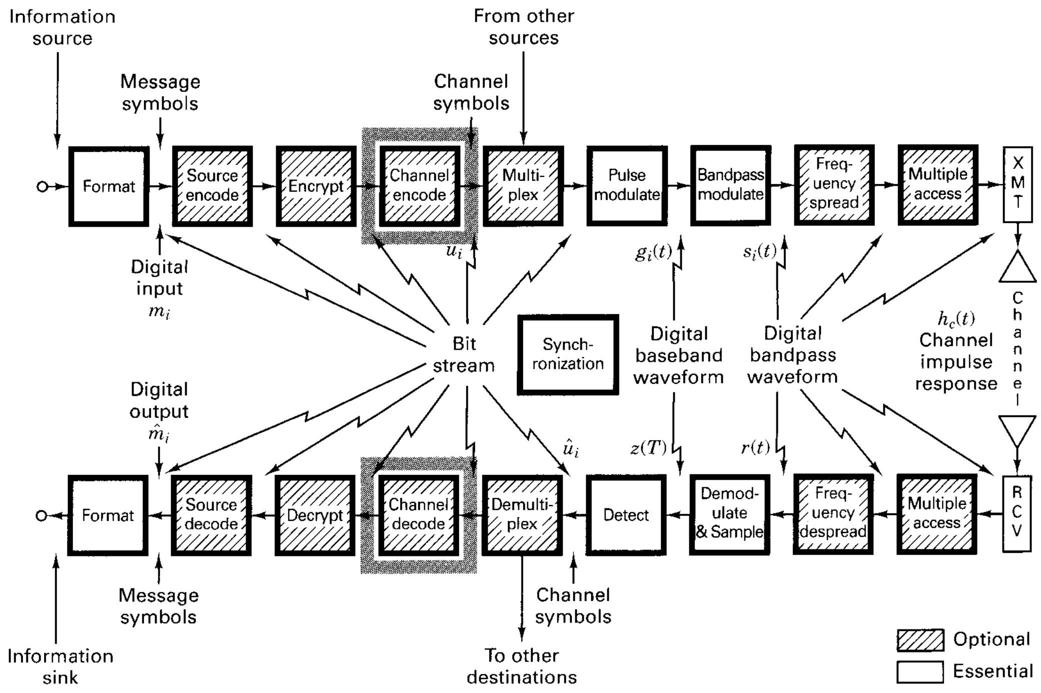
CHAPTER 8

Channel Coding: Part 3



8.1 REED–SOLOMON CODES

Reed–Solomon (R–S) codes are *nonbinary cyclic* codes with symbols made up of m-bit sequences, where m is any positive integer having a value greater than 2. R–S (n, k) codes on m-bit symbols exist for all n and k for which

$$0 < k < n < 2^m + 2 \tag{8.1}$$

where k is the number of data symbols being encoded, and n is the total number of code symbols in the encoded block. For the most conventional R–S (n, k) code,

$$(n, k) = (2^m - 1,\ 2^m - 1 - 2t) \tag{8.2}$$

where t is the symbol-error correcting capability of the code, and $n - k = 2t$ is the number of parity symbols. An extended R–S code can be made up with $n = 2^m$ or $n = 2^m + 1$, but not any further.

Reed–Solomon (R–S) codes achieve the *largest possible* code minimum distance for any linear code with the same encoder input and output block lengths. For nonbinary codes, the distance between two codewords is defined (analogous to Hamming distance) as the number of symbols in which the sequences differ. For Reed–Solomon codes the code minimum distance is given by [1]

$$d_{\min} = n - k + 1 \tag{8.3}$$

The code is capable of correcting any combination of t or fewer errors, where t obtained from Equation (6.44), can be expressed as

$$t = \left\lfloor \frac{d_{\min} - 1}{2} \right\rfloor = \left\lfloor \frac{n-k}{2} \right\rfloor \tag{8.4}$$

where $\lfloor x \rfloor$ means the largest integer not to exceed x. Equation (8.4) illustrates that for the case of R–S codes, correcting t symbol errors requires no more than $2t$ parity symbols. Equation (8.4) lends itself to the following intuitive reasoning. One can say that the decoder has $n - k$ redundant symbols "to spend," which is twice the amount of correctable errors. For each error, one redundant symbol is used to locate the error, and another redundant symbol is used to find its correct value.

The erasure-correcting capability of the code is

$$\rho = d_{\min} - 1 = n - k \tag{8.5}$$

Simultaneous error-correction and erasure-correction capability can be expressed by the requirement that

$$2\alpha + \gamma < d_{\min} < n - k \tag{8.6}$$

where α is the number of symbol error patterns that can be corrected, and γ is the number of symbol erasure patterns that can be corrected. An advantage of nonbinary codes such as a Reed–Solomon code can be seen by the following comparison. Consider a binary $(n, k) = (7, 3)$ code. The entire n-tuple space contains $2^n = 2^7 = 128$ n-tuples, of which $2^k = 2^3 = 8$ (or 1/16 of the n-tuples) are codewords. Next consider a nonbinary $(n, k) = (7, 3)$ code where each symbol comprises $m = 3$ bits. The n-tuple space amounts to $2^{nm} = 2^{21} = 2{,}097{,}152$ n-tuples, of which $2^{km} = 2^9 = 512$ (or 1/4096 of the n-tuples) are codewords. When dealing with nonbinary symbols, each made up of m bits, only a small fraction (i.e., 2^{km} of the large number 2^{nm}) of possible n-tuples are codewords. This fraction decreases with increasing values of m. The important point here is that, when a small fraction of the n-tuple space is used for codewords, a large $d_{\min}$ can be created.

Any linear code is capable of correcting $n - k$ symbol erasure patterns if the $n - k$ erased symbols all happen to lie on the parity symbols. However, R–S codes have the remarkable property that they are able to correct *any* set of $n - k$ symbol erasures within the block. R–S codes can be designed to have any redundancy. However, the complexity of a high speed implementation increases with redundancy. Thus, the most attractive R–S codes have high code rates (low redundancy).

8.1.1 Reed-Solomon Error Probability

The Reed–Solomon (R–S) codes are particularly useful for *burst-error correction*; that is, they are effective for channels that have memory. Also, they can be used efficiently on channels where the set of input symbols is large. An interesting feature of the R–S code is that as many as two information symbols can be added to an R–S code of length n without reducing its minimum distance. This extended R–S code has length $n + 2$ and the same number of parity check symbols as the original code. From Equation (6.46), the R–S decoded symbol error probability, P_E, in terms of the channel symbol error probability, p, can be written as follows [2]:

$$P_E \approx \frac{1}{2^m - 1} \sum_{j=t+1}^{2^m - 1} j \binom{2^m - 1}{j} p^j (1 - p)^{2^m - 1 - j} \tag{8.7}$$

where t is the symbol-error correcting capability of the code, and the symbols are made up of m bits each.

The bit error probability can be upper bounded by the symbol error probability for specific modulation types. For MFSK modulation with $M = 2^m$, the relationship between P_B and P_E as given in Equation (4.112) is repeated here:

$$\frac{P_B}{P_E} = \frac{2^{m-1}}{2^m - 1} \tag{8.8}$$

Figure 8.1 shows P_B versus the channel symbol error probability p, plotted from Equations (8.7) and (8.8) for various t-error-correcting 32-ary orthogonal Reed–Solomon codes with $n = 31$ (thirty-one 5-bit symbols per code block).

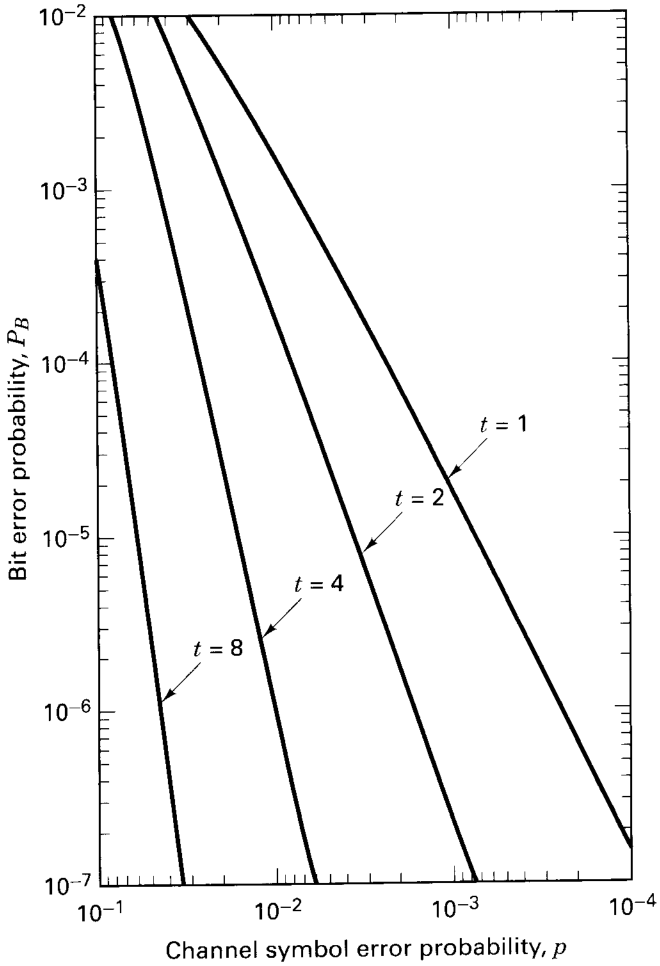


Figure 8.1 P_B versus p for 32-ary orthogonal signaling and $n = 31$, t-error-correcting Reed–Solomon coding. (Reprinted with permission from *Data Communications, Networks and Systems*, ed. Thomas C. Bartee, Howard W. Sams Company, Indianapolis, Ind., 1985, p. 311. Originally published in J. P. Odenwalder, *Error Control Coding Handbook*, M/A-COM LINKABIT, Inc., San Diego, Calif., July 15, 1976, p. 91.)

Figure 8.2 shows P_B versus E_b/N_0 for such a coded system using 32-ary MFSK modulation and noncoherent demodulation over an AWGN channel [2]. For R–S codes, error probability is an exponentially decreasing function of block length, n, and decoding complexity is proportional to a small power of the block length [1]. The R–S codes are sometimes used in a concatenated arrangement. In such a

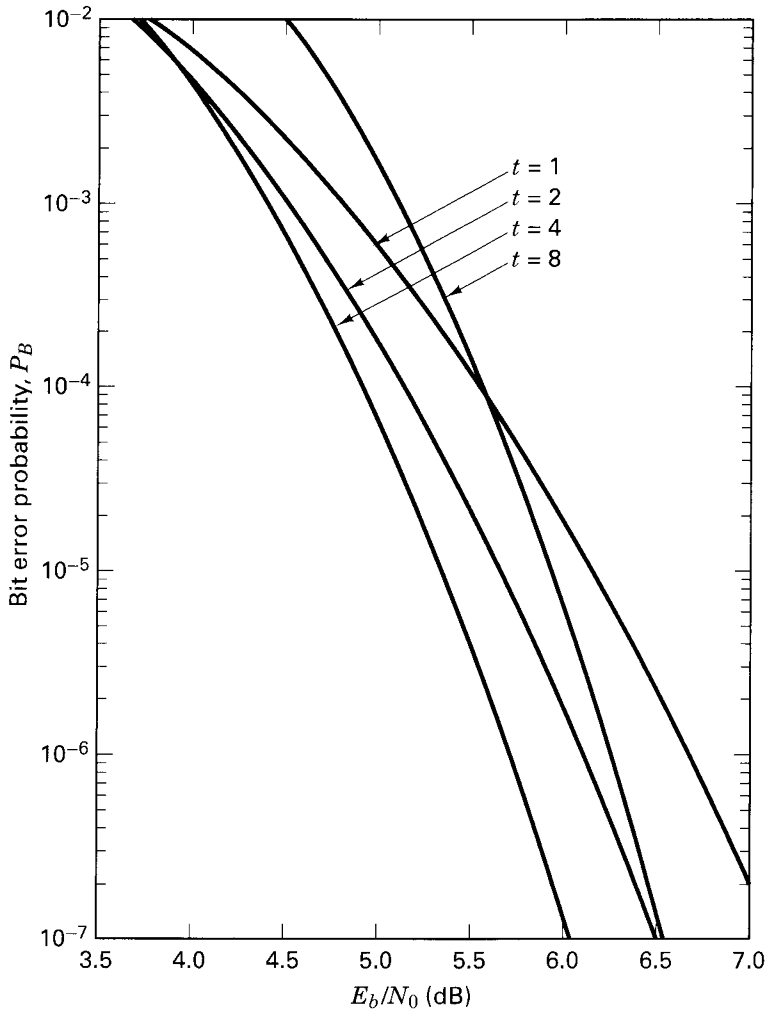


Figure 8.2 Bit error probability versus E_b/N_0 performance of several $n = 31$, t-error correcting Reed–Solomon coding systems with 32-ary MFSK modulation over an AWGN channel. (Reprinted with permission from *Data Communications, Networks, and Systems,* ed. Thomas C. Bartee, Howard W. Sams Company, Indianapolis, Ind., 1985, p. 312. Originally published in J. P. Odenwalder, *Error Control Coding Handbook,* M/A-COM LINKABIT, Inc. San Diego, Calif., July 15, 1976, p. 92.)

system, an inner convolutional decoder first provides some error control by operating on soft-decision demodulator outputs; the convolutional decoder then presents hard-decision data to the outer Reed–Solomon decoder, which further reduces the probability of error. In Sections 8.2.3 and 8.3 we discuss further the use of concatenated and R–S coding as applied to the compact disc (CD) digital audio system.

8.1.2 Why R–S Codes Perform Well Against Burst Noise

Consider an $(n, k) = (255, 247)$ R–S code, where each symbol is made up of $m = 8$ bits (such symbols are typically referred to as *bytes*). Since $n - k = 8$, Equation (8.4) indicates that this code can correct any 4 symbol errors in a block of 255. Imagine the presence of a noise burst, lasting for 25-bit durations and disturbing one block of data during transmission, as illustrated in Figure 8.3. In this example, notice that a burst of noise that lasts for a duration of 25 contiguous bits, must disturb exactly 4 symbols. The R–S decoder for the (255, 247) code will correct *any* 4-symbol errors without regard to the type of damage suffered by the symbol. In other words, when a decoder corrects a byte, it replaces the incorrect byte with the correct one, whether the error was caused by one bit being corrupted or all 8 bits being corrupted. Thus, if a symbol is wrong, it might as well be wrong in all of its bit positions. This gives a R–S code a tremendous burst-noise advantage over binary codes, even allowing for the interleaving of binary codes. In this example, if the 25-bit noise disturbance had occurred in a random fashion rather than as a contiguous burst, it should be clear that there would then be many more than 4 symbols affected (as many as 25 symbols might be disturbed). Of course, that would be beyond the capability of the (255, 247) code.

8.1.3 R–S Performance as a Function of Size, Redundancy, and Code Rate

For a code to successfully combat the effects of noise, the noise duration has to represent a relatively small percentage of the codeword. To ensure that this happens most of the time, the received noise should be averaged over a long period of time, reducing the effect of a sudden or unusual streak of bad luck. Hence, one can expect that error-correcting codes become more efficient (error performance improves) as the code block size increases, making R–S codes an attractive choice

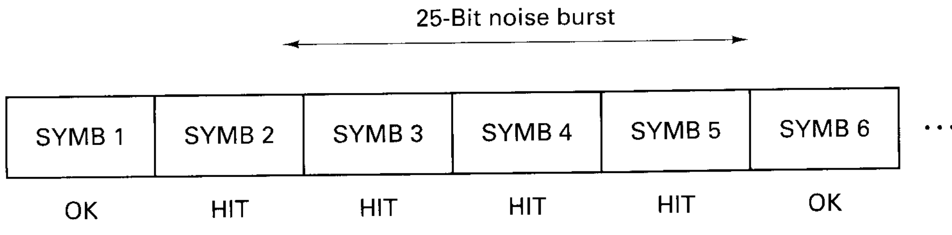


Figure 8.3 Data block disturbed by 25-bit noise burst.

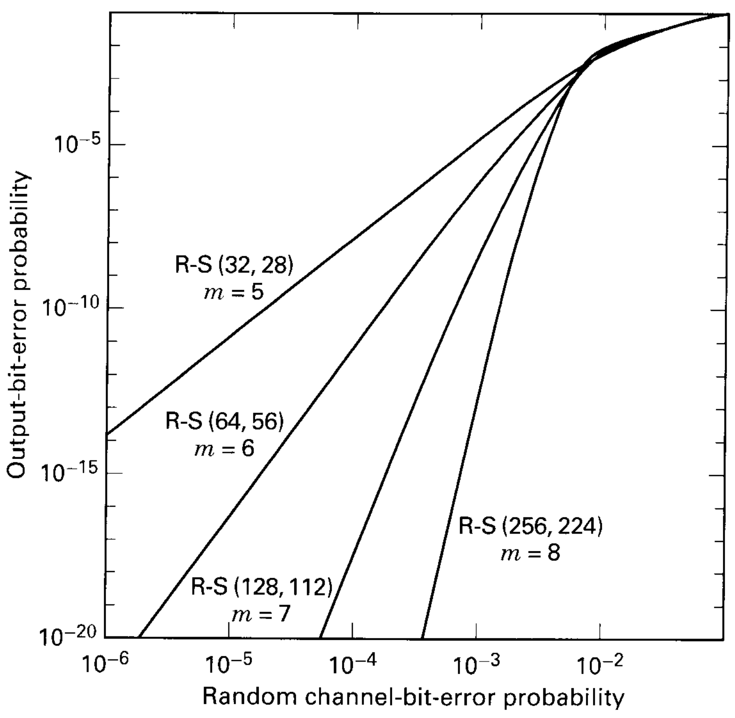


Figure 8.4 Reed–Solomon, rate 7/8, decoder performance as a function of symbol size.

whenever long block lengths are desired [3]. This is seen by the family of curves in Figure 8.4, where the rate of the code is held at a constant 7/8, while its block size increases from $n = 32$ symbols (with $m = 5$ bits per symbol) to $n = 256$ symbols (with $m = 8$ bits per symbol). Thus, the block size increases from 160 bits to 2048 bits.

As the redundancy of an R-S code increases (lower code rate), its implementation grows in complexity (especially for high speed devices). Also, the bandwidth expansion must grow for any real-time communications application. However, the benefit of increased redundancy, just like the benefit of increased symbol size, is the improvement in bit-error performance, as can be seen in Figure 8.5, where the code length n is held at a constant 64, while number of data symbols decreases from $k = 60$ to $k = 4$ (redundancy increases from 4 symbols to 60 symbols).

Figure 8.5 represents transfer functions (output bit-error probability versus input channel symbol-error probability) of hypothetical decoders. Because there is no system or channel in mind (only an output-versus-input of a decoder), one might get the idea that the improved error performance versus increased redundancy is a monotonic function that will continually provide system improvement even as the code rate approaches zero. However, this is not the case for codes operating in a real-time communication system. As the rate of a code varies from minimum to maximum (0 to 1), it is interesting to observe the effects shown in

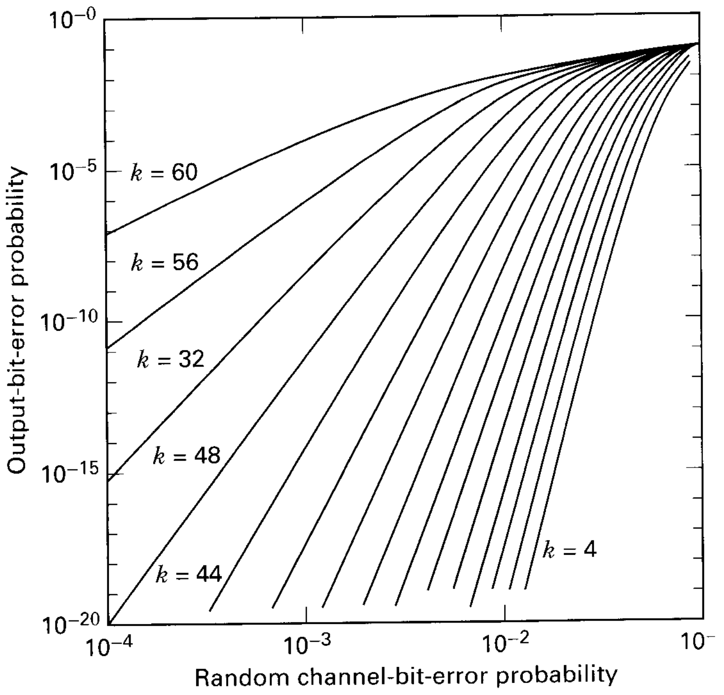


Figure 8.5 Reed–Solomon (64, *k*) decoder performance as a function of redundancy.

Figure 8.6. Here, the performance curves are plotted for BPSK modulation and an R–S (31, k) code for various channel types. Figure 8.6 reflects a real-time communication system, where the price paid for error-correction coding is bandwidth expansion by a factor equal to the inverse of the code rate. The curves plotted show clear optimum code rates which minimize the required E_b/N_0 [4]. The optimum code rate is about 0.6 to 0.7 for a Gaussian channel, 0.5 for a Rician-fading channel (with the ratio of direct to reflected received signal power, $K = 7$ dB), and 0.3 for a Rayleigh-fading channel. (Fading channels are treated in Chapter 15.) Why is there an E_b/N_0 degradation for very large rates (small redundancy) and very low rates (large redundancy)? It is easy to explain the degradation at high rates compared with the optimum rate. Any code generally provides a coding gain benefit; thus, as the code rate approaches unity (no coding), the system will suffer worse error performance. The degradation at low code rates is more subtle because in a real-time communication system using both modulation and coding, there are two mechanisms at work. One mechanism works to improve error performance, and the other works to degrade it. The improving mechanism is the coding; the greater the redundancy, the greater will be the error-correcting capability of the code. The degrading mechanism is the energy reduction per channel symbol (compared with the data symbol) which stems from the increased redundancy (and faster signaling in a real-time communication system). The reduced symbol energy causes the demodu-

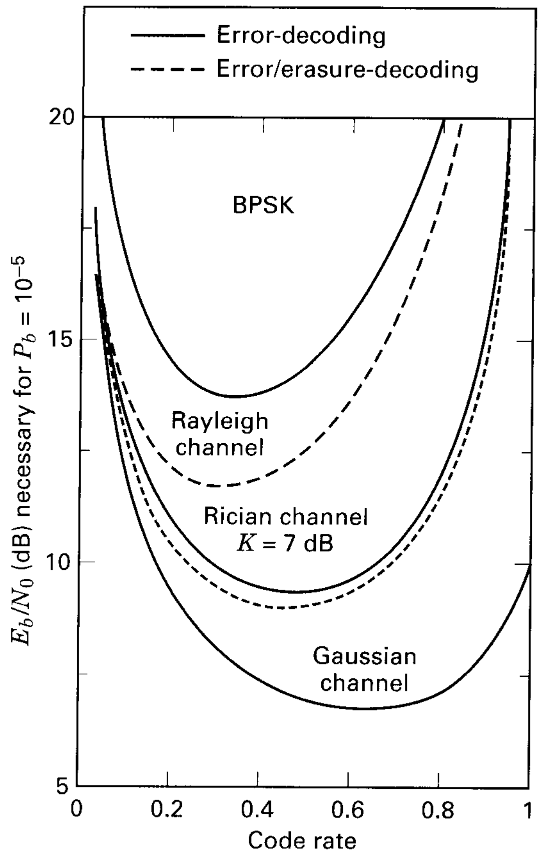


Figure 8.6 BPSK plus Reed–Solomon (31, *k*) decoder performance as a function of code rate.

lator to make more errors. Eventually, the second mechanism wins out, and thus, at very low code rates the system experiences error-performance degradation.

Let us see if we can corroborate the error performance versus code rate in Figure 8.6 with the curves in Figure 8.2. The figures are really not directly comparable because the modulation is BPSK in Figure 8.6, while it is 32-ary MFSK in Figure 8.2. However, perhaps we can verify that R-S error performance-versus-code rate exhibits the same general curvature with MFSK modulation as it does with BPSK. In Figure 8.2, the error performance over an AWGN channel, improves as the symbol error-correcting capability t increases from $t = 1$ to $t = 4$; the $t = 1$ and $t = 4$ cases correspond to R-S (31, 29) and R-S (31, 23) with code rates of 0.94 and 0.74, respectively. However at $t = 8$, which corresponds to R-S (31, 15) with code rate equal to 0.48, the error performance at $P_B = 10^{-5}$ degrades by about 0.5 dB of E_b/N_0, compared with the $t = 4$ case. From Figure 8.2, we can conclude that if we were to plot error performance versus code rate, the curve would have the same general shape as it does in Figure 8.6. Note that this manifestation cannot be gleaned from Figure 8.1, since that figure represents a decoder transfer function, which provides no information about the channel and the demodulation. Therefore, of the two mechanisms at work in the channel, the Figure 8.1 transfer function only presents the output-versus-input benefits of the decoder, and displays nothing about the loss of energy as a function of lower code rate. More is said about choosing a code in concert with a modulation type in Section 9.7.7.

8.1.4 Finite Fields

In order to understand the encoding and decoding principles of nonbinary codes, such as a Reed–Solomon (R-S) codes, it is necessary to venture into the area of finite fields known as *Galois Fields* (GF). For any prime number p there exists a finite field denoted GF(p), containing p elements. It is possible to extend GF(p) to a field of p^m elements, called an extension field of GF(p), and denoted by GF(p^m), where m is a nonzero positive integer. Note that GF(p^m) contains as a subset the elements of GF(p). Symbols from the extension field GF(2^m) are used in the construction of Reed–Solomon (R–S) codes.

The binary field GF(2) is a subfield of the extension field GF(2^m), much the same way as the real number field is a subfield of the complex number field. Besides the numbers 0 and 1, there are additional unique elements in the extension field that will be represented with a new symbol α. Each nonzero element in GF(2^m) can be represented by a power of α. An *infinite* set of elements, F, is formed by starting with the elements $\{0, 1, \alpha\}$ and generating additional elements by progressively multiplying the last entry by α which yields

$$F = \{0, 1, \alpha, \alpha^2, \cdots, \alpha^j, \cdots\} = \{0, \alpha^0, \alpha^1, \alpha^2, \cdots, \alpha^j, \cdots\} \tag{8.9}$$

To obtain the *finite* set of elements of GF(2^m) from F, a condition must be imposed on F so that it may contain only 2^m elements and is closed under multiplication. The condition that closes the set of field elements under multiplication is characterized by the irreducible polynomial

$$\alpha^{(2^m-1)} + 1 = 0$$

or equivalently,

$$\alpha^{(2^m-1)} = 1 = \alpha^0 \tag{8.10}$$

Using this polynomial constraint, any field element that has a power equal to or greater than $2^m - 1$ can be reduced to an element with a power less than $2^m - 1$ as follows:

$$\alpha^{(2^m+n)} = \alpha^{(2^m-1)}\alpha^{n+1} = \alpha^{n+1} \tag{8.11}$$

Thus, Equation (8.10) can be used to form the finite sequence F^* from the infinite sequence F, as follows:

$$\begin{aligned} F^* &= \{0, 1, \alpha, \alpha^2, \cdots, \alpha^{2^m-2}, \alpha^{2^m-1}, \alpha^{2^m}, \cdots\} \\ &= \{0, \alpha^0, \alpha^1, \alpha^2, \cdots, \alpha^{2^m-2}, \alpha^0, \alpha^1, \alpha^2, \cdots\} \end{aligned} \tag{8.12}$$

Therefore, it can be seen from Equation (8.12) that the elements of the finite field GF$(2)^m$ are given by

$$\mathrm{GF}(2^m) = \{0, \alpha^0, \alpha^1, \alpha^2, \cdots, \alpha^{2^m-2}\} \tag{8.13}$$

8.1.4.1 Addition in the Extension Field GF(2^m)

Each of the 2^m elements of the finite field GF(2^m) can be represented as a distinct polynomial of *degree* $m - 1$ or less. The degree of a polynomial is the value of its highest order exponent. We denote each of the nonzero elements of GF(2^m) as a polynomial $a_i(X)$, where at least one of the m coefficients of $a_i(X)$ is nonzero. For $i = 0, 1, 2, \ldots, 2^m - 2$,

$$\alpha^i = a_i(X) = a_{i,0} + a_{i,1}X + a_{i,2}X^2 + \cdots + a_{i,m-1}X^{m-1} \tag{8.14}$$

Consider the case of $m = 3$, where the finite field is denoted GF(2^3). Figure 8.7 shows the mapping (developed later) of the seven elements $\{\alpha^i\}$ and the zero element, in terms of the basis elements $\{X^0, X^1, X^2\}$ described by Equation (8.14). Since Equation (8.10) indicates that $\alpha^0 = \alpha^7$, there are seven nonzero elements or a total of eight elements in this field. Each row in the Figure 8.7 mapping comprises a sequence of binary values representing the coefficients $\alpha_{i,0}$, $\alpha_{i,1}$, and $\alpha_{i,2}$ in Equation (8.14). One of the benefits of using extension field elements $\{\alpha^i\}$ in place of binary elements is the compact notation that facilitates the mathematical representation of nonbinary encoding and decoding processes. Addition of two elements of the finite field is then defined as the modulo-2 sum of each of the polynomial coefficients of like powers, i.e.,

$$\alpha^i + \alpha^j = (a_{i,0} + a_{j,0}) + (a_{i,1} + a_{j,1})X + \cdots + (a_{i,m-1} + a_{j,m-1})X^{m-1} \tag{8.15}$$

8.1.4.2 A Primitive Polynomial is Used to Define the Finite Field

A class of polynomials called *primitive polynomials,* is of interest because such functions define the finite fields of GF(2^m) which in turn are needed to define R-S codes. The following condition is necessary and sufficient to guarantee that a

Field elements	Basis elements		
	X^0	X^1	X^2
0	0	0	0
α^0	1	0	0
α^1	0	1	0
α^2	0	0	1
α^3	1	1	0
α^4	0	1	1
α^5	1	1	1
α^6	1	0	1
α^7	1	0	0

Figure 8.7 Mapping field elements in terms of basis elements for GF(8) with $f(X) = 1 + X + X^3$.

polynomial is primitive. An irreducible polynomial, $f(X)$, of degree m is said to be primitive, if the smallest positive integer n for which $f(X)$ divides $X^n + 1$ is $n = 2^m - 1$. Note that an irreducible polynomial is one that cannot be factored to yield lower order polynomials, and that the statement A divides B means that A divided into B yields a nonzero quotient and a zero remainder. Polynomials will usually be shown low order-to-high order. Sometimes, it is convenient to follow the reverse format (e.g., when performing polynomial division).

Example 8.1 Recognizing a Primitive Polynomial

Based on the foregoing definition of a primitive polynomial, determine whether the following irreducible polynomials are primitive:

(a) $1 + X + X^4$

(b) $1 + X + X^2 + X^3 + X^4$

Solution

(a) We can verify whether or not this degree $m = 4$ polynomial is primitive by determining if it divides $X^n + 1 = X^{(2^m - 1)} + 1 = X^{15} + 1$, but does not divide $X^n + 1$, for values of n in the range of $1 \le n < 15$. It is easy to verify that $1 + X + X^4$ divides $X^{15} + 1$, and after repeated computations it can be verified that $1 + X + X^4$ will not divide $X^n + 1$ for any n in the range of $1 \le n < 15$. Therefore, $1 + X + X^4$ is a primitive polynomial.

(b) It is simple to verify that the polynomial $1 + X + X^2 + X^3 + X^4$ divides $X^{15} + 1$. Testing to see if it will divide $X^n + 1$ for some n that is less than 15, yields the fact that it also divides $X^5 + 1$. Thus, although $1 + X + X^2 + X^3 + X^4$ is irreducible, it is not primitive.

