of $Y + \hat{E}$ may be omitted if they are of no further interest.

The relationship between syndromes and error patterns also sheds some light on the design of error-correcting codes, since each of the $2^q - 1$ nonzero syndromes must represent a specific error pattern. Now there are $\binom{n}{1} = n$ single-error patterns for an n-bit word, $\binom{n}{2}$ double-error patterns, and so forth. Hence, if a code is to correct up to t errors per word, q and n must satisfy

$$2^q - 1 \geq n + \binom{n}{2} + \cdots + \binom{n}{t} \tag{13}$$

In the particular case of a single-error-correcting code, Eq. (13) reduces to $2^q - 1 \geq n$. Furthermore, when E corresponds to a single error in the jth bit of a codeword, we find from Eq. (12) that S is identical to the jth row of H^T. Therefore, to provide a distinct syndrome for each single-error pattern and for the no-error pattern, the rows of H^T (or columns of H) must all be different and each must contain at least one nonzero element. The generator matrix of a Hamming code is designed to satisfy this requirement on H, while q and n satisfy $2^q - 1 = n$.

Example 13.2-2 Let's apply table-lookup decoding to a (7, 4) Hamming code used for single-error correction. From Eq. (8) and the P submatrix given in Example 13.2-1, we obtain the 3×7 parity-check matrix

$$H = [P^T \mid I_q] = \left[\begin{array}{cccc|ccc} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{array}\right]$$

There are $2^3 - 1 = 7$ correctable single-error patterns, and the corresponding syndromes listed in Table 13.2-2 follow directly from the columns of H. To accommodate this table the decoder needs to store only $(q + n) \times 2^q = 80$ bits.

Table 13.2-2 Syndromes for the (7, 4) Hamming code

S	$\hat{E}$
0 0 0	0 0 0 0 0 0 0
1 0 1	1 0 0 0 0 0 0
1 1 1	0 1 0 0 0 0 0
1 1 0	0 0 1 0 0 0 0
0 1 1	0 0 0 1 0 0 0
1 0 0	0 0 0 0 1 0 0
0 1 0	0 0 0 0 0 1 0
0 0 1	0 0 0 0 0 0 1

But suppose a received word happens to have two errors, such that $E = (1\ 0\ 0\ 0\ 0\ 1\ 0)$. The decoder calculates $S = YH^T = EH^T = (1\ 1\ 1)$ and the syndrome table gives the assumed single-error pattern $\hat{E} = (0\ 1\ 0\ 0\ 0\ 0\ 0)$. The decoded output word $Y + \hat{E}$ therefore contains *three* errors, the two transmission errors plus the erroneous correction added by the decoder.

If multiple transmission errors per word are sufficiently infrequent, we need not be concerned about the occasional extra errors committed by the decoder. If multiple errors are frequent, a more powerful code would be required. For instance, an *extended* Hamming code has an additional check bit that provides double-error detection along with single-error correction; see Prob. 13.2-12.

Exercise 13.2-2 Use Eqs. (8) and (10) to show that the jth bit of S is given by

$$s_j = y_1 p_{1j} \oplus y_2 p_{2j} \oplus \cdots \oplus y_k p_{kj} \oplus y_{k+j}$$

Then diagram the syndrome-calculation circuit for a (7, 4) Hamming code, and compare it with Fig. 13.2-1.

Cyclic Codes★

The code for a forward-error-correction system must be capable of correcting $t \geq 1$ errors per word. It should also have a reasonably efficient code rate $R_c = k/n$. These two parameters are related by the inequality

$$1 - R_c \geq \frac{1}{n} \log_2 \left[\sum_{i=0}^{t} \binom{n}{i} \right] \tag{14}$$

which follows from Eq. (13) with $q = n - k = n(1 - R_c)$. This inequality underscores the fact that if we want $R_c \approx 1$, we must use codewords with $n \gg 1$ and $k \gg 1$. However, the hardware requirements for encoding and decoding long codewords may be prohibitive unless we impose further structural conditions on the code. *Cyclic codes* are a subclass of linear block codes with a cyclic structure that leads to more practical implementation. Thus, block codes used in FEC systems are almost always cyclic codes.

To describe a cyclic code, we'll find it helpful to change our indexing scheme and express an arbitrary n-bit code vector in the form

$$X = (x_{n-1} \quad x_{n-2} \quad \cdots \quad x_1 \quad x_0) \tag{15}$$

Now suppose that X has been loaded into a *shift register* with feedback connection from the first to last stage. Shifting all bits one position to the left yields the *cyclic shift* of X, written as

$$X' \triangleq (x_{n-2} \quad x_{n-3} \quad \cdots \quad x_1 \quad x_0 \quad x_{n-1}) \tag{16}$$

A second shift produces $X'' = (x_{n-3} \cdots x_1\ x_0\ x_{n-1}\ x_{n-2})$, and so forth. A linear code is *cyclic* if every cyclic shift of a code vector X is another vector in the code.

This cyclic property can be treated mathematically by associating a code vector X with the *polynomial*

$$X(p) = x_{n-1}p^{n-1} + x_{n-2}p^{n-2} + \cdots + x_1 p + x_0 \tag{17}$$

where p is an arbitrary real variable. The powers of p denote the positions of the codeword bits represented by the corresponding coefficients of p. Formally, binary code polynomials are defined in conjunction with *Galois fields*, a branch of modern algebra that provides the theory needed for a complete treatment of cyclic codes. For our informal overview of cyclic codes we'll manipulate code polynomials using ordinary algebra modified in two respects. First, to be in agreement with our earlier definition for the sum of two code vectors, the sum of two polynomials is obtained by *mod-2 addition* of their respective coefficients. Second, since all coefficients are either 0 or 1, and since $1 \oplus 1 = 0$, the *subtraction* operation is the same as mod-2 addition. Consequently, if $X(p) + Z(p) = 0$ then $X(p) = Z(p)$.

We develop the polynomial interpretation of cyclic shifting by comparing

$$pX(p) = x_{n-1}p^n + x_{n-2}p^{n-1} + \cdots + x_1 p^2 + x_0 p$$

with the shifted polynomial

$$X'(p) = x_{n-2}p^{n-1} + \cdots + x_1 p^2 + x_0 p + x_{n-1}$$

If we sum these polynomials, noting that $(x_1 \oplus x_1)p^2 = 0$, etc., we get

$$pX(p) + X'(p) = x_{n-1}p^n + x_{n-1}$$

and hence

$$X'(p) = pX(p) + x_{n-1}(p^n + 1) \tag{18}$$

Iteration yields similar expressions for multiple shifts.

The polynomial $p^n + 1$ and its factors play major roles in cyclic codes. Specifically, an (n, k) cyclic code is defined by a *generator polynomial* of the form

$$G(p) = p^q + g_{q-1}p^{q-1} + \cdots + g_1 p + 1 \tag{19}$$

where $q = n - k$ and the coefficients are such that $G(p)$ is a factor of $p^n + 1$. Each codeword then corresponds to the polynomial product

$$X(p) = Q_M(p)G(p) \tag{20}$$

in which $Q_M(p)$ represents a block of k message bits. All such codewords satisfy the cyclic condition in Eq. (18) since $G(p)$ is a factor of both $X(p)$ and $p^n + 1$.

Any factor of $p^n + 1$ that has degree q may serve as the generator polynomial for a cyclic code, but it does not necessarily generate a good code. Table 13.2-3 lists the generator polynomials of selected cyclic codes that have been demonstrated to possess desirable parameters for FEC systems. The table includes some cyclic Hamming codes, the famous Golay code, and a few members of the important family of BCH codes discovered by Bose, Chaudhuri, and Hocquenghem. The entries under $G(p)$ denote the polynomial's coefficients; thus, for instance, 1 0 1 1 means that $G(p) = p^3 + 0 + p + 1$.

Table 13.2-3 Selected cyclic codes

Type	n	k	R_c	d_{min}	$G(p)$
Hamming codes	7	4	0.57	3	1 011
	15	11	0.73	3	10 011
	31	26	0.84	3	100 101
BCH codes	15	7	0.46	5	111 010 001
	31	21	0.68	5	11 101 101 001
	63	45	0.71	7	1 111 000 001 011 001 111
Golay code	23	12	0.52	7	101 011 100 011

Cyclic codes may be systematic or nonsystematic, depending on the term $Q_M(p)$ in Eq. (20). For a systematic code, we define the message-bit and check-bit polynomials

$$M(p) = m_{k-1}p^{k-1} + \cdots + m_1 p + m_0$$

$$C(p) = c_{q-1}p^{q-1} + \cdots + c_1 p + c_0$$

and we want the codeword polynomials to be

$$X(p) = p^q M(p) + C(p) \tag{21}$$

Equations (20) and (21) therefore require $p^q M(p) + C(p) = Q_M(p)G(p)$, or

$$\frac{p^q M(p)}{G(p)} = Q_M(p) + \frac{C(p)}{G(p)} \tag{22a}$$

This expression says that $C(p)$ equals the *remainder* left over after dividing $p^q M(p)$ by $G(p)$, just as 14 divided by 3 leaves a remainder of 2 since $14/3 = 4 + 2/3$. Symbolically, we write

$$C(p) = \text{rem}\left[\frac{p^q M(p)}{G(p)}\right] \tag{22b}$$

where rem [ ] stands for the remainder of the division within the brackets.

