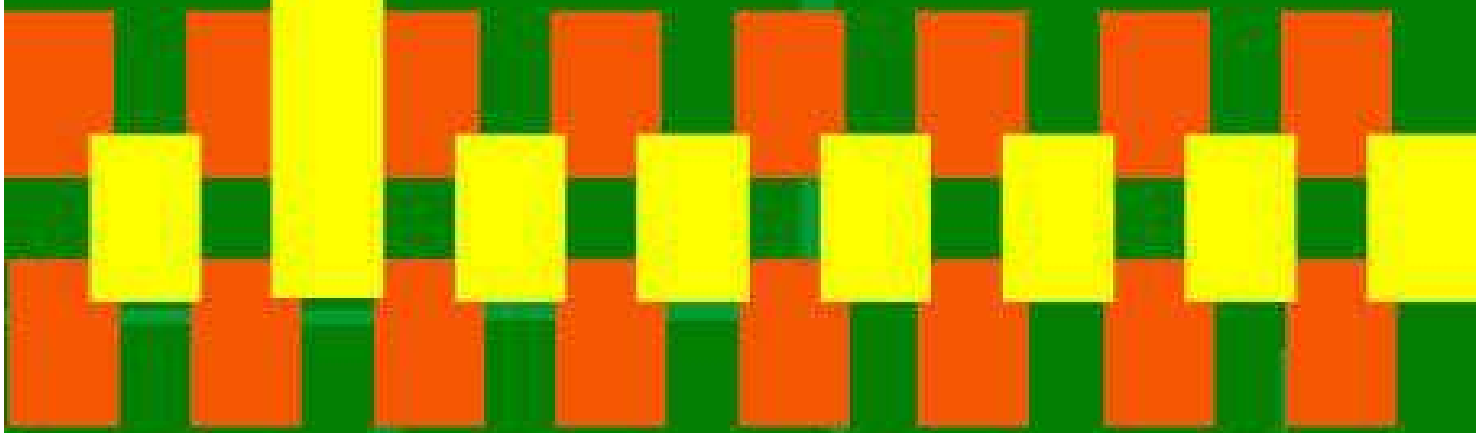
Algebraic Codes for Data Transmission

Richard E. Blahut



CAMBRIDGE

more information - www.cambridge.org/9780521553742

Algebraic Codes for Data Transmission

The need to transmit and store massive amount of data reliably and without error is a vital part of modern communications systems. Error-correcting codes play a fundamental role in minimizing data corruption caused by defects such as noise, interference, cross talk, and packet loss. This book provides an accessible introduction to the basic elements of algebraic codes, and discusses their use in a variety of applications. The author describes a range of important coding techniques, including Reed–Solomon codes, BCH codes, trellis codes, and turbocodes. Throughout the book, mathematical theory is illustrated by reference to many practical examples. The book is aimed at graduate students of electrical and computer engineering, and at practising engineers whose work involves communications or signal processing.

Professor Richard E. Blahut is Head of the Department of Electrical and Computer Engineering at the University of Illinois, Urbana Champaign. He is a Fellow of the Institute of Electrical and Electronics Engineers and the recipient of many awards including the IEEE Alexander Graham Bell Medal (1998), the Tau Beta Pi Daniel C. Drucker Eminent Faculty Award, and the IEEE Millennium Medal. He was named a Fellow of the IBM Corporation in 1980 (where he worked for over 30 years) and was elected to the National Academy of Engineering in 1990.

7 Algorithms Based on the Fourier Transform

An error-control code must be judged not only by its rate and minimum distance, but also by whether a decoder can be built for it economically. Usually, there are many ways to decode a given code. A designer can choose among several decoding algorithms and their variations, and must be familiar with all of them so that the best one for a particular application can be chosen. The choice will depend not only on the code parameters, such as blocklength and minimum distance, but also on how the implementation is to be divided between hardware and software, on the required decoding speed, and even on the economics of available circuit components.

In this chapter we shall broaden our collection of decoding techniques by working in the frequency domain. Included here are techniques for decoding in the presence of both erasures and errors, and techniques for decoding beyond the designed distance of a code.

7.1 Spectral estimation in a finite field

In Section 6.6 we studied the task of decoding BCH codes, including Reed–Solomon codes, up to the designed distance. This task will now be reinterpreted from the point of view of the Fourier transform. Using the terminology of the frequency domain, we shall develop alternative decoding procedures. These procedures will take the form of spectral estimation in a finite field.

A senseword $\boldsymbol{v} = \boldsymbol{c} + \boldsymbol{e}$, with components $v_i = c_i + e_i$ for $i = 0, \ldots, n-1$, is the sum of a codeword $\boldsymbol{c}$ and an error word $\boldsymbol{e}$. The decoder must process the senseword so that the error word $\boldsymbol{e}$ is removed; the data is then recovered from the codeword. By construction of a BCH code of designed distance $2t+1$, there are $2t$ consecutive spectral components of $\boldsymbol{c}$ equal to zero:

$$C_j = 0 \quad j = j_0, \ldots, j_0 + 2t - 1$$

where $\{\omega^{j_0}, \omega^{j_0+1}, \ldots, \omega^{j_0+2t-1}\}$ is the defining set of the BCH code.