8.1.4.3 The Extension Field GF(2^3)

Consider an example involving a primitive polynomial and the finite field that it defines. Table 8.1 contains a listing of some primitive polynomials. We choose the first one shown, $f(X) = 1 + X + X^3$ which defines a finite field GF(2^m), where the degree of the polynomial is $m = 3$. Thus, there are $2^m = 2^3 = 8$ elements in the field defined by $f(X)$. Solving for the roots of $f(X)$ means that the values of X that correspond to $f(X) = 0$ must be found. The familiar binary elements 1 and 0 do not satisfy (are not roots of) the polynomial $f(X) = 1 + X + X^3$, since $f(1) = 1$ and $f(0) = 1$ (using modulo-2 arithmetic). Yet, a fundamental theorem of algebra states that a polynomial of degree m must have precisely m roots. Therefore for this example, $f(X) = 0$ must yield 3 roots. Clearly a dilemma arises, since the 3 roots do not lie in the same finite field as the coefficients of $f(X)$. Therefore, they must lie somewhere else; the roots lie in the extension field GF(2^3). Let α, an element of the extension field, be defined as a root of the polynomial $f(X)$. Therefore, it is possible to write

$$f(\alpha) = 0$$

$$1 + \alpha + \alpha^3 = 0 \tag{8.16}$$

$$\alpha^3 = -1 - \alpha$$

TABLE 8.1 Some Primitive Polynomials

m		m	
3	$1+X+X^3$	14	$1+X+X^6+X^{10}+X^{14}$
4	$1+X+X^4$	15	$1+X+X^{15}$
5	$1+X^2+X^5$	16	$1+X+X^3+X^{12}+X^{16}$
6	$1+X+X^6$	17	$1+X^3+X^{17}$
7	$1+X^3+X^7$	18	$1+X^7+X^{18}$
8	$1+X^2+X^3+X^4+X^8$	19	$1+X+X^2+X^5+X^{19}$
9	$1+X^4+X^9$	20	$1+X^3+X^{20}$
10	$1+X^3+X^{10}$	21	$1+X^2+X^{21}$
11	$1+X^2+X^{11}$	22	$1+X+X^{22}$
12	$1+X+X^4+X^6+X^{12}$	23	$1+X^5+X^{23}$
13	$1+X+X^3+X^4+X^{13}$	24	$1+X+X^2+X^7+X^{24}$

Since in the binary field $+1 = -1$, then α^3 can be represented as

$$\alpha^3 = 1 + \alpha \tag{8.17}$$

Thus, α^3 is expressed as a weighted sum of α-terms having lower orders. In fact, all powers of α can be so expressed. For example, consider

$$\alpha^4 = \alpha \cdot \alpha^3 = \alpha \cdot (1 + \alpha) = \alpha + \alpha^2 \tag{8.18a}$$

Now consider

$$\alpha^5 = \alpha \cdot \alpha^4 = \alpha \cdot (\alpha + \alpha^2) = \alpha^2 + \alpha^3 \tag{8.18b}$$

From Equations (8.17) and (8.18b), we obtain

$$\alpha^5 = 1 + \alpha + \alpha^2 \tag{8.18c}$$

Now, using Equation (8.18c), we obtain

$$\alpha^6 = \alpha \cdot \alpha^5 = \alpha \cdot (1 + \alpha + \alpha^2) = \alpha + \alpha^2 + \alpha^3 = 1 + \alpha^2 \tag{8.18d}$$

And using Equation (8.18d), we obtain

$$\alpha^7 = \alpha \cdot \alpha^6 = \alpha \cdot (1 + \alpha^2) = \alpha + \alpha^3 = 1 = \alpha^0 \tag{8.18e}$$

Note that $\alpha^7 = \alpha^0$, and therefore, the eight finite field elements of GF(2^3) are

$$\{0, \alpha^0, \alpha^1, \alpha^2, \alpha^3, \alpha^4, \alpha^5, \alpha^6\} \tag{8.19}$$

The mapping of field elements in terms of basis elements, described by Equation (8.14) can be demonstrated with the linear feedback shift register (LFSR) circuit shown in Figure 8.8. The circuit generates (with $m = 3$) the $2^m - 1$ nonzero elements of the field, and thus summarizes the findings of Figure 8.7 and Equations (8.17) through (8.19). Note that in Figure 8.8, the circuit feedback connections correspond to the coefficients of the polynomial $f(X) = 1 + X + X^3$, just like for binary cyclic codes. (See Section 6.7.5.) By starting the circuit in any nonzero state, say 1 0 0, and performing a right-shift at each clock time, it is possible to verify that each of

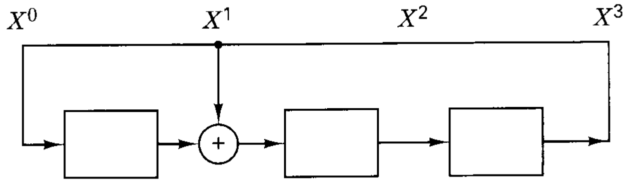


Figure 8.8 Extension field elements can be represented by the contents of a binary linear feedback shift register (LFSR) formed from a primitive polynomial.

the field elements shown in Figure 8.7 (except the all-zeros element) will cyclicly appear in the stages of the shift register. Two arithmetic operations, addition and multiplication, can be defined for this GF(2^3) finite field. Addition is shown in Table 8.2, and multiplication is shown in Table 8.3 for the nonzero elements only. The rules of addition follow from Equations (8.17) through (8.18e), and can be verified by noticing in Figure 8.7 that the sum of any field elements can be obtained by adding (modulo-2) the respective coefficients of their basis elements. The multiplication rules in Table 8.3 follow the usual procedure in which the product of the field elements is obtained by adding their exponents modulo-($2^m - 1$), or for this case, modulo-7.

TABLE 8.2 Addition Table for GF(8) with $f(X) = 1 + X + X^3$

	α^0	α^1	α^2	α^3	α^4	α^5	α^6
α^0	0	α^3	α^6	α^1	α^5	α^4	α^2
α^1	α^3	0	α^4	α^0	α^2	α^6	α^5
α^2	α^6	α^4	0	α^5	α^1	α^3	α^0
α^3	α^1	α^0	α^5	0	α^6	α^2	α^4
α^4	α^5	α^2	α^1	α^6	0	α^0	α^3
α^5	α^4	α^6	α^3	α^2	α^0	0	α^1
α^6	α^2	α^5	α^0	α^4	α^3	α^1	0

TABLE 8.3 Multiplication Table for GF(8) with $f(X) = 1 + X + X^3$

	α^0	α^1	α^2	α^3	α^4	α^5	α^6
α^0	α^0	α^1	α^2	α^3	α^4	α^5	α^6
α^1	α^1	α^2	α^3	α^4	α^5	α^6	α^0
α^2	α^2	α^3	α^4	α^5	α^6	α^0	α^1
α^3	α^3	α^4	α^5	α^6	α^0	α^1	α^2
α^4	α^4	α^5	α^6	α^0	α^1	α^2	α^3
α^5	α^5	α^6	α^0	α^1	α^2	α^3	α^4
α^6	α^6	α^0	α^1	α^2	α^3	α^4	α^5

8.1.4.4 A Simple Test to Determine if a Polynomial is Primitive

There is another way of defining a primitive polynomial that makes its verification relatively easy. For an irreducible polynomial to be a primitive polynomial, at least one of its roots must be a primitive element. A primitive element is one that when raised to higher order exponents will yield all the nonzero elements in the field. Since the field is a finite field, the number of such elements is finite.

Example 8.2 A Primitive Polynomial Must Have at Least one Primitive Element

Find the $m = 3$ roots of $f(X) = 1 + X + X^3$, and verify that the polynomial is primitive by checking that at least one of the roots is a primitive element. What are the roots? Which ones are primitive?

Solution

The roots will be found by enumeration. Clearly, $\alpha^0 = 1$ is not a root because $f(\alpha^0) = 1$. Now use Table 8.2 to check if a α^1 is a root. Since $f(\alpha) = 1 + \alpha + \alpha^3 = 1 + \alpha^0 = 0$, then α is a root. Now check if α^2 is a root. $f(\alpha^2) = 1 + \alpha^2 + \alpha^6 = 1 + \alpha^0 = 0$. Hence, α^2 is a root. Now check if α^3 is a root. $f(\alpha^3) = 1 + \alpha^3 + \alpha^9 = 1 + \alpha^3 + \alpha^2 = 1 + \alpha^5 = \alpha^4 \neq 0$. Hence, α^3 is *not* a root. Is α^4 a root? $f(\alpha^4) = \alpha^{12} + \alpha^4 + 1 = \alpha^5 + \alpha^4 + 1 = 1 + \alpha^0 = 0$. Yes, it is a root. Hence, the roots of $f(X) = 1 + X + X^3$, are α, α^2, and α^4. It is not difficult to verify that starting with any one of these roots and generating higher order exponents yields all of the 7 nonzero elements in the field. Hence, each of the roots is a primitive element. Since our verification requires that at least one root be a primitive element, the polynomial is primitive.

A relatively simple method to verify if a polynomial is primitive can be described in a manner that is related to this example. For any given polynomial under test, draw the LFSR, with the feedback connections corresponding to the polynomial coefficients as shown by the example of Figure 8.8. Load into the circuit-registers any nonzero setting, and perform a right shift with each clock pulse. If the circuit generates each of the nonzero field elements within one period, then the polynomial that defines this GF(2^m) field is a primitive polynomial.

8.1.5 Reed–Solomon Encoding

Equation (8.2) expresses the most conventional form of Reed–Solomon (R–S) codes in terms of the parameters *n, k, t,* and any positive integer $m > 2$. Repeated here, that equation is

$$(n, k) = (2^m - 1,\ 2^m - 1 - 2t) \tag{8.20}$$

where $n - k = 2t$ is the number of parity symbols, and t is the symbol-error correcting capability of the code. The generating polynomial for an R-S code takes the following form:

$$\mathbf{g}(X) = g_0 + g_1X + g_2X^2 + \cdots + g_{2t-1}X^{2t-1} + X^{2t} \tag{8.21}$$

The degree of the generator polynomial is equal to the number of parity symbols. R-S codes are a subset of the BCH codes described in Section 6.8.3 and Table 6.4. Hence, it should be no surprise that this relationship between the degree of the generator polynomial and the number of parity symbols holds just as it does for BCH codes. This can be verified by checking any of the generator polynomials in Table 6.4. Since the generator polynomial is of degree $2t$, there must be precisely $2t$ successive powers of α that are roots of the polynomial. We designate the roots of $\mathbf{g}(X)$ as: $\alpha, \alpha^2, \ldots, \alpha^{2t}$. It is not necessary to start with the root α; starting with any power of α is possible. Consider as an example, the (7, 3) double-symbol error correcting R-S code. We describe the generator polynomial in terms of its $2t = n - k = 4$ roots, as follows:

$$\begin{aligned}\mathbf{g}(X) &= (X-\alpha)(X-\alpha^2)(X-\alpha^3)(X-\alpha^4)\\ &= (X^2-(\alpha+\alpha^2)X+\alpha^3)(X^2-(\alpha^3+\alpha^4)X+\alpha^7)\\ &= (X^2-\alpha^4X+\alpha^3)(X^2-\alpha^6X+\alpha^0)\\ &= X^4-(\alpha^4+\alpha^6)X^3+(\alpha^3+\alpha^{10}+\alpha^0)X^2-(\alpha^4+\alpha^9)X+\alpha^3\\ &= X^4-\alpha^3X^3+\alpha^0X^2-\alpha^1X+\alpha^3\end{aligned}$$

Following the format of low order to high order, and changing negative signs to positive, since in the binary field $+1=-1$, the generator $\mathbf{g}(X)$ can be expressed as

$$\mathbf{g}(X) = \alpha^3+\alpha^1X+\alpha^0X^2+\alpha^3X^3+X^4 \tag{8.22}$$

8.1.5.1 Encoding in Systematic Form

Since R-S codes are cyclic codes, encoding in systematic form is analogous to the binary encoding procedure established in Section 6.7.3. We can think of shifting a message polynomial $\mathbf{m}(X)$ into the rightmost k stages of a codeword register and then appending a parity polynomial $\mathbf{p}(X)$ by placing it in the leftmost $n-k$ stages. Therefore we multiply $\mathbf{m}(X)$ by X^{n-k}, thereby manipulating the message polynomial algebraically so that it is right-shifted $n-k$ positions. In Chapter 6, this is shown in Equation (6.61) in the context of binary encoding. Next, we divide X^{n-k} $\mathbf{m}(X)$ by the generator polynomial $\mathbf{g}(X)$, which is written as

$$X^{n-k}\,\mathbf{m}(X) = \mathbf{q}(X)\,\mathbf{g}(X)+\mathbf{p}(X) \tag{8.23}$$

where $\mathbf{q}(X)$ and $\mathbf{p}(X)$ are quotient and remainder polynomials, respectively. As in the binary case, the remainder is the parity. Equation (8.23) can also be expressed as

$$\mathbf{p}(X) = X^{n-k}\,\mathbf{m}(X)\text{ modulo }\mathbf{g}(X) \tag{8.24}$$

The resulting codeword polynomial $\mathbf{U}(X)$, shown in Equation (6.64), is rewritten as

$$\mathbf{U}(X) = \mathbf{p}(X)+X^{n-k}\,\mathbf{m}(X) \tag{8.25}$$

We demonstrate the steps implied by Equations (8.24) and (8.25) by encoding the three-symbol message

$$\underbrace{010}_{\alpha^1}\quad\underbrace{110}_{\alpha^3}\quad\underbrace{111}_{\alpha^5}$$

with the (7, 3) R-S code whose generator polynomial is given in Equation (8.22). We first multiply (upshift) the message polynomial $\alpha^1+\alpha^3X+\alpha^5X^2$ by $X^{n-k}=X^4$, yielding $\alpha^1X^4+\alpha^3X^5+\alpha^5X^6$. We next divide this upshifted message polynomial by the generator polynomial in Equation (8.22), $\alpha^3+\alpha^1X+\alpha^0X^2+\alpha^3X^3+X^4$. Polynomial division with nonbinary coefficients is more tedious than its binary counterpart (see Example 6.9), because the required operations of addition (subtraction) and multiplication (division) must follow the rules in Tables 8.2 and 8.3, respectively. It is left as an exercise for the reader to verify that this polynomial division results in the following remainder (parity) polynomial:

$$\mathbf{p}(X) = \alpha^0 + \alpha^2 X + \alpha^4 X^2 + \alpha^6 X^3$$

Then, from Equation (8.25), the codeword polynomial can be written as

$$\mathbf{U}(X) = \alpha^0 + \alpha^2 X + \alpha^4 X^2 + \alpha^6 X^3 + \alpha^1 X^4 + \alpha^3 X^5 + \alpha^5 X^6$$

8.1.5.2 Systematic Encoding with an (n – k)-Stage Shift Register

Using circuitry to encode a 3-symbol sequence in systematic form with the (7, 3) R-S code described by $\mathbf{g}(X)$ in Equation (8.22) requires the implementation of a LFSR, as shown in Figure 8.9. It can be easily verified that the multiplier terms in Figure 8.9 taken from left to right correspond to the coefficients of the polynomial in Equation (8.22) (low order to high order). This encoding process is the nonbinary equivalent of the cyclic encoding that was described in Section 6.7.5. Here, corresponding to Equation (8.20), the (7, 3) R-S nonzero codewords are made up of $2^m - 1 = 7$ symbols, and each symbol is made of $m = 3$ bits.

Notice the similarity amongst Figures 8.9, 6.18, and 6.19. In all three cases the number of stages in the shift register is $n - k$. The figures in Chapter 6 illustrate binary examples where each shift-register stage holds 1 bit. Here the example is nonbinary, so that each stage in the shift register of Figure 8.9 holds a 3-bit symbol. In Figure 6.18, the coefficients labeled $g_1, g_2, \ldots$ are binary. Therefore, they take on values of 1 or 0, simply dictating the presence or absence of a connection in the LFSR. However in Figure 8.9, since each coefficient is specified by 3-bits, it can take on one of 8 values.

The nonbinary operation implemented by the encoder of Figure 8.9, forming codewords in a systematic format, proceeds in the same way as the binary one. The steps can be described as follows:

1. Switch 1 is closed during the first k clock cycles to allow shifting the message symbols into the (n – k)-stage shift register.

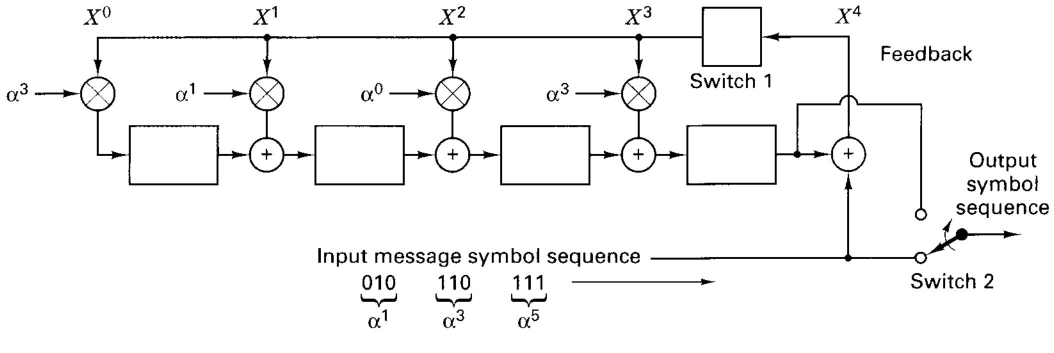


Figure 8.9 LFSR Encoder for a (7,3) R–S code.

2. Switch 2 is in the down position during the first k clock cycles in order to allow simultaneous transfer of the message symbols directly to an output register (not shown in Figure 8.9).
3. After transfer of the kth message symbol to the output register, switch 1 is opened and switch 2 is moved to the up position.
4. The remaining $(n - k)$ clock cycles clear the parity symbols contained in the shift register by moving them to the output register.
5. The total number of clock cycles is equal to n, and the contents of the output register is the codeword polynomial $\mathbf{p}(X) + X^{n-k}\mathbf{m}(X)$, where $\mathbf{p}(X)$ represents the parity symbols, and $\mathbf{m}(X)$ the message symbols in polynomial form.

We use the same symbol sequence that was chosen as a test message in Section 8.1.5.1, and we write

$$\underbrace{010}_{\alpha^1} \quad \underbrace{110}_{\alpha^3} \quad \underbrace{111}_{\alpha^5}$$

where the rightmost symbol is the earliest symbol, and the rightmost bit is the earliest bit. The operational steps during the first $k = 3$ shifts of the encoding circuit of Figure 8.9 are as follows:

INPUT QUEUE			CLOCK CYCLE	REGISTER CONTENTS				FEEDBACK
α^1	α^3	α^5	0	0	0	0	0	α^5
	α^1	α^3	1	α^1	α^6	α^5	α^1	α^0
		α^1	2	α^3	0	α^2	α^2	α^4
		—	3	α^0	α^2	α^4	α^6	—

After the third clock cycle, the register contents are the 4 parity symbols, α^0, α^2, α^4, and α^6, as shown. Then, switch 1 of the circuit is opened, switch 2 is toggled to the up position, and the parity symbols contained in the register are shifted to the output. Therefore the output codeword, written in polynomial form, can be expressed as

$$\mathbf{U}(X) = \sum_{n=0}^{6} u_n X^n$$

$$\mathbf{U}(X) = \alpha^0 + \alpha^2 X + \alpha^4 X^2 + \alpha^6 X^3 + \alpha^1 X^4 + \alpha^3 X^5 + \alpha^5 X^6 \tag{8.26}$$

$$= (100) + (001)\,X + (011)\,X^2 + (101)\,X^3 + (010)\,X^4 + (110)\,X^5 + (111)\,X^6$$

The process of verifying the contents of the register at various clock cycles is somewhat more tedious than in the binary case. Here, the field elements must be added and multiplied by using Table 8.2 and Table 8.3, respectively.

The roots of a generator polynomial $\mathbf{g}(X)$ must also be the roots of the codeword generated by $\mathbf{g}(X)$, because a valid codeword is of the form

$$\mathbf{U}(X) = \mathbf{m}(X)\,\mathbf{g}(X) \tag{8.27}$$

Therefore an arbitrary codeword, when evaluated at any root of $\mathbf{g}(X)$, must yield zero. It is of interest to verify that the codeword polynomial in Equation (8.26) does indeed yield zero when evaluated at the 4 roots of $\mathbf{g}(X)$. In other words, this means checking that

$$\mathbf{U}(\alpha) = \mathbf{U}(\alpha^2) = \mathbf{U}(\alpha^3) = \mathbf{U}(\alpha^4) = \mathbf{0}$$

Evaluating each term independently yields

$$\begin{aligned}
\mathbf{U}(\alpha) &= \alpha^0 + \alpha^3 + \alpha^6 + \alpha^9 + \alpha^5 + \alpha^8 + \alpha^{11} \\
&= \alpha^0 + \alpha^3 + \alpha^6 + \alpha^2 + \alpha^5 + \alpha^1 + \alpha^4 \\
&= \alpha^1 + \alpha^0 + \alpha^6 + \alpha^4 \\
&= \alpha^3 + \alpha^3 = \mathbf{0} \\
\mathbf{U}(\alpha^2) &= \alpha^0 + \alpha^4 + \alpha^8 + \alpha^{12} + \alpha^9 + \alpha^{13} + \alpha^{17} \\
&= \alpha^0 + \alpha^4 + \alpha^1 + \alpha^5 + \alpha^2 + \alpha^6 + \alpha^3 \\
&= \alpha^5 + \alpha^6 + \alpha^0 + \alpha^3 \\
&= \alpha^1 + \alpha^1 = \mathbf{0} \\
\mathbf{U}(\alpha^3) &= \alpha^0 + \alpha^5 + \alpha^{10} + \alpha^{15} + \alpha^{13} + \alpha^{18} + \alpha^{23} \\
&= \alpha^0 + \alpha^5 + \alpha^3 + \alpha^1 + \alpha^6 + \alpha^4 + \alpha^2 \\
&= \alpha^4 + \alpha^0 + \alpha^3 + \alpha^2 \\
&= \alpha^5 + \alpha^5 = \mathbf{0} \\
\mathbf{U}(\alpha^4) &= \alpha^0 + \alpha^6 + \alpha^{12} + \alpha^{18} + \alpha^{17} + \alpha^{23} + \alpha^{29} \\
&= \alpha^0 + \alpha^6 + \alpha^5 + \alpha^4 + \alpha^3 + \alpha^2 + \alpha^1 \\
&= \alpha^2 + \alpha^0 + \alpha^5 + \alpha^1 \\
&= \alpha^6 + \alpha^6 = \mathbf{0}
\end{aligned}$$

This demonstrates the expected results that a codeword evaluated at any root of $\mathbf{g}(X)$ must yield zero.

8.1.6 Reed–Solomon Decoding

In Section 8.1.5, a test message encoded in systematic form using a (7, 3) R–S code, resulted in a codeword polynomial described by Equation (8.26). Now, assume that during transmission, this codeword becomes corrupted so that 2 symbols are received in error. (This number of errors corresponds to the maximum error-correcting capability of the code.) For this 7-symbol codeword example, the error pattern can be described in polynomial form as

$$\mathbf{e}(X) = \sum_{n=0}^{6} e_n X^n \tag{8.28}$$

For this example, let the double-symbol error be such that

$$\mathbf{e}(X) = 0 + 0X + 0X^2 + \alpha^2 X^3 + \alpha^5 X^4 + 0X^5 + 0X^6 \tag{8.29}$$

$$= (000) + (000)X + (000)X^2 + (001)X^3 + (111)X^4 + (000)X^5 + (000)X^6$$

In other words, one parity symbol has been corrupted with a 1-bit error (seen as α^2), and one data symbol has been corrupted with a 3-bit error (seen as α^5). The received corrupted-codeword polynomial $\mathbf{r}(X)$ is then represented by the sum of the transmitted-codeword polynomial and the error-pattern polynomial as follows:

$$\mathbf{r}(X) = \mathbf{U}(X) + \mathbf{e}(X) \tag{8.30}$$

Following Equation (8.30), we add $\mathbf{U}(X)$ from Equation (8.26) to $\mathbf{e}(X)$ from Equation (8.29) to yield

$$\begin{aligned}\mathbf{r}(X) &= (100) + (001)X + (011)X^2 + (100)X^3 + (101)X^4 + (110)X^5 + (111)X^6 \\ &= \alpha^0 + \alpha^2 X + \alpha^4 X^2 + \alpha^0 X^3 + \alpha^6 X^4 + \alpha^3 X^5 + \alpha^5 X^6 \end{aligned} \tag{8.31}$$

In this 2-symbol error-correction example, there are four unknowns—two error locations and two error values. Notice an important difference between the nonbinary decoding of $\mathbf{r}(X)$ that we are faced with in Equation (8.31) and the binary decoding that was described in Chapter 6. In binary decoding, the decoder only needs to find the error locations. Knowledge that there is an error at a particular location dictates that the bit must be "flipped" from a 1 to a 0, or vice versa. But here, the nonbinary symbols require that we not only learn the error locations, but that we also determine the correct symbol values at those locations. Since there are four unknowns in this example, four equations are required for their solution.

8.1.6.1 Syndrome Computation

Recall from Section 6.4.7, that the *syndrome* is the result of a parity check performed on $\mathbf{r}$ to determine whether $\mathbf{r}$ is a valid member of the codeword set. If in fact $\mathbf{r}$ is a member, then the syndrome $\mathbf{S}$ has value $\mathbf{0}$. Any nonzero value of $\mathbf{S}$ indicates the presence of errors. Similar to the binary case, the syndrome $\mathbf{S}$ is made up of $n - k$ symbols, $\{S_i\}$ ($i = 1, \ldots, n - k$). Thus, for this (7, 3) R–S code, there are four symbols in every syndrome vector; their values can be computed from the received polynomial $\mathbf{r}(X)$. Note how the computation is facilitated by the structure of the code, given by Equation (8.27) and rewritten as

$$\mathbf{U}(X) = \mathbf{m}(X)\,\mathbf{g}(X)$$

From this structure it can be seen that every valid codeword polynomial $\mathbf{U}(X)$ is a multiple of the generator polynomial $\mathbf{g}(X)$. Therefore, the roots of $\mathbf{g}(X)$ must also be the roots of $\mathbf{U}(X)$. Since $\mathbf{r}(X) = \mathbf{U}(\mathrm{x}) + \mathbf{e}(X)$, then $\mathbf{r}(X)$ evaluated at each of the roots of $\mathbf{g}(X)$ should yield zero only when it is a valid codeword. Any nonzero result is an indication that an error is present. The computation of a syndrome symbol can be described as

$$S_i = \mathbf{r}(X)\Big|_{X=\alpha^i} = \mathbf{r}(\alpha^i) \quad i = 1, \cdots, n - k \tag{8.32}$$

where $\mathbf{r}(X)$ contains the postulated 2-symbol errors as shown in Equation (8.29). If $\mathbf{r}(X)$ were a valid codeword, it would cause each syndrome symbol S_i to equal 0. For this example, the four syndrome symbols are found as follows:

$$\begin{aligned} S_1 = \mathbf{r}(\alpha) &= \alpha^0 + \alpha^3 + \alpha^6 + \alpha^3 + \alpha^{10} + \alpha^8 + \alpha^{11} \\ &= \alpha^0 + \alpha^3 + \alpha^6 + \alpha^3 + \alpha^2 + \alpha^1 + \alpha^4 \\ &= \alpha^3 \end{aligned} \tag{8.33}$$

$$\begin{aligned} S_2 = \mathbf{r}(\alpha^2) &= \alpha^0 + \alpha^4 + \alpha^8 + \alpha^6 + \alpha^{14} + \alpha^{13} + \alpha^{17} \\ &= \alpha^0 + \alpha^4 + \alpha^1 + \alpha^6 + \alpha^0 + \alpha^6 + \alpha^3 \\ &= \alpha^5 \end{aligned} \tag{8.34}$$

$$\begin{aligned} S_3 = \mathbf{r}(\alpha^3) &= \alpha^0 + \alpha^5 + \alpha^{10} + \alpha^9 + \alpha^{18} + \alpha^{18} + \alpha^{23} \\ &= \alpha^0 + \alpha^5 + \alpha^3 + \alpha^2 + \alpha^4 + \alpha^4 + \alpha^2 \\ &= \alpha^6 \end{aligned} \tag{8.35}$$

$$\begin{aligned} S_4 = \mathbf{r}(\alpha^4) &= \alpha^0 + \alpha^6 + \alpha^{12} + \alpha^{12} + \alpha^{22} + \alpha^{23} + \alpha^{29} \\ &= \alpha^0 + \alpha^6 + \alpha^5 + \alpha^5 + \alpha^1 + \alpha^2 + \alpha^1 \\ &= 0 \end{aligned} \tag{8.36}$$

The results confirm that the received codeword contains an error (which we inserted) since $\mathbf{S} \neq \mathbf{0}$.

Example 8.3 A Secondary Check on the Syndrome Values

For the (7, 3) R–S code example under consideration, the error pattern is known since it was chosen earlier. Recall the property of codes presented in Section 6.4.8.1 when describing the standard array. Each element of a coset (row) in the standard array has the same syndrome. Show that this property is also true for the R-S code by evaluating the error polynomial $\mathbf{e}(X)$ at the roots of $\mathbf{g}(X)$ to demonstrate that it must yield the same syndrome values as when $\mathbf{r}(X)$ is evaluated at the roots of $\mathbf{g}(X)$. In other words, it must yield the same values obtained in Equations (8.33) through (8.36).

Solution

$$S_i = \mathbf{r}(X)\Big|_{X=\alpha^i} = \mathbf{r}(\alpha^i) \quad i = 1, 2, \cdots, n-k$$

$$S_i = [\mathbf{U}(X) + \mathbf{e}(X)]\Big|_{X=\alpha^i} = \mathbf{U}(\alpha^i) + \mathbf{e}(\alpha^i)$$

$$S_i = \mathbf{r}(\alpha^i) = \mathbf{U}(\alpha^i) + \mathbf{e}(\alpha^i) = 0 + \mathbf{e}(\alpha^i)$$

From Equation (8.29), $\mathbf{e}(X) = \alpha^2 X^3 + \alpha^5 X^4$; therefore,

$$\begin{aligned} S_1 = \mathbf{e}(\alpha^1) &= \alpha^5 + \alpha^9 \\ &= \alpha^5 + \alpha^2 \\ &= \alpha^3 \end{aligned}$$

$$\begin{aligned} S_2 = \mathbf{e}(\alpha^2) &= \alpha^8 + \alpha^{13} \\ &= \alpha^1 + \alpha^6 \\ &= \alpha^5 \end{aligned}$$

$$\begin{aligned} S_3 = \mathbf{e}(\alpha^3) &= \alpha^{11} + \alpha^{17} \\ &= \alpha^4 + \alpha^3 \\ &= \alpha^6 \end{aligned}$$

$$\begin{aligned} S_4 = \mathbf{e}(\alpha^4) &= \alpha^{14} + \alpha^{21} \\ &= \alpha^0 + \alpha^0 \\ &= 0 \end{aligned}$$

These results confirm that the syndrome values are the same, whether obtained by evaluating $\mathbf{e}(X)$ at the roots of $\mathbf{g}(X)$, or $\mathbf{r}(X)$ at the roots of $\mathbf{g}(X)$.