The division operation needed to generate a systematic cyclic code is easily and efficiently performed by the shift-register encoder diagrammed in Fig. 13.2-3.

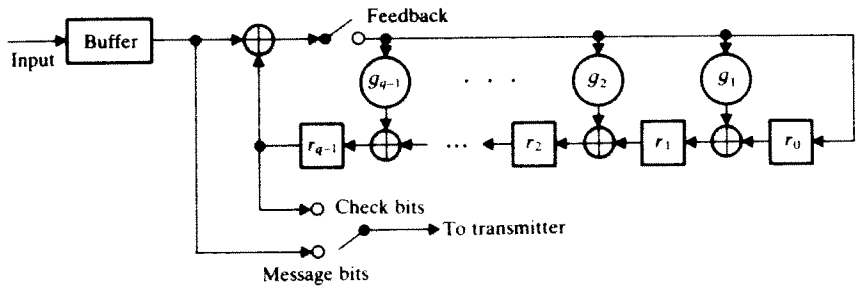


Figure 13.2-3 Shift-register encoder.

Encoding starts with the feedback switch closed, the output switch in the message-bit position, and the register initialized to the all-zero state. The k message bits are shifted into the register and simultaneously delivered to the transmitter. After k shift cycles, the register contains the q check bits. The feedback switch is now opened and the output switch is moved to deliver the check bits to the transmitter.

Syndrome calculation at the receiver is equally simple. Given a received vector Y, the syndrome is determined from

$$S(p) = \text{rem}\left[\frac{Y(p)}{G(p)}\right] \tag{23}$$

If $Y(p)$ is a valid code polynomial, then $G(p)$ will be a factor of $Y(p)$ and $Y(p)/G(p)$ has zero remainder. Otherwise we get a nonzero syndrome polynomial indicating detected errors.

Besides simplified encoding and syndrome calculation, cyclic codes have other advantages over noncyclic block codes. The foremost advantage comes from the ingenious error-correcting decoding methods that have been devised for specific cyclic codes. These methods eliminate the storage needed for table-lookup decoding and thus make it practical to use powerful and efficient codes with $n \gg 1$. Another advantage is the ability of cyclic codes to detect error bursts that span many successive bits. Detailed exposition of these properties are presented in texts such as Lin and Costello (1983).

Example 13.2-3 Consider the cyclic (7, 4) Hamming code generated by $G(p) = p^3 + 0 + p + 1$. We'll use long division to calculate the check-bit polynomial $C(p)$ when $M = (1\ 1\ 0\ 0)$. We first write the message-bit polynomial

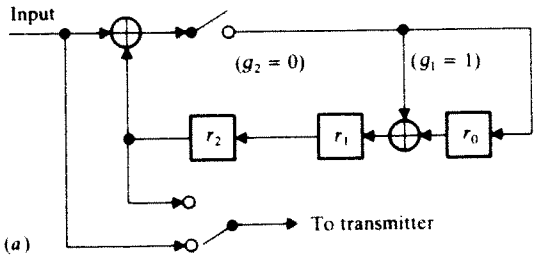
$$M(p) = p^3 + p^2 + 0 + 0$$

so $p^q M(p) = p^3 M(p) = p^6 + p^5 + 0 + 0 + 0 + 0 + 0$. Next, we divide $G(p)$ into $p^q M(p)$, keeping in mind that subtraction is the same as addition in mod-2 arithmetic. Thus,

$$\begin{array}{r|l} & Q_M(p) = p^3 + p^2 + p + 0 \\ p^3 + 0 + p + 1 & p^6 + p^5 + 0 + 0 + 0 + 0 + 0 \\ & \underline{p^6 + 0 + p^4 + p^3} \\ & \quad p^5 + p^4 + p^3 + 0 \\ & \quad \underline{p^5 + 0 + p^3 + p^2} \\ & \quad\quad p^4 + 0 + p^2 + 0 \\ & \quad\quad \underline{p^4 + 0 + p^2 + p} \\ & \quad\quad\quad 0 + 0 + p + 0 \\ & \quad\quad\quad \underline{0 + 0 + 0 + 0} \\ & \quad\quad\quad C(p) = 0 + p + 0 \end{array}$$

so the complete code polynomial is

$$X(p) = p^3 M(p) + C(p) = p^6 + p^5 + 0 + 0 + 0 + p + 0$$



(b)

Input bit m	Register bits before shift r_2	r_1	r_0	Register bits after shift $r_2' = r_1$	$r_1' = r_0 \oplus r_2 \oplus m$	$r_0' = r_2 \oplus m$
1	0	0	0	0	1	1
1	0	1	1	1	0	1
0	1	0	1	0	0	1
0	0	0	1	0	1	0

Figure 13.2-4 (*a*) Shift-register encoder for (7, 4) Hamming code; (*b*) register bits when $M = (1\ 1\ 0\ 0)$.

which corresponds to the codeword

$$X = (1\ \ 1\ \ 0\ \ 0\ \ |\ \ 0\ \ 1\ \ 0)$$

You'll find this codeword back in Table 13.2-1, where you'll also find the cyclic shift $X' = (1\ 0\ 0\ 0\ |\ 1\ 0\ 1)$ and all multiple shifts.

Finally, Fig. 13.2-4 shows the shift-register encoder and the register bits for each cycle of the encoding process when the input is $M = (1\ 1\ 0\ 0)$. After four shift cycles, the register holds $C = (0\ 1\ 0)$—in agreement with our manual division.

Exercise 13.2-3 Let $Y(p) = X(p) + E(p)$ where $E(p)$ is the error polynomial. Use Eqs. (20) and (23) to show that the syndrome polynomial $S(p)$ depends on $E(p)$ but not on $X(p)$.

13.3 CONVOLUTIONAL CODES

Convolutional codes have a structure that effectively extends over the entire transmitted bit stream, rather than being limited to codeword blocks. The convolutional structure is especially well suited to space and satellite communication systems that require simple encoders and achieve high performance by sophisticated decoding methods. Our treatment of this important family of codes consists of selected examples that introduce the salient features of convolutional encoding and decoding.

Convolutional Encoding

The fundamental hardware unit for convolutional encoding is a tapped shift register with $L + 1$ stages, as diagrammed in Fig. 13.3-1. Each tap gain g is a binary digit representing a short-circuit connection or an open circuit. The message bits in the register are combined by mod-2 addition to form the encoded bit

$$x_j = m_{j-L}g_L \oplus \cdots \oplus m_{j-1}g_1 \oplus m_j g_0$$
$$= \sum_{i=0}^{L} m_{j-i}g_i \qquad \text{(mod-2)} \tag{1}$$

The name *convolutional encoding* comes from the fact that Eq. (1) has the form of a *binary convolution*, analogous to the convolutional integral

$$x(t) = \int m(t - \lambda)g(\lambda)\, d\lambda$$

Notice that x_j depends on the current input m_j and on the *state* of the register defined by the previous L message bits. Also notice that a particular message bit influences a span of $L + 1$ successive encoded bits as it shifts through the register.

To provide the extra bits needed for error control, a complete convolutional encoder must generate output bits at a rate greater than the message bit rate r_b. This is achieved by connecting two or more mod-2 summers to the register and interleaving the encoded bits via a commutator switch. For example, the encoder in Fig. 13.3-2 generates $n = 2$ encoded bits

$$x'_j = m_{j-2} \oplus m_{j-1} \oplus m_j \qquad x''_j = m_{j-2} \oplus m_j \tag{2}$$

which are interleaved by the switch to produce the output stream

$$X = x'_1 x''_1 x'_2 x''_2 x'_3 x''_3 \cdots$$

The output bit rate is therefore $2r_b$ and the code rate is $R_c = 1/2$—like an (n, k) block code with $R_c = k/n = 1/2$.