The Fourier transform of the senseword $\boldsymbol{v}$ has components $V_j = C_j + E_j$ for $j = 0, \ldots, n-1$. The (frequency-domain) syndromes are defined as those $2t$ components

of this spectrum, from j_0 to $j_0 + 2t - 1$, corresponding to those spectral components where C_j is zero. It is convenient to index the syndromes starting with index one. Define

$$S_j = V_{j+j_0-1} = E_{j+j_0-1} \quad j = 1, \ldots, 2t.$$

The block of $2t$ syndromes gives us a window through which we can look at $2t$ of the n components of the error spectrum $\boldsymbol{E}$. But the BCH bound suggests that if the error pattern has a weight of at most t, then these $2t$ components of the error spectrum are enough to uniquely determine the error pattern.

Recall the linear complexity property of the Fourier transform, which says that if the error pattern $\boldsymbol{e}$ has weight ν, then the error spectrum satisfies the recursion

$$E_k = -\sum_{j=1}^{\nu} \Lambda_j E_{((k-j))} \quad k = 0, \ldots, n-1$$

for appropriate connection weights $\Lambda_1, \ldots, \Lambda_\nu$. To redevelop this equation here, suppose that there are ν errors in the senseword $\boldsymbol{v} = (v_0, \ldots, v_{n-1})$ at locations ω^{i_ℓ} for $\ell = 1, \ldots, \nu$. The error-locator polynomial

$$\Lambda(x) = \prod_{\ell=1}^{\nu} (1 - x\omega^{i_\ell})$$

defines a vector $\boldsymbol{\Lambda} = (\Lambda_0, \Lambda_1, \ldots, \Lambda_\nu, 0, \ldots, 0)$ with inverse Fourier transform $\boldsymbol{\lambda} = (\lambda_0, \lambda_1, \ldots, \lambda_{n-1})$ given by

$$\lambda_i = \frac{1}{n} \sum_{j=0}^{n-1} \Lambda_j \omega^{-ij} = \frac{1}{n} \Lambda(\omega^{-i}).$$

Clearly, $\Lambda(\omega^{-i})$ equals zero if and only if i is an error location. Thus $\lambda_i = 0$ whenever $e_i \neq 0$. Therefore $\lambda_i e_i = 0$ for all i. By the convolution theorem, the Fourier transform of this equation becomes a cyclic convolution in the frequency domain set equal to zero:

$$\sum_{j=0}^{\nu} \Lambda_j E_{((k-j))} = 0 \quad k = 0, \ldots, n-1$$

where the upper limit is ν because $\Lambda(x)$ is a polynomial of degree ν. Because Λ_0 equals one, this reduces to the recursion mentioned above. Furthermore, if $\nu \leq t$, then $\Lambda_j = 0$ for $j = \nu + 1, \ldots, t$, so the recursion then can be rewritten in the form:

$$E_k = -\sum_{j=1}^{t} \Lambda_j E_{((k-j))} \quad k = 0, \ldots, n-1$$

which may be preferred because the upper summation index t is now fixed. This system of n equations involves $2t$ known components of $\boldsymbol{E}$, given by the $2t$ syndromes, and $n - t$ unknowns, of which t are unknown coefficients of $\Lambda(x)$ and $n - 2t$ are unknown components of $\boldsymbol{E}$. Of the n equations, there are t equations that involve only the known

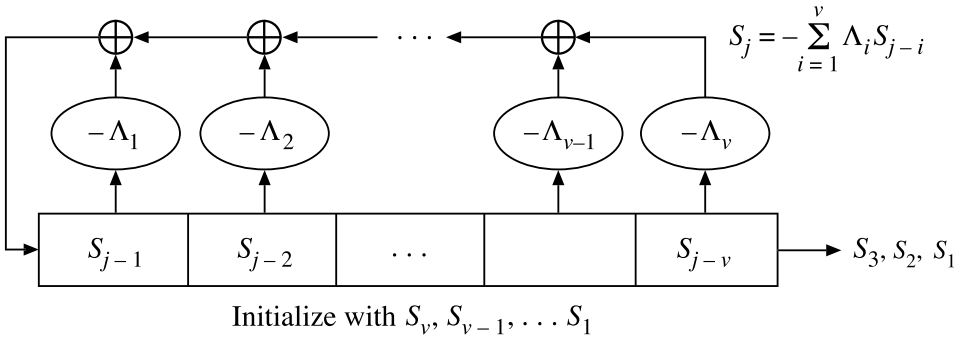


Figure 7.1. Error-locator polynomial as a shift-register circuit

components of $\boldsymbol{E}$, as expressed by the syndromes $S_j = E_{j+j_0-1}$ for $j = 1, \ldots, 2t$, and the t unknown components of $\boldsymbol{\Lambda}$, which are the coefficients of the unknown polynomial $\Lambda(x)$. That is, the t equations

$$S_k = -\sum_{j=1}^{t} \Lambda_j S_{((k-j))} \quad k = t+1, \ldots, 2t$$

involve only the syndromes, which are known, and the t unknown components of $\boldsymbol{\Lambda}$. This linear recursion, written in the form

$$S_k = -\sum_{j=1}^{v} \Lambda_j S_{k-j} \quad k = v+1, \ldots, 2t,$$

must be solved for $\boldsymbol{\Lambda}$ using the smallest possible value of v. The recursion, denoted $(\boldsymbol{\Lambda}, v)$ or $(\Lambda(x), v)$, can be depicted as a *linear-feedback shift register*, shown in Figure 7.1. This system of equations is always solvable for $\boldsymbol{\Lambda}$ and v. One way to solve it is by the Peterson algorithm, which was given in Section 6.6.