8.1.6.2 Error Location

Suppose there are v errors in the codeword at location $X^{j_1}, X^{j_2}, \ldots X^{j_v}$. Then, the error polynomial shown in Equations (8.28) and (8.29) can be written as

$$\mathbf{e}(X) = e_{j_1} X^{j_1} + e_{j_2} X^{j_2} + \cdots + e_{j_v} X^{j_v} \tag{8.37}$$

The indices 1, 2, . . . , v refer to the 1st, 2nd, . . . , vth errors, and the index j refers to the error location. To correct the corrupted codeword, each error value e_{j_ℓ} and its location X^{j_ℓ}, where $\ell = 1, 2, \ldots v$ must be determined. We define an error locator number as $\beta_\ell = \alpha^{j_\ell}$. Next, we obtain the $n - k = 2t$ syndrome symbols by substituting α^i into the received polynomial for $i = 1, 2, \ldots, 2t$:

$$\begin{aligned} S_1 = \mathbf{r}(\alpha) &= e_{j_1}\beta_1 + e_{j_2}\beta_2 + \cdots + e_{j_v}\beta_v \\ S_2 = \mathbf{r}(\alpha^2) &= e_{j_1}\beta_1^2 + e_{j_2}\beta_2^2 + \cdots + e_{j_v}\beta_v^2 \\ &\vdots \\ S_{2t} = \mathbf{r}(\alpha^{2t}) &= e_{j_1}\beta_1^{2t} + e_{j_2}\beta_2^{2t} + \cdots + e_{j_v}\beta_v^{2t} \end{aligned} \tag{8.38}$$

There are $2t$ unknowns (t error values and t locations), and $2t$ simultaneous equations. However, these $2t$ simultaneous equations cannot be solved in the usual way because they are nonlinear (as some of the unknowns have exponents). Any technique that solves this system of equations is known as a Reed–Solomon decoding algorithm.

When a nonzero syndrome vector (one or more of its symbols are nonzero) has been computed, it signifies that an error has been received. Next, it is necessary to learn the location of the error or errors. An error-locator polynomial can be defined as

$$\begin{aligned} \boldsymbol{\sigma}(X) &= (1 + \beta_1 X)(1 + \beta_2 X) \cdots (1 + \beta_v X) \\ &= 1 + \sigma_1 X + \sigma_2 X^2 + \cdots + \sigma_v X^v \end{aligned} \tag{8.39}$$

The roots of $\boldsymbol{\sigma}(X)$ are $1/\beta_1, 1/\beta_2, \ldots, 1/\beta_v$. The reciprocal of the roots of $\boldsymbol{\sigma}(X)$ are the error-location numbers of the error pattern $\mathbf{e}(X)$. Then using autoregressive modeling techniques [5], we form a matrix from the syndromes, where the first t syndromes are used to predict the next syndrome. That is,

$$\begin{bmatrix} S_1 & S_2 & S_3 & \cdots & S_{t-1} & S_t \\ S_2 & S_3 & S_4 & \cdots & S_t & S_{t+1} \\ & & & \vdots & & \\ S_{t-1} & S_t & S_{t+1} & \cdots & S_{2t-3} & S_{2t-2} \\ S_t & S_{t+1} & S_{t+2} & \cdots & S_{2t-2} & S_{2t-1} \end{bmatrix} \begin{bmatrix} \sigma_t \\ \sigma_{t-1} \\ \vdots \\ \sigma_2 \\ \sigma_1 \end{bmatrix} = \begin{bmatrix} -S_{t+1} \\ -S_{t+2} \\ \vdots \\ -S_{2t-1} \\ -S_{2t} \end{bmatrix} \tag{8.40}$$

We apply the autoregressive model of Equation (8.40) by using the largest dimensioned matrix that has a nonzero determinant. For the (7, 3) double symbol error-correcting R-S code, the matrix size is 2×2, and the model is written as

$$\begin{bmatrix} S_1 & S_2 \\ S_2 & S_3 \end{bmatrix} \begin{bmatrix} \sigma_2 \\ \sigma_1 \end{bmatrix} = \begin{bmatrix} S_3 \\ S_4 \end{bmatrix} \tag{8.41}$$

$$\begin{bmatrix} \alpha^3 & \alpha^5 \\ \alpha^5 & \alpha^6 \end{bmatrix} \begin{bmatrix} \sigma_2 \\ \sigma_1 \end{bmatrix} = \begin{bmatrix} \alpha^6 \\ 0 \end{bmatrix} \tag{8.42}$$

To solve for the coefficients σ_1 and σ_2 of the error-locator polynomial $\boldsymbol{\sigma}(X)$, we first take the inverse of the matrix in Equation (8.42). The inverse of a matrix $[A]$ is found as follows:

$$\text{Inv}\,[A] = \frac{\text{cofactor}\,[A]}{\det\,[A]}$$

Therefore,

$$\det \begin{bmatrix} \alpha^3 & \alpha^5 \\ \alpha^5 & \alpha^6 \end{bmatrix} = \alpha^3\alpha^6 - \alpha^5\alpha^5 = \alpha^9 + \alpha^{10} \tag{8.43}$$

$$= \alpha^2 + \alpha^3 = \alpha^5$$

$$\text{cofactor} \begin{bmatrix} \alpha^3 & \alpha^5 \\ \alpha^5 & \alpha^6 \end{bmatrix} = \begin{bmatrix} \alpha^6 & \alpha^5 \\ \alpha^5 & \alpha^3 \end{bmatrix} \tag{8.44}$$

and

$$\text{Inv} \begin{bmatrix} \alpha^3 & \alpha^5 \\ \alpha^5 & \alpha^6 \end{bmatrix} = \frac{\begin{bmatrix} \alpha^6 & \alpha^5 \\ \alpha^5 & \alpha^3 \end{bmatrix}}{\alpha^5} = \alpha^{-5} \begin{bmatrix} \alpha^6 & \alpha^5 \\ \alpha^5 & \alpha^3 \end{bmatrix} \tag{8.45}$$

$$= \alpha^2 \begin{bmatrix} \alpha^6 & \alpha^5 \\ \alpha^5 & \alpha^3 \end{bmatrix} = \begin{bmatrix} \alpha^8 & \alpha^7 \\ \alpha^7 & \alpha^5 \end{bmatrix} = \begin{bmatrix} \alpha^1 & \alpha^0 \\ \alpha^0 & \alpha^5 \end{bmatrix}$$

Safety Check. If the inversion was performed correctly, then the multiplication of the original matrix by the inverted matrix should yield an identity matrix:

$$\begin{bmatrix} \alpha^3 & \alpha^5 \\ \alpha^5 & \alpha^6 \end{bmatrix} \begin{bmatrix} \alpha^1 & \alpha^0 \\ \alpha^0 & \alpha^5 \end{bmatrix} = \begin{bmatrix} \alpha^4 + \alpha^5 & \alpha^3 + \alpha^{10} \\ \alpha^6 + \alpha^6 & \alpha^5 + \alpha^{11} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{8.46}$$

Continuing from Equation (8.42), we begin our search for the error locations by solving for the coefficients of the error-locator polynomial $\boldsymbol{\sigma}(X)$, as follows:

$$\begin{bmatrix} \sigma_2 \\ \sigma_1 \end{bmatrix} = \begin{bmatrix} \alpha^3 & \alpha^5 \\ \alpha^5 & \alpha^6 \end{bmatrix}^{-1} \begin{bmatrix} \alpha^6 \\ 0 \end{bmatrix} = \begin{bmatrix} \alpha^1 & \alpha^0 \\ \alpha^0 & \alpha^5 \end{bmatrix} \begin{bmatrix} \alpha^6 \\ 0 \end{bmatrix} = \begin{bmatrix} \alpha^7 \\ \alpha^6 \end{bmatrix} = \begin{bmatrix} \alpha^0 \\ \alpha^6 \end{bmatrix} \tag{8.47}$$

From Equations (8.39) and (8.47),

$$\boldsymbol{\sigma}(X) = \alpha^0 + \sigma_1 X + \sigma_2 X^2 \tag{8.48}$$

$$= \alpha^0 + \alpha^6 X + \alpha^0 X^2$$

The roots of $\boldsymbol{\sigma}(X)$ are the reciprocals of the error locations. Once these roots are located, the error locations will be known. In general, the roots of $\boldsymbol{\sigma}(X)$ may be one or more of the elements of the field. We determine these roots by exhaustive testing of the $\boldsymbol{\sigma}(X)$ polynomial with each of the field elements, as shown below. Any element X that yields $\boldsymbol{\sigma}(X) = \mathbf{0}$ is a root, and allows us to locate an error:

$$\boldsymbol{\sigma}(\alpha^0) = \alpha^0 + \alpha^6 + \alpha^0 = \alpha^6 \neq \mathbf{0}$$

$$\boldsymbol{\sigma}(\alpha^1) = \alpha^0 + \alpha^7 + \alpha^2 = \alpha^2 \neq \mathbf{0}$$

$$\boldsymbol{\sigma}(\alpha^2) = \alpha^0 + \alpha^8 + \alpha^4 = \alpha^6 \neq \mathbf{0}$$

$$\boldsymbol{\sigma}(\alpha^3) = \alpha^0 + \alpha^9 + \alpha^6 = \mathbf{0} \Rightarrow \text{ERROR}$$

$$\boldsymbol{\sigma}(\alpha^4) = \alpha^0 + \alpha^{10} + \alpha^8 = \mathbf{0} \Rightarrow \text{ERROR}$$

$$\boldsymbol{\sigma}(\alpha^5) = \alpha^0 + \alpha^{11} + \alpha +^{10} = \alpha^2 \neq \mathbf{0}$$

$$\boldsymbol{\sigma}(\alpha^6) = \alpha^0 + \alpha^{12} + \alpha^{12} = \alpha^0 \neq \mathbf{0}$$

As seen in Equation (8.39), the error locations are at the inverse of the roots of the polynomial. Therefore $\boldsymbol{\sigma}(\alpha^3) = \mathbf{0}$ indicates that one root exits at $1/\beta_\ell = \alpha^3$. Thus, $\beta_\ell = 1/\alpha^3 = \alpha^4$. Similarly, $\boldsymbol{\sigma}(\alpha^4) = \mathbf{0}$ indicates that another root exits at $1/\beta_{\ell'} = 1/\alpha^4 = \alpha^3$, where (for this example) ℓ and ℓ' refer to the 1st and 2nd error respectively. Since there are 2-symbol errors here, the error polynomial is of the form

$$\mathbf{e}(X) = e_{j_1} X^{j_1} + e_{j_2} X^{j_2} \tag{8.49}$$

The two errors were found at locations α^3 and α^4. Note that the indexing of the error-location numbers is completely arbitrary. Thus, for this example, we can designate the $\beta_\ell = \alpha^{j_\ell}$ values as $\beta_1 = \alpha^{j_1} = \alpha^3$ and $\beta_2 = \alpha^{j_2} = \alpha^4$.

8.1.6.3 Error Values

An error had been denoted e_{j_ℓ}, where the index j refers to the error location and the index ℓ identifies the ℓth error. Since each error value is coupled to a particular location, the notation can be simplified by denoting e_{j_ℓ} simply as e_ℓ. Now, preparing to determine the error values e_1 and e_2, associated with locations $\beta_1 = \alpha^3$ and $\beta_2 = \alpha^4$, any of the four syndrome equations can be used. From Equation (8.38), let us use S_1 and S_2:

$$S_1 = \mathbf{r}(\alpha) = e_1\beta_1 + e_2\beta_2 \tag{8.50}$$

$$S_2 = \mathbf{r}(\alpha^2) = e_1\beta_1^2 + e_2\beta_2^2$$

We can write these equations in matrix form as follows:

$$\begin{bmatrix} \beta_1 & \beta_2 \\ \beta_1^2 & \beta_2^2 \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \end{bmatrix} = \begin{bmatrix} S_1 \\ S_2 \end{bmatrix} \tag{8.51}$$

$$\begin{bmatrix} \alpha^3 & \alpha^4 \\ \alpha^6 & \alpha^8 \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \end{bmatrix} = \begin{bmatrix} \alpha^3 \\ \alpha^5 \end{bmatrix} \tag{8.52}$$

To solve for the error values e_1 and e_2, the matrix in Equation (8.52) is inverted in the usual way, yielding

$$\begin{aligned}
\text{Inv}\begin{bmatrix} \alpha^3 & \alpha^4 \\ \alpha^6 & \alpha^1 \end{bmatrix} &= \frac{\begin{bmatrix} \alpha^1 & \alpha^4 \\ \alpha^6 & \alpha^3 \end{bmatrix}}{\alpha^3\alpha^1 - \alpha^6\alpha^4} \\
&= \frac{\begin{bmatrix} \alpha^1 & \alpha^4 \\ \alpha^6 & \alpha^3 \end{bmatrix}}{\alpha^4 + \alpha^3} = \alpha^{-6}\begin{bmatrix} \alpha^1 & \alpha^4 \\ \alpha^6 & \alpha^3 \end{bmatrix} = \alpha^1\begin{bmatrix} \alpha^1 & \alpha^4 \\ \alpha^6 & \alpha^3 \end{bmatrix} \qquad (8.53) \\
&= \begin{bmatrix} \alpha^2 & \alpha^5 \\ \alpha^7 & \alpha^4 \end{bmatrix} = \begin{bmatrix} \alpha^2 & \alpha^5 \\ \alpha^0 & \alpha^4 \end{bmatrix}
\end{aligned}$$

Now, we solve Equation (8.52) for the error values, as follows:

$$\begin{bmatrix} e_1 \\ e_2 \end{bmatrix} = \begin{bmatrix} \alpha^2 & \alpha^5 \\ \alpha^0 & \alpha^4 \end{bmatrix}\begin{bmatrix} \alpha^3 \\ \alpha^5 \end{bmatrix} = \begin{bmatrix} \alpha^5 + \alpha^{10} \\ \alpha^3 + \alpha^9 \end{bmatrix} = \begin{bmatrix} \alpha^5 + \alpha^3 \\ \alpha^3 + \alpha^2 \end{bmatrix} = \begin{bmatrix} \alpha^2 \\ \alpha^5 \end{bmatrix} \qquad (8.54)$$

8.1.6.4 Correcting the Received Polynomial with Estimates of the Error Polynomial

From Equation (8.49) and (8.54), the estimated error polynomial is formed, to yield

$$\begin{aligned}
\hat{\mathbf{e}}(X) &= e_1X^{j_1} + e_2X^{j_2} \qquad (8.55) \\
&= \alpha^2X^3 + \alpha^5X^4
\end{aligned}$$

The demonstrated algorithm repairs the received polynomial yielding an estimate of the transmitted codeword, and ultimately delivers a decoded message. That is,

$$\hat{\mathbf{U}}(X) = \mathbf{r}(X) + \hat{\mathbf{e}}(X) = \mathbf{U}(X) + \mathbf{e}(X) + \hat{\mathbf{e}}(X) \qquad (8.56)$$

$$\mathbf{r}(X) = (100) + (001)X + (011)X^2 + (100)X^3 + (101)X^4 + (110)X^5 + (111)X^6$$

$$\hat{\mathbf{e}}(X) = (000) + (000)X + (000)X^2 + (001)X^3 + (111)X^4 + (000)X^5 + (000)X^6$$

$$\begin{aligned}
\hat{\mathbf{U}}(X) &= (100) + (001)X + (011)X^2 + (101)X^3 + (010)X^4 + (110)X^5 + (111)X^6 \\
&= \alpha^0 + \alpha^2X + \alpha^4X^2 + \alpha^6X^3 + \alpha^1X^4 + \alpha^3X^5 + \alpha^5X^6 \qquad (8.57)
\end{aligned}$$

Since the message symbols constitute the rightmost $k = 3$ symbols, the decoded message is

$$\underbrace{010}_{\alpha^1} \quad \underbrace{110}_{\alpha^3} \quad \underbrace{111}_{\alpha^5}$$

which is exactly the test message that was chosen in Section 8.1.5 for this example. (For further reading on R-S coding, see the collection of papers in reference [6].)

8.2 INTERLEAVING AND CONCATENATED CODES

Throughout this and earlier chapters we have assumed that the channel is *memoryless*, since we have considered codes that are designed to combat random independent errors. A channel that has *memory* is one that exhibits mutually dependent signal transmission impairments. A channel that exhibits *multipath fading,* where signals arrive at the receiver over two or more paths of different lengths, is an example of a channel with memory. The effect is that the signals can arrive out of phase with each other, and the cumulative received signal is distorted. Wireless mobile communication channels, as well as ionospheric and tropospheric propagation channels, suffer from such phenomena. (See Chapter 15 for details on fading channels.) Also, some channels suffer from switching noise and other burst noise (e.g., telephone channels or channels disturbed by pulse jamming). All of these time-correlated impairments result in statistical dependence among successive symbol transmissions. That is, the disturbances tend to cause errors that occur in bursts, instead of as isolated events.

Under the assumption that the channel has memory, the errors no longer can be characterized as single randomly distributed bit errors whose occurrence is independent from bit to bit. Most block or convolutional codes are designed to combat random independent errors. The result of a channel having memory on such coded signals is to cause degradation in error performance. Coding techniques for channels with memory have been proposed, but the greatest problem with such coding is the difficulty in obtaining accurate models of the often time-varying statistics of such channels. One technique, which only requires a knowledge of the *duration or span* of the channel memory, *not* its exact statistical characterization, is the use of time diversity or *interleaving.*

Interleaving the coded message before transmission and deinterleaving after reception causes bursts of channel errors to be spread out in time and thus to be handled by the decoder as if they were random errors. Since, in all practical cases, the channel memory decreases with time separation, the idea behind interleaving is to separate the codeword symbols in time. The intervening times are similarly filled by the symbols of other codewords. Separating the symbols in time effectively transforms a channel with memory to a *memoryless* one, and thereby enables the random-error-correcting codes to be useful in a burst-noise channel.

The interleaver shuffles the code symbols over a span of several block lengths (for block codes) or several constraint lengths (for convolutional codes). The span required is determined by the burst duration. The details of the bit redistribution pattern must be known to the receiver in order for the symbol stream to be deinterleaved before being decoded. Figure 8.10 illustrates a simple interleaving example. In Figure 8.10a we see seven uninterleaved codewords, *A* through *G*. Each codeword is comprised of seven code symbols. Let us assume that the code has a single-error-correcting capability within each seven-symbol sequence. If the memory span of the channel is one codeword in duration, such a seven-symbol-time noise burst could destroy the information contained in one or two codewords. However, suppose that, after having encoded the data, the code symbols were then *interleaved* or shuffled, as shown in Figure 8.10b. That is, each code symbol of each codeword is

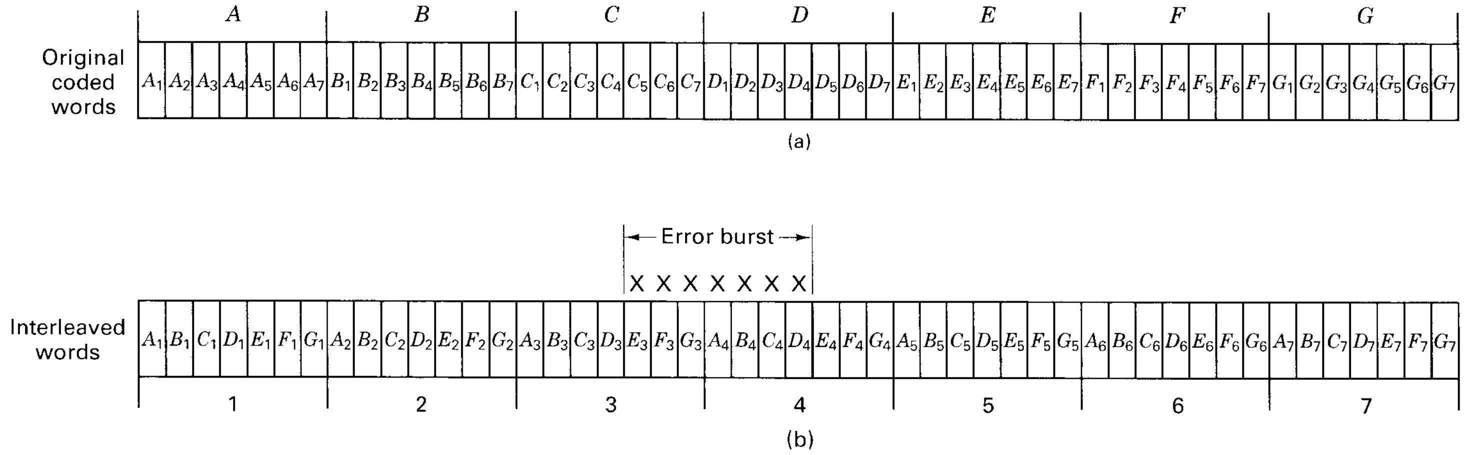

A B C D E F G
Original coded words
A_1 A_2 A_3 A_4 A_5 A_6 A_7 B_1 B_2 B_3 B_4 B_5 B_6 B_7 C_1 C_2 C_3 C_4 C_5 C_6 C_7 D_1 D_2 D_3 D_4 D_5 D_6 D_7 E_1 E_2 E_3 E_4 E_5 E_6 E_7 F_1 F_2 F_3 F_4 F_5 F_6 F_7 G_1 G_2 G_3 G_4 G_5 G_6 G_7
(a)
Error burst
X X X X X X X
Interleaved words
A_1 B_1 C_1 D_1 E_1 F_1 G_1 A_2 B_2 C_2 D_2 E_2 F_2 G_2 A_3 B_3 C_3 D_3 E_3 F_3 G_3 A_4 B_4 C_4 D_4 E_4 F_4 G_4 A_5 B_5 C_5 D_5 E_5 F_5 G_5 A_6 B_6 C_6 D_6 E_6 F_6 G_6 A_7 B_7 C_7 D_7 E_7 F_7 G_7
1 2 3 4 5 6 7
(b)

Figure 8.10 Interleaving example. (a) Original uninterleaved codewords, each comprised of seven code symbols. (b) Interleaved code symbols.

separated from its preinterleaved neighbors by a span of seven symbol times. The interleaved stream is then used to modulate a waveform that is transmitted over the channel. A contiguous channel noise burst occupying seven symbol times is seen in Figure 8.10b, to affect one code symbol from each of the original seven codewords. Upon reception, the stream is first deinterleaved so that it resembles the original coded sequence in Figure 8.10a. Then the stream is decoded. Since each codeword possesses a single-error-correcting capability, the burst noise has no degrading effect on the final sequence.

Interleaving techniques have proven useful for all the block and convolutional codes described here and in earlier chapters. Two types of interleavers are commonly used, *block interleavers* and *convolutional interleavers*. They are each described below.

8.2.1 Block Interleaving

A block interleaver accepts the coded symbols in blocks from the encoder, permutes the symbols, and then feeds the rearranged symbols to the modulator. The usual permutation of the block is accomplished by *filling the columns* of an M-row-by N-column ($M \times N$) array with the encoded sequence. After the array is completely filled, the symbols are then fed to the modulator *one row at a time* and transmitted over the channel. At the receiver, the deinterleaver performs the inverse operation; it accepts the symbols from the demodulator, deinterleaves them, and feeds them to the decoder. Symbols are entered into the deinterleaver array by rows, and removed by columns. Figure 8.11a illustrates an example of an interleaver with $M = 4$ rows and $N = 6$ columns. The entries in the array illustrate the order in which the 24 code symbols are placed into the interleaver. The output sequence to the transmitter consists of code symbols removed from the array by rows, as shown in the figure. The most important characteristics of such a block interleaver are as follows:

1. Any burst of less than N contiguous channel symbol errors results in isolated errors at the deinterlever output that are separated from each other by at least M symbols.
2. Any bN burst of errors, where $b > 1$, results in output bursts from the deinterleaver of no more than $\lceil b \rceil$ symbol errors. Each output burst is separated from the other bursts by no less than $M - \lfloor b \rfloor$ symbols. The notation $\lceil x \rceil$ means the smallest integer no less than x, and $\lfloor x \rfloor$ means the largest integer no greater than x.
3. A periodic sequence of single errors spaced N symbols apart results in a single burst of errors of length M at the deinterleaver output.
4. The interleaver/deinterleaver end-to-end delay is approximately $2MN$ symbol times. To be precise, only $M(N-1)+1$ memory cells need to be filled before transmission can begin (as soon as the first symbol of the last column of the $M \times N$ array is filled). A corresponding number needs to be filled at the receiver before decoding begins. Thus the minimum end-to-end delay is $(2MN - 2M + 2)$ symbol times, not including any channel propagation delay.

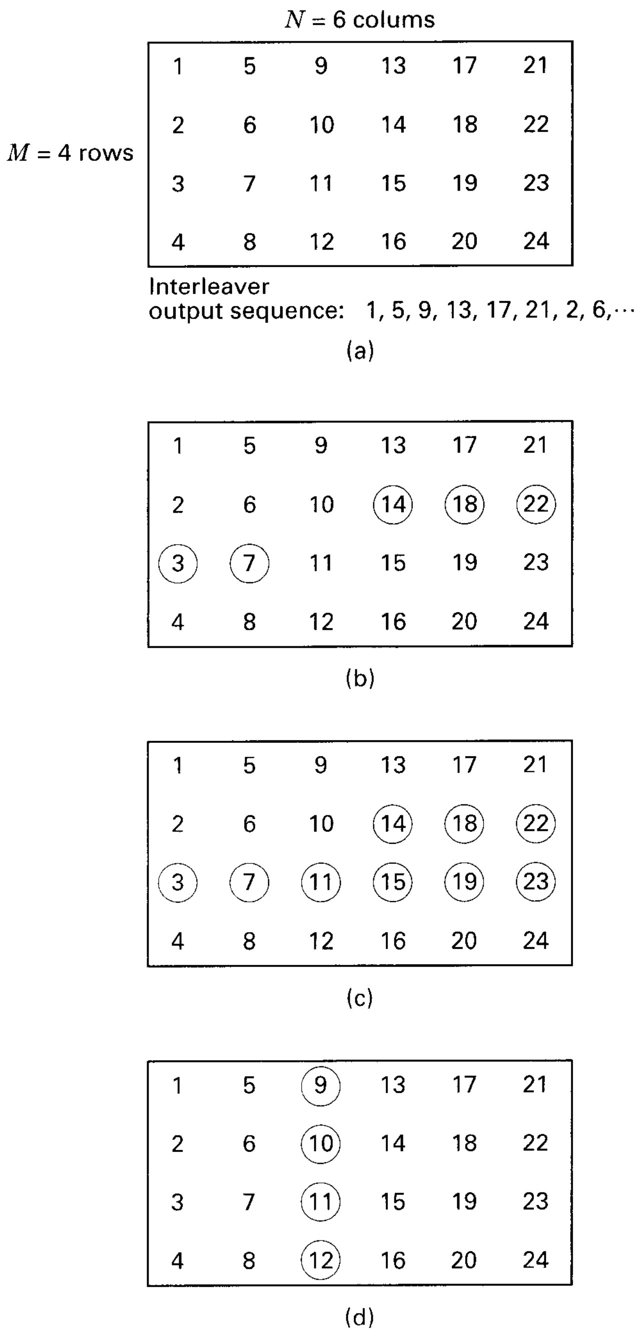


Figure 8.11 Block interleaver example. (a) $M \times N$ block interleaver. (b) Five-symbol error burst. (c) Nine-symbol error burst. (d) Periodic single-error sequence spaced $N = 6$ symbols apart.

5. The memory requirement is MN symbols for each location (interleaver and deinterleaver). However, since the $M \times N$ array needs to be (mostly) filled before it can be read out, a memory of $2MN$ symbols is generally implemented at each location to allow the emptying of one $M \times N$ array while the other is being filled, and vice versa.

Example 8.4 Interleaver Characteristics

Using the $M = 4$, $N = 6$ interleaver structure of Figure 8.11a, verify each of the block interleaver characteristics described above.

Solution

1. Let there be a noise burst of five symbol times, such that the symbols shown encircled in Figure 8.11b experience errors in transmission. After deinterleaving at the receiver, the sequence is

   1 2 ③ 4 5 6 ⑦ 8 9 10 11 12
   13 ⑭ 15 16 17 ⑱ 19 20 21 ㉒ 23 24

   where the encircled symbols are in error. It is seen that the smallest separation between symbols in error is $M = 4$.

2. Let $b = 1.5$ so that $bN = 9$. Figure 8.11c illustrates an example of nine-symbol error burst. After deinterleaving at the receiver, the sequence is

   1 2 ③ 4 5 6 ⑦ 8 9 10 ⑪ 12
   13 ⑭ ⑮ 16 17 ⑱ ⑲ 20 21 ㉒ ㉓ 24

   Again, the encircled symbols are in error. It is seen that the bursts consist of no more than $\lceil 1.5 \rceil = 2$ contiguous symbols and that they are separated by at least $M - \lfloor 1.5 \rfloor = 4 - 1 = 3$ symbols.

3. Figure 8.11d illustrates a sequence of single errors spaced by $N = 6$ symbols apart. After deinterleaving at the receiver, the sequence is

   1 2 3 4 5 6 7 8 ⑨ ⑩ ⑪ ⑫
   13 14 15 16 17 18 19 20 21 22 23 24

   It is seen that the deinterleaved sequence has a singe error burst of length $M = 4$ symbols.

4. End-to-end delay: The minimum end-to-end delay due to the interleaver and deinterleaver is $(2MN - 2M + 2) = 42$ symbol times.

5. Memory requirement: The interleaver and the deinterleaver arrays are each of size $M \times N$. Therefore, storage for $MN = 24$ symbols is required at each end of the channel. As mentioned earlier, storage for $2MN = 48$ symbols would generally be implemented.

Typically, for use with a single-error-correcting code the interleaver parameters are selected such that the number of columns N overbounds the *expected burst length*. The choice of the number of rows M is dependent on the coding scheme used. For block codes, M should be larger than the code block length, while for convolutional codes, M should be larger than the constraint length. Thus a burst of length N can cause at most a single error in any block codeword; similarly, with convolutional codes, there will be at most a single error in any decoding constraint length. For t-error-correcting codes, the choice of N need only overbound the expected burst length divided by t.

8.2.2 Convolutional Interleaving

Convolutional interleavers have been proposed by Ramsey [7] and Forney [8]. The structure proposed by Forney appears in Figure 8.12. The code symbols are sequentially shifted into the bank of N registers; each successive register provides J symbols more storage than did the preceding one. The zeroth register provides no storage (the symbol is transmitted immediately). With each new code symbol the commutator switches to a new register, and the new code symbol is shifted in while the oldest code symbol in that register is shifted out to the modulator/transmitter. After the $(N - 1)$th register, the commutator returns to the zeroth register and starts again. The deinterleaver performs the inverse operation, and the input and output commutators for both interleaving and deinterleaving must be synchronized.

Figure 8.13 illustrates an example of a simple convolutional four-register ($J = 1$) interleaver being loaded by a sequence of code symbols. The synchronized deinterleaver is shown simultaneously feeding the deinterleaved symbols to the decoder. Figure 8.13a shows symbols 1 to 4 being loaded; the $\times$s represent unknown states. Figure 8.13b shows the first four symbols shifted within the registers and the entry of symbols 5 to 8 to the interleaver input. Figure 8.13c shows symbols 9 to 12 entering the interleaver. The deinterleaver is now filled with message symbols, but nothing useful is being fed to the decoder yet. Finally, Figure 8.13d shows symbols

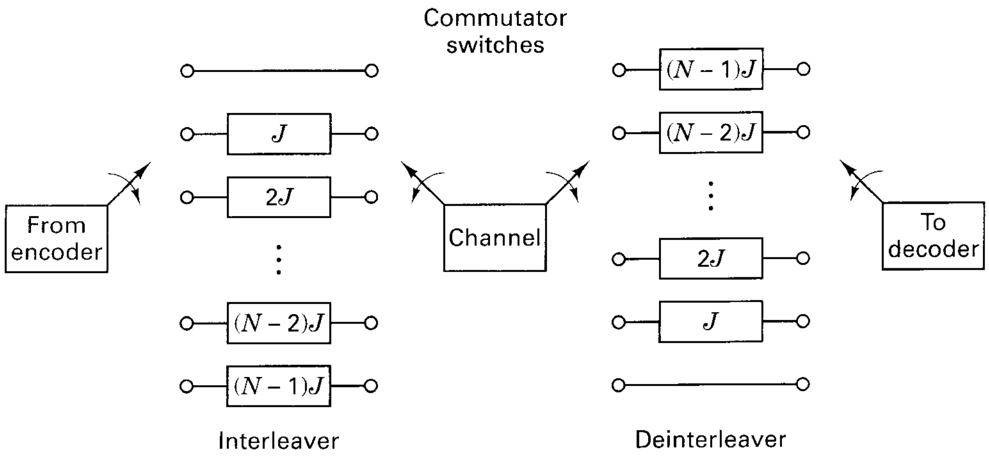


Figure 8.12 Shift register implementation of a convolutional interleaver/deinterleaver.