However, unlike a block code, the input bits have not been grouped into words. Instead, each message bit influences a span of $n(L + 1) = 6$ successive output bits. The quantity $n(L + 1)$ is called the *constraint length* measured in terms

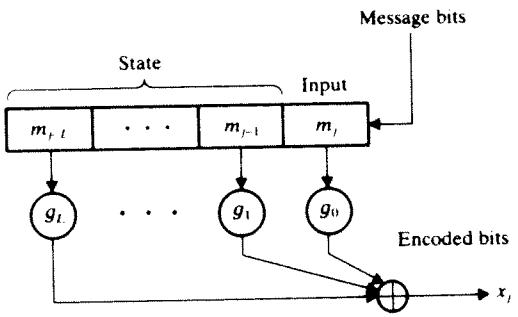


Figure 13.3-1 Tapped shift register for convolutional encoding.

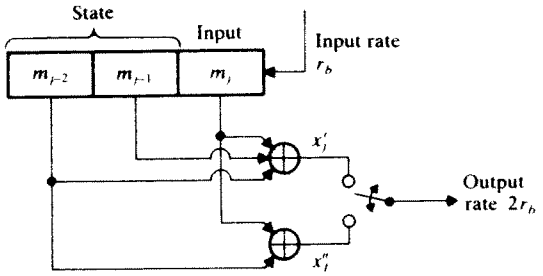


Figure 13.3-2 Convolutional encoder with $n = 2$, $k = 1$, and $L = 2$.

of encoded output bits, whereas L is the encoder's *memory* measured in terms of input message bits. We say that this encoder produces an (n, k, L) convolutional code† with $n = 2$, $k = 1$, and $L = 2$.

Three different but related graphical representations have been devised for the study of convolutional encoding: the code tree, the code trellis, and the state diagram. We'll present each of these for our (2, 1, 2) encoder in Fig. 13.3-2, starting with the code tree. In accordance with normal operating procedure, we presume that the register has been cleared to contain all 0s when the first message bit m_1 arrives. Hence, the initial state is $m_{-1}m_0 = 00$ and Eq. (2) gives the output $x'_1x''_1 = 00$ if $m_1 = 0$ or $x'_1x''_1 = 11$ if $m_1 = 1$.

The *code tree* drawn in Fig. 13.3-3 begins at a branch point or *node* labeled *a* representing the initial state. If $m_1 = 0$, you take the upper branch from node *a* to find the output 00 and the next state, which is also labeled *a* since $m_0 m_1 = 00$ in this case. If $m_1 = 1$, you take the lower branch from *a* to find the output 11 and the next state $m_0 m_1 = 01$ signified by the label *b*. The code tree progressively evolves in this fashion for each new input bit. Nodes are labeled with letters denoting the current state $m_{j-2}m_{j-1}$; you go up or down from a node, depending on the value of m_j; each branch shows the resulting encoded output $x'_j x''_j$ calculated from Eq. (2), and it terminates at another node labeled with the next state. There are 2^j possible branches for the jth message bit, but the branch pattern begins to repeat at $j = 3$ since the register length is $L + 1 = 3$.

Having observed repetition in the code tree, we can construct a more compact picture called the *code trellis* and shown in Fig. 13.3-4*a*. Here, the nodes on the left denote the four possible current states, while those on the right are the resulting next states. A solid line represents the state transition or branch for $m_j = 0$, and a broken line represents the branch for $m_j = 1$. Each branch is labeled with the resulting output bits $x'_j x''_j$. Going one step further, we coalesce the left and right sides of the trellis to obtain the *state diagram* in Fig. 13.3-4*b*. The self-loops at nodes *a* and *d* represent the state transitions *a*-*a* and *d*-*d*.

Given a sequence of message bits and the initial state, you can use either the code trellis or state diagram to find the resulting state sequence and output bits. The procedure is illustrated in Fig. 13.3-4*c*, starting at initial state *a*.

† Notation for convolutional codes has not been standardized and varies from author to author, as does the definition of constraint length.

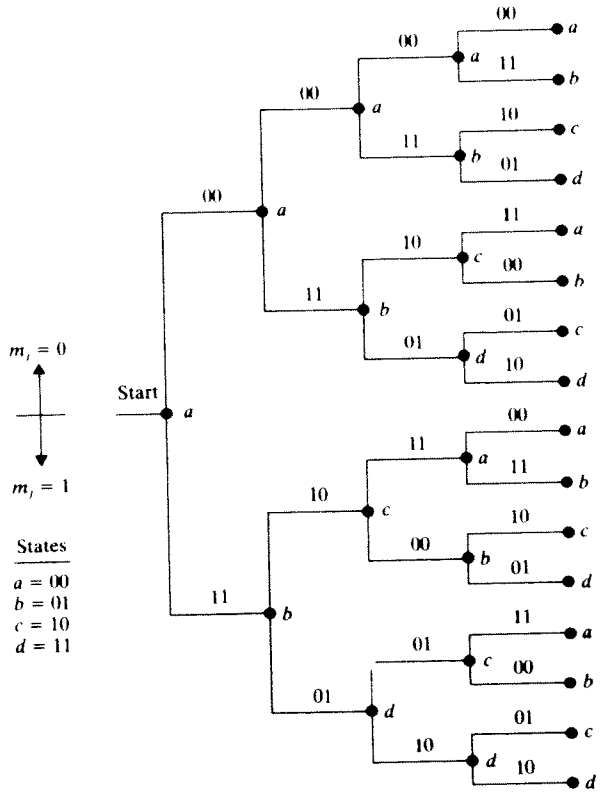


Figure 13.3-3 Code tree for (2, 1, 2) encoder.

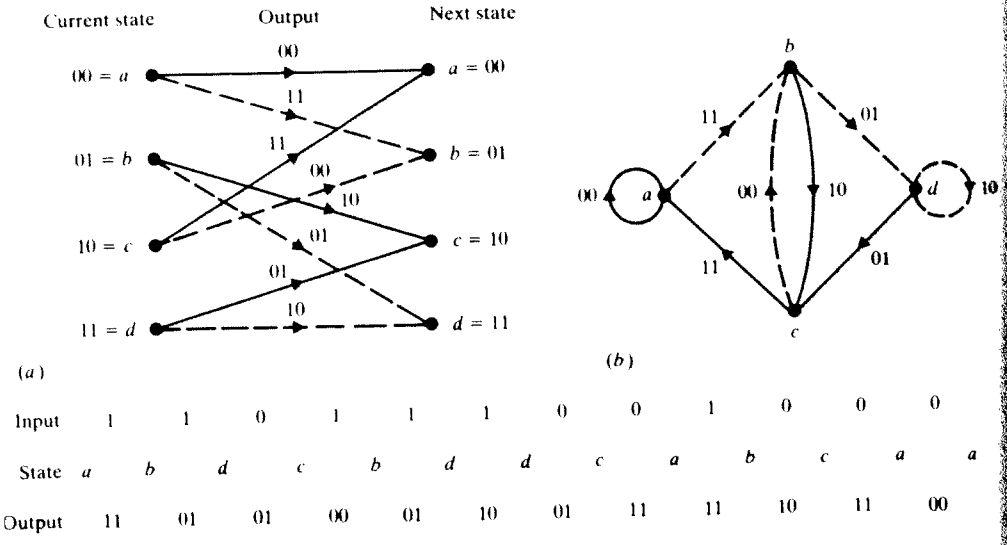


Figure 13.3-4 (*a*) Code trellis; (*b*) state diagram for (2, 1, 2) encoder.

Numerous other convolutional codes are obtained by modifying the encoder in Fig. 13.3-2. If we just change the connections to the mod-2 summers, then the code tree, trellis, and state diagram retain the same structure since the states and branching pattern reflect only the register contents. The output bits would be different, of course, since they depend specifically on the summer connections.

If we extend the shift register to an arbitrary length $L + 1$ and connect it to $n \geq 2$ mod-2 summers, we get an (n, k, L) convolutional code with $k = 1$ and code rate $R_c = 1/n \leq 1/2$. The state of the encoder is defined by L previous input bits, so the code trellis and state diagram have 2^L different states, and the code-tree pattern repeats after $j = L + 1$ branches. Connecting one commutator terminal directly to the first stage of the register yields the encoded bit stream

$$X = m_1 x_1'' x_1''' \cdots m_2 x_2'' x_2''' \cdots m_3 x_3'' x_3''' \cdots \tag{3}$$

which defines a *systematic* convolutional code with $R_c = 1/n$.