After $\boldsymbol{\Lambda}$ is computed, the remaining $n - 2t$ components of $\boldsymbol{S}$ can be obtained from $\Lambda(x)$ and the known components of $\boldsymbol{S}$ by the Gorenstein–Zierler algorithm. Alternatively, a procedure known as *recursive extension* can be chosen, using the linear recursion first to find S_{2t+1} from the known components of $\boldsymbol{S}$ and $\boldsymbol{\Lambda}$, then to find S_{2t+2}, and so on. Recursive extension can be described as the operation of the linear-feedback shift register with connection weights given by the components of $\boldsymbol{\Lambda}$ and initialized with $S_{t+1}, \ldots, S_{2t}$. In this way, S_j is computed for all j. Then E_j equals S_{j-j_0+1} and

$$C_j = V_j - E_j,$$

from which the codeword is easily recovered.

The decoder can be used with an encoder in the time domain, as shown in Figure 7.2, or an encoder in the frequency domain, as shown in Figure 7.3. If a time-domain encoder is used, then the inverse Fourier transform of the corrected codeword spectrum $\boldsymbol{C}$ must be computed to obtain the time-domain codeword $\boldsymbol{c}$, from which the data is recovered. If, instead, the encoder uses the data symbols in the frequency domain to specify the values of the spectrum, then the corrected spectrum gives the data symbols directly. That decoder does not compute an inverse Fourier transform.

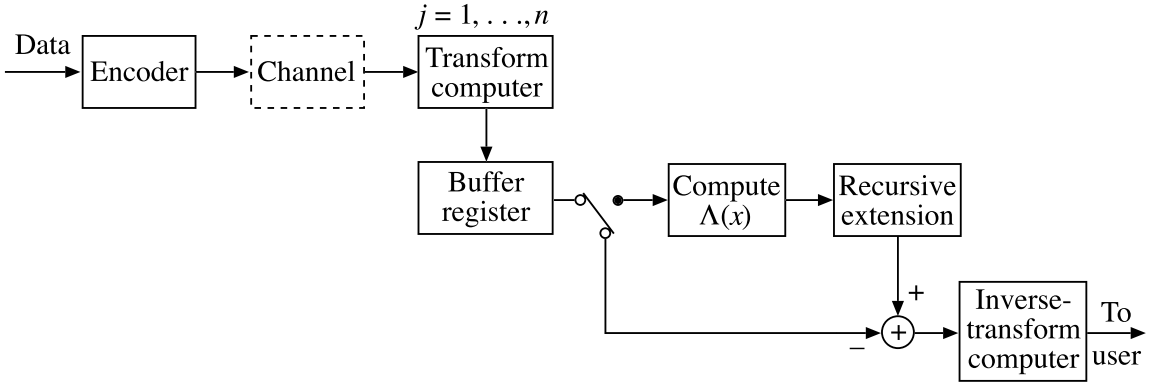


Figure 7.2. An encoder/decoder for BCH codes

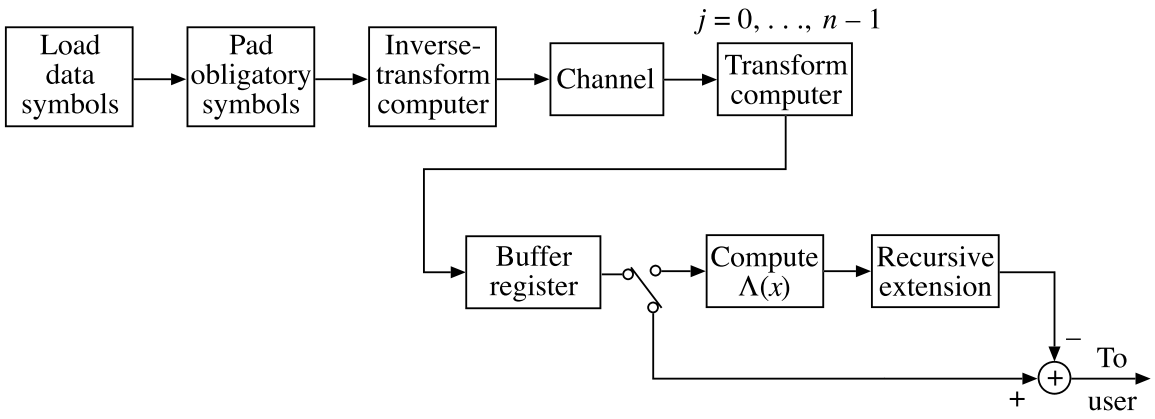


Figure 7.3. Frequency-domain encoder/decoder for BCH codes

Our main task in this chapter is to find efficient algorithms to compute a minimum-length linear recursion $(\Lambda(x), \nu)$ that produces the syndromes cyclically. This task can be reposed in the language of polynomials, the formulation of which we will find useful later. Define the (frequency-domain) *syndrome polynomial* as

$$S(x) = \sum_{j=1}^{2t} S_j x^j.$$

Let $\Lambda(x)$ be any polynomial of degree $\nu \leq t$. Define the *discrepancy polynomial* as $\Delta(x) = \Lambda(x)S(x)$. The discrepancy polynomial can always be decomposed as

$$\Delta(x) = \Lambda(x)S(x) = \Gamma(x) + x^{\nu+1} O(x) + x^{2t+1} \Theta(x)$$

where $\deg \Gamma(x) \leq \nu$ and $\deg x^{\nu+1} O(x) \leq 2t$. In this polynomial formulation, the coefficients of $x^{\nu+1} O(x)$ have the form $\sum_{j=0}^{\nu} \Lambda_j S_{k-j}$ for $k = \nu + 1, \ldots, 2t$, so $O(x) = 0$ if and only if the linear recursion $(\Lambda(x), \nu)$ produces the sequence $S_1, \ldots, S_{2t}$. Thus the decomposition

$$\Lambda(x)S(x) = \Gamma(x) + x^{2t+1} \Theta(x) \quad \deg \Gamma(x) \leq \nu$$

is just a way of saying that the discrepancy polynomial $\Delta(x) = \Lambda(x)S(x)$ has zero coefficients for $k = \nu + 1, \ldots, 2t$ if and only if the linear recursion $(\Lambda(x), \nu)$ produces

the sequence $S_1, \ldots, S_{2t}$. Any algorithm for solving this polynomial equation is an algorithm for finding the locator polynomial $\Lambda(x)$.