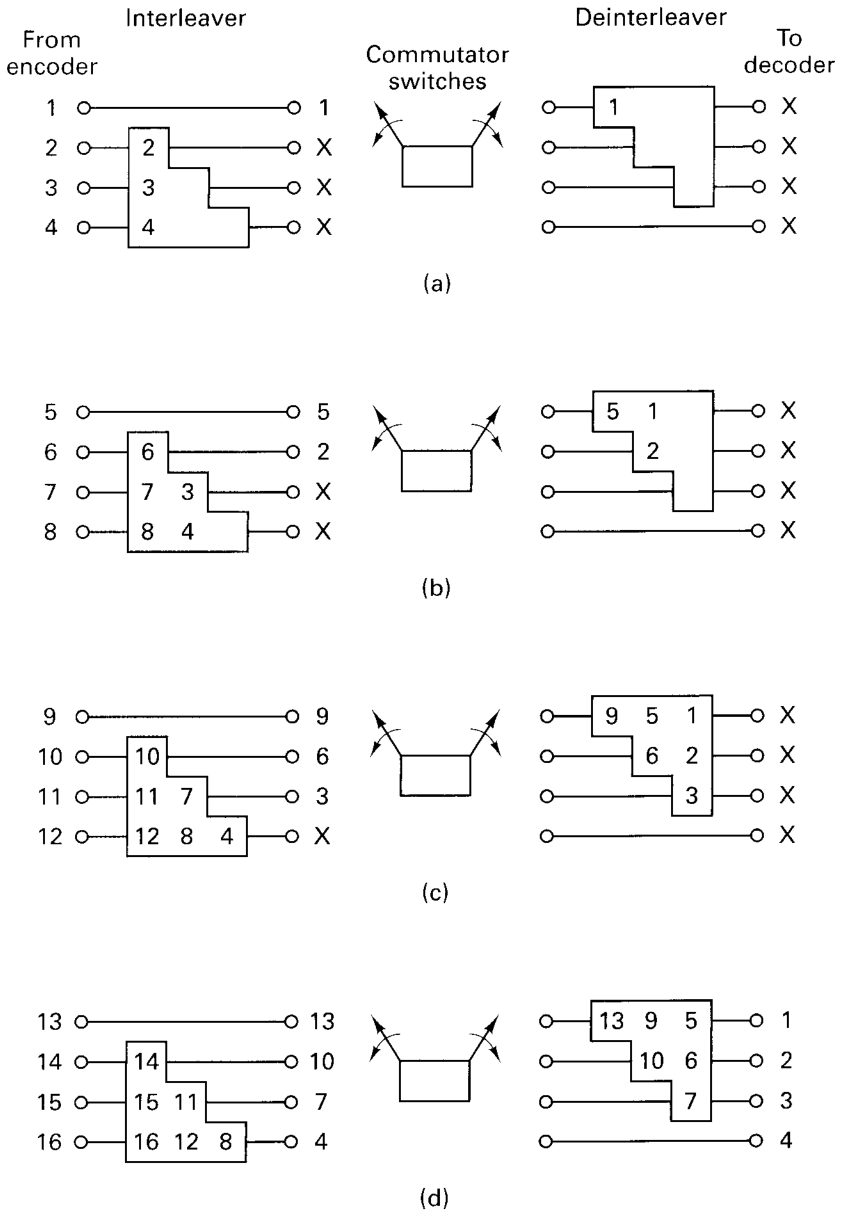


Figure 8.13 Convolutional interleaver/deinterleaver example.

13 to 16 entering the interleaver, and at the output of the deinterleaver, symbols 1 to 4 are being passed to the decoder. The process continues in this way until the entire codeword sequence, in its original preinterleaved form, is presented to the decoder.

The performance of a convolutional interleaver is very similar to that of a block interleaver. The important advantage of convolutional over block interleaving is that with convolutional interleaving the end-to-end delay is $M(N-1)$ symbols, where $M = NJ$, and the memory required is $M(N-1)/2$ at both ends of the channel. Therefore, there is a reduction of one-half in delay and memory over the block interleaving requirements [9].

8.2.3 Concatenated Codes

A concatenated code is one that uses two levels of coding, an inner code and an outer code, to achieve the desired error performance. Figure 8.14 illustrates the order of encoding and decoding. The inner code, the one that interfaces with the modulator/demodulator and channel, is usually configured to correct most of the channel errors. The outer code, usually a higher-rate (lower-redundancy) code, then reduces the probability of error to the specified level. The primary reason for using a concatenated code is to achieve a low error rate with an overall implementation complexity which is less than that which would be required by a single coding operation. In Figure 8.14 an interleaver is shown between the two coding steps. This is usually required to spread any error bursts that may appear at the output of the inner coding operation.

One of the most popular concatenated coding systems uses a Viterbi-decoded convolutional inner code and a Reed–Solomon (R–S) outer code, with interleaving between the two coding steps [2]. Operation of such systems with E_b/N_0 in the range 2.0 to 2.5 dB to achieve $P_B = 10^{-5}$ is feasible with practical hardware [9]. In this system, the demodulator outputs soft quantized code symbols to the inner convolutional decoder, which in turn outputs hard quantized code symbols with bursty errors to the R–S decoder. (In a Viterbi-decoded system, the output errors tend to

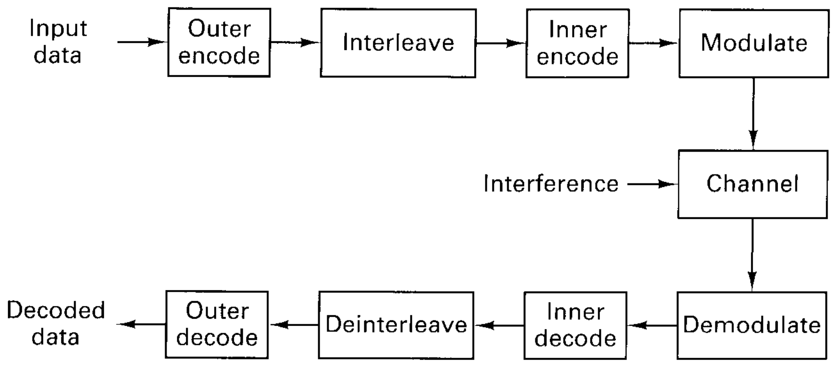


Figure 8.14 Block diagram of a concatenated coding system.

occur in bursts.) The outer R–S code is formed from m-bit segments of the binary data stream. The performance of such a (nonbinary) R–S code depends only on the number of *symbol errors* in the block. The code is undisturbed by burst errors within an m-bit symbol. That is, for a given symbol error, the R–S code performance is the same whether the symbol error is due to one bit being in error or m bits being in error. However, the concatenated system performance is severely degraded by correlated errors among successive symbols. Hence interleaving between codes at the symbol level (not at the bit level) needs to be provided. Reference [10] presents a review of concatenated codes that have been investigated for deep-space communications. In the next section we consider a popular consumer application of symbol interleaving in a concatenated system.

8.3 CODING AND INTERLEAVING APPLIED TO THE COMPACT DISC DIGITAL AUDIO SYSTEM

In 1979, Philips Corp. of the Netherlands and Sony Corp. of Japan defined a standard for the digital storage and reproduction of audio signals, known as the *compact disc (CD) digital audio system.* This CD system has become the world standard for achieving fidelity of sound reproduction that far surpasses any other available technique. A plastic disc 120 mm in diameter is used to store the digitized audio waveform. The waveform is sampled at 44.1 kilosamples/s to provide a recorded bandwidth of 20 kHZ; each audio sample is uniformly quantized to one of 2^{16} levels (16 bits/sample), resulting in a dynamic range of 96 dB and a total harmonic distortion of 0.005%. A single disc (playing time approximately 70 minutes) stores about 10^{10} bits in the form of minute *pits* that are optically scanned by a laser.

There are several sources of channel errors: (1) small unwanted particles or air bubbles in the plastic material or pit inaccuracies arising in manufacturing, and (2) fingerprints or scratches during handling. It is difficult to predict how, on the average, a CD will get damaged; but in the absence of an accurate channel model, it is safe to assume that the channel mainly has a *burstlike* error behavior, since a scratch or fingerprint will cause *several* consecutive data samples to be in error. An important aspect of the system design contributing to the high-fidelity performance is a concatenated error-control scheme called the *cross-interleave Reed–Solomon* code (CIRC). The data are rearranged in time so that digits stemming from contiguous samples of the waveform are *spread out in time.* In this way, error bursts are made to appear as single random events (see the earlier sections on interleaving). The digital information is protected by adding parity bytes derived in two Reed–Solomon (R–S) encoders. Error control applied to the compact disc depends mostly on R–S coding and multiple layers of interleaving.

In digital audio applications, an undetected decoding error is very serious since it results in clicks, while occasional *detected* failures are not so serious because they can be concealed. The CIRC error-control scheme in the CD system involves both *correction* and *concealment* of errors. The performance specifications for the CIRC are given in Table 8.4. From the specifications in the table it would appear

TABLE 8.4 Specifications for the CD Cross-Interleave Reed–Solomon Code

Maximum correctable burst length	$\approx$ 4000 bits (2.5-mm track length on the disc)
Maximum interpolatable burst length	$\approx$ 12,000 bits (8 mm)
Sample interpolation rate	One sample every 10 hours at $P_B = 10^{-4}$ 1000 samples/min at $P_B = 10^{-3}$
Undetected error samples (clicks)	Less than one every 750 hours at $P_B = 10^{-3}$ Negligible at $P_B \leq 10^{-4}$
New discs are characterized by	$P_B \approx 10^{-4}$

that the CD can endure much damage (e.g., 8-mm holes punched in the disc) without any noticeable effect on the sound quality.

The CIRC system achieves its error control by a hierarchy of the following techniques:

1. The decoder provides a level of error correction.
2. If the error correction capability is exceeded, the decoder provides a level of erasure correction (see Section 6.5.5).
3. If the erasure correction capability is exceeded, the decoder attempts to conceal unreliable data samples by *interpolating* between reliable neighboring samples.
4. If the interpolation capability is exceeded, the decoder blanks out or *mutes* the system for the duration of the unreliable samples.

8.3.1 Circ Encoding

Figure 8.15 illustrates the basic CIRC encoder block diagram (within the CD recording equipment) and the decoder block diagram (within the CD player equipment). Encoding consists of the encoding and interleaving steps designated as Δ interleave, C_2 encode, D^* interleave, C_1 encode, and D interleave. The decoder steps, consisting of deinterleaving and decoding, are preformed in the *reverse* order of the encoding steps and are designated as D deinterleave, C_1 decode, D^* deinterleave, C_2 decode, and Δ deinterleave.

Figure 8.16 illustrates the basic system frame time, comprising six sampling periods, each made up of a stereo sample pair (16-bit left sample and 16-bit right sample). The bits are organized into symbols or bytes of 8 bits each. Therefore, each sample pair contains 4 bytes, and the uncoded frame contains k = 24 bytes. Figure 8.16a–e summarizes the *five encoding steps* that characterize the CIRC system. The function of each of these steps will best be understood when we consider the decoding operation. The steps are as follows:

(a) Δ *interleave.* Even-numbered samples are separated from odd-numbered samples by two frame times in order to scramble uncorrectable but detectable byte errors. This facilitates the interpolation process.

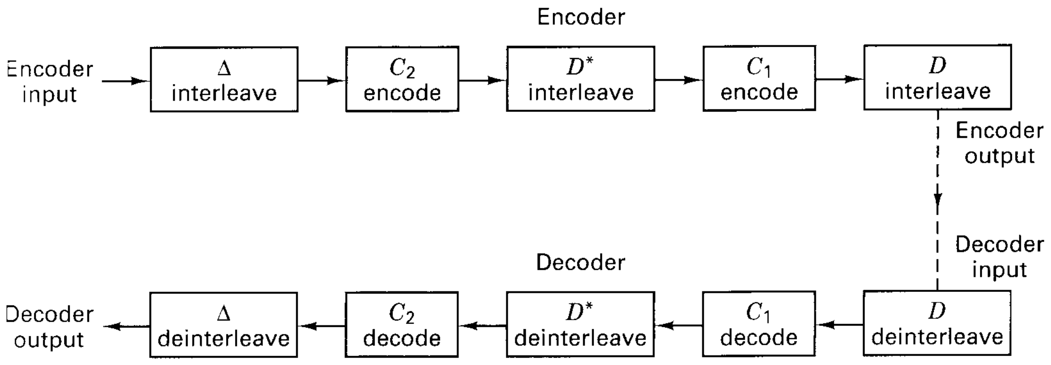


Figure 8.15 CIRC encoder and decoder.

(b) *C_2 encode.* Four Reed–Solomon (R–S) parity bytes are added to the Δ-interleaved 24-byte frame, resulting in a total of $n = 28$ bytes. This (28, 24) code is called the *outer code.*

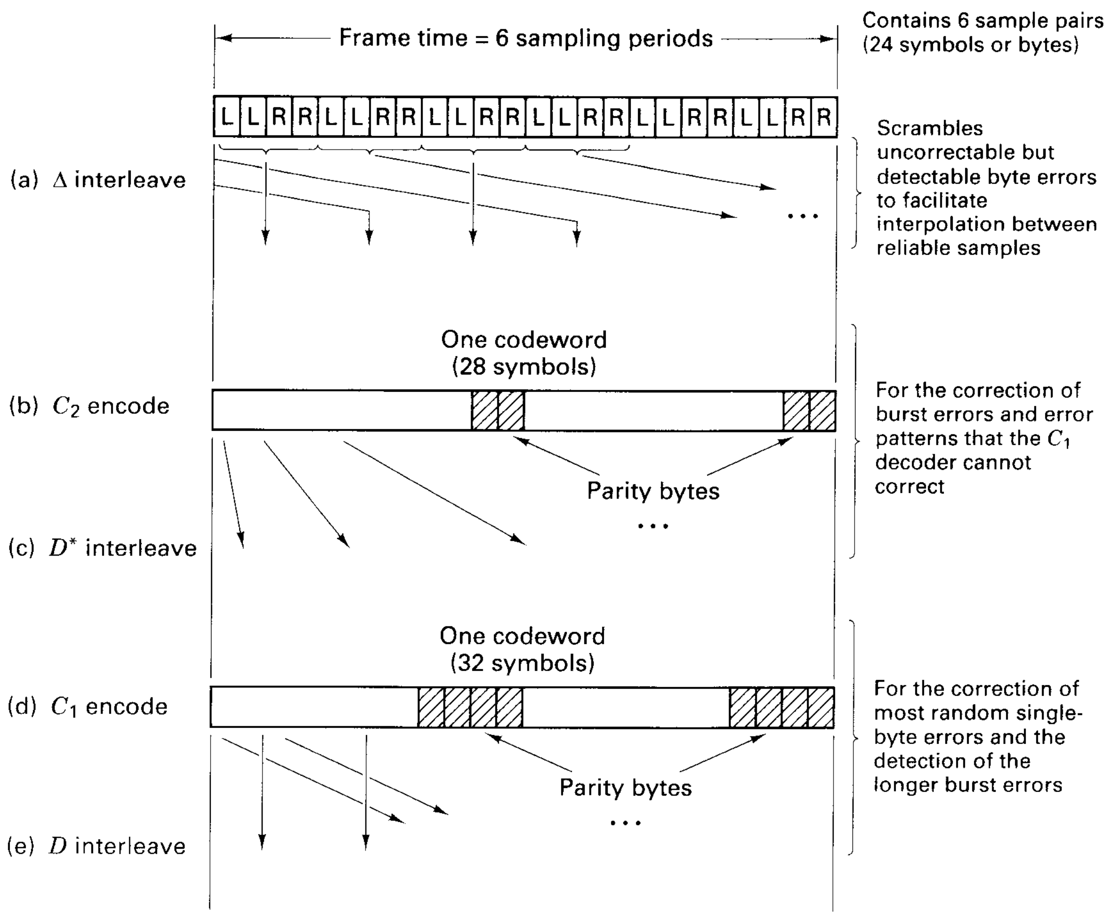
(c) *D^* interleave.* Here each byte is delayed a different length, thereby spreading errors over several codewords. C_2 encoding together with D^* interleaving have the function of providing for the correction of burst errors and error patterns that the C_1 decoder cannot correct.

(d) *C_1 encode.* Four R-S parity bytes are added to the $k = 28$ bytes of the D^*-interleaved frame, resulting in a total of $n = 32$ bytes. This (32, 28) code is called the *inner code.*

(e) *D interleave.* The purpose is to *cross-interleave* the *even bytes* of a frame with the *odd bytes* of the next frame. By this procedure, two consecutive bytes on the disc will always end up in two different codewords. Upon decoding, this interleaving, together with the C_1 decoding, results in the correction of most random single errors and the detection of longer burst errors.

8.3.1.1 Shortening the R-S Code

In Section 8.1 an (n, k) R–S code is expressed in terms of $n = 2^m - 1$ total symbols and $k = 2^m - 1 - 2t$ data symbols, where m is the number of bits per symbol and t is the error-correcting capability of the code in symbols. For the CD system, where a symbol is made up of 8 bits, a 2-symbol error-correcting code can be configured as a (255, 251) code. However, the CD system uses a considerably shorter block length. Any block code (in systematic form) can be shortened without affecting the number of errors that can be corrected within a block length. In terms of the (255, 251) R–S code, imagine that 227 of the 251 data symbols are a set of all-zero symbols (which are not actually transmitted and hence are not subject to any errors). Then the code is really a (28, 24) code with the same 2-symbol error-correcting capability. This is what is done in the C_2 encoder of the CD system.



Frame time = 6 sampling periods
Contains 6 sample pairs (24 symbols or bytes)
L L R R L L R R L L R R L L R R L L R R L L R R
(a) Δ interleave
Scrambles uncorrectable but detectable byte errors to facilitate interpolation between reliable samples
One codeword (28 symbols)
(b) C_2 encode
Parity bytes
For the correction of burst errors and error patterns that the C_1 decoder cannot correct
(c) D^* interleave
One codeword (32 symbols)
(d) C_1 encode
Parity bytes
For the correction of most random single-byte errors and the detection of the longer burst errors
(e) D interleave

Figure 8.16 Compact disc encoder. (a) Δ interleave. (b) C_2 encode. (c) D^* interleave. (d) C_1 encode. (e) D interleave.

We can think of the 28 total symbols out of the C_2 encoder as the data symbols into the C_1 encoder. Again, we can configure a shortened 2-symbol error-correcting (255, 251) code by throwing away 223 data symbols—the result being a (32, 28) code.

8.3.2 CIRC Decoding

The inner and outer R–S codes with (n, k) values (32, 28) and 28, 24) each use four parity bytes. The code rate of the CIRC is $(k_1/n_1)(k_2/n_2) = 24/32 = 3/4$. From Equation (8.3) the minimum distance of the C_1 and C_2 R-S codes is $d_{\min} = n - k + 1 = 5$. From Equations (8.4) and (8.5)

$$t \le \left\lfloor \frac{d_{\min} - 1}{2} \right\rfloor = \left\lfloor \frac{n-k}{2} \right\rfloor \tag{8.58}$$

and

$$\rho \le d_{\min} - 1 \tag{8.59}$$

where t is the error-correcting capability and ρ is the erasure-correcting capability, it is seen that the C_1 or C_2 decoder can correct a maximum of 2 symbol errors or 4 symbol erasures per codeword. Or, as described by Equation (8.6), it is possible to correct any pattern of α errors and γ erasures simultaneously, provided that

$$2\alpha + \gamma < d_{\min} < n - k \tag{8.60}$$

There is a trade-off between error correction and erasure correction; the larger the error correcting capability used, the smaller will be the erasure correcting capability.

The benefits of CIRC are best seen at the *decoder,* where the processing steps, shown in Figure 8.17 are in the reverse order of the encoder steps. The decoder steps are as follows:

1. *D deinterleave.* This function is performed by the alternating delay lines marked D. The 32 bytes $(B_{i1}, \ldots, B_{i32})$ of an encoded frame are applied in

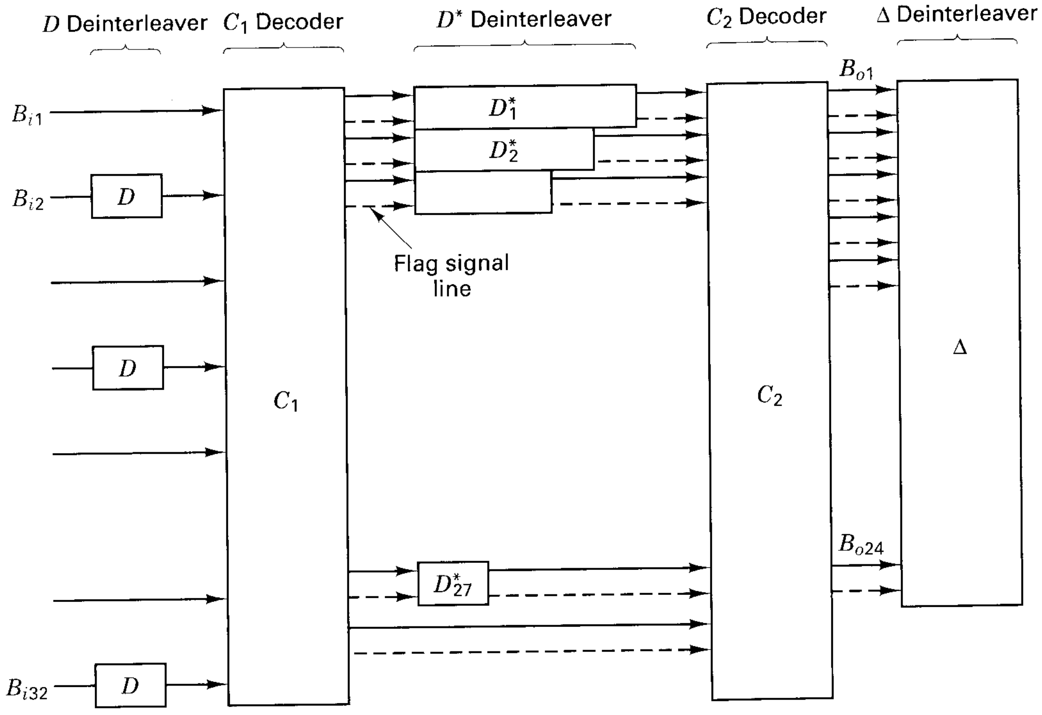


Figure 8.17 Compact disc decoder.

parallel to the 32 inputs of the D deinterleaver. Each delay is equal to the duration of 1 byte, so that the information of the *even bytes* of a frame is cross-deinterleaved with that of the *odd bytes* of the next frame.

2. C_1 *decode.* The D deinterleaver and the C_1 decoder are designed to correct a single byte error in the block of 32 bytes and to detect larger burst errors. If multiple errors occur, the C_1 decoder passes them on unchanged, attaching to all 28 remaining bytes an erasure flag, sent via the dashed lines (the four parity bytes used in the C_1 decoder are no longer retained).
3. D^* *deinterleave.* Due to the different lengths of the deinterleaving delay lines $D^*(1, \ldots, 27)$, errors that occur in one word at the output of the C_1 decoder are *spread over a number of words* at the input of the C_2 decoder. This results in reducing the number of errors per input word of the C_2 decoder, enabling the C_2 decoder to correct these errors.
4. C_2 *decode.* The C_2 decoder is intended for the correction of burst errors that the C_1 decoder could not correct. If the C_2 decoder cannot correct these errors, the 24-byte codeword is passed on unchanged to the Δ deinterleaver and the associated positions are given an *erasure flag* via the dashed output lines, $B_{o1}, \ldots, B_{o24}$.
5. Δ *deinterleave.* The final operation deinterleaves uncorrectable but detected byte errors in such a way that *interpolation* can be used between reliable neighboring samples.

Figure 8.18 highlights the decoder steps 2, 3, and 4. At the output of the C_1 decoder is seen a sequence of four 28-byte codewords that have exceeded the 1 byte per codeword error correction design. Therefore, each of the symbols in these codewords is tagged with an erasure flag (shown with circles). The D^* deinterleaver provides a staggered delay for each byte of a codeword, so that the bytes of a given codeword arrive in different codewords at the input to the C_2 decoder. If we assume that the delay increments of the D^* deinterleaver in Figure 8.18 are 1 byte, it would be possible to correct error bursts of as many as four consecutive C_1 codewords (since the C_2 decoder is capable of four erasure corrections per codeword). In the actual CD system, the delay increments are 4 bytes; therefore, the maximum burst error correction capability consists of 16 consecutive uncorrectable C_1 words.

8.3.3 Interpolation and Muting

Samples that cannot be corrected by the C_2 decoder could cause audible disturbances. The function of the *interpolation* process is to insert new samples, estimated from reliable neighbors, in place of the unreliable ones. If an entire C_2 word is detected as unreliable, this would make it impossible to apply interpolation without additional interleaving, since both even- and odd-numbered samples are unreliable. This can happen if the C_1 decoder fails to detect an error but the C_2 decoder detects it. It is the purpose of Δ deinterleaving (over a span of two frame times) to obtain a pattern where even-numbered samples can be interpolated from reliable odd-numbered samples, or vice versa.

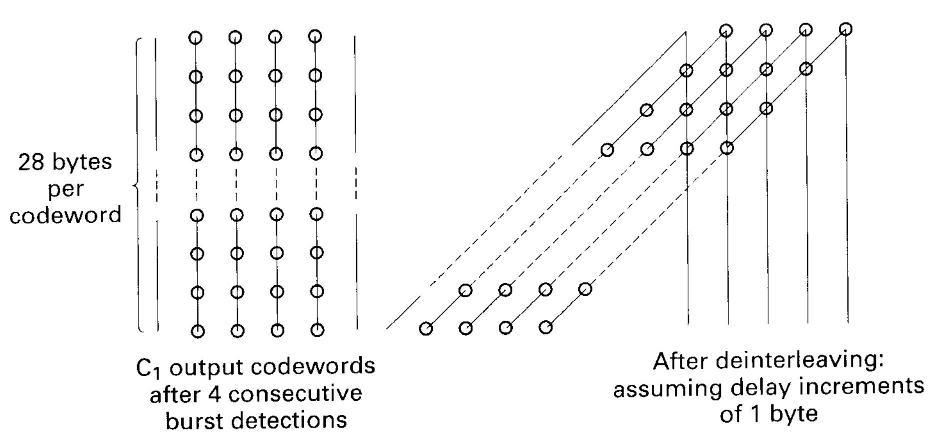

28 bytes per codeword
C1 output codewords after 4 consecutive burst detections
After deinterleaving: assuming delay increments of 1 byte

Figure 8.18 Example of 4-byte erasure capability. (Rightmost event is at the earliest time.)

Two successive unreliable words consisting of 12 sample pairs are shown in Figure 8.19. A sample pair consists of a sample (2 bytes) from the right audio channel and a sample from the left audio channel. The numbers indicate the ordering of the sets of samples. An encircled sample set denotes an *erasure* flag. After Δ deinterleaving, the unreliable samples shown in the figure are estimated by a first-order linear interpolation between neighboring samples that stem from a different location on the disc.

In CD players, another level of error control is provided in case a burst length of 48 frames is exceeded and 2 or more consecutive unreliable samples result. In this case the system is *muted* (audio is softly blanked out), which is not discernible to the human ear if the muting time does not exceed a few milliseconds. For a more detailed treatment of the CIRC coding scheme in the CD system, see References [11–15].

8.4 TURBO CODES

Concatenated coding schemes were first proposed by Forney [16} as a method for achieving large coding gains by combining two or more relatively simple building-block or *component codes* (sometimes called *constituent codes*). The resulting codes had the error-correction capability of much longer codes, and they were endowed with a structure that permitted relatively easy to moderately complex decoding. A serial concatenation of codes is most often used for power-limited systems such as transmitters on deep-space probes. The most popular of these schemes consists of a Reed–Solomon outer (applied first, removed last) code followed by a convolutional inner (applied last, removed first) code [10]. A turbo code can be thought of as a refinement of the concatenated encoding structure plus an iterative algorithm for decoding the associated code sequence. Because of

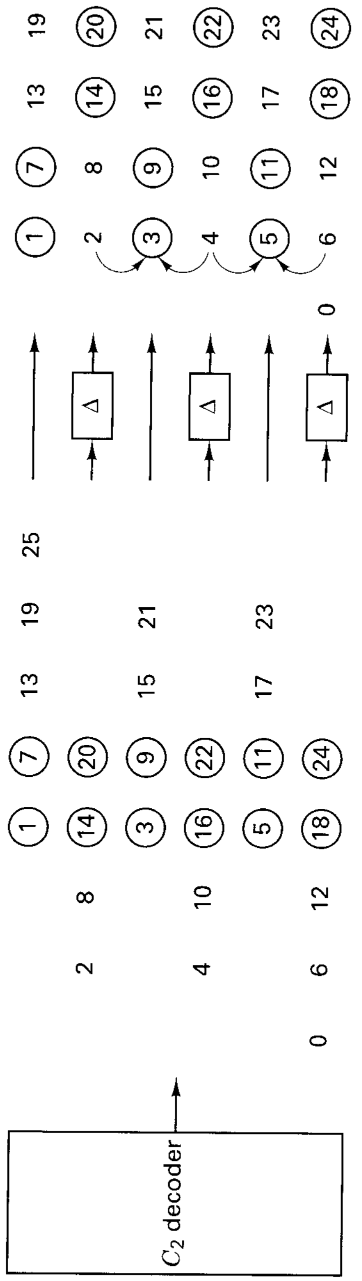


Figure 8.19 Effect of interleaving. (Rightmost event is at the earliest time.)

its unique iterative form, we choose to list *turbo* as a separate category under structured sequences in Figure 1.3.

Turbo codes were first introduced in 1993 by Berrou, Glavieux, and Thitimajshima, and reported in [17, 18], where a scheme is described that achieves a bit-error-probability of 10^{-5}, using a rate 1/2 code over an additive white Gaussian noise (AWGN) channel and BPSK modulation at an E_b/N_0 of 0.7 dB. The codes are constructed by using two or more component codes on different interleaved versions of the same information sequence. Whereas, for conventional codes, the final step at the decoder yields hard-decision decoded bits (or more generally, decoded symbols), for a concatenated scheme, such as a turbo code, to work properly, the decoding algorithm should not limit itself to passing hard-decisions among the decoders. To best exploit the information learned from each decoder, the decoding algorithm must effect an exchange of soft decisions rather than hard decisions. For a system with two component codes, the concept behind turbo decoding is to pass soft decisions from the output of one decoder to the input of the other decoder, and to iterate this process several times so as to produce more reliable decisions.