Code rates higher than $1/n$ require $k \geq 2$ shift registers and an input distributor switch. This scheme is illustrated by the (3, 2, 1) encoder in Fig. 13.3-5. The message bits are distributed alternately between $k = 2$ registers, each of length $L + 1 = 2$. We regard the pair of bits $m_{j-1}m_j$ as the current input, while the pair $m_{j-3}m_{j-2}$ constitute the state of the encoder. For each input pair, the mod-2 summers generate $n = 3$ encoded output bits given by

$$x_j' = m_{j-3} \oplus m_{j-2} \oplus m_j \qquad x_j'' = m_{j-3} \oplus m_{j-1} \oplus m_j$$
$$x_j''' = m_{j-2} \oplus m_j \tag{4}$$

Thus, the output bit rate is $3r_b/2$, corresponding to the code rate $R_c = k/n = 2/3$. The constraint length is $n(L + 1) = 6$, since a particular input bit influences a span of $n = 3$ output bits from each of its $L + 1 = 2$ register positions.

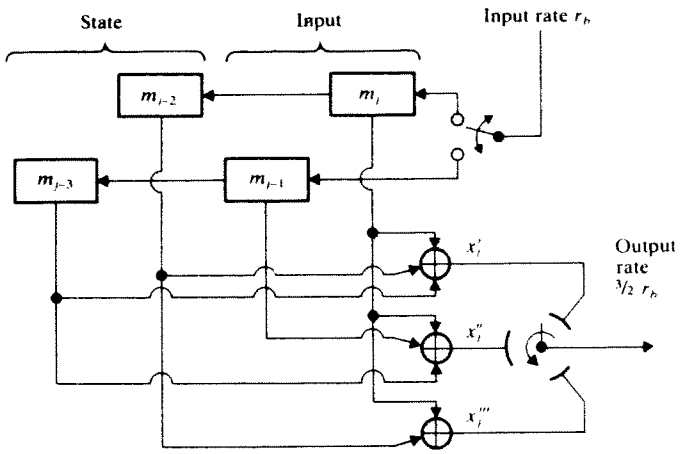


Figure 13.3-5 (3, 2, 1) encoder.

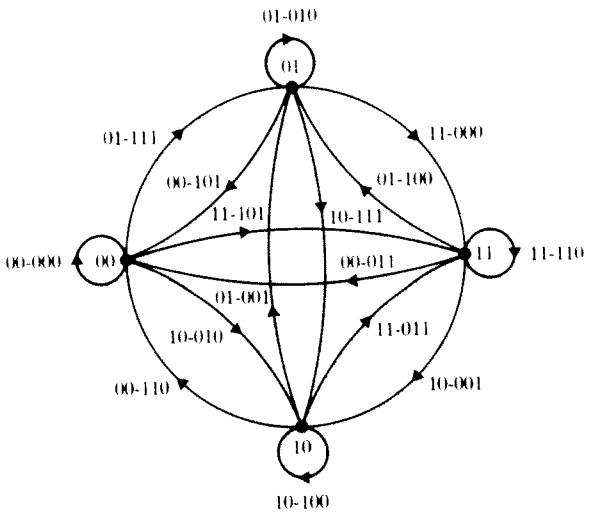


Figure 13.3-6 State diagram for (3, 2, 1) encoder.

Graphical representation becomes more cumbersome for convolutional codes with $k > 1$ because we must deal with input bits in groups of 2^k. Consequently, 2^k branches emanate and terminate at each node, and there are 2^{kL} different states. As an example, Fig. 13.3-6 shows the state diagram for the (3, 2, 1) encoder in Fig. 13.3-5. The branches are labeled with the $k = 2$ input bits followed by the resulting $n = 3$ output bits.

The convolutional codes employed for FEC systems usually have small values of n and k, while the constraint length typically falls in the range of 10 to 30. All convolutional encoders require a commutator switch at the output, as shown in Figs. 13.3-2 and 13.3-5. For codes with $k > 1$, the input distributor switch can be eliminated by using a single register of length kL and shifting the bits in groups of k. In any case, convolutional encoding hardware is simpler than the hardware for block encoding since message bits enter the register unit at a steady rate r_b and an input buffer is not needed.

Exercise 13.3-1 Consider a systematic (3, 1, 3) convolutional code. List the possible states and determine the state transitions produced by $m_j = 0$ and $m_j = 1$. Then construct and label the state diagram, taking the encoded output bits to be m_j, $m_{j-2} \oplus m_j$, and $m_{j-3} \oplus m_{j-1}$. (See Fig. P13.3-4 for a convenient eight-state pattern.)

Free Distance and Coding Gain

We previously found that the error-control power of a block code depends upon its minimum distance, determined from the weights of the codewords. A convolutional code does not subdivide into codewords, so we consider instead the weight $w(X)$ of an entire transmitted sequence X generated by some message sequence.

The *free distance* of a convolutional code is then defined to be

$$d_f \triangleq [w(X)]_{\min} \qquad X \neq 000 \cdots \tag{5}$$

The value of d_f serves as a measure of error-control power.

It would be an exceedingly dull and tiresome task to try to evaluate d_f by listing all possible transmitted sequences. Fortunately, there's a better way based on the normal operating procedure of appending a "tail" of 0s at the end of a message to clear the register unit and return the encoder to its initial state. This procedure eliminates certain branches from the code trellis for the last L transitions.

Take the code trellis in Fig. 13.3-4*a*, for example. To end up at state *a*, the next-to-last state must be either *a* or *c* so the last few branches of any transmitted sequence X must follow one of the paths shown in Fig. 13.3-7. Here the final state is denoted by *e*, and each branch has been labeled with the number of 1s in the encoded bits—which equals the *weight* associated with that branch. The total weight of a transmitted sequence X equals the sum of the branch weights along the path of X. In accordance with Eq. (5), we seek the path that has the smallest branch-weight sum, other than the trivial all-zero path.

Looking backwards $L + 1 = 3$ branches from *e*, we locate the last path that emanates from state *a* before terminating at *e*. Now suppose all earlier transitions followed the all-zero path along the top line, giving the state sequence $aa \cdots abce$. Since an *a-a* branch has weight 0, this state sequence corresponds to a *minimum-weight nontrivial path*. We therefore conclude that $d_f = 0 + 0 + \cdots + 0 + 2 + 1 + 2 = 5$. There are other minimum-weight paths, such as $aa \cdots abcae$ and $aa \cdots abcbce$, but no nontrivial path has less weight than $d_f = 5$.

Another approach to the calculation of free distance involves the *generating function* of a convolutional code. The generating function may be viewed as the transfer function of the encoder with respect to state transitions. Thus, instead of relating the input and output bits streams by *convolution*, the generating function relates the initial and final states by *multiplication*. Generating functions provide important information about code performance, including the free distance and decoding error probability.

We'll develop the generating function for our (2, 1, 2) encoder using the modified state diagram in Fig. 13.3-8*a*. This diagram has been derived from Fig.

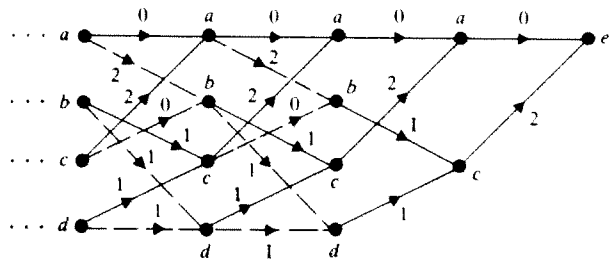


Figure 13.3-7 Termination of (2, 1, 2) code trellis.

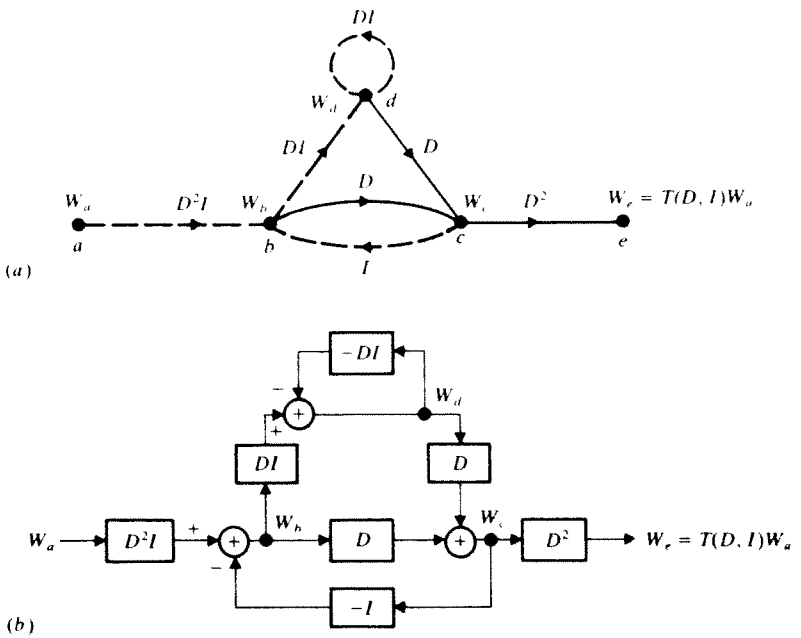


Figure 13.3-8 (*a*) Modified state diagram for (2, 1, 2) encoder; (*b*) equivalent block diagram.