7.2 Synthesis of linear recursions

To decode a BCH code or a Reed–Solomon code, as discussed in Section 7.1, one solves the following problem. Find the connection weights $(\Lambda_1, \ldots, \Lambda_\nu)$ for the smallest $\nu \leq t$ for which the system of equations

$$S_k = -\sum_{j=1}^{\nu} \Lambda_j S_{k-j} \quad k = \nu + 1, \ldots, 2t$$

has a solution. The Peterson algorithm solves for the linear recursion $\mathbf{\Lambda}$ by inverting a ν by ν matrix equation for various values of ν. For small ν, matrix inversion is not unreasonable; the number of multiplications necessary to invert a ν by ν matrix is proportional to ν^3. However, when ν is large, one should use a more efficient method of solution. The structure of the problem is used to find a method that is conceptually more intricate than direct matrix inversion, but computationally much simpler.

The *Berlekamp–Massey algorithm* solves the following modified problem. Find the connection weights $(\Lambda_1, \ldots, \Lambda_\nu)$ for the smallest $\nu \leq 2t$ for which the system of equations

$$S_k = -\sum_{j=1}^{\nu} \Lambda_j S_{k-j} \quad k = \nu + 1, \ldots, 2t$$

has a solution. In the modified problem, values of the integer ν as large as $2t$ are allowed, so for any arbitrary sequence $S_1, \ldots, S_{2t}$, the problem must have a solution. In particular, the value $\nu = 2t$, together with arbitrary values for $\Lambda_1, \ldots, \Lambda_{2t}$, will always satisfy the system of equations.

The modified problem is not the same as the original problem, which for an arbitrary sequence $S_1, \ldots, S_{2t}$ need not have a solution satisfying the constraint $\nu \leq t$. It is sufficient to solve the modified problem, however, because if the original problem has a solution, it is the same as the solution to the modified problem.

The modified problem can be viewed as the task of designing the shortest linear recursion $(\Lambda(x), \nu)$ that will produce the known sequence of syndromes. If, for this connection polynomial $\Lambda(x)$, either $\deg \Lambda(x) > t$ or $\Lambda_\nu = 0$, then $\Lambda(x)$ is not a legitimate error-locator polynomial. In this case, the syndromes cannot be explained by a correctable error pattern – a pattern of more than t errors has been detected.

The procedure that we will develop applies in any field and does not presume any special properties for the sequence $S_1, S_2, \ldots, S_{2t}$. The procedure is iterative. For each r, starting with $r = 1$, we will design a minimum-length linear recursion $(\Lambda^{(r)}(x), L_r)$ to produce $S_1, \ldots, S_r$. It need not be unique. At the start of iteration r, we will have

already constructed a list of linear recursions

$$\begin{aligned}&\big(\Lambda^{(1)}(x), L_1\big),\\ &\big(\Lambda^{(2)}(x), L_2\big),\\ &\qquad\vdots\\ &\big(\Lambda^{(r-1)}(x), L_{r-1}\big).\end{aligned}$$

The mth entry on the list produces the truncated sequence $S_1, \ldots, S_m$, and the last entry on the list produces the truncated sequence $S_1, \ldots, S_{r-1}$.

The main trick of the Berlekamp–Massey algorithm is to use these earlier iterates to compute a new minimum-length linear recursion $(\Lambda^{(r)}(x), L_r)$ that will produce the sequence $S_1, \ldots, S_{r-1}, S_r$. This is done by using as a trial candidate the linear recursion that produces the shorter sequence $S_1, \ldots, S_{r-2}, S_{r-1}$, and then possibly increasing its length and adjusting the connection weights.

At iteration r, compute the next output of the $(r-1)$th linear recursion:

$$\hat{S}_r = -\sum_{j=1}^{L_{r-1}} \Lambda_j^{(r-1)} S_{r-j}.$$

Subtract $\hat{S}_r$ from the desired output S_r to get a quantity Δ_r, known as the rth *discrepancy*:

$$\Delta_r = S_r - \hat{S}_r = S_r + \sum_{j=1}^{L_{r-1}} \Lambda_j^{(r-1)} S_{r-j}.$$

Equivalently,

$$\Delta_r = \sum_{j=0}^{L_{r-1}} \Lambda_j^{(r-1)} S_{r-j},$$

which is the rth coefficient of the polynomial product $\Lambda^{(r-1)}(x)S(x)$. If Δ_r is zero, then set $(\Lambda^{(r)}(x), L_r) = (\Lambda^{(r-1)}(x), L_{r-1})$, and the rth iteration is complete. Otherwise, the connection polynomial is modified as follows:

$$\Lambda^{(r)}(x) = \Lambda^{(r-1)}(x) + Ax^{\ell}\Lambda^{(m-1)}(x)$$

where A is an appropriate field element, ℓ is an integer, and $\Lambda^{(m-1)}(x)$ is one of the earlier connection polynomials. With the new connection polynomial, the discrepancy is

$$\begin{aligned}\Delta_r' &= \sum_{j=0}^{L_r} \Lambda_j^{(r)} S_{r-j}\\ &= \sum_{j=0}^{L_r} \Lambda_j^{(r-1)} S_{r-j} + A\sum_{j=0}^{L_r} \Lambda_j^{(m-1)} S_{r-j-\ell}\end{aligned}$$

where L_r has not been specified, but it is at least as large as $\deg \Lambda^{(r-1)}(x)$ and at least as large as $\deg x^{\ell}\Lambda^{(m-1)}(x)$.