8.4.1 Turbo Code Concepts

8.4.1.1 Likelihood Functions

The mathematical foundations of hypothesis testing rests on Bayes' theorem, which is developed in Appendix B. For communications engineering, where applications involving an AWGN channel are of great interest, the most useful form of Bayes' theorem expresses the a posteriori probability (APP) of a decision in terms of a continuous-valued random variable x as

$$P(d = i|x) = \frac{p(x|d = i)\, P(d = i)}{p(x)} \qquad i = 1, \cdots, M \tag{8.61}$$

and

$$p(x) = \sum_{i=1}^{M} p(x|d = i)\, P(d = i) \tag{8.62}$$

where $P\ (d = i|x)$ is the APP, and $d = i$ represents data d belonging to the ith signal class from a set of M classes. Further, $p(x|d = i)$ represents the probability density function (pdf) of a received continuous-valued data-plus-noise signal x, conditioned on the signal class $d = i$. Also, $p(d = i)$, called the a priori probability, is the probability of occurrence of the ith signal class. Typically x is an "observable" random variable or a test statistic that is obtained at the output of a demodulator or some other signal processor. Therefore, $p(x)$ is the pdf of the received signal x, yielding the test statistic over the entire space of signal classes. In Equation (8.61), for a particular observation, $p(x)$ is a scaling factor since it is obtained by averaging over all the classes in the space. Lower case p is used to designate the pdf of a continuous-valued random variable, and upper case P is used to designate probability (a priori and APP). Determining the APP of a received signal from Equation (8.61) can be thought of as the result of an experiment. Before the experiment, there generally exists (or one can estimate) an a priori probability $P(d = i)$. The experiment consists of using Equation (8.61) for computing the APP, $P(d = i|x)$, which can be thought of as a "refinement" of the prior knowledge about the data, brought about by examining the received signal x.

8.4.1.2 The Two-Signal Class Case

Let the binary logical elements 1 and 0 be represented electronically by voltages +1 and −1, respectively. The variable d is used to represent the transmitted data bit, whether it appears as a voltage or as a logical element. Sometimes one format is more convenient than the other; the reader should be able to recognize the difference from the context. Let the binary 0 (or the voltage value −1) be the null element under addition. For signal transmission over an AWGN channel, Figure 8.20 shows the conditional pdfs, referred to as likelihood functions. The rightmost function $p(x|d = +1)$ shows the pdf of the random variable x conditioned on $d = +1$ being transmitted. The leftmost function $p(x|d = -1)$ illustrates a similar pdf conditioned on $d = -1$ being transmitted. The abscissa represents the full range of possible values of the test statistic x generated at the receiver. In Figure 8.20, one such arbitrary value x_k is shown, where the index denotes an observation in the kth time interval. A line subtended from x_k intercepts the two likelihood functions yielding two likelihood values $\ell_1 = p(x_k|d_k = +1)$ and $\ell_2 = p\ (x_k|d_k = -1)$. A well-known hard-decision rule, known as *maximum likelihood*, is to choose the data $d_k = +1$ or $d_k = -1$ associated with the larger of the two intercept values ℓ_1 or ℓ_2, respectively. For each data bit at time k, this is tantamount to deciding that $d_k = +1$ if x_k falls on the right side of the decision line labeled γ_0, otherwise deciding that $d_k = -1$.

A similar decision rule, known as *maximum a posteriori* (MAP), which can be shown to be a *minimum-probability-of-error rule,* takes into account the a priori probabilities of the data. The general expression for the MAP rule in terms of APPs is

$$P(d = +1|x) \underset{H_2}{\overset{H_1}{\gtrless}} P(d = -1|x) \qquad (8.63)$$

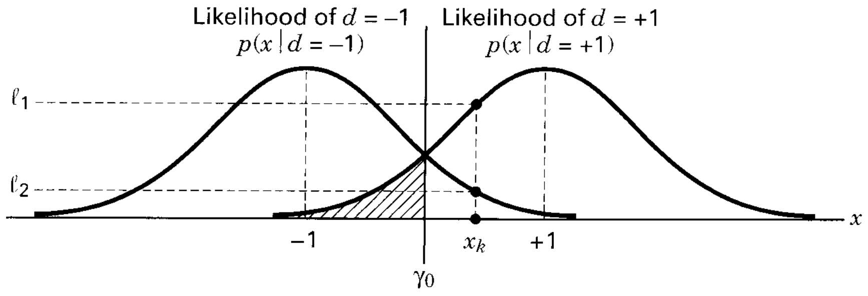


Figure 8.20 Likelihood functions.

Equation (8.63) states that one should choose the hypothesis H_1, $(d = +1)$ if the APP, $P(d = +1|x)$, is greater than the APP, $P(d = -1|x)$. Otherwise, one should choose hypothesis H_2, $(d = -1)$. Using the Bayes' theorem of Equation (8.61), the APPs in Equation (8.63) can be replaced by their equivalent expressions, yielding

$$p(x|d = +1)\,P(d = +1) \overset{H_1}{\underset{H_2}{\gtrless}} p(x|d = -1)\,P(d = -1) \tag{8.64}$$

where the pdf $p(x)$ appearing on both sides of the inequality in Equation (8.61) has been canceled. Equation (8.64) is generally expressed in terms of a ratio, yielding the so-called *likelihood ratio test,* as follows:

$$\frac{p(x|d = +1)}{p(x|d = -1)} \overset{H_1}{\underset{H_2}{\gtrless}} \frac{P(d = -1)}{P(d = +1)} \quad \text{or} \quad \frac{p(x|d = +1)\,P(d = +1)}{p(x|d = -1)\,P(d = -1)} \overset{H_1}{\underset{H_2}{\gtrless}} 1 \tag{8.65}$$

8.4.1.3 Log-Likelihood Ratio

By taking the logarithm of the likelihood ratio developed in Equations (8.63) through (8.65), we obtain a useful metric called the log-likelihood ratio (LLR). It is a real number representing a soft decision out of a detector, designated by

$$L(d|x) = \log\left[\frac{P(d = +1|x)}{P(d = -1|x)}\right] = \log\left[\frac{p(x|d = +1)\,P(d = +1)}{p(x|d = -1)\,P(d = -1)}\right] \tag{8.66}$$

so that

$$L(d|x) = \log\left[\frac{p(x|d = +1)}{p(x|d = -1)}\right] + \log\left[\frac{P(d = +1)}{P(d = -1)}\right] \tag{8.67}$$

or

$$L(d|x) = L(x|d) + L(d) \tag{8.68}$$

where $L(x|d)$ is the LLR of the test statistic x obtained by measurements of the channel output x under the alternate conditions that $d = +1$ or $d = -1$ may have been transmitted, and $L(d)$ is the a priori LLR of the data bit d. To simplify the notation, Equation (8.68) is rewritten as

$$L'(\hat{d}) = L_c(x) + L(d) \tag{8.69}$$

where the notation $L_c(x)$ emphasizes that this LLR term is the result of a channel measurement made at the receiver. Equations (8.61) through (8.69) were developed with only a data detector in mind. Next, the introduction of a decoder will typically yield decision-making benefits. For a systematic code, it can be shown [17] that the LLR (soft output) out of the decoder is equal to

$$L(\hat{d}) = L'(\hat{d}) + L_e(\hat{d}) \tag{8.70}$$

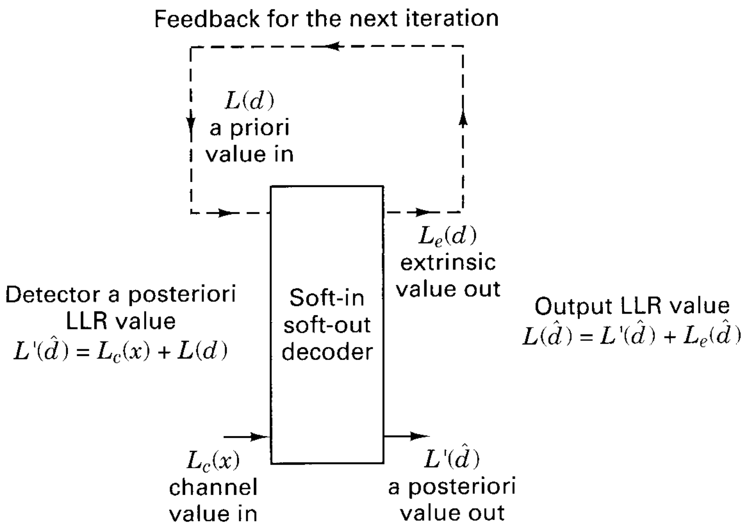
where $L'(\hat{d})$ is the LLR of a data bit out of the demodulator (input to the decoder), and $L_e(\hat{d})$, called the *extrinsic* LLR, represents extra knowledge that is gleaned from the decoding process. The output sequence of a systematic decoder is made up of values representing data bits and parity bits. From Equations (8.69) and (8.70), the output LLR of the decoder is now written as

$$L(\hat{d}) = L_c(x) + L(d) + L_e(\hat{d}) \tag{8.71}$$

Equation (8.71) shows that the output LLR of a systematic decoder can be represented as having three LLR elements—a channel measurement, a priori knowledge of the data, and an extrinsic LLR stemming solely from the decoder. To yield the final $L(\hat{d})$, each of the individual LLRs can be added as shown in Equation (8.71), because the three terms are statistically independent [17, 19]. The proof is left as an exercise for the reader. (See Problem 8.18.) This soft decoder output $L(\hat{d})$ is a real number that provides a hard decision as well as the reliability of that decision. The sign of $L(\hat{d})$ denotes the hard decision—that is, for positive values of $L(\hat{d})$ decide that $d = +1$, and for negative values that $d = -1$. The magnitude of $L(\hat{d})$ denotes the reliability of that decision. Often the value of $L_e(\hat{d})$ due to the decoding has the same sign as $L_c(x) + L(d)$ and therefore acts to improve the reliability of $L(\hat{d})$.

8.4.1.4 Principles of Iterative (Turbo) Decoding

In a typical communications receiver, a demodulator is often designed to produce soft decisions which are then transferred to a decoder. In Chapter 7, the error-performance improvement of systems utilizing such soft decisions compared with hard decisions were quantified as being approximately 2 dB in AWGN. Such a decoder could be called a soft-input/hard-output decoder, because the final decoding process out of the decoder must terminate in bits (hard decisions). With turbo codes, where two or more component codes are used, and decoding involves feeding outputs from one decoder to the inputs of other decoders in an iterative fashion, a hard-output decoder would not be suitable. That is because hard decisions into a decoder degrades system performance (compared with soft decisions). Hence, what is needed for the decoding of turbo codes is a *soft-input/soft-output* decoder. For the first decoding iteration of such a soft-input/soft-output decoder, illustrated in Figure 8.21, one generally assumes the binary data to be equally likely, yielding an initial a priori LLR value of $L(d) = 0$ for the third term in Equation (8.67). The channel LLR value $L_c(x)$ is measured by forming the logarithm of the ratio of the values of ℓ_1 and ℓ_2 for a particular observation of x (see Figure 8.20), which appears as the second term in Equation (8.67). The output $L(\hat{d})$ of the decoder in Figure 8.21 is made up of the LLR from the detector $L'(\hat{d})$ and the extrinsic LLR output $L_e(\hat{d})$, representing knowledge gleaned from the decoding process. As illustrated in Figure 8.21, for iterative decoding, the extrinsic likelihood is fed back to the input (of another component decoder) to serve as a refinement of the a-priori probability of the data for the next iteration.



Feedback for the next iteration
$L(d)$ a priori value in
$L_e(d)$ extrinsic value out
Detector a posteriori LLR value $L'(\hat{d}) = L_c(x) + L(d)$
Soft-in soft-out decoder
Output LLR value $L(\hat{d}) = L'(\hat{d}) + L_e(\hat{d})$
$L_c(x)$ channel value in
$L'(\hat{d})$ a posteriori value out

Figure 8.21 Soft input/soft output decoder (for a systematic code).

8.4.2 Log-Likelihood Algebra

To best explain the iterative feedback of soft decoder outputs, the concept of a log-likelihood algebra [19] is introduced. For statistically independent data d, the sum of two log likelihood ratios (LLRs) is defined as

$$L(d_1) \boxplus L(d_2) \triangleq L(d_1 \oplus d_2) = \log_e \left[\frac{e^{L(d_1)} + e^{L(d_2)}}{1 + e^{L(d_1)} e^{L(d_2)}} \right] \tag{8.72}$$

$$\approx (-1) \times \text{sgn}\left[L(d_1)\right] \times \text{sgn}\left[L(d_2)\right] \times \min\left(|L(d_1)|, |L(d_2)|\right) \tag{8.73}$$

where the natural logarithm is used, and the function sgn $(\cdot)$ represents the "polarity of." There are three addition operations in Equation (8.72). The + sign is used for ordinary addition. The $\oplus$ sign is used to denote the modulo-2 sum of data expressed as binary digits. The $\boxplus$ sign denotes log-likelihood addition, or equivalently, the mathematical operation described by Equation (8.72). The sum of two LLRs denoted by the operator $\boxplus$ is defined as the LLR of the modulo-2 sum of the underlying statistically independent data bits. The development of Equation (8.72) is shown in Appendix 8A. Equation (8.73) is an approximation of Equation (8.72) that will prove useful later in a numerical example. The sum of LLRs, as described by Equations (8.72) or (8.73), yields the following interesting results when one of the LLRs is very large or very small:

$$L(d) \boxplus \infty = -L(d)$$

and

$$L(d) \boxplus 0 = 0$$

Note that the log-likelihood algebra described here differs slightly from that used in [19] because of a different choice of the null element. In this treatment, the null element of the binary set (1, 0) has been chosen to be 0.

8.4.3 Product Code Example

Consider the 2-dimensional code (product code) depicted in Figure 8.22. The configuration can be described as a data array made up of k_1 rows and k_2 columns. The k_1 rows contain codewords made up of k_2 data bits and $n_2 - k_2$ parity bits. Thus each of the k_1 rows represents a codeword from an (n_2, k_2) code. Similarly, the k_2 columns contain codewords made up of k_1 data bits and $n_1 - k_1$ parity bits. Thus, each of the k_2 columns represents a codeword from an (n_1, k_1) code. The various portions of the structure are labeled d for data, p_h for horizontal parity (along the rows), and p_v for vertical parity (along the columns). In effect, the block of $k_1 \times k_2$ data bits is encoded with two codes—a horizontal code, and a vertical code.

Additionally, in Figure 8.22, there are blocks labeled L_{eh} and L_{ev} containing the extrinsic LLR values learned from the horizontal and vertical decoding steps, respectively. Error-correction codes generally provide some improved performance. We will see that the extrinsic LLRs represent a measure of that improvement. Notice that this product code is a simple example of a concatenated code. Its structure encompasses two separate encoding steps—horizontal and vertical.

Recall that the final decoding decision for each bit and its reliability hinges on the value of $L(\hat{d})$, as shown in Equation (8.71). With this equation in mind, an algorithm yielding the extrinsic LLRs (horizontal and vertical) and a final $L(\hat{d})$ can be described. For the product code, this iterative decoding algorithm proceeds as follows:

1. Set the a-priori LLR $L(d) = 0$ (unless the a priori probabilities of the data bits are other than equally likely).

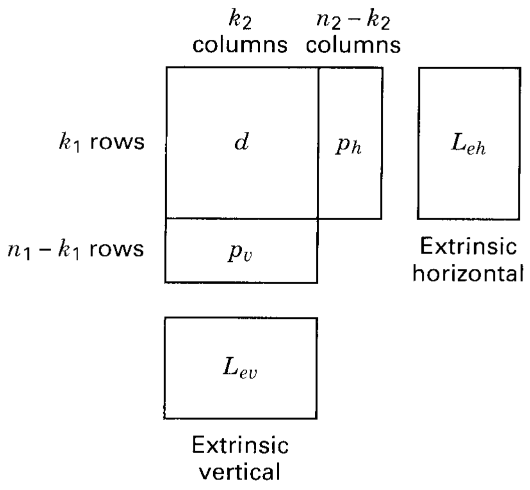


Figure 8.22 Two-dimensional product code.

2. Decode horizontally, and, using Equation (8.71), obtain the horizontal extrinsic LLR

$$L_{eh}(\hat{d}) = L(\hat{d}) - L_c(x) - L(d)$$

3. Set $L(d) = L_{eh}(\hat{d})$ for the vertical decoding of step 4.
4. Decode vertically, and, using Equation (8.71), obtain the vertical extrinsic LLR

$$L_{ev}(\hat{d}) = L(\hat{d}) - L_c(x) - L(d)$$

5. Set $L(d) = L_{ev}(\hat{d})$ for the step 2 horizontal decoding. Then repeat steps 2 through 5.
6. After enough iterations (i.e., repetitions of steps 2 through 5) to yield a reliable decision, go to step 7.
7. The soft output is

$$L(\hat{d}) = L_c(x) + L_{eh}(\hat{d}) + L_{ev}(\hat{d}) \tag{8.74}$$

An example is next used to demonstrate the application of this algorithm to a very simple product code.

8.4.3.1 Two-Dimensional Single-Parity Code Example

At the encoder, let the data bits and parity bits take on the values shown in Figure 8.23a, where the relationships between data and parity bits within a particular row (or column) expressed as the binary digits (1, 0) are

$$d_i \oplus d_j = p_{ij} \tag{8.75}$$

and

$$d_i = d_j \oplus p_{ij} \qquad i, j \in \{(1, 2),\ (3, 4),\ (1, 3),\ (2, 4)\} \tag{8.76}$$

in which $\oplus$ denotes modulo-2 addition. The transmitted bits are represented by the sequence d_1, d_2, d_3, d_4, p_{12}, p_{34}, p_{13}, p_{24}. At the receiver input, the noise-corrupted bits are represented by the sequence $\{x_i\}$, $\{x_{ij}\}$, where $x_i = d_i + n$ for each received data bit, $x_{ij} = p_{ij} + n$ for each received parity bit, and n represents the noise contribution that is statistically independent for both d_i and p_{ij}. The indices i and j represent position in the encoder output array shown in Figure 8.23a. However, it is often more useful to denote the received sequence as $\{x_k\}$, where k is a time index. Both conventions will be followed below—using i and j when focusing on the positional relationships within the product code, and using k when focusing on the more general aspect of a time-related signal. The distinction as to which convention is being used should be clear from the context. Using the relationships developed in Equations (8.67) through (8.69), and assuming an AWGN interference model, the LLR for the channel measurement of a signal x_k received at time k, is written

$$L_c(x_k) = \log_e \left[\frac{p(x_k \mid d_k = +1)}{p(x_k \mid d_k = -1)} \right] \tag{8.77a}$$

$d_1 = 1$	$d_2 = 0$	$p_{12} = 1$
$d_3 = 0$	$d_4 = 1$	$p_{34} = 1$
$p_{13} = 1$	$p_{24} = 1$	

(a) Encoder output binary digits

$L_c(x_1) = 1.5$	$L_c(x_2) = 0.1$	$L_c(x_{12}) = 2.5$
$L_c(x_3) = 0.2$	$L_c(x_4) = 0.3$	$L_c(x_{34}) = 2.0$
$L_c(x_{13}) = 6.0$	$L_c(x_{24}) = 1.0$	

(b) Decoder input log-likelihood ratios $L_c(x)$ **Figure 8.23** Product code example.

$$= \log_e \left(\frac{\frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{1}{2}\left(\frac{x_k - 1}{\sigma}\right)^2 \right]}{\frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{1}{2}\left(\frac{x_k + 1}{\sigma}\right)^2 \right]} \right) \tag{8.77b}$$

$$= -\frac{1}{2}\left(\frac{x_k - 1}{\sigma}\right)^2 + \frac{1}{2}\left(\frac{x_k + 1}{\sigma}\right)^2 = \frac{2}{\sigma^2} x_k \tag{8.77c}$$

where the natural logarithm is used. If a further simplifying assumption is made that the noise variance σ^2 is unity, then

$$L_c(x_k) = 2x_k \tag{8.78}$$

Consider the following example, where the data sequence d_1, d_2, d_3, d_4 is made up of the binary digits 1 0 0 1, as shown in Figure 8.23a. By the use of Equation (8.75), it is seen that the parity sequence p_{12}, p_{34}, p_{13}, p_{24} must be equal to the digits 1 1 1 1. Thus, the transmitted sequence is

$$\{d_i\}, \{p_{ij}\} = 1\ 0\ 0\ 1\ 1\ 1\ 1\ 1 \tag{8.79}$$

When the data bits are expressed as bipolar voltage values of +1 and −1 corresponding to the binary logic levels 1 and 0, the transmitted sequence is

$$\{d_i\}, \{p_{ij}\} = +1\ \ -1\ \ -1\ \ +1\ \ +1\ \ +1\ \ +1\ \ +1$$

Assume now that the noise transforms this data-plus-parity sequence into the received sequence

$$\{x_i\}, \{x_{ij}\} = 0.75,\ 0.05,\ 0.10,\ 0.15,\ 1.25,\ 1.0,\ 3.0,\ 0.5 \tag{8.80}$$

where the members of $\{x_i\}$, $\{x_{ij}\}$ positionally correspond to the data and parity $\{d_i\}$, $\{p_{ij}\}$ that was transmitted. Thus, in terms of the positional subscripts, the received sequence can be denoted as

$$\{x_i\}, \{x_{ij}\} = x_1, x_2, x_3, x_4, x_{12}, x_{34}, x_{13}, x_{24}$$

From Equation (8.78), the assumed channel measurements yield the LLR values

$$\{L_c(x_i)\}, \{L_c(x_{ij})\} = 1.5,\ 0.1,\ 0.20,\ 0.3,\ 2.5,\ 2.0,\ 6.0,\ 1.0 \tag{8.81}$$

These values are shown in Figure 8.23b as the decoder input measurements. It should be noted that, given equal prior probabilities for the transmitted data, if hard decisions are made based on the $\{x_k\}$ or the $\{L_c(x_k)\}$ values shown above, such a process would result in two errors, since d_2 and d_3 would each be incorrectly classified as binary 1.

8.4.3.2 Extrinsic Likelihoods

For the product-code example in Figure 8.23, we use Equation (8.71) to express the soft output for the received signal corresponding to data d_1 as

$$L(\hat{d}_1) = L_c(x_1) + L(d_1) + \{[L_c(x_2) + L(d_2)] \boxplus L_c(x_{12})\} \tag{8.82}$$

where the terms $\{[L_c(x_2) + L(d_2)] \boxplus L_c(x_{12})\}$ represent the extrinsic LLR contributed by the code (i.e., the reception corresponding to data d_2 and its a priori probability, in conjunction with the reception corresponding to parity p_{12}). In general the soft output $L(\hat{d}_i)$ for the received signal corresponding to data d_i is

$$L(\hat{d}_i) = L_c(x_i) + L(d_i) + \{[L_c(x_j) + L(d_j)] \boxplus L_c(x_{ij})\} \tag{8.83}$$

where $L_c(x_i)$, $L_c(x_j)$, and $L_c(x_{ij})$ are the channel LLR measurements of the reception corresponding to d_i, d_j, and p_{ij}, respectively. $L(d_i)$ and $L(d_j)$ are the LLRs of the a priori probabilities of d_i and d_j respectively, and $\{[L_c(x_j) + L(d_j)] \boxplus L_c(x_{ij})\}$ is the extrinsic LLR contribution from the code. Equations (8.82) and (8.83) can best be understood in the context of Figure 8.23b. For this example, assuming equally likely signaling, the soft output $L(\hat{d}_1)$ is represented by the detector LLR measurement of $L_c(x_1) = 1.5$ for the reception corresponding to data d_1, plus the extrinsic LLR of $[L_c(x_2) = 0.1] \boxplus [L_c(x_{12}) = 2.5]$ gleaned from the fact that the data d_2 and the parity p_{12} also provide knowledge about the data d_1 as seen from Equations (8.75) and (8.76).

8.4.3.3 Computing the Extrinsic Likelihoods

For the example in Figure 8.23, the horizontal calculations for $L_{eh}(\hat{d})$ and the vertical calculations for $L_{ev}(\hat{d})$ are expressed as follows:

$$L_{eh}(\hat{d}_1) = [L_c(x_2) + L(d_2)] \boxplus L_c(x_{12}) \tag{8.84a}$$

$$L_{ev}(\hat{d}_1) = [L_c(x_3) + L(d_3)] \boxplus L_c(x_{13}) \tag{8.84b}$$

$$L_{eh}(\hat{d}_2) = [L_c(x_1) + L(d_1)] \boxplus L_c(x_{12}) \tag{8.85a}$$

$$L_{ev}(\hat{d}_2) = [L_c(x_4) + L(d_4)] \boxplus L_c(x_{24}) \tag{8.85b}$$

$$L_{eh}(\hat{d}_3) = [L_c(x_4) + L(d_4)] \boxplus L_c(x_{34}) \tag{8.86a}$$

$$L_{ev}(\hat{d}_3) = [L_c(x_1) + L(d_1)] \boxplus L_c(x_{13}) \tag{8.86b}$$

$$L_{eh}(\hat{d}_4) = [L_c(x_3) + L(d_3)] \boxplus L_c(x_{34}) \tag{8.87a}$$

$$L_{ev}(\hat{d}_4) = [L_c(x_2) + L(d_2)] \boxplus L_c(x_{24}) \tag{8.87b}$$

The LLR values shown in Figure 8.23 are entered into the $L_{eh}(\hat{d})$ expressions in Equations (8.84) through (8.87), and, assuming equally likely signaling, the $L(d)$ values are initially set equal to zero, yielding

$$L_{eh}(\hat{d}_1) = (0.1 + 0) \boxplus 2.5 \approx -0.1 = \text{new } L(d_1) \tag{8.88}$$

$$L_{eh}(\hat{d}_2) = (1.5 + 0) \boxplus 2.5 \approx -1.5 = \text{new } L(d_2) \tag{8.89}$$

$$L_{eh}(\hat{d}_3) = (0.3 + 0) \boxplus 2.0 \approx -0.3 = \text{new } L(d_3) \tag{8.90}$$

$$L_{eh}(\hat{d}_4) = (0.2 + 0) \boxplus 2.0 \approx -0.2 = \text{new } L(d_4) \tag{8.91}$$

where the log-likelihood addition has been calculated using the approximation in Equation (8.73). Next, we proceed to obtain the first vertical calculations, using the $L_{ev}(\hat{d})$ expressions in Equations (8.84) through (8.87). Now, the values of $L(d)$ can be refined by using the new $L(d)$ values gleaned from the first horizontal calculations, shown in Equations (8.88) through (8.91). That is,

$$L_{ev}(\hat{d}_1) = (0.2 - 0.3) \boxplus 6.0 \approx 0.1 = \text{new } L(d_1) \tag{8.92}$$

$$L_{ev}(\hat{d}_2) = (0.3 - 0.2) \boxplus 1.0 \approx -0.1 = \text{new } L(d_2) \tag{8.93}$$

$$L_{ev}(\hat{d}_3) = (1.5 - 0.1) \boxplus 6.0 \approx -1.4 = \text{new } L(d_3) \tag{8.94}$$

$$L_{ev}(\hat{d}_4) = (0.1 - 1.5) \boxplus 1.0 \approx 1.0 = \text{new } L(d_4) \tag{8.95}$$

The results of the first full iteration of the two decoding steps (horizontal and vertical) are as follows:

Original $L_c(x_k)$ measurements

1.5	0.1
0.2	0.3

–0.1	–1.5
–0.3	–0.2

$L_{eh}(\hat{d})$ after first horizontal decoding

0.1	–0.1
–1.4	1.0

$L_{ev}(\hat{d})$ after first vertical decoding

Each decoding step improves the original LLRs that are based on channel measurements only. This is seen by calculating the decoder output LLR, using Equation (8.74). The original LLR plus the horizontal extrinsic LLRs yield the following improvement (the extrinsic vertical terms are not yet being considered):

Improved LLRs due to $L_{eh}(\hat{d})$

1.4	−1.4
−0.1	0.1

The original LLR plus both the horizontal and vertical extrinsic LLRs yield the following improvement:

Improved LLRs due to $L_{eh}(\hat{d}) + L_{ev}(\hat{d})$

1.5	−1.5
−1.5	1.1

For this example, it is seen that the knowledge gained from horizontal decoding alone is sufficient to yield the correct hard decisions out of the decoder, but with very low confidence for data bits d_3 and d_4. After incorporating the vertical extrinsic LLRs into the decoder, the new LLR values exhibit a higher level of reliability or confidence. Let us pursue one additional horizontal and vertical decoding iteration to determine if there are any significant changes in the results. We again use the relationships shown in Equations (8.84) through (8.87) and proceed with the second horizontal calculations for $L_{eh}(\hat{d})$, using the new $L(d)$ from the first vertical calculations, shown in Equations (8.92) through (8.95), so that

$$L_{eh}(\hat{d}_1) = (0.1 - 0.1) \boxplus 2.5 \approx 0 = \text{new } L(d_1) \tag{8.96}$$

$$L_{eh}(\hat{d}_2) = (1.5 + 0.1) \boxplus 2.5 \approx -1.6 = \text{new } L(d_2) \tag{8.97}$$

$$L_{eh}(\hat{d}_3) = (0.3 + 1.0) \boxplus 2.0 \approx -1.3 = \text{new } L(d_3) \tag{8.98}$$

$$L_{eh}(\hat{d}_4) = (0.2 - 1.4) \boxplus 2.0 \approx 1.2 = \text{new } L(d_4) \tag{8.99}$$

Next, we proceed with the second vertical calculations for $L_{ev}(\hat{d})$, using the new $L(d)$ from the second horizontal calculations, shown in Equations (8.96) through (8.99). This yields

$$L_{ev}(\hat{d}_1) = (0.2 - 1.3) \boxplus 6.0 \approx 1.1 = \text{new } L(d_1) \tag{8.100}$$

$$L_{ev}(\hat{d}_2) = (0.3 + 1.2) \boxplus 1.0 \approx -1.0 = \text{new } L(d_2) \tag{8.101}$$

$$L_{ev}(\hat{d}_3) = (1.5 + 0) \boxplus 6.0 \approx -1.5 = \text{new } L(d_3) \tag{8.102}$$

$$L_{ev}(\hat{d}_4) = (0.1 - 1.6) \boxplus 1.0 \approx 1.0 = \text{new } L(d_4) \tag{8.103}$$

The second iteration of horizontal and vertical decoding, yielding the preceding values, results in soft-output LLRs that are again calculated from Equation (8.74), which is rewritten below:

$$L(\hat{d}) = L_c(x) + L_{eh}(\hat{d}) + L_{ev}(\hat{d}) \tag{8.104}$$

The horizontal and vertical extrinsic LLRs of Equations (8.96) through (8.103) and the resulting decoder LLRs are displayed below. For this example, the second horizontal and vertical iteration (yielding a total of four iterations) suggests a modest improvement over a single horizontal and vertical iteration. The results show a balancing of the confidence values amongst each of the four data decisions:

Original $L_c(x)$ Measurements

1.5	0.1
0.2	0.3

0	−1.6
−1.3	1.2

$L_{ev}(\hat{d})$ after second vertical decoding

1.1	−1.0
−1.5	1.0

$L_{eh}(\hat{d})$ after second horizontal decoding

The soft output is $L(\hat{d}) = L_c(x) + L_{eh}(\hat{d}) + L_{ev}(\hat{d})$, which, after a total of four iterations, yields the values for $L(\hat{d})$ of

2.6	−2.5
−2.6	2.5

Observe that correct decisions about the four data bits will result, and the level of confidence about these decisions is high. The iterative decoding of turbo codes is similar to the process used when solving a crossword puzzle. The first pass through the puzzle is likely to contain a few errors. Some words seem to fit, but when the letters intersecting a row and column do not match, it is necessary to go back and correct the first-pass answers.

8.4.4 Encoding with Recursive Systematic Codes

The basic concepts of concatenation, iteration, and soft decision decoding using a simple product-code example have been described. These ideas are next applied to the implementation of turbo codes that are formed by the parallel concatenation of component convolutional codes [17, 20].