13.3-4*b* with four modifications. First, we've eliminated the *a*-*a* loop which contributes nothing to the weight of a sequence X. Second, we've drawn the *c*-*a* branch as the final *c*-*e* transition. Third, we've assigned a *state variable* W_a at node *a*, and likewise at all other nodes. Fourth, we've labeled each branch with two "gain" variables D and I such that the exponent of D equals the branch weight (as in Fig. 13.3-7), while the exponent of I equals the corresponding number of nonzero message bits (as signified by the solid or dashed branch line). For instance, since the *c*-*e* branch represents $x_j' x_j'' = 11$ and $m_j = 0$, it is labeled with $D^2 I^0 = D^2$. This exponential trick allows us to perform *sums* by *multiplying* the D and I terms, which will become the independent variables of the generating function.

Our modified state diagram now looks like a *signal-flow graph* of the type sometimes used to analyze feedback systems. Specifically, if we treat the nodes as summing junctions and the DI terms as branch gains, then Fig. 13.3-8*a* represents the set of algebraic *state equations*

$$\begin{aligned} W_b &= D^2 I W_a + I W_c \qquad & W_c &= D W_b + D W_d \\ W_d &= DI W_b + DI W_d \qquad & W_e &= D^2 W_c \end{aligned} \tag{6a}$$

The encoder's generating function $T(D, I)$ can now be defined by the *input-output equation*

$$T(D, I) \triangleq W_e / W_a \tag{6b}$$

These equations are also equivalent to the *block diagram* in Fig. 13.3-8*b*, which further emphasizes the relationships between the state variables, the branch gains, and the generating function. Note that minus signs have been introduced here so that the two feedback paths *c*-*b* and *d*-*d* correspond to negative feedback.

Next, the expression for $T(D, I)$ is obtained by algebraic solution of Eq. (6), or by block-diagram reduction of Fig. 13.3-8*b* using the transfer-function relations for parallel, cascade, and feedback connections in Fig. 3.1-8. (If you know *Mason's rule*, you could also apply it to Fig. 13.3-8*a*.) Any of these methods produces the final result

$$T(D, I) = \frac{D^5 I}{1 - 2DI} \tag{7a}$$

$$= D^5 I + 2D^6 I^2 + 4D^7 I^3 + \cdots$$

$$= \sum_{d=5}^{\infty} 2^{d-5} D^d I^{d-4} \tag{7b}$$

where we've expanded $(1 - 2DI)^{-1}$ to get the series in Eq. (7*b*). Keeping in mind that $T(D, I)$ represents all possible transmitted sequences that terminate with a *c*-*e* transition, Eq. (7*b*) has the following interpretation: for any $d \geq 5$, there are exactly 2^{d-5} valid paths with weight $w(X) = d$ that terminate with a *c*-*e* transition, and those paths are generated by messages containing $d - 4$ nonzero bits. The smallest value of $w(X)$ is the free distance, so we again conclude that $d_f = 5$.

As a generalization of Eq. (7), the generating function for an arbitrary convolutional code takes the form

$$T(D, I) = \sum_{d=d_f}^{\infty} \sum_{i=1}^{\infty} A(d, i) D^d I^i \tag{8}$$

Here, $A(d, i)$ denotes the number of different input-output paths through the modified state diagram that have weight d and are generated by messages containing i nonzero bits.

Now consider a received sequence $Y = X + E$, where E represents transmission errors. The path of Y then diverges from the path of X and may or may not be a valid path for the code in question. When Y does not correspond to a valid path, a *maximum-likelihood decoder* should seek out the valid path that has the smallest Hamming distance from Y. Before describing how such a decoder might be implemented, we'll state the relationship between generating functions, free distance, and error probability in maximum-likelihood decoding of convolutional codes.

If transmission errors occur with equal and independent probability α per bit, then the probability of a decoded message-bit error is upper-bounded by

$$P_{be} \leq \frac{1}{k} \left. \frac{\partial T(D, I)}{\partial I} \right|_{D = 2\sqrt{\alpha(1-\alpha)},\ I = 1} \tag{9}$$

The derivation of this bound is given in Lin and Costello (1983, chap. 11) or Viterbi and Omura (1979, chap. 4). When α is sufficiently small, series expansion

of $T(D, I)$ yields the approximation

$$P_{be} \approx \frac{M(d_f)}{k} 2^{d_f} \alpha^{d_f/2} \qquad \sqrt{\alpha} \ll 1 \tag{10}$$

where

$$M(d_f) = \sum_{i=1}^{\infty} iA(d_f, i)$$

The quantity $M(d_f)$ simply equals the total number of nonzero message bits over all minimum-weight input-output paths in the modified state diagram.

Equation (10) supports our earlier assertion that the error-control power of a convolutional code depends upon its free distance. For a performance comparison with uncoded transmission, we'll make the usual assumption of gaussian white noise and $(S/N)_R = 2R_c \gamma_b \geq 10$ so Eq. (10), Sect. 13.1, gives the transmission error probability

$$\alpha \approx (4\pi R_c \gamma_b)^{-1/2} e^{-R_c \gamma_b}$$

The decoded error probability then becomes

$$P_{be} \approx \frac{M(d_f) 2^{d_f}}{k(4\pi R_c \gamma_b)^{d_f/4}} e^{-(R_c d_f/2)\gamma_b} \tag{11}$$

whereas uncoded transmission would yield

$$P_{ube} \approx \frac{1}{(4\pi\gamma_b)^{1/2}} e^{-\gamma_b} \tag{12}$$

Since the exponential terms dominate in these expressions, we see that convolutional coding improves reliability when $R_c d_f/2 > 1$. Accordingly, the quantity $R_c d_f/2$ is known as the *coding gain*, usually expressed in dB.

Explicit design formulas for d_f do not exist, unfortunately, so good convolutional codes must be discovered by computer search and simulation. Table 13.3-1 lists the maximum free distance and coding gain of convolutional codes for selected values of n, k, and L. Observe that the free distance and coding gain increase

Table 13.3-1 Maximum free distance and coding gain of selected convolutional codes

n	k	R_c	L	d_f	$R_c d_f/2$
4	1	1/4	3	13	1.63
3	1	1/3	3	10	1.68
2	1	1/2	3	6	1.50
			6	10	2.50
			9	12	3.00
3	2	2/3	3	7	2.33
4	3	3/4	3	8	3.00

with increasing memory L when the code rate R_c is held fixed. All listed codes are nonsystematic; a systematic convolutional code has a smaller d_f than an optimum nonsystematic code with the same rate and memory.

Example 13.3-1 The (2, 1, 2) encoder back in Fig. 13.3-2 has $T(D, I) = D^5 I/(1 - 2DI)$, so $\partial T(D, I)/\partial I = D^5/(1 - 2DI)^2$. Equation (9) therefore gives

$$P_{be} \leq \frac{2^5[\alpha(1-\alpha)]^{5/2}}{[1 - 4\sqrt{\alpha(1-\alpha)}]^2} \approx 2^5 \alpha^{5/2}$$

and the small-α approximation agrees with Eq. (10). Specifically, in Fig. 13.3-8*a* we find just one minimum-weight nontrivial path *abce*, which has $w(X) = 5 = d_f$ and is generated by a message containing one nonzero bit, so $M(d_f) = 1$.

If $\gamma_b = 10$, then $R_c \gamma_b = 5$, $\alpha \approx 8.5 \times 10^{-4}$, and maximum-likelihood decoding yields $P_{be} \approx 6.7 \times 10^{-7}$, as compared with $P_{ube} \approx 4.1 \times 10^{-6}$. This rather small reliability improvement agrees with the small coding gain $R_c d_f/2 = 5/4$.

Exercise 13.3-2 Let the connections to the mod-2 summers in Fig. 13.3-2 be changed such that $x'_j = m_j$ and $x''_j = m_{j-2} \oplus m_{j-1} \oplus m_j$.

(*a*) Construct the code trellis and modified state diagram for this systematic code. Show that there are two minimum-weight paths in the state diagram, and that $d_f = 4$ and $M(d_f) = 3$. It is not necessary to find $T(D, I)$.

(*b*) Now assume $\gamma_b = 10$. Calculate α, P_{be}, and P_{ube}. What do you conclude about the performance of a convolutional code when $R_c d_f/2 = 1$?