We are now ready to specify m, ℓ, and A. Choose an m smaller than r for which $\Delta_m \neq 0$, choose $\ell = r - m$, and choose $A = -\Delta_m^{-1}\Delta_r$. Then

$$\Delta_r' = \Delta_r - \frac{\Delta_r}{\Delta_m}\Delta_m = 0,$$

and also $\Delta_k = 0$ for $k < r$. Thus the new recursion $(\Lambda^{(r)}(x), L_r)$ produces the sequence $S_1, \ldots, S_{r-1}, S_r$.

We do not want just any such recursion, however; we want one of the shortest length. We still have not specified which of the m satisfying $\Delta_m \neq 0$ should be chosen. We now specify that m is the most recent iteration at which $L_m > L_{m-1}$, which ensures that we will get a shortest-length linear recursion at every iteration, but the proof of this statement will take some time to develop.

The computation is illustrated by Figure 7.4, with the recursions depicted as shift registers. The shift registers at the start of iterations m and r are shown in Figure 7.4(a). Iteration m is chosen such that, at iteration m, the shift register $(\Lambda^{(m-1)}(x), L_{m-1})$ failed to produce syndrome S_m, and $L_m > L_{m-1}$. Figure 7.4(b) shows the shift register $(\Lambda^{(m-1)}(x), L_{m-1})$ made into an auxiliary shift register by lengthening it, positioning it, and scaling its output so that it can compensate for the failure of $(\Lambda^{(r-1)}(x), L_{r-1})$ to produce S_r. The auxiliary shift register has an extra tap with a weight of one due to the term $\Lambda_0^{(m-1)}$. At the rth symbol time, a nonzero valve Δ_m comes from the auxiliary shift register. The coefficient A is selected to modify this nonzero value so that it can be added to the rth feedback of the main shift register, thereby producing the required syndrome S_r. Figure 7.4(c) shows the two shift registers merged into one shift register, which does not change the behavior. This gives the new shift register $(\Lambda^{(r)}(x), L_r)$. Sometimes $L_r = L_{r-1}$, and sometimes $L_r > L_{r-1}$. When the latter happens, we replace m with r for use in future iterations.

The following theorem gives the precise procedure for producing the shortest-length linear recursion with the desired property. The theorem uses the scratch polynomial $B(x)$ in place of $\Delta_m^{-1}\Lambda^{(m)}(x)$. The proof of the theorem occupies the remainder of the section.

Algorithm 7.2.1 (Berlekamp–Massey Algorithm). *In any field, let $S_1, \ldots, S_{2t}$ be given. With initial conditions $\Lambda^{(0)}(x) = 1$, $B^{(0)}(x) = 1$, and $L_0 = 0$, let the following set of equations for $r = 1, \ldots, 2t$ be used iteratively to compute $\Lambda^{(2t)}(x)$:*

$$\Delta_r = \sum_{j=0}^{n-1} \Lambda_j^{(r-1)} S_{r-j}$$

$$L_r = \delta_r(r - L_{r-1}) + \overline{\delta}_r L_{r-1}$$

$$\begin{bmatrix} \Lambda^{(r)}(x) \\ B^{(r)}(x) \end{bmatrix} = \begin{bmatrix} 1 & -\Delta_r x \\ \Delta_r^{-1}\delta_r & \overline{\delta}_r x \end{bmatrix} \begin{bmatrix} \Lambda^{(r-1)}(x) \\ B^{(r-1)}(x) \end{bmatrix}$$