A short review of simple binary rate 1/2 convolutional encoders with constraint length K and memory $K - 1$ is in order. The input to the encoder at time k is a bit d_k, and the corresponding codeword is the bit pair (u_k, v_k), where

$$u_k = \sum_{i=0}^{K-1} g_{1i}\, d_{k-i} \quad \text{modulo-2}, \quad g_{1i} = 0, 1 \tag{8.105}$$

and

$$v_k = \sum_{i=0}^{K-1} g_{2i}\, d_{k-i} \quad \text{modulo-2}, \quad g_{2i} = 0, 1 \tag{8.106}$$

$\mathbf{G}_1 = \{g_{1i}\}$ and $\mathbf{G}_2 = \{g_{2i}\}$ are the code generators, and d_k is represented as a binary digit. This encoder can be visualized as a discrete-time finite impulse response (FIR) linear system, giving rise to the familiar nonsystematic convolutional (NSC) code, an example of which is shown in Figure 8.24. Its trellis structure can be seen in Figure 7.7. In this example, the constraint length is $K = 3$, and the two code generators are described by $\mathbf{G}_1 = \{111\}$ and $\mathbf{G}_2 = \{101\}$. It is well known that at large E_b/N_0 values, the error performance of a NSC is better than that of a systematic code having the same memory. At small E_b/N_0 values, it is generally the other way around [17]. A class of infinite impulse response (IIR) convolutional codes [17] has been proposed as building blocks for a turbo code. Such building blocks are also referred to as recursive systematic convolutional (RSC) codes because previously encoded information bits are continually fed back to the encoder's input. For high code rates, RSC codes result in better error performance than the best NSC codes at any value of E_b/N_0. A binary rate 1/2 RSC code is obtained from a NSC code by using a feedback loop, and setting one of the two outputs (u_k or v_k) equal to d_k. Figure 8.25a illustrates an example of such an RSC code, with $K = 3$, where a_k is recursively calculated as

$$a_k = d_k + \sum_{i=1}^{K-1} g_i' a_{k-i} \quad \text{modulo-2} \tag{8.107}$$

and g_i' is equal to g_{1i} if $u_k = d_k$, and to g_{2i} if $v_k = d_k$. Figure 8.25b shows the trellis structure for the RSC code in Figure 8.25a.

It is assumed that an input bit d_k takes on values of 1 or 0 with equal probability. Furthermore, $\{a_k\}$ exhibits the same statistical properties as $\{d_k\}$ [17]. The free distance is identical for the RSC code of Figure 8.25a and the NSC code of Figure 8.24. Similarly, their trellis structures are identical with respect to state transitions and their corresponding output bits. However, the two output sequences $\{u_k\}$ and $\{v_k\}$ do not correspond to the same input sequence $\{d_k\}$ for RSC and NSC codes. For the same code generators, it can be said that the weight distribution of the output codewords from an RSC encoder is not modified compared with the weight distribution from the NSC counterpart. The only change is the mapping between input data sequences and output codeword sequences.

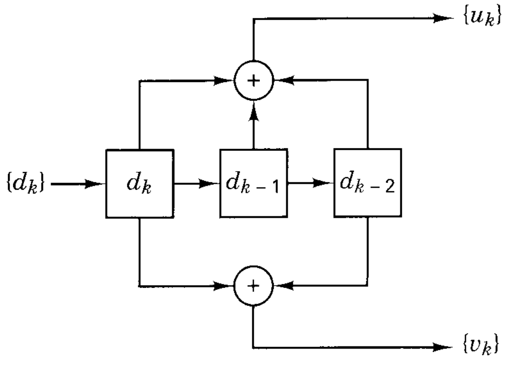


Figure 8.24 Nonsystematic convolutional (NSC) code.

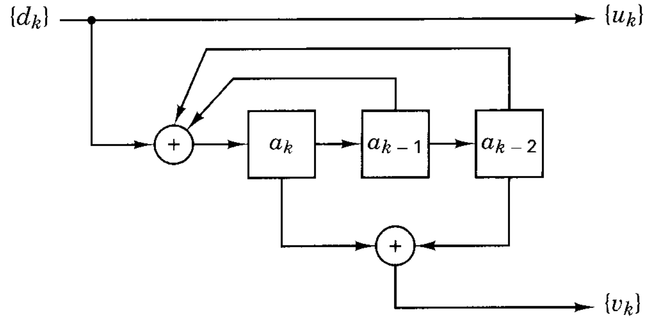


Figure 8.25a Recursive systematic convolutional (RSC) code.

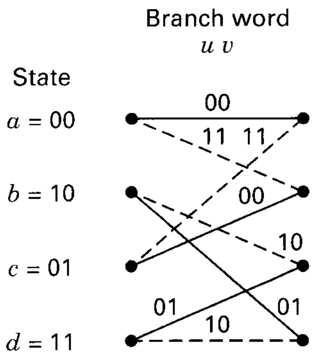


Figure 8.25b Trellis structure for the RSC code in part a).

Example 8.5 Recursive Encoders and their Trellis Diagrams

a) Using the RSC encoder in Figure 8.25a, verify the section of the trellis structure (diagram) shown in Figure 8.25b.

b) For the encoder in part a), start with the input data sequence $\{d_k\}$ = 1 1 1 0, and show the step-by-step encoder procedure for finding the output codeword.

Solution

a) For NSC encoders, keeping track of the register contents and state transitions is a straightforward procedure. However, when the encoders are recursive, more care must be taken. Table 8.5 is made up of 8 rows corresponding to the 8 possible transitions in this 4-state machine. The first four rows represent transitions when the input data bit, d_k, is a binary zero, and the last four rows represent transitions when d_k is a one. For this example, the step-by-step encoding procedure can be described with reference to Table 8.5 and Figure 8.25 as follows:

1. At any input-bit time, k, the (starting) state of a transition, is denoted by the contents of the two rightmost stages in the register, namely a_{k-1} and a_{k-2}.
2. For any row on the table (transition on the trellis), the contents of the a_k stage is found by the modulo-2 addition of bits d_k, a_{k-1} and a_{k-2} on that row.
3. The output code-bit sequence, $u_k v_k$, for each possible starting state (i.e., a = 00, b = 10, c = 01, and d = 11) is found by appending the modulo-2 addition of a_k and a_{k-2} to the current data bit $d_k = u_k$.

TABLE 8.5 Validation of the Figure 8.25b Trellis Section

Input bit	Current bit	Starting state		Code bits		Ending state	
$d_k = u_k$	a_k	a_{k-1}	a_{k-2}	u_k	v_k	a_k	a_{k-1}
	0	0	0	0	0	0	0
	1	1	0	0	1	1	1
0	1	0	1	0	0	1	0
	0	1	1	0	1	0	1
	1	0	0	1	1	1	0
	0	1	0	1	0	0	1
1	0	0	1	1	1	0	0
	1	1	1	1	0	1	1

It is easy to verify that the details in Table 8.5 correspond to the trellis section of Figure 8.25b. An interesting property of the most useful recursive shift registers used as component codes for turbo encoders is that the two transitions entering a state *should not* correspond to the same input bit value (i.e., two solid lines or two dashed lines should not enter a given state). This property is assured if the polynomial describing the feedback in the shift register is of full degree, which means one of the feedback lines must emanate from the highest-order stage, in this example, stage a_{k-2}.

b) There are two ways to proceed with encoding the input data sequence $\{d_k\}$ = 1 1 1 0. One way uses the trellis diagram, and the other way uses the encoder circuit. Using the trellis section in Figure 8.25b, we choose the dashed-line transition (representing input bit binary one) from the state a = 00 (a natural choice for the starting state) to the next state b =10 (which becomes the starting state for the next input bit). We denote the bits shown on that transition as the output coded-bit sequence 11. This procedure is repeated for each input bit. Another way to proceed is to build a table, such as Table 8.6, based on the encoder circuit in Figure 8.25a. Here, time, k, is shown from start to finish (5 time instances and 4 time intervals). Table 8.6 is read as follows:

1. At any instant of time, a data bit d_k becomes transformed to a_k by summing it (modulo-2) to the bits a_{k-1} and a_{k-2} on the same row.
2. For example, at time k = 2, the data bit d_k = 1 is tranformed to a_k = 0 by summing it to the bits a_{k-1} and a_{k-2} on the same k = 2 row.
3. The resulting output, $u_k v_k$ = 10 dictated by the encoder logic circuitry, is the coded-bit sequence associated with time k = 2 (actually the time interval between times k = 2 and k = 3).

TABLE 8.6 Encoding a Bit Sequence with the Figure 8.25a Encoder

Time	Input bit	First stage	State at time k		Code bits	
k	$d_k = u_k$	a_k	a_{k-1}	a_{k-2}	u_k	v_k
1	1	1	0	0	1	1
2	1	0	1	0	1	0
3	1	0	0	1	1	1
4	0	0	0	0	0	0
5			0	0		

4. At time $k = 2$, the contents, 10, of the rightmost two stages, $a_{k-1}\ a_{k-2}$, represents the state of the machine at the start of that transition.
5. The state at the end of that transition is seen as the contents, 01, in the two leftmost stages, $a_k\ a_{k-1}$, on that same row. Since the bits shift from left to right, this transition-terminating state reappears as the starting state for time $k = 3$ on the next row.
6. Each row can be described in the same way. Thus, the encoded sequence seen in the final column of Table 8.6 is 1 1 1 0 1 1 0 0.

8.4.4.1 Concatenation of RSC Codes

Consider the parallel concatenation of two RSC encoders of the type shown in Figure 8.25. Good turbo codes have been constructed from component codes having short constraint lengths ($K = 3$ to 5). An example of such a turbo encoder is shown in Figure 8.26, where the switch yielding v_k provides puncturing, making the overall code rate 1/2. Without the switch, the code rate would be 1/3. There is no limit to the number of encoders that may be concatenated, and, in general, the component codes need not be identical with regard to constraint length and rate. The goal in designing turbo codes is to choose the best component codes by maximizing the effective free distance of the code [21]. At large values of E_b/N_0, this is tantamount to maximizing the minimum weight codeword. However, at low values of E_b/N_0 (the region of greatest interest), optimizing the weight distribution of the codewords is more important than maximizing the minimum weight codeword [20].

The turbo encoder in Figure 8.26 produces codewords from each of two component encoders. The weight distribution for the codewords out of this parallel concatenation depends on how the codewords from one of the component encoders are combined with codewords from the other encoder.

Intuitively, we should avoid pairing low-weight codewords from one encoder with low-weight codewords from the other encoder. Many such pairings can be avoided by proper design of the interleaver. An interleaver that permutes the data in a random fashion provides better performance than the familiar block interleaver [22].

If the component encoders are not recursive, the unit weight input sequence (0 0 . . . 0 0 1 0 0 . . . 0 0) will always generate a low weight codeword at the input of a second encoder for any interleaver design. In other words, the interleaver would not influence the output codeword weight distribution if the component codes were not recursive. However, if the component codes are recursive, a weight-1 input sequence generates an infinite impulse response (infinite-weight output). Therefore, for the case of recursive codes, the weight-1 input sequence does not yield the minimum weight codeword out of the encoder. The encoded output weight is kept finite only by trellis termination, a process that forces the coded sequence to terminate in such a way that the encoder returns to the zero state. In effect, the convolutional code is converted to a block code.

For the encoder of Figure 8.26, the minimum weight codeword for each component encoder is generated by the weight-3 input sequence (0 0 . . . 0 0 1 1 1 0 0 0 . . . 0 0), with three consecutive 1's. Another input that produces fairly low weight codewords is the weight-2 sequence (0 0 . . . 0 0 1 0 0 1 0 0 . . . 0 0). However, after the permutations introduced by an interleaver, either of these deleterious input

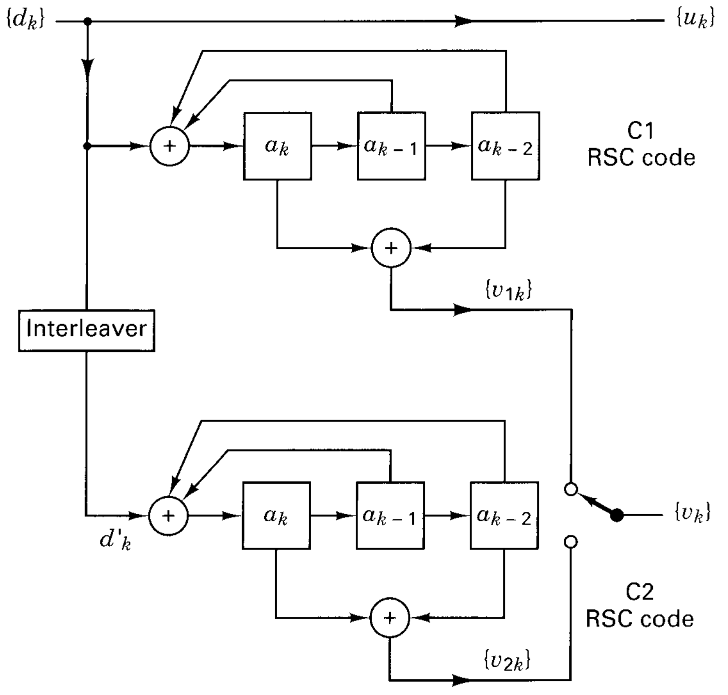


Figure 8.26 Parallel concatenation of two RSC encoders.

patterns is not likely to appear again at the input to another encoder, making it unlikely that a minimum weight codeword will be combined with another minimum weight codeword.

The important aspect of the building blocks used in turbo codes is that they are recursive (the systematic aspect is merely incidental). It is the RSC code's IIR property that protects against the generation of low-weight codewords that cannot be remedied by an interleaver. One can argue that turbo code performance is largely influenced by minimum weight codewords that result from the weight-2 input sequence. The argument is that weight-1 inputs can be ignored since they yield large codeword weights due to the IIR encoder structure. For input sequences having weight-3 and larger, a properly designed interleaver makes the occurrence of low weight output codewords relatively rare [21–25].

8.4.5 A Feedback Decoder

The Viterbi algorithm (VA) is an optimal decoding method for minimizing the probability of sequence error. Unfortunately, the (hard-decision output) VA is not suited to generate the a posteriori probability (APP) or soft-decision output for each decoded bit. A relevant algorithm for doing this has been proposed by Bahl et. al. [26]. The Bahl algorithm was modified by Berrou, et. al. [17] for use in decoding RSC codes. The APP that a decoded data bit $d_k = i$ can be derived from the joint probability $\lambda_k^{i,m}$ defined by

$$\lambda_k^{i,m} = P\{d_k = i, S_k = m \,|\, R_1^N\} \tag{8.108}$$

where $S_k = m$ is the encoder state at time k, and R_1^N is a received binary sequence from time $k = 1$ through some time N.

Thus, the APP that a decoded data bit $d_k = i$, represented as a binary digit, is obtained by summing the joint probability over all states, as follows:

$$P\{d_k = i \,|R_1^N\} = \sum_m \lambda_k^{i,m} \qquad i = 0, 1 \tag{8.109}$$

Next, the log-likelihood ratio (LLR) is written as the logarithm of the ratio of APPs, as

$$L(\hat{d}_k) = \log \left[\frac{\sum_m \lambda_k^{1,m}}{\sum_m \lambda_k^{0,m}} \right] \tag{8.110}$$

The decoder makes a decision, known as the *maximum a posteriori* (MAP) decision rule, by comparing $L(\hat{d}_k)$ to a zero threshold. That is,

$$\begin{aligned} \hat{d}_k &= 1 \quad \text{if} \quad L(\hat{d}_k) > 0 \\ \hat{d}_k &= 0 \quad \text{if} \quad L(\hat{d}_k) < 0 \end{aligned} \tag{8.111}$$

For a systematic code, the LLR $L(\hat{d}_k)$ associated with each decoded bit $\hat{d}_k$ can be described as the sum of the LLR of $\hat{d}_k$, out of the demodulator and of other LLRs generated by the decoder (extrinsic information), as was expressed in Equations (8.72) and (8.73). Consider the detection of a noisy data sequence that stems from the encoder of Figure 8.26, with the use of a decoder shown in Figure 8.27.

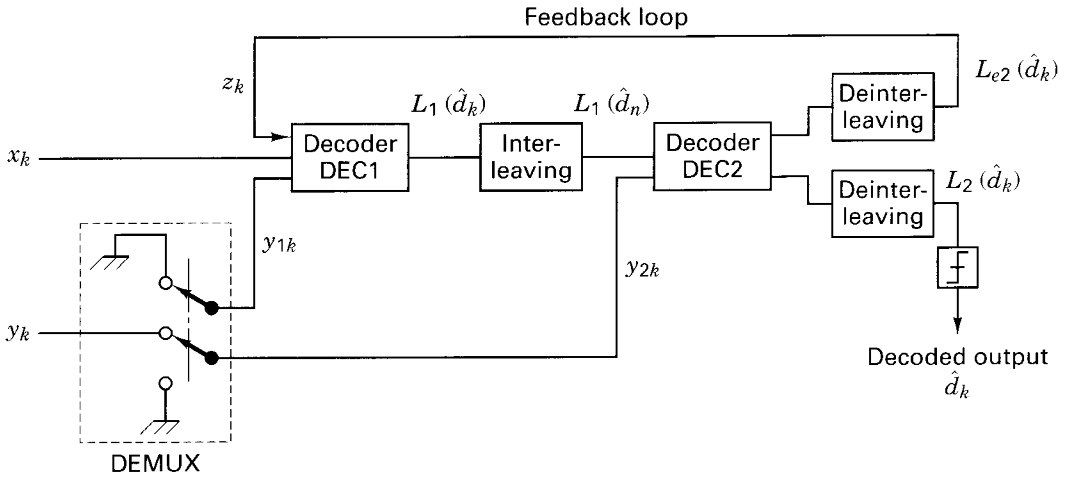


Figure 8.27 Feedback decoder.

Assume binary modulation and a discrete memoryless Gaussian channel. The decoder input is made up of a set R_k of two random variables x_k and y_k. For the bits d_k and v_k at time k, expressed as binary numbers (1, 0), the conversion to received bipolar (+1, −1) pulses can be expressed as

$$x_k = (2d_k - 1) + i_k \tag{8.112}$$

and

$$y_k = (2v_k - 1) + q_k \tag{8.113}$$

where i_k and q_k are two statistically independent random variables with the same variance σ^2, accounting for the noise contribution. The redundant information y_k is demultiplexed and sent to decoder DEC1 as y_{1k}, when $v_k = v_{1k}$, and to decoder DEC2 as y_{2k}, when $v_k = v_{2k}$. When the redundant information of a given encoder (C1 or C2) is not emitted, the corresponding decoder input is set to zero. Note that the output of DEC1 has an interleaver structure identical to the one used at the transmitter between the two component encoders. This is because the information processed by DEC1 is the noninterleaved output of C1 (corrupted by channel noise). Conversely, the information processed by DEC2 is the noisy output of C2 whose input is the same data going into C1, however permuted by the interleaver. DEC2 makes use of the DEC1 output, provided this output is time ordered in the same way as the input to C2 (i.e., the two sequences into DEC2 must appear "in step" with respect to the positional arrangement of the signals in each sequence).

8.4.5.1 Decoding with a Feedback Loop

We rewrite Equation (8.71) for the soft-decision output at time k, with the a priori LLR $L(d_k)$ initially set to zero. This follows from the assumption that the data bits are equally likely. Therefore,

$$\begin{aligned} L(\hat{d}_k) &= L_c(x_k) + L_e(\hat{d}_k) \qquad (8.114) \\ &= \log\left[\frac{p(x_k|d_k = 1)}{p(x_k|d_k = 0)}\right] + L_e(\hat{d}_k) \end{aligned}$$

where $L(\hat{d}_k)$ is the soft-decision output at the decoder, and $L_c(x_k)$ is the LLR channel measurement, stemming from the ratio of likelihood functions $p(x_k|d_k = i)$ associated with the discrete memoryless channel model. $L_e\hat{d}_k = L(\hat{d}_k)|_{x_k=0}$ is a function of the redundant information. It is the extrinsic information supplied by the decoder and does not depend on the decoder input x_k. Ideally $L_c(x_k)$ and $L_e(\hat{d}_k)$ are corrupted by uncorrelated noise, and thus $L_e(\hat{d}_k)$ may be used as a new observation of d_k by another decoder to form an iterative process. The fundamental principle for feeding back information to another decoder is that a decoder should never be supplied with information that stems from its own input (because the input and output corruption will be highly correlated).

For the Gaussian channel, the natural logarithm in Equation (8.114) is used to describe the channel LLR $L_c(x_k)$, as was done in Equations (8.77). We rewrite the Equation (8.77c) LLR result as

$$L_c(x_k) = -\frac{1}{2}\left(\frac{x_k - 1}{\sigma}\right)^2 + \frac{1}{2}\left(\frac{x_k + 1}{\sigma}\right)^2 = \frac{2}{\sigma^2}x_k \tag{8.115}$$

Both decoders, DEC1 and DEC2 use the modified Bahl algorithm [26]. If the inputs $L_1(\hat{d}_k)$ and y_{2k} to decoder DEC2 (see Figure 8.27) are statistically independent, then the LLR $L_2(\hat{d}_k)$ at the output of DEC2 can be written as

$$L_2(\hat{d}_k) = f[L_1(\hat{d}_k)] + L_{e2}(\hat{d}_k) \tag{8.116}$$

with

$$L_1(\hat{d}_k) = \frac{2}{\sigma_0^2}x_k + L_{e1}(\hat{d}_k) \tag{8.117}$$

where $f[\cdot]$ indicates a functional relationship. The extrinsic information $L_{e2}(\hat{d}_k)$ out of DEC2 is a function of the sequence $\{L_1(\hat{d}_k)\}_{n \neq k}$. Since $L_1(\hat{d}_n)$ depends on the observation R_1^N, then the extrinsic information $L_{e2}(\hat{d}_k)$ is correlated with observations x_k and y_{1k}. Nevertheless, the greater $|n - k|$ is, the less correlated are $L_1(\hat{d}_n)$ and the observations x_k, y_k. Thus, due to the interleaving between DEC1 and DEC2, the extrinsic information $L_{e2}(\hat{d}_k)$ and the observations x_k, y_{1k} are weakly correlated. Therefore, they can be jointly used for the decoding of bit d_k [17]. In Figure 8.27, the parameter $z_k = L_{e2}(\hat{d}_k)$ feeding into DEC1 acts as a diversity effect in an iterative process. In general, $L_{e2}(\hat{d}_k)$ will have the same sign as d_k. Therefore, $L_{e2}(\hat{d}_k)$ may increase the associated LLR and thereby improve the reliability of each decoded data bit.

The algorithmic details for computing the LLR $L(\hat{d}_k)$ of the a posteriori probability (APP) for each data bit has been described by several authors [17–18, 30]. Suggestions for decreasing the implementational complexity of the algorithms can be found in [27–31]. A reasonable way to think of the process that produces APP values for each data bit is to imagine implementing a maximum likelihood sequence estimation or Viterbi algorithm (VA) and computing it in two directions over a block of code bits. Proceeding with this bi-directional VA in a sliding-window fashion—and thereby obtaining metrics associated with states in the forward and backward direction—allows computing the APP for each data bit represented in the block. With this view in mind, the decoding of turbo codes can be estimated to be at least two times more complex than decoding one of its component codes using the VA.

8.4.5.2 Turbo Code Error-Performance Example

Performance results using Monte Carlo simulations have been presented in [17] for a rate 1/2, $K = 5$ encoder implemented with generators $\mathbf{G}_1 = \{1\ 1\ 1\ 1\ 1\}$ and $\mathbf{G}_2 = \{1\ 0\ 0\ 0\ 1\}$, using parallel concatenation and a 256×256 array interleaver. The modified Bahl algorithm was used with a data block length of 65,536 bits. After 18 decoder iterations, the bit-error probability P_B was less than 10^{-5} at $E_b/N_0 = 0.7$ dB. The error-performance improvement as a function of the number of decoder iterations is seen in Figure 8.28. Note that as the Shannon limit of -1.6 dB is approached, the required system bandwidth approaches infinity, and the capacity

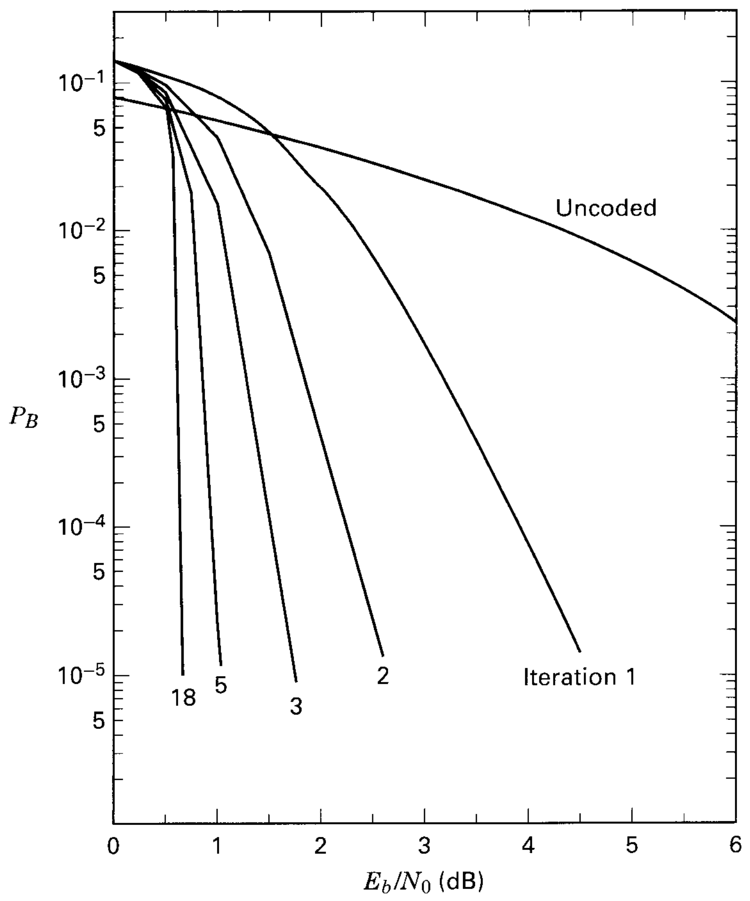


Figure 8.28 Bit-error probability as a function of E_b/N_o and multiple iterations.
REF: Berrou, C., Glavieux, A., and Thitimajshima, P, "Near Shannon Limit Error-Correcting Coding and Decoding: Turbo Codes," IEEE Proc. of Int'l. Conf. on Communications, Geneva, Switzerland, May 1993 (ICC '93), pp. 1064-1070.

(code rate) approaches zero. Therefore, the Shannon limit represents an interesting theoretical bound, but it is not a practical goal. For binary modulation, several authors use $P_B = 10^{-5}$ and $E_b/N_0 = 0.2$ dB as a *pragmatic* Shannon limit reference for a rate 1/2 code. Thus, with parallel concatenation of RSC convolutional codes and feedback decoding, the error performance of a turbo code at $P_B = 10^{-5}$ is within 0.5 dB of the (pragmatic) Shannon limit. A class of codes that use serial instead of parallel concatenation of the interleaved building blocks has been proposed. It has been suggested that serial concatenation of codes may have superior performance [28] to those that use parallel concatenation.

8.4.6 The MAP Algorithm

The process of turbo-code decoding starts with the formation of *a posteriori probabilities* (APPs) for each data bit, which is followed by choosing the data-bit value that corresponds to the *maximum a posteriori* (MAP) probability for that data bit. Upon reception of a corrupted code-bit sequence, the process of decision making with APPs, allows the MAP algorithm to determine the most likely information bit to have been transmitted at each bit time. This is unlike the Viterbi algorithm (VA), where the APP for each data bit is not available. Instead, the VA finds the most likely sequence to have been transmitted. There are, however, similarities in the implementation of the two algorithms. (See Section 8.4.6.3.) When the decoded P_B is small, there is very little performance difference between the MAP and a soft-output Viterbi algorithm called SOVA. However, at low E_b/N_0 and high P_B values, the MAP algorithm can outperform SOVA decoding by 0.5 dB or more [30, 31]. For turbo codes, this can be very important, since the first decoding iterations can yield poor error performance. The implementation of the MAP algorithm proceeds somewhat like performing a Viterbi algorithm in two directions over a block of code bits. Once this bi-directional computation yields state and branch metrics for the block, the APPs and the MAP can be obtained for each data bit represented within the block. We describe here a derivation of the MAP decoding algorithm for systematic convolutional codes assuming an AWGN channel model, as presented by Pietrobon [30]. We start with the ratio of the APPs, known as the likelihood ratio $\Lambda(\hat{d}_k)$, or its logarithm, $L(\hat{d}_k)$ called the log-likelihood ratio (LLR), as shown earlier in Equation (8.110):

$$\Lambda(\hat{d}_k) = \frac{\sum_m \lambda_k^{1,m}}{\sum_m \lambda_k^{0,m}} \tag{8.118a}$$

and

$$L(\hat{d}_k) = \log \left[\frac{\sum_m \lambda_k^{1,m}}{\sum_m \lambda_k^{0,m}} \right] \tag{8.118b}$$

Here $\lambda_k^{i,m}$—the joint probability that data $d_k = i$ and state $S_k = m$, conditioned on the received binary sequence R_1^N, observed from time $k = 1$ through some time N—is described by Equation (8.108), and rewritten below:

$$\lambda_k^{i,m} = P(d_k = i, S_k = m \mid R_1^N) \tag{8.119}$$

R_1^N represents a corrupted code-bit sequence after it has been transmitted through the channel, demodulated, and presented to the decoder in soft-decision form. In effect, the MAP algorithm requires that the output sequence from the demodulator be presented to the decoder as a block of N bits at a time. Let R_1^N be written as follows:

$$R_1^N = \{R_1^{k-1}, R_k, R_{k+1}^N\} \tag{8.120}$$

To facilitate the use of Bayes' rule, Equation (8.119) is partitioned using the letters A, B, C, D, and Equation (8.120). Thus, Equation (8.119) can be written in the form

$$\lambda_k^{i,m} = P(\underbrace{d_k = i, S_k = m}_{A} \,|\underbrace{R_1^{k-1}}_{B},\ \underbrace{R_k}_{C},\ \underbrace{R_{k+1}^N}_{D}) \tag{8.121}$$

Recall from Bayes' rule that

$$\begin{aligned} P(A|B,\ C,\ D) &= \frac{P(A,\ B,\ C,\ D)}{P(B,\ C,\ D)} = \frac{P(B|A,\ C,\ D)\,P(A,\ C,\ D)}{P(B,\ C,\ D)} \\ &= \frac{P(B|A,\ C,\ D)\,P(D|A,\ C)\,P(A,\ C)}{P(B,\ C,\ D)} \end{aligned} \tag{8.122}$$

Hence, application of this rule to Equation (8.121) yields

$$\begin{aligned} \lambda_k^{i,m} &= P(R_1^{k-1}|d_k = i, S_k = m, R_k^N)\, P(R_{k+1}^N|d_k = i, S_k = m, R_k) \\ &\quad \times P(d_k = i, S_k = m, R_k)/P(R_1^N) \end{aligned} \tag{8.123}$$

where $R_k^N = \{R_k,\ R_{k+1}^N\}$. Equation (8.123) can be expressed in a way that gives greater meaning to the probability terms contributing to $\lambda_k^{i,m}$. In the sections that follow, the three numerator factors on the right side of Equation (8.123) will be defined and developed as the forward state metric, the reverse state metric, and the branch metric.

8.4.6.1 The State Metrics and the Branch Metric

We define the first numerator factor on the right side of Equation (8.123) as the forward state metric at time k and state m, and denote it as α_k^m. Thus, for $i = 1, 0$

$$P(R_1^{k-1}\,|\overbrace{d_k = i}^{\text{IRRELEVANT}}, S_k = m, \overbrace{R_k^N}^{\text{IRRELEVANT}}) = P(R_1^{k-1}\,|S_k = m) \triangleq \alpha_k^m \tag{8.124}$$

Note that $d_k = i$ and R_k^N are designated as irrelevant, since the assumption that $S_k = m$ implies that events before time k are not influenced by observations after time k. In other words, the past is not affected by the future; hence, $P(R_1^{k-1})$ is independent of the fact that $d_k = i$ and the sequence R_k^N. However, since the encoder has memory, the encoder state $S_k = m$ is based on the past, so this term is relevant and must be left in the expression. The form of Equation (8.124) is intuitively satisfying since it presents the forward state metric α_k^m, at time k, as being a probability of the past sequence that is only dependent on the current state induced by this sequence and nothing more. This should be familiar to us from the convolutional encoder and its state representation as a Markov process in Chapter 7.