Decoding Methods

There are three generic methods for decoding convolutional codes. At one extreme, the Viterbi algorithm executes *maximum-likelihood decoding* and achieves optimum performance but requires extensive hardware for computation and storage. At thd other extreme, *feedback decoding* sacrifices performance in exchange for simplified hardware. Between these extremes, *sequential decoding* approaches optimum performance to a degree that depends upon the decoder's complexity. We'll describe how these methods work with a (2, 1, L) code. The extension to other codes is conceptually straightforward, but becomes messy to portray for $k > 1$.

Recall that a maximum-likelihood decoder must examine an entire received sequence Y and find a valid path that has the smallest Hamming distance from Y. However, there are 2^N possible paths for an arbitrary message sequence of N bits (or Nn/k bits in Y), so an exhaustive comparison of Y with all valid paths would be an absurd task in the usual case of $N \gg 1$. The *Viterbi algorithm* applies maximum-likelihood principles to limit the comparison to 2^{kL} *surviving paths*, independent of N, thereby bringing maximum-likelihood decoding into the realm of feasibility.

A Viterbi decoder assigns to each branch of each surviving path a *metric* that equals its Hamming distance from the corresponding branch of Y. (We assume here that 0s and 1s have the same transmission-error probability; if not, the branch metric must be redefined to account for the differing probabilities.) Summing the branch metrics yields the path metric, and Y is finally decoded as the surviving path with the smallest metric. To illustrate the metric calculations and explain how surviving paths are selected, we'll walk through an example of Viterbi decoding.

Suppose that our (2, 1, 2) encoder is used at the transmitter, and the received sequence starts with $Y = 11\ 01\ 11$. Figure 13.3-9 shows the first three branches of the valid paths emanating from the initial node a_0 in the code trellis. The number in parentheses beneath each branch is the branch metric, obtained by counting the differences between the encoded bits and the corresponding bits in Y. The circled number at the right-hand end of each branch is the running path metric, obtained by summing branch metrics from a_0. For instance, the metric of the path $a_0\, b_1 c_2 b_3$ is $0 + 2 + 2 = 4$.

Now observe that another path $a_0 a_1 a_2 b_3$ also arrives at node b_3 and has a smaller metric $2 + 1 + 0 = 3$. Regardless of what happens subsequently, this path will have a smaller Hamming distance from Y than the other path arriving at b_3 and is therefore more likely to represent the actual transmitted sequence. Hence, we discard the larger-metric path, marked by an X, and we declare the path with the smaller metric to be the survivor at this node. Likewise, we discard the larger-metric paths arriving at nodes a_3, c_3, and d_3, leaving a total of $2^{kL} = 4$ surviving paths. The fact that none of the surviving path metrics equals zero indicates the presence of *detectable errors* in Y.

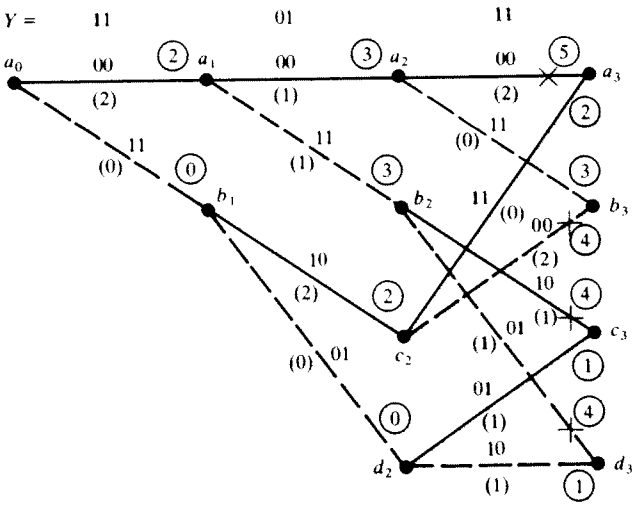


Figure 13.3-9

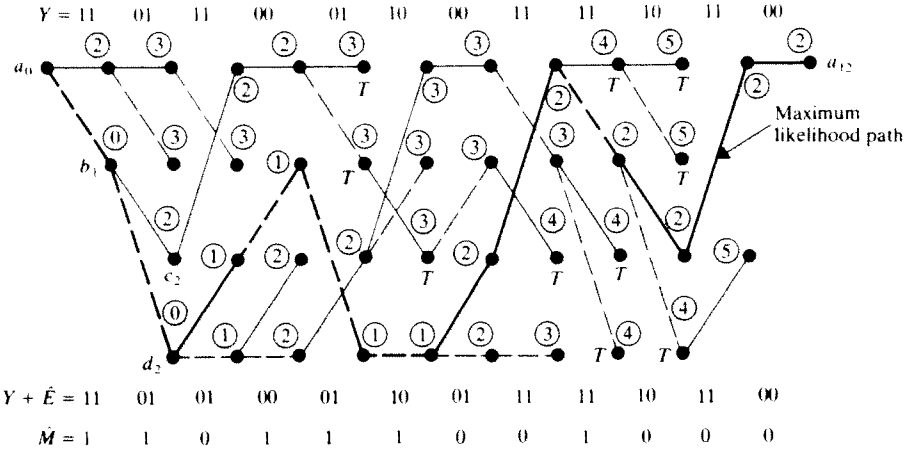


Figure 13.3-10 Illustration of the Viterbi algorithm for maximum-likelihood decoding.

Figure 13.3-10 depicts the continuation of Fig. 13.3-9 for a complete message of $N = 12$ bits, including tail 0s. All discarded branches and all labels except the running path metrics have been omitted for the sake of clarity. The letter T under a node indicates that the two arriving paths had equal running matrics, in which case we just flip a coin to choose the survivor (why?). The maximum-likelihood path follows the heavy line from a_0 to a_{12}, and the final value of the path metric signifies at least two transmission errors in Y. The decoder assumes the corresponding transmitted sequence $Y + \hat{E}$ and message sequence $\hat{M}$ written below the trellis.

A Viterbi decoder must calculate two metrics for each node and store 2^{kL} surviving paths, each consisting of N branches. Hence, decoding complexity increases exponentially with L and linearly with N. The exponential factor limits practical application of the Viterbi algorithm to codes with small values of L.

When $N \gg 1$, storage requirements can be reduced by a truncation process based on the following metric-divergence effect: if two surviving paths emanate from the same node at some point, then the running metric of the less likely path tends to increase more rapidly than the metric of the other survivor within about $5L$ branches from the common node. This effect appears several times in Fig. 13.3-10; consider, for instance, the two paths emanating from node b_1. Hence, decoding need not be delayed until the end of the transmitted sequence. Instead, the first k message bits can be decoded and the first set of branches can be deleted from memory after the first $5Ln$ received bits have been processed. Successive groups of k message bits are then decoded for each additional n bits received thereafter.

Sequential decoding, which was invented before the Viterbi algorithm, also relies on the metric-divergence effect. A simplified version of the sequential algorithm is illustrated in Fig. 13.3-11*a*, using the same trellis, received sequence, and

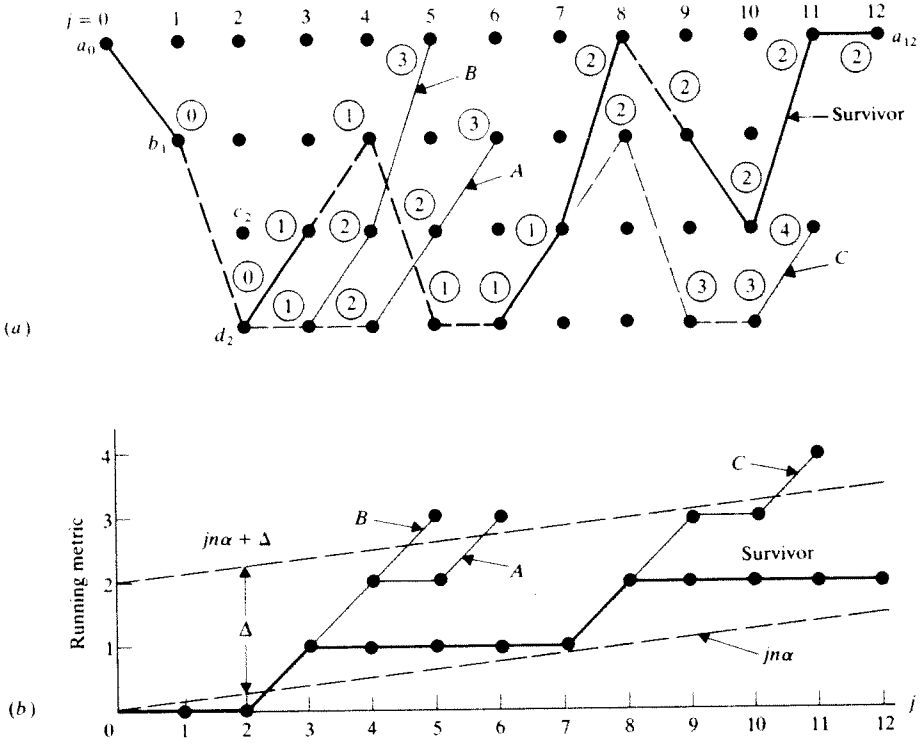


Figure 13.3-11 Illustration of sequential decoding.

metrics as in Fig. 13.3-10. Starting at a_0, the sequential decoder pursues a single path by taking the branch with the smallest branch metric at each successive node. If two or more branches from one node have the same metric, such as at node d_2, the decoder selects one at random and continues on. Whenever the current path happens to be unlikely, the running metric rapidly increases and the decoder eventually decides to go back to a lower-metric node and try another path. There are three of these abandoned paths in our example. Even so, a comparison with Fig. 13.3-10 shows that sequential decoding involves less computation than Viterbi decoding.