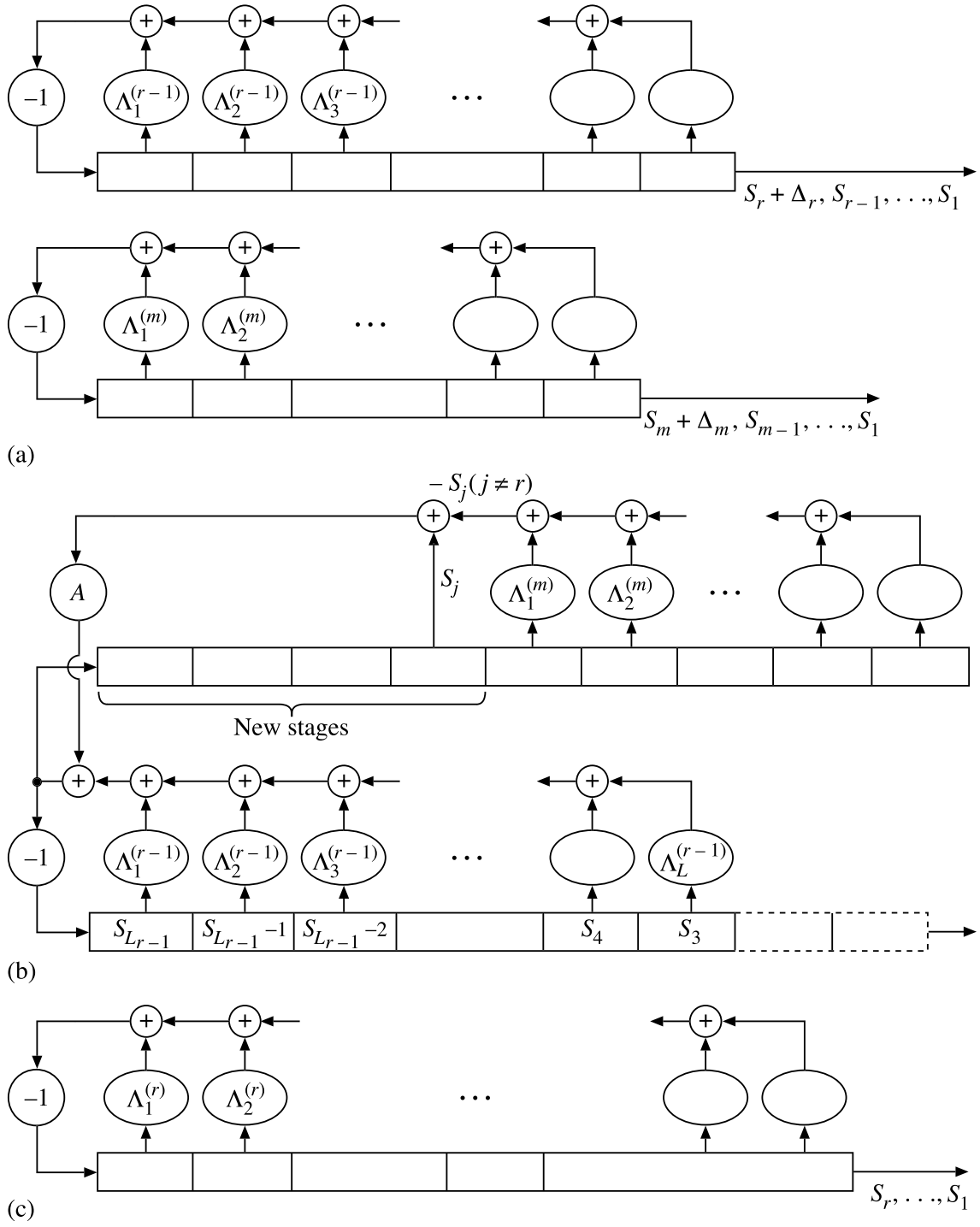


Figure 7.4. Berlekamp–Massey construction

where $\delta_r = (1 - \overline{\delta}_r) = 1$ *if both* $\Delta_r \neq 0$ *and* $2L_{r-1} \leq r - 1$, *and otherwise* $\delta_r = 0$. *Then* $(\Lambda^{(2t)}(x), L_{2t})$ *is a linear recursion of shortest length that produces* $S_1, \ldots, S_{2t}$.

Figure 7.5 restates the Berlekamp–Massey algorithm graphically in the form of a flow chart. In each of $2t$ iterations, the discrepancy Δ_r is computed. If the discrepancy is nonzero, $\Lambda(x)$ is changed by adding a multiple of $x\,B(x)$, which is a translated multiple