Similarly, the second numerator factor on the right side of Equation (8.123) represents a reverse state metric β_k^m, at time k and state m, described by

$$P(R_{k+1}^{N} \mid d_k = i, S_k = m, R_k) = P(R_{k+1}^{N} \mid S_{k+1} = f(i,m)) \triangleq \beta_{k+1}^{f(i,m)} \quad (8.125)$$

where $f(i, m)$ is the next state, given an input i and state m, and $\beta_{k+1}^{f(i,m)}$ is the reverse state metric at time $k + 1$ and state $f(i, m)$. The form of Equation (8.125) is intuitively satisfying since it presents the reverse state metric β_{k+1}^{m}, at future time $k + 1$, as being a probability of the future sequence, which depends on the state (at future time $k + 1$), which in turn is a function of the input bit and the state (at current time k). This should be familiar to us because it engenders the basic definition of a finite-state machine (see Section 7.2.2).

We define the third numerator factor on the right side of Equation (8.123) as the branch metric at time k and state m, which we denote $\delta_k^{i,m}$. Thus, we write

$$P(d_k = i, S_k = m, R_k) \triangleq \delta_k^{i,m} \quad (8.126)$$

Substituting Equations (8.124) through (8.126) into Equation (8.123) yields the following more compact expression for the joint probability, as follows:

$$\lambda_k^{i,m} = \frac{\alpha_k^{m} \, \delta_k^{i,m} \, \beta_{k+1}^{f(i,m)}}{P(R_1^N)} \quad (8.127)$$

Equation (8.127) can be used to express Equation (8.118) as

$$\Lambda(\hat{d}_k) = \frac{\sum_m \alpha_k^{m} \, \delta_k^{1,m} \, \beta_{k+1}^{f(1,m)}}{\sum_m \alpha_k^{m} \, \delta_k^{0,m} \, \beta_{k+1}^{f(0,m)}} \quad (8.128a)$$

and

$$L(\hat{d}_k) = \log \left[\frac{\sum_m \alpha_k^{m} \, \delta_k^{1,m} \, \beta_{k+1}^{f(1,m)}}{\sum_m \alpha_k^{m} \, \delta_k^{0,m} \, \beta_{k+1}^{f(0,m)}} \right] \quad (8.128b)$$

where $\Lambda(\hat{d}_k)$ is the likelihood ratio of the k-th data bit, and $L(\hat{d}_k)$ the logarithm of $\Lambda(\hat{d}_k)$, is the LLR of the k-th data bit, where the logarithm is generally taken to the base e.

8.4.6.2 Calculating the Forward State Metric

Starting from Equation (8.124), α_k^m can be expressed as the summation of all possible transition probabilities from time $k - 1$, as follows:

$$\alpha_k^{m} = \sum_{m'} \sum_{j=0}^{1} P(d_{k-1} = j, S_{k-1} = m', R_1^{k-1} \mid S_k = m) \quad (8.129)$$

We can rewrite R_1^{k-1} as $\{R_1^{k-2}, R_{k-1}\}$, and from Bayes' Rule,

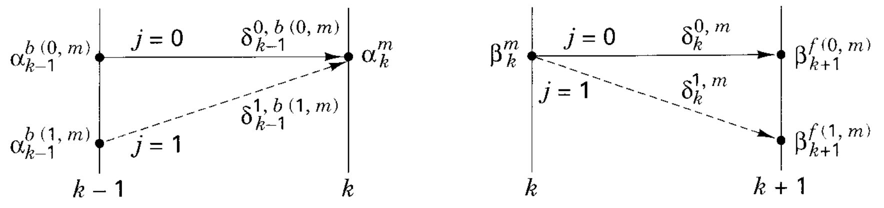
$$\alpha_k^m = \sum_{m'} \sum_{j=0}^{1} P(R_1^{k-2} \,|\, S_k = m, d_{k-1} = j, S_{k-1} = m', R_{k-1}) \times P(d_{k-1} = j, S_{k-1} = m', R_{k-1} \,|\, S_k = m) \qquad (8.130a)$$

$$= \sum_{j=0}^{1} P(R_1^{k-2} | S_{k-1} = b(j,m))\, P(d_{k-1} = j,\, S_{k-1} = b(j,m),\, R_{k-1}) \qquad (8.130b)$$

where $b(j, m)$ is the state going backwards in time from state m, via the previous branch corresponding to input j. Equation (8.130b) can replace Equation (8.130a), since knowledge about the state m' and the input j, at time $k - 1$, completely defines the path resulting in state $S_k = m$. Using Equations (8.124) and (8.126) to simplify the notation of Equation (8.130b) yields

$$\alpha_k^m = \sum_{j=0}^{1} \alpha_{k-1}^{b(j,m)} \; \delta_{k-1}^{j,\,b(j,m)} \qquad (8.131)$$

Equation (8.131) indicates that a new forward state metric at time k and state m is obtained by summing two weighted state metrics from time $k - 1$. The weighting consists of the branch metrics associated with the transitions corresponding to data bits 0 and 1. Figure 8.29a illustrates the use of two different types of notation for the parameter alpha. We use $\alpha_{k-1}^{b(j,m)}$ for the forward state metric at time $k - 1$, when there are two possible underlying states (depending upon whether $j = 0$ or 1). And we use α_k^m for the forward state metric at time k, when the two possible transitions from the previous time terminate on the same state m at time k.



(a) Forward state metric:

$$\alpha_k^m = \alpha_{k-1}^{b(0,m)} \delta_{k-1}^{0,b(0,m)} + \alpha_{k-1}^{b(1,m)} \delta_{k-1}^{1,b(1,m)}$$

Where $b(j, m)$ is the state going backwards in time corresponding to an input j

(b) Reverse state metric:

$$\beta_k^m = \beta_{k+1}^{f(0,m)} \delta_k^{0,m} + \beta_{k+1}^{f(1,m)} \delta_k^{1,m}$$

Where $f(j, m)$ is the next state given an input j and state m

Branch metric:

$$\delta_k^{i,m} = \pi_k^i \exp\left(x_k u_k^i + y_k v_k^{i,m}\right)$$

Figure 8.29 Graphical representation for calculating α_k^m and β_k^m.
REF: Pietrobon, S.S., "Implementation and Performance of a Turbo/Map Decoder," Int'l. J. of Satellite Communications, vol. 16, Jan.-Feb. 1998, pp. 23–46.

8.4.6.3 Calculating the Reverse State Metric

Starting from Equation (8.125), where $\beta_{k+1}^{f(i,m)} = P[R_{k+1}^N \mid S_{k+1} = f(i, m)]$, we have

$$\beta_k^m = P(R_k^N \mid S_k = m) = P(R_k, R_{k+1}^N \mid S_k = m) \tag{8.132}$$

We can express β_k^m as the summation of all possible transition probabilities to time $k + 1$, as follows:

$$\beta_k^m = \sum_{m'} \sum_{j=0}^{1} P(d_k = j, S_{k+1} = m', R_k, R_{k+1}^N \mid S_k = m) \tag{8.133}$$

Using Bayes' Rule,

$$\begin{aligned}\beta_k^m = \sum_{m'} \sum_{j=0}^{1} & P(R_{k+1}^N \mid S_k = m, d_k = j, S_{k+1} = m', R_k) \\ & \times P(d_k = j, S_{k+1} = m', R_k \mid S_k = m)\end{aligned} \tag{8.134}$$

$S_k = m$ and $d_k = j$ in the first term on the right side of Equation (8.134) completely defines the path resulting in $S_{k+1} = f(j, m)$, the next state given an input j and state m. Thus, these conditions allow replacing $S_{k+1} = m'$ with $S_k = m$ in the second term of Equation (8.134), yielding

$$\begin{aligned}\beta_k^m &= \sum_{j=0}^{1} P(R_{k+1}^N \mid S_{k+1} = f(j, m))\, P(d_k = j, S_k = m, R_k) \\ &= \sum_{j=0}^{1} \delta_k^{j,m}\, \beta_{k+1}^{f(j,m)}\end{aligned} \tag{8.135}$$

Equation (8.135) indicates that a new reverse state metric at time k and state m is obtained by summing two weighted state metrics from time $k + 1$. The weighting consists of the branch metrics associated with the transitions corresponding to data bits 0 and 1. Figure 8.29b illustrates the use of two different types of notation for the parameter beta. First, we use $\beta_{k+1}^{f(j,m)}$ for the reverse state metric at time $k + 1$, when there are two possible underlying states (depending on whether $j = 0$ or 1). Second, we use β_k^m for the reverse state metric at time k, where the two possible transitions arriving at time $k + 1$ stem from the same state m at time k. Figure 8.29 presents a graphical illustration for calculating the forward and reverse state metrics.

Implementing the MAP decoding algorithm has some similarities to implementing the Viterbi decoding algorithm (see Section 7.3). In the Viterbi algorithm, we add branch metrics to state metrics. Then we compare and select the minimum distance (maximum likelihood) in order to form the next state metric. The process is called Add-Compare-Select (ACS). In the MAP algorithm, we multiply (add, in the logarithmic domain) state metrics by branch metrics. Then, instead of comparing them, we sum them to form the next forward (or reverse) state metric, as seen in Figure 8.29. The differences should make intuitive sense. With the Viterbi algorithm, the most likely sequence (path) is being sought; hence, there is a continual comparison and selection to find the best path. With the MAP algorithm, a soft number (likelihood or log-likelihood) is being sought; hence, the process uses all the metrics from all the possible transitions within a time interval, in order to come up with the best overall statistic regarding the data bit associated with that time interval.

8.4.6.4 Calculating the Branch Metric

We start with Equation (8.126),

$$\begin{aligned}\delta_k^{i,m} &= P(d_k = i, S_k = m, R_k) \\ &= P(R_k|d_k = i, S_k = m)\, P(S_k = m|d_k = i)\, P(d_k = i)\end{aligned} \tag{8.136}$$

where R_k represents the sequence $\{x_k, y_k\}$, x_k is the noisy received data bit, and y_k is the corresponding noisy received parity bit. Since the noise affecting the data and the parity are independent, the current state is independent of the current input and can therefore be any one of the 2^υ states, where υ is the number of memory elements in the convolutional code system. That is, the constraint length K of the code is equal to $\upsilon + 1$. Hence,

$$P(S_k = m|d_k = i) = \frac{1}{2^\upsilon}$$

and

$$\delta_k^{i,m} = P(x_k|d_k = i, S_k = m)\, P(y_k|d_k = i, S_k = m)\, \frac{\pi_k^i}{2^\upsilon} \tag{8.137}$$

where π_k^i is defined as $P(d_k = i)$, the a priori probability of d_k.

From Equation (1.25d) in Chapter 1, the probability $P(X_k = x_k)$ of a random variable, X_k taking on the value x_k, is related to the probability density function (pdf) $p_{x_k}(x_k)$, as follows:

$$P(X_k = x_k) = p_{x_k}(x_k)\, dx_k \tag{8.138}$$

For notational convenience, the random variable X_k, which takes on values x_k, is often termed "the random variable x_k", which represents the meanings of x_k and y_k in Equation (8.137). Thus, for an AWGN channel, where the noise has zero mean and variance σ^2, we use Equation (8.138) in order to replace the probability terms in Equation (8.137) with their pdf equivalents, and we write

$$\delta_k^{i,m} = \frac{\pi_k^i}{2^\upsilon\sqrt{2\pi}\sigma} \exp\left[-\frac{1}{2}\left(\frac{x_k - u_k^i}{\sigma}\right)^2\right] dx_k \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{1}{2}\left(\frac{y_k - v_k^{i,m}}{\sigma}\right)^2\right] dy_k \tag{8.139}$$

where u_k and v_k represent the transmitted data bits and parity bits, respectively (in bipolar form), and dx_k and dy_k are the differentials of x_k and y_k, and get absorbed into the constant A_k, below. Note that the parameter u_k^i represents data that have no dependence on the state m. However, the parameter $v_k^{i,m}$ represents parity that does depend on the state m, since the code has memory. Simplifying the notation by eliminating all terms that will appear in both the numerator and denominator of the likelihood ratio, resulting in cancellation, we can write

$$\delta_k^{i,m} = A_k \, \pi_k^i \exp\left[\frac{1}{\sigma^2}(x_k u_k^i + y_k v_k^{i,m})\right] \tag{8.140}$$

If we substitute Equation (8.140) into Equation (8.128a), we obtain

$$\Lambda(\hat{d}_k) = \pi_k \exp\left(\frac{2x_k}{\sigma^2}\right) \frac{\sum_m \alpha_k^m \exp\left(\frac{y_k v_k^{1,m}}{\sigma^2}\right) \beta_{k+1}^{f(1,m)}}{\sum_m \alpha_k^m \exp\left(\frac{y_k v_k^{0,m}}{\sigma^2}\right) \beta_{k+1}^{f(0,m)}} \tag{8.141a}$$

$$= \pi_k \exp\left(\frac{2x_k}{\sigma^2}\right) \pi_k^e \tag{8.141b}$$

and

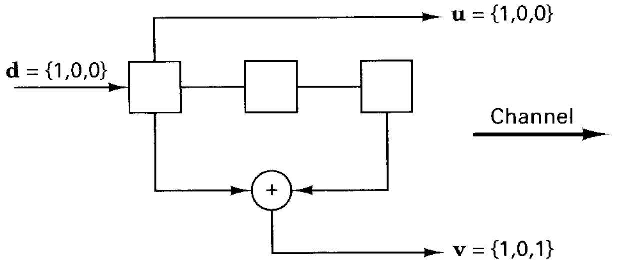
$$L(\hat{d}_k) = L(d_k) + L_c(x_k) + L_e(\hat{d}_k) \tag{8.141c}$$

where $\pi_k = \pi_k^1/\pi_k^0$ is the input a priori probability ratio (prior likelihood), and π_k^e is the output extrinsic likelihood, each at time k. In Equation (8.141b), one can think of π_k^e as a correction term (due to the coding) that changes the input prior knowledge about a data bit. In a turbo code, such correction terms are passed from one decoder to the next, in order to improve the likelihood ratio for each data bit, and thus minimize the probability of decoding error. Thus, the decoding process entails the use of Equation (8.141b) to compute $\Lambda(\hat{d}_k)$ for several iterations. The extrinsic likelihood π_k^e, resulting from a particular iteration replaces the a priori likelihood ratio π_{k+1} for the next iteration. Taking the logarithm of $\Lambda(\hat{d}_k)$ in Equation (8.141b) yields Equation (8.141c), which is the same result provided by Equation (8.71) showing that the final soft number $L(\hat{d}_k)$ is made up of three LLR terms—the a priori LLR, the channel-measurement LLR, and the extrinsic LLR.

The MAP algorithm can be implemented in terms of a likelihood ratio $\Lambda(\hat{d}_k)$, as shown in Equation (8.128a) or (8.141a,b). However, implementation using likelihood ratios is very complex because of the multiply operations that are required. By operating the MAP algorithm in the logarithmic domain [30, 31], as described by the LLR in Equation (8.128b) or (8.141c), the complexity can be greatly reduced by eliminating the multiply operations.

8.4.7 MAP Decoding Example

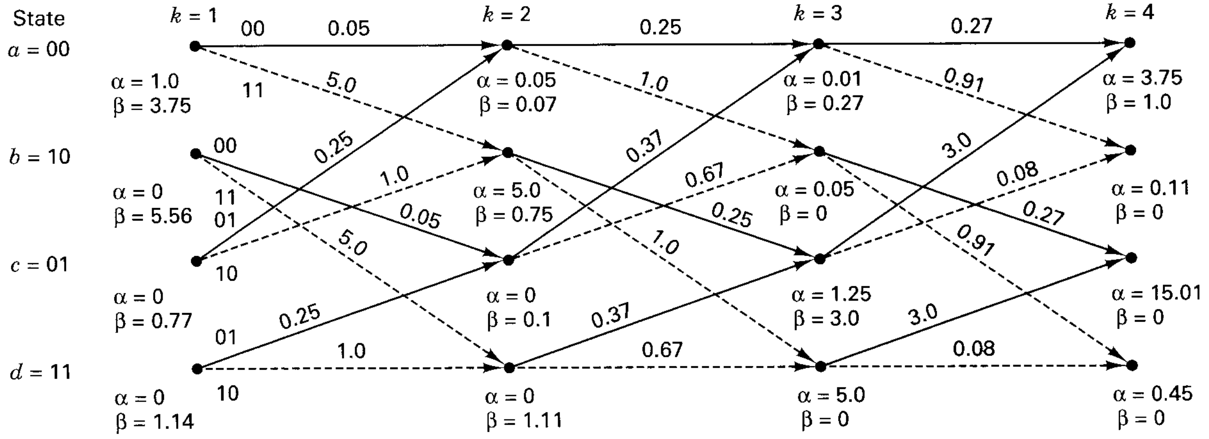
Figure 8.30 illustrates a MAP decoding example. Figure 8.30a shows a simple systematic convolutional encoder, with constraint length, $K = 3$, and rate $\frac{1}{2}$. The input data consists of the sequence $\mathbf{d} = \{1, 0, 0\}$, corresponding to the times $k = 1, 2, 3$. The output code-bit sequence, being systematic, is formed by consecutively taking one bit from the sequence $\mathbf{u} = \{1, 0, 0\}$, followed by one bit from the parity-bit sequence $\mathbf{v} = \{1, 0, 1\}$. In each case, the leftmost bit is the earliest bit. Thus, the output sequence is 1 1 0 0 0 1, or in bipolar form the sequence is +1 +1 −1 −1 −1 +1. Figure 8.30b shows the results of some postulated noise vectors $\mathbf{n}_x$ and $\mathbf{n}_y$, having cor-



(a) Systematic convolutional encoder
($K = 3$, rate 1/2)

	$k = 1$	$k = 2$	$k = 3$
$\mathbf{x} =$	{1.5,	0.5,	−0.6}
$\mathbf{y} =$	{0.8,	0.2,	1.2}
$\pi^1 = \pi^0 =$	{0.5,	0.5,	0.5}

(b) Received channel bits
(bipolar) plus noise



$L(\hat{d}_1) = \log\left(\frac{3.75}{0.0035}\right) = 3.03$ $\quad L(\hat{d}_2) = \log\left(\frac{0}{3.75}\right) = -\infty$ $\quad L(\hat{d}_3) = \log\left(\frac{0}{3.75}\right) = -\infty$

(c) Decoder trellis diagram

Figure 8.30 Example of MAP decoding ($K = 3$, rate ½, systematic).

rupted sequences **u** and **v**, so they are now designated as $\mathbf{x} = \mathbf{u} + \mathbf{n}_x$ and $\mathbf{y} = \mathbf{v} + \mathbf{n}_y$. As shown in Figure 8.30b, the demodulator outputs arriving at the decoder corresponding to the times $k = 1, 2, 3$, have values of 1.5, 0.8, 0.5, 0.2, –0.6, 1.2. Also shown are the a priori probabilities of a data bit being 1 or 0, designated as π^1 and π^0, respectively, and assumed to be equally likely for all k times. For this example, all information is now available to calculate the branch metrics and the state metrics, and enter their values onto the decoder trellis diagram of Figure 8.30c. On the trellis diagram, each transition occurring between times k and $k + 1$ corresponds to a data bit d_k that appears at the encoder input at the transition-start time k. At time k, the encoder will be in some state m, and at time $k + 1$ it transitions to a new state (possibly the same state). When such a trellis diagram is used to depict a sequence of code bits (representing N data bits), the sequence is characterized by N transition intervals and $N + 1$ states (from start to finish).

8.4.7.1 Calculating the Branch Metrics

We start with Equation (8.140), with $\pi_k^i = 0.5$ (for this exercise, data bits are assumed equally likely for all time), and for simplicity assume that $A_k = 1$ for all time and that $\sigma^2 = 1$. Thus, $\delta_k^{i,m}$ becomes

$$\delta_k^{i,m} = 0.5 \exp\,(x_k\, u_k^i + y_k\, v_k^{i,m}) \tag{8.142}$$

What basic receiver function does Equation (8.142) resemble? The expression looks somewhat like a correlation process. At the decoder, a pair of receptions (data-bit related x_k, and parity-bit related y_k) arrive at each time k. The branch metric is calculated by taking the product of the received x_k with each of the prototype signals u_k, and similarly the product of the received y_k with each of the prototype signals v_k. For each trellis transition, the magnitude of the branch metric will be a function of how good a match there is between the pair of noisy receptions and the code-bit meaning of that trellis transition. For $k = 1$, Equation (8.142) is used with the data in Figure 8.30b for evaluating eight branch metrics (a transition from each state m and for each data value i), as shown below. For notational simplicity, we designate the trellis states as follows: $a = 00$, $b = 10$, $c = 01$, $d = 11$. Note that the code-bit meaning, u_k, v_k, of each trellis transition is written on the transition in Figure 8.30c (for $k = 1$ only) and was obtained from the encoder structure in the usual way. (See Section 7.2.4.) Also, for the trellis transitions of Figure 8.30c, the convention that dashed lines and solid lines correspond to the underlying data bits 1 and 0, respectively, is used:

$$\delta_{k=1}^{1,m=a} = \delta_{k=1}^{1,m=b} = 0.5 \exp\,[(1.5)\,(1) + (0.8)\,(1)] = 5.0$$

$$\delta_{k=1}^{0,m=a} = \delta_{k=1}^{0,m=b} = 0.5 \exp\,[(1.5)\,(-1) + (0.8)\,(-1)] = 0.05$$

$$\delta_{k=1}^{1,m=c} = \delta_{k=1}^{1,m=d} = 0.5 \exp\,[(1.5)\,(1) + (0.8)\,(-1)] = 1.0$$

$$\delta_{k=1}^{0,m=c} = \delta_{k=1}^{0,m=d} = 0.5 \exp\,[(1.5)\,(-1) + (0.8)\,(1)] = 0.25$$

Next, we repeat these calculations using Equation (8.142) for the eight branch metric values at time $k = 2$:

$$\delta_{k=2}^{1,m=a} = \delta_{k=2}^{1,m=b} = 0.5 \exp[(0.5)(1) + (0.2)(1)] = 1.0$$
$$\delta_{k=2}^{0,m=a} = \delta_{k=2}^{0,m=b} = 0.5 \exp[(1.5)(-1) + (0.2)(-1)] = 0.25$$
$$\delta_{k=2}^{1,m=c} = \delta_{k=2}^{1,m=d} = 0.5 \exp[(0.5)(1) + (0.2)(-1)] = 0.67$$
$$\delta_{k=2}^{0,m=c} = \delta_{k=2}^{0,m=d} = 0.5 \exp[(0.5)(-1) + (0.2)(1)] = 0.37$$

Again, we repeat the calculations for the eight branch metric values at time $k = 3$:

$$\delta_{k=3}^{1,m=a} = \delta_{k=3}^{1,m=b} = 0.5 \exp[(-0.6)(1) + (1.2)(1)] = 0.91$$
$$\delta_{k=3}^{0,m=a} = \delta_{k=3}^{0,m=b} = 0.5 \exp[(-0.6)(-1) + (1.2)(-1)] = 0.27$$
$$\delta_{k=3}^{1,m=c} = \delta_{k=3}^{1,m=d} = 0.5 \exp[(-0.6)(1) + (1.2)(-1)] = 0.08$$
$$\delta_{k=3}^{0,m=c} = \delta_{k=3}^{0,m=d} = 0.5 \exp[(-0.6)(-1) + (1.2)(1)] = 3.0$$

8.4.7.2 Calculating the State Metrics

Once the eight values of $\delta_k^{i,m}$ are computed for each k, the forward state metrics α_k^m can be calculated with the help of Figures 8.29, 8.30c, and Equation (8.131), rewritten below

$$\alpha_{k+1}^m = \sum_{j=0}^{1} \delta_k^{j,b(j,m)} \alpha_k^{b(j,m)}$$

Assume that the encoder starting state is $a = 00$. Then,

$$\alpha_{k=1}^{m=a} = 1.0 \quad \text{and} \quad \alpha_{k=1}^{m=b} = \alpha_{k=1}^{m=c} = \alpha_{k=1}^{m=d} = 0$$
$$\alpha_{k=2}^{m=a} = (0.05)(1.0) + (0.25)(0) = 0.05$$
$$\alpha_{k=2}^{m=b} = (5.0)(1.0) + (1.0)(0) = 5.0$$
$$\alpha_{k=2}^{m=c} = \alpha_{k=2}^{m=d} = 0$$

and so forth, as shown on the trellis diagram of Figure 8.30c. Similarly, the reverse state metric β_k^m can be calculated with the help of Figures 8.29, 8.30c, and Equation (8.135), rewritten below

$$\beta_k^m = \sum_{j=0}^{1} \delta_k^{j,m} \beta_{k+1}^{f(j,m)}$$

The data sequence and the code in this example were purposely chosen so that the final state of the trellis at time $k = 4$ is the $a = 00$ state. Otherwise, it would be necessary to use tail bits to force the final state into such a known state. Thus, for this example, illustrated in Figure 8.30, knowing that the final state is $a = 00$, the reverse state metrics can be calculated as follows:

$$\beta_{k=4}^{m=a} = 1.0 \quad \text{and} \quad \beta_{k=4}^{m=b} = \beta_{k=4}^{m=c} = \beta_{k=4}^{m=d} = 0$$
$$\beta_{k=3}^{m=a} = (0.27)(1.0) + (0.91)(0) = 0.27$$

$$\beta_{k=3}^{m=b} = \beta_{k=3}^{m=d} = 0$$

$$\beta_{k=3}^{m=c} = (3.0)(1.0) + (0.08)(0) = 3.0$$

and so forth. All the reverse state metric values are shown on the trellis of Figure 8.30c.

8.4.7.3 Calculating the Log-Likelihood Ratio

Now that the metrics δ, α, and β have all been computed for the code-bit sequence in this example, the turbo decoding process can use Equation (8.128) or (8.141) for finding a soft decision, $\Lambda(\hat{d}_k)$ or $L(\hat{d}_k)$, for each data bit. When using turbo codes, this process can be iterated several times to improve the reliability of that decision. This is generally accomplished by using the extrinsic likelihood parameter of Equation (8.141b) to compute and re-compute the likelihood ratio $\Lambda(\hat{d}_k)$ for several iterations. The extrinsic likelihood π_k^e of any iteration is used to replace the a priori likelihood ratio π_{k+1} for the next iteration.

For this example, let us now use the metrics calculated above (with a single pass through the decoder). We choose Equation (8.128b) to compute the LLR for each data bit in the sequence $\{d_k\}$, and then use the decision rules of Equation (8.111) to transform the resulting soft numbers into hard decisions. For $k = 1$, omitting some of the zero factors, we obtain

$$L(\hat{d}_k) = \log\left(\frac{1.0 \times 5.0 \times 0.75}{1.0 \times 0.05 \times 0.07}\right) = \log\left(\frac{3.75}{0.0035}\right) = 3.03$$

For $k = 2$, again omitting some of the zero factors, we obtain

$$L(\hat{d}_2) = \log\left[\frac{(0.05 \times 1.0 \times 0) + (5.0 \times 1.0 \times 0)}{(0.05 \times 0.25 \times 0.27) + (5.0 \times 0.25 \times 3.0)}\right] = \log\left(\frac{0}{3.75}\right) = -\infty$$

For $k = 3$, we obtain

$$L(\hat{d}_3) = \log\left[\frac{(0.01 \times 0.91 \times 0) + (0.05 \times 0.91 \times 0) + (1.25 \times 0.08 \times 0) + (5.0 \times 0.08 \times 0)}{(0.01 \times 0.27 \times 1.0) + (0.05 \times 0.27 \times 0) + (1.25 \times 3.0 \times 1.0) + (5.0 \times 3.0 \times 0)}\right]$$

$$= \log\left(\frac{0}{3.75}\right) = -\infty$$

Using Equation (8.111) to make the final decisions about the bits at times $k = 1, 2, 3$, the sequence is decoded as {1 0 0}. This is clearly correct, given the specified input to the encoder.

8.4.7.4 Shift Register Representation for Finite State Machines

The shift registers used throughout this book, whether feed-forward or feedback, are mostly represented with storage stages and connecting lines. It is important to point out that it is often useful to represent an encoder shift register, particularly a recursive encoder, in a slightly different way. Some authors use blocks labeled with the letter D or T to denote time delays (typically 1-bit delays). The junctions outside the blocks, carrying voltage or logic levels, represent the storage in the encoder between clock times. The two formats—storage blocks versus delay blocks—do not change the characteristics or the operation of the underlying process in any way. For some finite-state machines, with many recursive connections, it may be somewhat easier to track the signal when the delay-block format is used. Problems 8.23 and 8.24 employ such encoders in Figures P8.2 and P8.3, respectively. For the storage-stage format, the current state of a machine is described by the contents of the rightmost $K - 1$ stages. For the delay-block format, the current state is similarly described by the logic levels at the outputs of the rightmost $K - 1$ delay blocks. For both formats, the relationship between memory v and constraint length K is the same, that is $v = K - 1$. Thus, in Figure P8.2, three delay blocks means that $v = 3$ and $K = 4$. Similarly, in Figure P8.3, two delay blocks means that $v = 2$ and $K = 3$.

8.5 CONCLUSION

In this chapter, we examined Reed–Solomon (R–S) codes, an important class of nonbinary block codes, particularly useful for correcting burst errors. Because coding efficiency increases with code length, R–S codes have a special attraction. They can be configured with long block lengths (in bits) with less decoding time than other codes of similar lengths. That is because the decoder logic works with symbol-based, not bit-based, arithmetic. Hence, for 8-bit symbols, the arithmetic operations would all be at the byte level. This increases the complexity of the logic, compared with binary codes of the same length, but it also increases the throughput.

We next described a technique called interleaving, which allows the popular block and convolutional coding schemes to be used over channels that exhibit bursty noise or periodic fading without suffering degradation. We used the CD digital audio system as an example of how both R–S coding and interleaving play an important role in ameliorating the effects of burst noise.

We described concatenated codes and the concept of turbo coding, whose basic configuration depends on the concatenation of two or more component codes. Basic statistical measures, such as a posteriori probability and likelihood also were reviewed. We then used these measures for describing the error performance of a soft-input/soft-output decoder. We showed how performance is improved when soft outputs from concatenated decoders are used in an iterative decoding process. We then proceeded to apply these concepts to the parallel concatenation of recursive systematic convolutional (RSC) codes, and we explained why such codes are the preferred building blocks in turbo codes. A feedback decoder was described in general ways, and its remarkable performance was presented. We next developed the mathematics of a maximum a posteriori (MAP) decoder, and used a numerical example (traversing a trellis diagram in two directions) that resulted in soft-decision outputs.