The decision to backtrack and try again is based on the expected value of the running metric at a given node. Specifically, if α is the transmission error probability per bit, then the expected running metric at the jth node of the correct path equals $jn\alpha$, the expected number of bit errors in Y at that point. The sequential decoder abandons a path when its metric exceeds some specified *threshold* Δ above $jn\alpha$. If no path survives the threshold test, the value of Δ is increased and the decoder backtracks again. Figure 13.3-11*b* plots the running metrics versus j, along with $jn\alpha$ and the threshold line $jn\alpha + \Delta$ for $\alpha = 1/16$ and $\Delta = 2$.

Sequential decoding approaches the performance of maximum-likelihood decoding when the threshold is loose enough to permit exploration of all probable paths. However, the frequent backtracking requires more computations and results in a decoding delay significantly greater than Viterbi decoding. A tighter threshold reduces computations and decoding delay but may actually eliminate the most probable path, thereby increasing the output error probability compared to that of maximum-likelihood decoding with the same coding gain. As compensation, sequential decoding permits practical application of convolutional codes with large L and large coding gain since the decoder's complexity is essentially independent of L.

We've described sequential decoding and Viterbi decoding in terms of *algorithms* rather than block diagrams of hardware. Indeed, these methods are usually implemented as software for a computer or microprocessor that performs the metric calculations and stores the path data. When circumstances preclude algorithmic decoding, and a higher error probability is tolerable, *feedback decoding* may be the appropriate method. A feedback decoder acts in general like a "sliding block decoder" that decodes message bits one by one based on a block of L or more successive tree branches. We'll focus on the special class of feedback decoding that employs *majority logic* to achieve the simplest hardware realization of a convolutional decoder.

Consider a message sequence $M = m_1 m_2 \cdots$ and the *systematic* (2, 1, L) encoded sequence

$$X = x_1' x_1'' x_2' x_2'' \cdots \tag{13a}$$

where

$$x_j' = m_j \qquad x_j'' = \sum_{i=0}^{L} m_{j-i} g_j \qquad \text{(mod-2)} \tag{13b}$$

We'll view the entire sequence X as a codeword of indefinite length. Then, borrowing from the matrix representation used for block codes, we'll define a *generator matrix* G and a *parity-check matrix* H such that

$$X = MG \qquad XH^T = 0 \quad 0 \quad \cdots$$

To represent Eq. (13), G must be a semi-infinite matrix with a diagonal structure given by

$$G = \begin{bmatrix} 1 & g_0 & 0 & g_1 & 0 & \cdots & 0 & g_L & & & \\ & 1 & g_0 & 0 & g_1 & 0 & \cdots & 0 & g_L & & \\ & & \cdot & \cdot & \cdot & & & & \cdot & & \\ & & & \cdot & \cdot & \cdot & & & & \cdot & \\ & & & & \cdot & \cdot & & \cdot & & & \cdot \end{bmatrix} \tag{14a}$$

This matrix extends indefinitely to the right and down, and the triangular blank spaces denote elements that equal zero. The parity-check matrix is

$$H = \begin{bmatrix} g_0 & 1 & & & & & & & \\ g_1 & 0 & g_0 & 1 & & & & & \\ \cdot & \cdot & g_1 & 0 & g_0 & 1 & & & \\ \cdot & \cdot & \cdot & \cdot & & & \cdot & \cdot & \\ \cdot & \cdot & \cdot & \cdot & & & & \cdot & \cdot \\ g_L & 0 & \cdot & \cdot & & & & & \cdot \\ & & g_L & 0 & & & & & \\ & & & & \cdot & \cdot & & & \\ & & & & & & \cdot & \cdot & \\ & & & & & & & & \cdot \end{bmatrix} \tag{14b}$$

which also extends indefinitely to the right and down.

Next, let E be the transmission error pattern in a received sequence $Y = X + E$. We'll write these sequences as

$$Y = y'_1 y''_1 y'_2 y''_2 \cdots \qquad E = e'_1 e''_1 e'_2 e''_2 \cdots$$

so that $y'_j = m_j \oplus e'_j$. Hence, given the error bit e'_j, the jth message bit is

$$m_j = y'_j \oplus e'_j \tag{15}$$

A feedback decoder estimates errors from the *syndrome* sequence

$$S = YH^T = (X + E)H^T = EH^T$$

Using Eq. (14*b*) for H, the jth bit of S is

$$s_j = \sum_{i=0}^{L} y'_{j-i} g_i \oplus y''_j = \sum_{i=0}^{L} e'_{j-i} g_i \oplus e''_j \tag{16}$$

where the sums are mod-2 and it's understood that $y'_{j-i} = e'_{j-i} = 0$ for $j - i \leq 0$.

As a specific example, take a (2, 1, 6) encoder with $g_0 = g_2 = g_5 = g_6 = 1$ and $g_1 = g_3 = g_4 = 0$, so

$$s_j = y'_{j-6} \oplus y'_{j-5} \oplus y'_{j-2} \oplus y'_j \oplus y''_j \tag{17a}$$

$$= e'_{j-6} \oplus e'_{j-5} \oplus e'_{j-2} \oplus e'_j \oplus e''_j \tag{17b}$$

Equation (17*a*) leads directly to the shift-register circuit for syndrome calculation diagrammed in Fig. 13.3-12. Equation (17*b*) is called a *parity-check sum* and will leads us eventually to the remaining portion of the feedback decoder.

To that end, consider the parity-check table in Fig. 13.3-13*a* where checks indicate which error bits appear in the sums s_{j-6}, s_{j-4}, s_{j-1}, and s_j. This table brings out the fact that e'_{j-6} is checked by all four of the listed sums, while no other error bit is checked by more than one. Accordingly, this set of check sums is said to be *orthogonal* on e'_{j-6}. The tap gains of the encoder were carefully chosen to obtain orthogonal check sums.

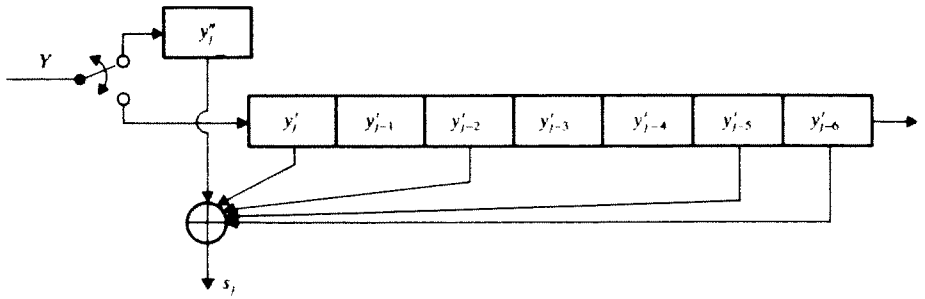


Figure 13.3-12 Shift-register circuit for syndrome calculation for a systematic (2, 1, 6) code.

	e'_{j-12}	e'_{j-11}	e'_{j-10}	e'_{j-9}	e'_{j-8}	e'_{j-7}	e'_{j-6}	e'_{j-6}	e'_{j-5}	e'_{j-4}	e''_{j-4}	e'_{j-3}	e'_{j-2}	e'_{j-1}	e''_{j-1}	e'_j	e''_j
s_{j-6}	√	√			√		√	√									
s_{j-4}			√	√			√			√	√						
s_{j-1}						√	√					√		√	√		
s_j							√		√				√			√	√

Figure 13.3-13 Parity-check table for a systematic (2, 1, 6) code.