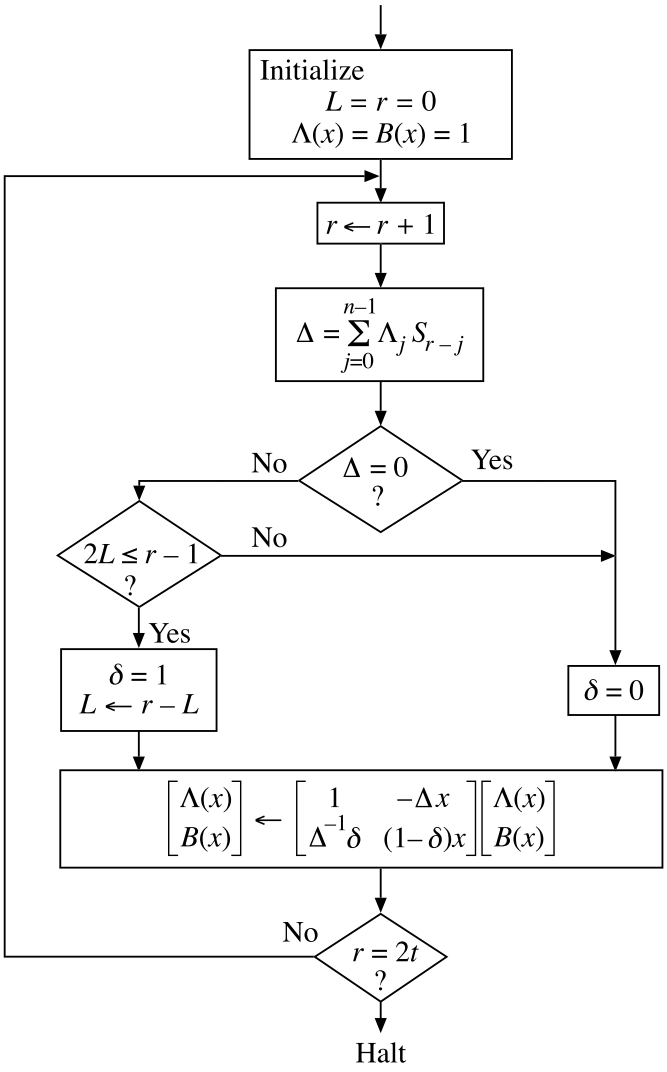


Figure 7.5. The Berlekamp–Massey algorithm

of a previous value of $\Lambda(x)$ that was stored when the length L_r was most recently changed. During each iteration when L_r is not changed, $B(x)$ is remultiplied by x so that a multiplication by x^{r-m} will be precomputed in time for the rth iteration. The decision to change L is based on the conditions $\Delta_r \neq 0$ and $2L \leq r - 1$, as required by the algorithm statement. In the computations, Δ_r may be zero, but only when δ_r is zero. Then the term $\Delta_r^{-1}\delta_r$ is defined to be zero. The matrix update requires at most $2t$ multiplications per iteration, and the calculation of Δ_r requires at most t multiplications per iteration. There are $2t$ iterations and so not more than $6t^2$ multiplications. In most instances, the algorithm is better than direct matrix inversion, which requires of the order of t^3 operations.

We shall illustrate the Berlekamp–Massey algorithm by decoding a binary (15, 5) BCH code in Section 7.3 and by decoding a (15, 9) Reed–Solomon code in Section 7.4. Both examples have a spectrum in $GF(16)$. The inner workings of the algorithm can be studied in these examples. One will notice that at the rth iteration, there may be more than one recursion of minimum length that produces the given sequence. During later

iterations, whenever possible, another of the linear recursions of this length is selected. If no linear recursion of this length produces the next required symbol, then the linear recursion must be replaced by a longer one. The test to see if lengthening is required, however, uses the next symbol only to determine that $\Delta_r \neq 0$.

The proof of Algorithm 7.2.1 is broken into lemmas. In Lemma 7.2.3, which is based on Lemma 7.2.2, we find an inequality relationship between L_r and L_{r-1}. In Lemma 7.2.4, we use Algorithm 7.2.1 for the iterative construction of a linear recursion that produces the given sequence, and we conclude that the construction gives the shortest such linear recursion because it satisfies Lemma 7.2.3.

Lemma 7.2.2. *If the sequence* $\boldsymbol{S} = (S_1, \ldots, S_r)$ *has linear complexity* L' *and the sequence* $\boldsymbol{T} = (T_1, \ldots, T_r)$ *has linear complexity* L''*, then the sequence,* $\boldsymbol{T} - \boldsymbol{S} = (T_1 - S_1, \ldots, T_r - S_r)$ *has linear complexity not larger than* $L' + L''$.

Proof: Let L be the length of a linear recursion of minimum length that produces the sequence $\boldsymbol{T} - \boldsymbol{S} = (T_1 - S_1, \ldots, T_r - S_r)$. Let $(\Lambda'(x), L')$ be a linear recursion of minimum length that produces sequence $\boldsymbol{S}$. Let $(\Lambda''(x), L'')$ be a linear recursion of minimum length that produces sequence $\boldsymbol{T}$. Clearly, the jth coefficient of the polynomial product $\Lambda'(x)S(x)$ is equal to zero for $j = L' + 1, \ldots, r$, and the jth coefficient of the polynomial product $\Lambda''(x)T(x)$ is equal to zero for $j = L'' + 1, \ldots, r$. We can express these facts as a pair of polynomial relationships

$$\Lambda'(x)S(x) = \Gamma'(x) + x^{r+1}\Theta'(x)$$
$$\Lambda''(x)T(x) = \Gamma''(x) + x^{r+1}\Theta''(x)$$

for some $\Gamma'(x)$ and $\Gamma''(x)$ that satisfy $\deg \Gamma'(x) \leq L'$ and $\deg \Gamma''(x) \leq L''$, and for some $\Theta'(x)$ and $\Theta''(x)$.

Clearly, $L \leq r$, so whenever $L' + L'' > r$, we can conclude that $L < L' + L''$, and the lemma is true. Otherwise, $L' + L'' \leq r$. In this case, the two polynomial equations above imply that

$$\Lambda'(x)\Lambda''(x)[T(x) - S(x)] = [\Lambda'(x)\Gamma''(x) - \Lambda''(x)\Gamma'(x)] + x^{r+1}[\Lambda''(x)\Theta''(x) - \Lambda''(x)\Theta'(x)]$$

which has the form

$$\Lambda(x)[T(x) - S(x)] = \Gamma(x) + x^{r+1}\Theta(x)$$

with $\deg \Gamma(x) \leq \deg \Lambda(x) = L' + L''$, from which we observe that the linear recursion $(\Lambda'(x)\Lambda''(x), L' + L'')$ produces the sequence $\boldsymbol{T} - \boldsymbol{S}$. We conclude that the inequality $L \leq L' + L''$ must always be satisfied. □

Lemma 7.2.3. *Suppose that* $(\Lambda^{(r-1)}(x), L_{r-1})$ *is a linear recursion of the shortest length that produces* $S_1, \ldots, S_{r-1}$*;* $(\Lambda^{(r)}(x), L_r)$ *is a linear recursion of the shortest*

length that produces $S_1, \ldots, S_{r-1}, S_r$*; and* $\Lambda^{(r)}(x) \neq \Lambda^{(r-1)}(x)$*. Then*

$$L_r \geq \max\left[L_{r-1}, r - L_{r-1}\right].$$

Proof: The inequality to be proved is a combination of two inequalities:

$$L_r \geq L_{r-1},$$
$$L_r \geq r - L_{r-1}.$$

The first inequality is obvious because, if a linear recursion produces a sequence, it must also produce any beginning segment of the sequence.