APPENDIX 8A THE SUM OF LOG-LIKELIHOOD RATIOS

Following are the algebraic details yielding the results shown in Equation (8.72). rewritten below:

$$L(d_1) \boxplus L(d_2) \triangleq L(d_1 \oplus d_2) = \log_e \left(\frac{e^{L(d_1)} + e^{L(d_2)}}{1 + e^{L(d_1)} e^{L(d_1)}} \right) \tag{8A.1}$$

We start with a likelihood ratio of the APP that a data bit equals +1 compared to the APP that it equals –1. Since the logarithm of this likelihood ratio, denoted $L(d)$, has been conveniently taken to the base e, it can be expressed as

$$L(d) = \log_e \left[\frac{P(d = +1)}{P(d = -1)} \right] = \log_e \left[\frac{P(d = +1)}{1 - P(d = +1)} \right] \tag{8A.2}$$

so that

$$e^{L(d)} = \left[\frac{P(d = +1)}{1 - P(d = +1)} \right] \tag{8A.3}$$

Solving for $P(d + 1)$, we obtain

$$e^{L(d)} - e^{L(d)} \times P(d = +1) = P(d = +1) \tag{8A.4}$$

$$e^{L(d)} = P(d = +1) \times [1 + e^{L(d)}] \tag{8A.5}$$

and

$$P(d = +1) = \frac{e^{L(d)}}{1 + e^{L(d)}} \qquad 8A.6)$$

Observe from Equation 8A.6 that

$$P(d = -1) = 1 - P(d = +1) = 1 - \frac{e^{L(d)}}{1 + e^{L(d)}} = \frac{1}{1 + e^{L(d)}} \tag{8A.7}$$

Let d_1 and d_2 be two statistically independent data bits taking on voltage values of +1 and –1 corresponding to logic levels 1 and 0 respectively.

When formatted in this way, the modulo-2 summation of d_1 and d_2 yields –1 whenever d_1 and d_2 have identical values (both +1 or both –1), and the summation yields +1 whenever d_1 and d_2 have different values. Then

$$\begin{aligned} L(d_1 \oplus d_2) &= \log_e \left[\frac{P(d_1 \oplus d_2 = 1)}{P(d_1 \oplus d_2 = -1)} \right] \\ &= \log_e \left[\frac{P(d_1 = +1) \times P(d_2 = -1) + [1 - P(d_1 = +1)][1 - P(d_2 = -1)]}{P(d_1 = +1) \times P(d_2 = +1) + [1 - P(d_1 = +1)][1 - P(d_2 = +1)]} \right] \end{aligned} \tag{8A.8}$$

Using Equations (8A.6) and 8A.7) to replace the probability terms of Equation (8A.8), we obtain

$$L(d_1 \oplus d_2) = \log_e \left[\frac{\left(\frac{e^{L(d_1)}}{1 + e^{L(d_1)}}\right)\left(\frac{1}{1 + e^{L(d_2)}}\right) + \left(\frac{1}{1 + e^{L(d_1)}}\right)\left(\frac{e^{L(d_2)}}{1 + e^{L(d_2)}}\right)}{\left(\frac{e^{L(d_1)}}{1 + e^{L(d_1)}}\right)\left(\frac{e^{L(d_2)}}{1 + e^{L(d_2)}}\right) + \left(\frac{1}{1 + e^{L(d_1)}}\right)\left(\frac{1}{1 + e^{L(d_2)}}\right)} \right] \tag{8A.9}$$

$$= \log_e \left[\frac{\left(\frac{e^{L(d_1)} + e^{L(d_2)}}{[1 + e^{L(d_1)}]\,[1 + e^{L(d_2)}]}\right)}{\left(\frac{e^{L(d_1)} \; e^{L(d_2)} + 1}{[1 + e^{L(d_1)}]\,[1 + e^{L(d_2)}]}\right)} \right] \tag{8A.10}$$

$$= \log_e \left[\frac{e^{L(d_1)} + e^{L(d_2)}}{1 + e^{L(d_2)} e^{L(d_2)}} \right] \tag{8A.11}$$

REFERENCES

1. Gallager, R. G., *Information Theory and Reliable Communication,* John Wiley and Sons, New York, 1968.
2. Odenwalder, J. P., *Error Control Coding Handbook,* Linkabit Corporation, San Diego, CA, July 15, 1976.
3. Berlekamp, E. R., Peile, R. E., and Pope, S. P., "The Application of Error Control to Communications," *IEEE Communications Magazine,* vol. 25, no. 4, April 1987, pp. 44–57.
4. Hagenauer, J., and Lutz, E., "Forward Error Correction Coding for Fading Compensation in Mobile Satellite Channels," *IEEE J. on Selected Areas in Comm.,* vol. SAC-5, no. 2, February 1987, pp. 215–225.
5. Blahut, R. E., *Theory and Practice of Error Control Codes,* Addison-Wesley Publishing Co., Reading, Massachusetts, 1983.
6. *Reed–Solomon Codes and Their Applications,* ed. Wicker, S. B., and Bhargava, V. K., IEEE Press, Piscataway, New Jersey, 1983.
7. Ramsey, J. L., "Realization of Optimum Interleavers, *IEEE Trans. Inform. Theory,* vol. IT-16, no. 3, May 1970, pp 338–345.
8. Forney, G. D., "Burst-Correcting Codes for the Classic Bursty Channel,"*IEEE Trans. Commun. Technol., vol. COM-19, Oct. 1971, pp. 772–781.*
9. Clark, G. C., Jr., and Cain, J. B., *Error-Correction Coding for Digital Communications,* Plenum Press, New York, 1981.
10. J. H. Yuen, et. al., "Modulation and Coding for Satellite and Space Communications," *Proc. IEEE,* vol. 78, no. 7, July 1990, pp. 1250–1265.
11. Peek, J. B. H., "Communications Aspects of the Compact Disc Digital Audio System," *IEEE Communications Magazine,* vol. 23, no. 2, February 1985, pp. 7–20.
12. Berkhout, P. J., and Eggermont, L. D. J., "Digital Audio Systems," *IEEE ASSP Magazine,* October 1985, pp. 45–67.

13. Driessen, L. M. H. E., and Vries, L. B., "Performance Calculations of the Compact Disc Error Correcting Code on Memoryless Channel," *Fourth Int'l. Conf. Video and Data Recording,* Southampton, England, April 20–23, 1982, IERE Conference Proc #54, pp. 385–395.
14. Hoeve, H., Timmermans, J., and Vries, L. B., "Error Correction in the Compact Disc System," *Philips Tech. Rev.,* vol. 40, no. 6, 1982, pp. 166–172.
15. Pohlmann, K. C., *The Compact Disc Handbook,* A-R Editions, Inc., Madison, Wisconsin, 1992.
16. Forney, G. D., Jr., *Concatenated Codes,* Cambridge, Massachusetts: M. I. T. Press, 1966.
17. Berrou, C., Glavieux, A., and Thitimajshima, P. "Near Shannon Limit Error-Correcting Coding and Decoding: Turbo Codes," *IEEE Proceedings of the Int. Conf. on Communications,* Geneva, Switzerland, May 1993 (ICC '93), pp. 1064–1070.
18. Berrou, C. and Glavieux, A. "Near Optimum Error Correcting Coding and Decoding: Turbo-Codes," *IEEE Trans. On Communications,* vol. 44, no. 10, October 1996, pp. 1261–1271.
19. Hagenauer, J. "Iterative Decoding of Binary Block and Convolutional Codes," *IEEE Trans. On Information Theory,* vol. 42, no. 2, March 1996, pp. 429–445.
20. Divsalar, D. and Pollara, F. "On the Design of Turbo Codes," *TDA Progress Report 42–123,* Jet Propulsion Laboratory, Pasadena, California, November 15, 1995, pp. 99–121.
21. Divsalar, D. and McEliece, R. J. "Effective Free Distance of Turbo Codes," *Electronic Letters,* vol. 32, no. 5, Feb. 29, 1996, pp. 445–446.
22. Dolinar, S. and Divsalar, D. "Weight distributions for Turbo Codes Using Random and Nonrandom Permutations," *TDA Progress Report 42–122,* Jet Propulsion Laboratory, Pasadena, California, August 15, 1995, pp. 56–65.
23. Divsalar, D. and Pollara, F. "Turbo Codes for Deep-Space Communications," *TDA Progress Report 42–120,* Jet Propulsion Laboratory, Pasadena, California, February 15, 1995, pp. 29–39.
24. Divsalar, D. and Pollara, F. "Multiple Turbo Codes for Deep-Space Communications," *TDA Progress Report 42–121,* Jet Propulsion Laboratory, Pasadena, California, May 15, 1995, pp. 66–77.
25. Divsalar, D. and Pollara, F. "Turbo Codes for PCS Applications," *Proc. ICC '95,* Seattle, Washington, June 18–22, 1995.
26. Bahl, L. R., Cocke, J., Jelinek, F. and Raviv, J. "Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate," *Trans. Inform. Theory,* vol. IT-20, March 1974, pp. 248–287.
27. Benedetto, S. et. al., "Soft Output Decoding Algorithm in Iterative Decoding of Turbo Codes," *TDA Progress Report 42–124,* Jet Propulsion Laboratory, Pasadena, California, February 15, 1996, pp. 63–87.
28. Benedetto, S. et. al., "A Soft-Input Soft-Output Maximum A Posteriori (MAP) Module to Decode Parallel and Serial Concatenated Codes," *TDA Progress Report 42–127,* Jet Propulsion Laboratory, Pasadena, California, November 15, 1996, pp. 63–87.
29. Benedetto, S. et. al., "A Soft-Input Soft-Output APP Module for Iterative Decoding of Concatenated Codes," *IEEE Communications Letters,* vol. 1, no. 1, January 1997, pp. 22–24.
30. Pietrobon, S., "Implementation and Performance of a Turbo/MAP Decoder," *Int'l. J. Satellite Commun.,* vol. 15, Jan–Feb 1998, pp. 23–46.

31. Robertson, P., Villebrun, E., and Hoeher, P., "A Comparison of Optimal and Sub-Optimal MAP Decoding Algorithms Operating in the Log Domain," *Proc. of ICC '95*, Seattle, Washington, June 1995, pp. 1009–1013.

PROBLEMS

8.1. Determine which if any of the following polynomials are primitive. Hint: One of the easiest way is with the use of an LFSR, similar to the one shown in Figure 8.8.

a) $1 + X^2 + X^3$
b) $1 + X + X^2 + X^3$
c) $1 + X^2 + X^4$
d) $1 + X^3 + X^4$
e) $1 + X + X^2 + X^3 + X^4$
f) $1 + X + X^5$
g) $1 + X^2 + X^5$
h) $1 + X^3 + X^5$
i) $1 + X^4 + X^5$

8.2. a) What is the symbol-error correcting capability of a (7, 3) R–S code? How many bits are there per symbol?
b) Compute the number of rows and columns in the standard array (see Section 6.6) required to represent the (7, 3) R–S code in part a).
c) Use the dimensions of the standard array in part b) to corroborate the symbol-error correcting capability found in part a).
d) Is the (7, 3) R–S code a perfect code? If not, how much residual symbol-error correcting capability does it have?

8.3. a) Define a set of elements $\{0, \alpha^0, \alpha^1, \alpha^2, \ldots, \alpha^{2^m-2}\}$ in terms of basis elements from the finite field GF (2^m), where $m = 4$.
b) For the finite field defined in part a), develop an addition table similar to Table 8.2.
c) Develop a multiplication table similar to Table 8.3.
d) Find the generator polynomial for the (31, 27) R–S code.
e) Encode the message {96 leading zeros followed by 110010001111} (rightmost bit is earliest) with the (31, 27) R–S code in systematic form. Why do you suppose the message was configured with so many leading zeros?

8.4. Use the generator polynomial for the (7, 3) R–S code to encode the message 010110111 (rightmost bit is earliest bit) in systematic form. Use polynomial division to find the parity polynomial, and show the resulting codeword in polynomial form and in binary form.

8.5. a) Use a LFSR to encode the symbols {6, 5, 1} (rightmost symbol is the earliest) with a (7, 3) R–S code in systematic form. Show the resulting codeword in binary form.
b) Verify the encoding results from part a) by evaluating the codeword polynomial at the roots of the (7, 3) R–S generator polynomial, $\mathbf{g}(X)$.

8.6. a) Suppose that the codeword found in Problem 8.5 was degraded during transmission, so that its rightmost 6 bits are inverted. Find the value of each syndrome by evaluating the flawed codeword polynomial at the roots of the generator polynomial $\mathbf{g}(X)$.
b) Verify that the same syndrome values found in part a) can be found by evaluating the error polynomial, $\mathbf{e}(X)$, at the roots of $\mathbf{g}(X)$.

8.7. **a)** Use the autoregressive model in Equation (8.40) with the flawed codeword from Problem 8.6 to find the location of each symbol error.

b) Find the value of each symbol error.

c) Use the information found in parts a) and b) to correct the flawed codeword.

8.8. The sequence 1011011000101100 is the input to a 4×4 block interleaver. What is the output sequence? The same input sequence is applied to the convolutional interleaver of Figure 8.13. What is the output sequence?

8.9. For each of the following conditions, design an interleaver for a communication system operating over a bursty noise channel at a transmission rate of 19.200 code symbols/s.

a) A contiguous noise burst typically lasts for 250 ms. The system code consists of a (127, 36) BCH code with $d_{\min} = 31$. The end-to-end delay is not to exceed 5 s.

b) A contiguous noise burst typically lasts for 20 ms. The system code consists of a rate $\frac{1}{2}$ convolutional code with a feedback decoding algorithm that corrects an average of 3 symbols in a sequence of 21 symbols. The end-to-end delay is not to exceed 160 ms.

8.10. **a)** Calculate the probability of a byte (symbol) error after decoding the data stored on a compact disc (CD) as described in Section 8.3. Assume that the probability of a channel-symbol error for the disc is 10^{-3}. Also assume that the inner and outer R–S decoders are each configured to correct all 2-symbol errors, and that the interleaving process results in channel symbol errors being uncorrelated from one another.

b) Repeat part a) for a disc that has a probability of channel-symbol error equal to 10^{-2}.

8.11. A BPSK system receives equiprobable bipolar symbols (+1 or −1) plus AWGN. Assume unity noise variance. At time k, the value of the received signal x_k is equal to 0.11.

a) Calculate the two likelihood values for this received signal.

b) What would be the maximum a posteriori decision, +1 or −1?

c) The a priori probability that the transmitted symbol was +1 is equal to 0.3. What would be the maximum a posteriori decision, +1 or −1?

d) Assuming the a priori probabilities from part c), calculate the log-likelihood ratio $L(d_k \mid x_k)$.

8.12. Consider the two-dimensional parity-check code example described in Section 8.4.3. As outlined there, the transmitted symbols are represented by the sequence d_1, d_2, d_3, d_4, p_{12}, p_{34}, p_{13}, p_{24}, resulting in a code rate of 1/2. A particular application requiring a higher data rate allows the output sequence from this code to be *punctured* by discarding every other parity bit, resulting in an overall code rate of 2/3. The transmitted output is now given by the sequence d_1, d_2, d_3, d_4, p_{12}, _, p_{13}, _ (parity bits p_{34} and p_{24} are not transmitted). The transmitted sequence is $\{d_i\}$, $\{p_{ij}\}$ = +1 −1 −1 +1 +1 +1, where i and j are location indices. The noise transforms this data plus parity sequence into the received sequence $\{x_k\}$ = 0.75, 0.05, 0.10, 0.15, 1.25, 3.0, where k is a time index. Calculate the values of the soft outputs for the data bits after two horizontal and two vertical decoding iterations. Assume unity noise variance.

8.13. Consider the parallel concatenation of two RSC component encoders as shown in Figure 8.26. An interleaver of block size 10, maps a sequence of input bits $\{d_k\}$ to bits $\{d'_k\}$ where the interleaver permutation is given by [6, 3, 8, 9, 5, 7, 1, 4, 10, 2], i.e., the 1st bit of the incoming data block is mapped to position 6, the 2nd bit is mapped to position 3 etc. The input sequence is given by (0, 1, 1, 0, 0, 1, 0, 1, 1, 0). Assume that the component encoders start in the all-zeros state and that no termination bits are added to force them back to the all-zeros state.

a) Calculate the 10-bit parity sequence $\{v_{1k}\}$.
b) Calculate the 10-bit parity sequence $\{v_{2k}\}$.
c) The switch yielding the sequence $\{v_k\}$ performs puncturing such that $\{v_k\}$ is given by the following: $v_{1k}, v_{2(k+1)}, v_{1(k+2)}, v_{2(k+3)}, \ldots$, and the code rate is $\frac{1}{2}$. Calculate the weight of the output codeword.
d) When decoding with the MAP algorithm, what changes do you think need to be made with regard to initializing the state metrics and branch metrics, if the encoders are left unterminated?

8.14. **a)** For the nonrecursive encoder shown in Figure P8.1, calculate the minimum distance of the overall code.
b) For the recursive encoder shown in Figure 8.26, calculate the minimum distance of the overall code. Assume that there is no puncturing, so that the code rate is $\frac{1}{3}$.
c) For the encoder shown in Figure 8.26, discuss the effect on the output code weight if the input to each component encoder is given by the weight-2 sequence (00 . . . 00100100 . . . 00) (Assume no puncturing).
d) Repeat part c) for the case where the weight-2 sequence is given by (00 . . . 0010100 . . . 00).

8.15. Consider that the encoder in Figure 8.25a is used as a component code within a turbo code. Its 4-state trellis structure is shown in Figure 8.25b. The code rate is $\frac{1}{2}$ and the branch labeling, u v, represents the output branch word (code bits) for that branch, where u is a data bit (systematic code) and v is a parity bit, and at each time k, a data bit and parity bit are transmitted. Signals received from a demodulator have the noise-disturbed u, v values of 1.9, 0.7 at time $k = 1$, and –0.4, 0.8 at time $k = 2$. Assume that the a priori probability for the data bit being a 1 or 0 is equally likely and that the encoder begins in the all-zeros state at starting time $k = 1$. Also assume that the noise variance is equal to 1.3. Recall that a data sequence of N bits is character-

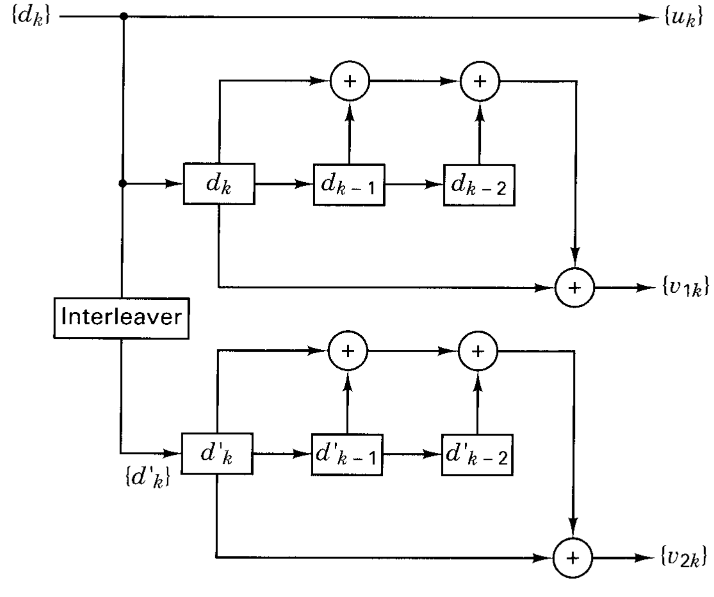


Figure P8.1 Encoder with non-recursive component codes.

ized by N transition-time intervals and $N + 1$ states (from start to finish). Thus, for this example, bits are launched at times $k = 1, 2$, and we are interested in the state metrics at times $k = 1, 2, 3$.

a) Calculate the branch metrics for times $k = 1$ and $k = 2$, that are needed for using the MAP algorithm.

b) Calculate the forward state metrics for times $k = 1, 2$, and 3.

c) The values of the reverse state metrics at times $k = 2$ and 3 are given below in Table P8.1 for each valid state. Based on the values in the table and the values calculated in parts a) and b) calculate the values for the log-likelihood ratio associated with the data bits at time $k = 1$ and $k = 2$. Use the MAP decision rule to find the most likely data bit sequence that was transmitted.

TABLE P8.1

β_k^m	$k = 2$	$k = 3$
$m = a$	4.6	2.1
$m = b$	2.4	11.5
$m = c$	5.7	3.4
$m = d$	4.3	0.9

8.16. Suppose the received sequence obtained in Problem 8.15 is in fact for a rate $\frac{2}{3}$ code which is obtained by puncturing the rate $\frac{1}{2}$ code (defined by the trellis in Figure 8.25b). The puncturing is such that only every second parity bit generated is transmitted. Therefore the four-signal sequence received represents data symbol, parity symbol, data symbol, data symbol. Calculate the branch metrics and forward state metrics for times $k = 1$ and $k = 2$ that would be needed for using the MAP algorithm.

8.17. The trellis for a four-state code used as a component code within a turbo code is shown in Figure 8.25b. The code rate is $\frac{1}{2}$ and the branch labeling, $u\ v$ represents the output, branch word (code bits) for that branch, where u is a data bit (systematic code) and v is a parity bit. A block of $N = 1024$ samples are received from a demodulator. Assume that the first signals in the block arrive at time $k = 1$, and at each time k, a noisy data bit and parity bit is received. At time $k = 1023$, the received signals have noisy u, v values of 1.3, –0.8, and at time $k = 1024$, the values are –1.4, –0.9. Assume that the a priori probability for the data bit being a 1 or 0 is equally likely and the encoder ends in a state $a = 00$ at termination time $k = 1025$. Also, assume that the noise variance is equal to 2.5.

a) Calculate the branch metrics for time $k = 1023$ and $k = 1024$.

b) Calculate the reverse state metrics for time $k = 1023$, 1024, and 1025.

c) The values of the forward state metrics at time $k = 1023$ and $k = 1024$ are given below in Table P8.2 for each valid state. Based on the values in the table and the values calculated in parts a) and b) calculate the values for the likelihood ratio associated with the data bits at time $k = 1023$ and $k = 1024$, and using the MAP decision rule, find the most likely data bit sequence that was transmitted.

TABLE P8.2

α_k^m	$k = 1023$	$k = 1024$
$m = a$	6.6	12.1
$m = b$	7.0	1.5
$m = c$	4.2	13.4
$m = d$	4.0	5.9

8.18. Given two statistically independent observations of a noisy signal x_1 and x_2, prove that the log-likelihood ratio (LLR) $L(d \mid x_1, x_2)$ can be expressed in terms of individual LLRs as

$$L(d|x_1, x_2) = L(x_1|d) + L(x_2|d) + L(d)$$

where $L(d)$ is the a priori LLR of the underlying data bit d.

8.19. **a)** Using Bayes' theorem, show the detailed steps that transform α_k^m in Equation (8.129) to Equation (8.130b). Hint: Use a simple lettering scheme as is used in Equations (8.121) and (8.122).

b) Explain how the summation over the states m' in Equation (8.130a) results in the expression seen in Equation (130b).

c) Repeat part a) to show in detail how Equation (8.133) evolves to Equation (8.135). Also explain how the summation over the states m' at the future time $k + 1$ results in the form of Equation (8.135).

8.20. Starting with Equation (8.139) for the branch metric $\delta_k^{i,m}$, show the detailed development resulting in Equation (8.140), and indicate which terms can be identified as the constant A_k in Equation (8.140). Why does the A_k term disappear in Equation (8.141a)?

8.21. The interleaver in Figure 8.27 (identical to the interleaver in the corresponding encoder) is needed to insure that the sequence out of DEC1 is time ordered in the same way as the sequence $\{y_{2k}\}$ Can this be implemented in a simpler way? What about using a deinterleaver in the lower line? Wouldn't that accomplish the same time ordering more simply? If we did that, then the two deinterleavers, just prior to the output, could be eliminated. Explain why that would not work.

8.22. In the implementation of the Viterbi decoding algorithm, an add-compare-select (ACS) processor is used. But, in performing the maximum a posteriori (MAP) algorithm in turbo decoding, there is no such concept as comparing and selecting one transition over another. Instead the MAP algorithm incorporates all of the branch and state metrics at each time interval. Explain the reason for this fundamental difference between the two algorithms.

8.23. Figure P8.2 illustrates a recursive systematic convolutional (RSC) rate $\frac{1}{2}$, $K = 4$, encoder. Note that the figure uses the format of 1-bit delay blocks rather than storage stages (see Section 8.4.7.4). Thus, the current state of this circuit can be described by

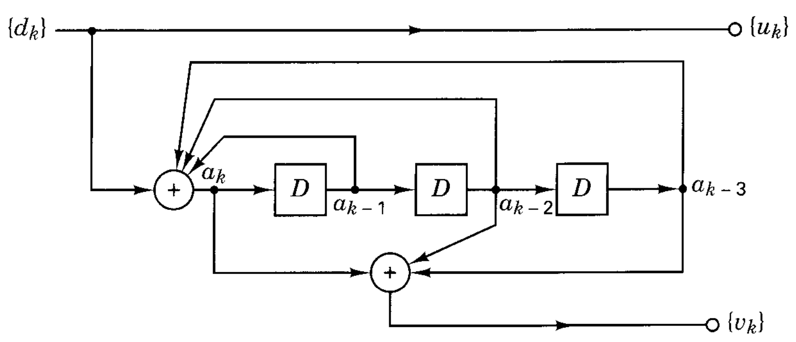


Figure P8.2 Recursive systematic convolutional (RSC) encoder, rate ½, $K = 4$.

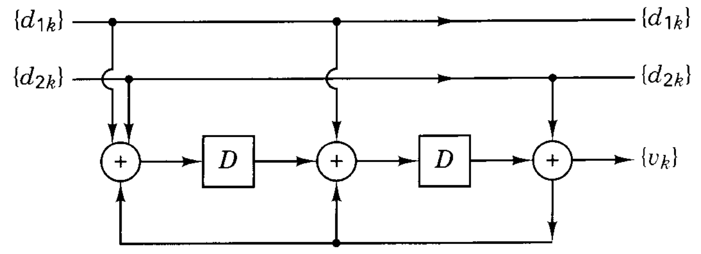


Figure P8.3 Recursive systematic convolutional (RSC) encoder, rate ⅔, $K = 3$.

the signal levels at points a_{k-1}, a_{k-2}, and a_{k-3}, similar to the way a state is described in the format using storage stages. Form a table, similar to Table 8.5 that describes all possible transitions in this circuit, and use the table to draw a trellis section.

8.24. Figure P8.3 illustrates a recursive systematic convolutional (RSC) rate $\frac{2}{3}$, $K = 3$, encoder. Note that the figure uses the format of 1-bit delay blocks rather than storage stages (see Section 8.4.7.4). Form a table, similar to Table 8.5 that describes all possible transitions in this circuit, and use the table to draw a trellis section. Use a table, similar to Table 8.6 to find the output codeword for the message sequence 1100110011. At each clock time, data bits enter the circuit in pairs $\{d_{1k}, d_{2k}\}$, and each output branch word $\{d_{1k}, d_{2k}, v_k\}$ is made up of that pair of data bits plus one parity bit, v_k.

8.25. Consider a turbo code consisting of two, four state convolutional codes as the component codes, both of which are described by the same trellis as shown in Figure 8.25b. The code rate is equal to $\frac{1}{2}$ and the block length is equal to 12. The second encoder is left unterminated. The branch metrics, forward state metrics, and reverse state metrics for the data bits associated with the terminated encoder are described by the matrices that follow. The received 12-signal vector is of the form data signal, parity signal, data signal, parity signal, and so forth, and has the following values:

$$1.2 \quad 1.3 \quad -1.2 \quad 0.6 \quad -0.4 \quad 1.9 \quad -0.7 \quad -1.9 \quad -2.2 \quad 0.2 \quad -0.1 \quad 0.6$$

Branch $\delta_k^{i,m}$ *matrix*

$$\delta_k^{i,m} = \begin{bmatrix} \delta_1^{0,a} & \delta_2^{0,a} & \cdots & \delta_6^{0,a} \\ \delta_1^{1,a} & \ddots & \cdots & \delta_6^{1,a} \\ \delta_1^{0,b} & \ddots & \ddots & \vdots \\ \delta_1^{1,b} & \ddots & \ddots & \vdots \\ \delta_1^{0,c} & \ddots & \ddots & \vdots \\ \delta_1^{1,c} & \ddots & \ddots & \vdots \\ \delta_1^{0,d} & \ddots & \ddots & \vdots \\ \delta_1^{1,d} & \cdots & \cdots & \delta_6^{1,d} \end{bmatrix} = \begin{bmatrix} 1.00 & 1.00 & 1.00 & 1.00 & 1.00 & 1.00 \\ 3.49 & 0.74 & 2.12 & 0.27 & 0.37 & 1.28 \\ 1.00 & 1.00 & 1.00 & 1.00 & 1.00 & 1.00 \\ 3.49 & 0.74 & 2.12 & 0.27 & 0.37 & 1.28 \\ 1.92 & 1.35 & 2.59 & 0.39 & 1.11 & 1.35 \\ 1.82 & 0.55 & 0.82 & 0.70 & 0.33 & 0.95 \\ 1.92 & 1.35 & 2.59 & 0.39 & 1.11 & 1.35 \\ 1.82 & 0.55 & 0.82 & 0.70 & 0.33 & 0.95 \end{bmatrix}$$

Alpha (α_k^m) *matrix*

$$\alpha_k^m = \begin{bmatrix} \alpha_1^a & \alpha_2^a & \cdots & \alpha_7^a \\ \alpha_1^b & \ddots & \cdots & \alpha_7^b \\ \alpha_1^c & \cdots & \ddots & \vdots \\ \alpha_1^d & \cdots & \cdots & \alpha_7^d \end{bmatrix} = \begin{bmatrix} 1.00 & 1.00 & 1.00 & 5.05 & 8.54 & 10.41 & 24.45 \\ 0.00 & 0.00 & 1.92 & 12.79 & 5.07 & 10.93 & 31.48 \\ 0.00 & 3.49 & 0.74 & 4.03 & 14.16 & 8.22 & 24.30 \\ 0.00 & 0.00 & 4.71 & 5.77 & 5.63 & 17.53 & 27.76 \end{bmatrix}$$

Beta (β_k^m) *matrix*

$$\beta_k^m = \begin{bmatrix} \beta_1^a & \beta_2^a & \cdots & \beta_7^a \\ \beta_1^b & \ddots & \cdots & \beta_7^b \\ \beta_1^c & \cdots & \ddots & \vdots \\ \beta_1^d & \cdots & \cdots & \beta_7^d \end{bmatrix} = \begin{bmatrix} 24.45 & 5.44 & 2.83 & 1.12 & 1.00 & 1.00 & 1.00 \\ 24.43 & 5.62 & 3.17 & 0.70 & 0.37 & 1.28 & 0.00 \\ 21.32 & 5.45 & 3.53 & 0.81 & 0.43 & 0.00 & 0.00 \\ 21.31 & 5.79 & 2.75 & 1.14 & 1.42 & 0.00 & 0.00 \end{bmatrix}$$

Calculate the log-likelihood ratio for each of the six data bits $\{d_k\}$, and by using the MAP decision rule, find the most likely data-bit sequence that was transmitted.

QUESTIONS

8.1. Explain why R–S codes perform so well in a *bursty-noise* environment. (See Section 8.1.2.)

8.2. Explain why the curves in Figure 8.6 of the text show error-performance *degradation* at low values of code rate. (See Section 8.1.3.)

8.3. Considering all the ways that there are to determine whether a polynomial is *primitive,* the method involving a linear feedback shift register (LFSR) is one of the simplest. Explain the procedure. (See Example 8.2.)

8.4. Explain why a *syndrome* can be calculated by evaluating the received polynomial at each of the roots of the code's generator polynomial. (See Section 8.1.6.1.)

8.5. What key transformation does an interleaver/deinterleaver system perform on *bursty noise?* (See Section 8.2.1.)

8.6. Why is the *Shannon limit* of –1.6 dB not a useful goal in the design of real systems? (See Section 8.4.5.2.)

8.7. What are the consequences of the *Viterbi decoding algorithm* not yielding *a posteriori* probabilities? (See Section 8.4.6.)

8.8. What is a more descriptive name for the Viterbi algorithm? (See Section 8.4.6.)

8.9. Describe the similarities and differences between implementing a Viterbi decoding algorithm and implementing a *maximum a posteriori* (MAP) decoding algorithm? (See Section 8.4.6.)

EXERCISES

Using the Companion CD, run the exercises associated with Chapter 8.