When the transmission error probability is reasonably small, we expect to find at most one or two errors in the 17 transmitted bits represented by the parity-check table. If one of the errors corresponds to $e'_{j-6} = 1$, then the four check sums will contain three or four 1s. Otherwise, the check sums contain less than three 1s. Hence, we can apply these four check sums to a *majority-logic gate* to generate the most likely estimate of e'_{j-6}.

Figure 13.3-14 diagrams a complete majority-logic feedback decoder for our systematic (2, 1, 6) code. The syndrome calculator from Fig. 13.3-12 has two

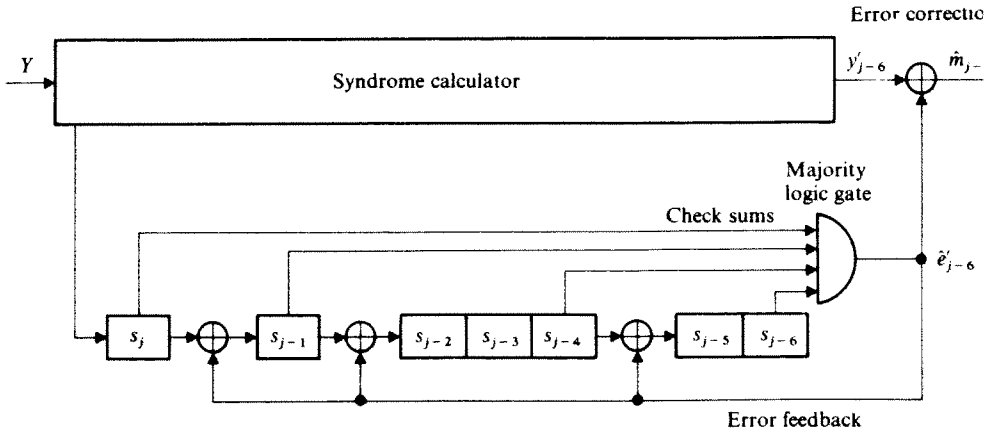


Figure 13.3-14 Majority-logic feedback decoder for a systematic (2, 1, 6) code.

outputs, y'_{j-6} and s_j. The syndrome bit goes into another shift register with taps that connect the check sums to the majority-logic gate, whose output equals the estimated error $\hat{e}'_{j-6}$. The mod-2 addition $y'_{j-6} \oplus \hat{e}'_{j-6} = \hat{m}_{j-6}$ carries out error correction based on Eq. (15). The error is also fed back to the syndrome register to improve the reliability of subsequent check sums. This feedback path accounts for the name *feedback* decoding.

Our example decoder can correct any single-error or double-error pattern in six consecutive message bits. However, more than two transmission errors produces erroneous corrections and error propagation via the feedback path. These effects result in a higher output error than that of maximum-likelihood decoding. See Lin and Costello (1983, chap. 13) for the error analysis and further treatment of majority-logic decoding.

13.4 PROBLEMS

13.1-1 Calculate the probabilities that a word has no errors, detected errors, and undetected errors when a parity-check code with $n = 4$ is used and $\alpha = 0.1$.

13.1-2 Do Prob. 13.1-1 with $n = 9$ and $\alpha = 0.05$.

13.1-3 Consider the square-array code in Fig. 13.1-1. (*a*) Confirm that if a word has two errors, then they can be detected but not corrected. (*b*) Discuss what happens when a word contains three errors.

13.1-4 An FEC system contaminated by gaussian white noise must achieve $P_{be} \leq 10^{-4}$ with minimum transmitted power. Three block codes under consideration have the following parameters:

n	k	$d_{\min}$
31	26	3
31	21	5
31	16	7

Determine which code should be used, and calculate the power saving in dB compared to uncoded transmission.

13.1-5 Do Prob. 13.1-4 with $P_{be} \leq 10^{-6}$.

13.1-6 Calculate α, P_{be}, and P_{ube} at $\gamma_b = 2$, 5, and 10 for an FEC system with gaussian white noise using a (31, 26) block code having $d_{\min} = 3$. Plot your results in a form like Fig. 13.1-5.

13.1-7 Do Prob. 13.1-6 for a (31, 21) code having $d_{\min} = 5$.

13.1-8 A selective-repeat ARQ system with gaussian white noise is to have $P_{be} = 10^{-5}$ using one of the following blocks codes for error detection:

n	k	$d_{\min}$
12	11	2
15	11	3
16	11	4

Calculate r_b/r and γ_b for each code and for uncoded transmission. Then plot γ_b in dB versus r_b/r.

13.1-9 Do Prob. 13.1-8 for $P_{be} = 10^{-6}$.

13.1-10 A go-back-N ARQ system has gaussian white noise, $\gamma_c = 6$ dB, $r = 500$ kbps, and a one-way path length of 45 km. Find P_{be}, the minimum value of N, and the maximum value of r_b using a (15, 11) block code with $d_{\min} = 3$ for error detection.

13.1-11 Do Prob. 13.1-10 using a (16, 11) block code with $d_{\min} = 4$.

13.1-12 A stop-and-wait ARQ system uses simple parity checking with $n = k + 1$ for error detection. The system has gaussian white noise, $r = 10$ kbps, and a one-way path length of 18 km. Find the smallest value of k such that $P_{be} \leq 10^{-6}$ and $r_b \geq 7200$ bps. Then calculate γ_b in dB.

13.1-13 Do Prob. 13.1-12 with a 60-km path length.

13.1-14‡ Derive m as given in Eq. (19) for a go-back-N ARQ system. *Hint*: If a given word has detected errors in i successive transmissions, then the total number of transmitted words equals $1 + Ni$.

13.1-15 Consider a hybrid ARQ system using a code that corrects t errors and detects $\ell > t$ errors per n-bit word. Obtain an expression for the retransmission probability p when $\alpha \ll 1$. Then take $d_{\min} = 4$ and compare your result with Eq. (15).

13.1-16 Suppose a hybrid selective-repeat ARQ system uses an (n, k) block code with $d_{\min} = 2t + 2$ to correct up to t errors and detect $t + 1$ errors per word. (*a*) Assume $\alpha \ll 1$ to obtain an approximate expression for the retransmission probability p, and show that

$$P_{be} \approx \binom{n-1}{t+1}\alpha^{t+2}$$

(*b*) Evaluate α and p for a (24, 12) code with $d_{\min} = 8$ when $P_{be} = 10^{-5}$. Then assume gaussian white noise and find P_{be} for uncoded transmission with the same value of γ_b.

13.2-1 Let U and V be n-bit vectors. (*a*) By considering the number of 1's in U, V, and $U + V$, confirm that $d(U, V) \leq w(U) + w(V)$. (*b*) Now let $U = X + Y$ and $V = Y + Z$. Show that $U + V = X + Z$ and derive the *triangle inequality*

$$d(X, Z) \leq d(X, Y) + d(Y, Z)$$

13.2-2 Let X be a code vector, let Z be any other vector in the code, and let Y be the vector that results when X is received with i bit errors. Use the triangle inequality in Prob. 13.2-1 to show that if the code has $d_{\min} \geq \ell + 1$ and if $i \leq \ell$, then the errors in Y are detectable.

13.2-3 Let X be a code vector, let Z be any other vector in the code, and let Y be the vector that results when X is received with i bit errors. Use the triangle inequality in Prob. 13.2-1 to show that if the code has $d_{\min} \geq 2t + 1$ and if $i \leq t$, then the errors in Y are correctable.

13.2-4 A triple-repetition code is a systematic (3, 1) block code generated using the submatrix $P = [1 \quad 1]$. Tabulate all possible received vectors Y and $S = YH^T$. Then determine the corresponding maximum-likelihood errors patterns and corrected vectors $Y + \hat{E}$.

13.2-5 Construct the lookup table for the (6, 3) block code in Exercise 13.2-1.

13.2-6 Consider a (5, 3) block code obtained by deleting the last column of the P submatrix in Exercise 13.2-1. Construct the lookup table, and show that this code could be used for error detection but not correction.

13.2-7 Let the P submatrix for a (15, 11) Hamming code be arranged such that the row words increasing in numerical value from top to bottom. Construct the lookup table and write the check-bit equations.

13.2-8 Suppose a block code with $t = 1$ is required to have $k = 6$ message bits per word. (*a*) Find the minimum value of n and the number of bits stored in the lookup table. (*b*) Construct an appropriate P submatrix.

13.2-9 Do Prob. 13.2-8 with $k = 8$.

13.2-10 It follows from Eq. (4) that $XH^T = MA$ with $A = GH^T$. Prove Eq. (9) by showing that any element of A has the property $a_{ij} = 0$.

13.2-11 The original (7, 4) Hamming code is a *nonsystematic* code with

$$H = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$