Let $\boldsymbol{\Delta} = (\Delta_1, \ldots, \Delta_r) = (0, 0, \ldots, 0, \Delta_r)$ where $\Delta_r \neq 0$. The sequence $\boldsymbol{\Delta}$ can be produced only by a linear recursion of length $L = r$. Suppose that $(\Lambda^{(r-1)}(x), L_{r-1})$ produces $\boldsymbol{S} + \boldsymbol{\Delta}$ and $(\Lambda^{(r)}(x), L_r)$ produces $\boldsymbol{S}$. Then with $\boldsymbol{T} = \boldsymbol{S} + \boldsymbol{\Delta}$ in the conclusion of Lemma 7.2.2, we have

$$r = L \leq L_{r-1} + L_r.$$

Consequently,

$$L_r \geq r - L_{r-1},$$

as was to be proved. □

If we can find a linear recursion that produces the given sequence and satisfies the inequality of Lemma 7.2.3 with equality, then it must be the shortest such linear recursion. The following lemma shows that Algorithm 7.2.1 accomplishes this.

Lemma 7.2.4. *Suppose that* $(\Lambda^{(i)}(x), L_i)$*, with* $i = 1, \ldots, r-1$*, is a sequence of minimum-length linear recursions such that* $\Lambda^{(i)}(x)$ *produces* $S_1, \ldots, S_i$*. If* $(\Lambda^{(r-1)}(x), L_{r-1})$ *does not produce* $S_1, \ldots, S_r$*, then any minimum-length linear recursion of length* L_r *that produces* $S_1, \ldots, S_r$ *has length*

$$L_r = \max\left[L_{r-1}, r - L_{r-1}\right].$$

Algorithm 7.2.1 gives such a recursion.

Proof: By Lemma 7.2.3, L_r cannot be smaller than $\max[L_{r-1}, r - L_{r-1}]$, so any linear recursion of length L_r that produces the required sequence must be a minimum-length linear recursion. We give a construction for a linear recursion of length L_r, assuming that we have constructed such linear recursions for each $i \leq r - 1$. For each i from 0 to $r - 1$, let $(\Lambda^{(i)}(x), L_i)$ be the minimum-length linear recursion that produces $S_1, \ldots, S_i$. For the induction argument, assume that

$$L_i = \max[L_{i-1}, i - L_{i-1}] \quad i = 1, \ldots, r-1$$

whenever $\Lambda^{(i)}(x) \neq \Lambda^{(i-1)}(x)$. This assumption is true for i such that S_i is the first nonzero syndrome because then $L_0 = 0$ and $L_i = i$. Let m denote the value of i at the

most recent iteration step that required a change in the recursion length. At the end of iteration $r-1$, m is that integer such that $L_{r-1} = L_m > L_{m-1}$.

We now have

$$S_j + \sum_{i=1}^{L_{r-1}} \Lambda_i^{(r-1)} S_{j-i} = \sum_{i=0}^{L_{r-1}} \Lambda_i^{(r-1)} S_{j-i} = \begin{cases} 0 & j = L_{r-1}, \ldots, r-1 \\ \Delta_r & j = r. \end{cases}$$

If $\Delta_r = 0$, then the linear recursion $(\Lambda^{(r-1)}(x), L_{r-1})$ also produces the first r components. Thus

$$L_r = L_{r-1} \quad \text{and} \quad \Lambda^{(r)}(x) = \Lambda^{(r-1)}(x).$$

If $\Delta_r \neq 0$, then a new linear recursion must be designed. Recall that a change in length occurred at iteration $i = m$. Hence

$$S_j + \sum_{i=1}^{L_{m-1}} \Lambda_i^{(m-1)} S_{j-i} = \begin{cases} 0 & j = L_{m-1}, \ldots, m-1 \\ \Delta_m \neq 0 & j = m, \end{cases}$$

and by the induction hypothesis,

$$\begin{aligned} L_{r-1} = L_m &= \max[L_{m-1}, m - L_{m-1}] \\ &= m - L_{m-1} \end{aligned}$$

because $L_m > L_{m-1}$. Now choose the new polynomial

$$\Lambda^{(r)}(x) = \Lambda^{(r-1)}(x) - \Delta_r \Delta_m^{-1} x^{r-m} \Lambda^{(m-1)}(x),$$

and let $L_r = \deg \Lambda^{(r)}(x)$. Then because $\deg \Lambda^{(r-1)}(x) \leq L_{r-1}$ and

$$\deg[x^{r-m} \Lambda^{(m-1)}(x)] \leq r - m + L_{m-1},$$

we conclude that

$$L_r \leq \max[L_{r-1}, r - m + L_{m-1}] \leq \max[L_{r-1}, r - L_{r-1}].$$

Hence, recalling Lemma 7.2.2, if $\Lambda^{(r)}(x)$ produces $S_1, \ldots, S_r$, then $L_r = \max[L_{r-1}, r - L_{r-1}]$. It remains only to prove that the linear recursion $(\Lambda^{(r)}(x), L_r)$ produces the required sequence. But the discrepancy in the next term is

$$\begin{aligned} S_j + \sum_{i=1}^{L_r} \Lambda_i^{(r)} S_{j-i} &= S_j + \sum_{i=1}^{L_{r-1}} \Lambda_i^{(r-1)} S_{j-i} - \Delta_r \Delta_m^{-1} \left(S_{j-r+m} + \sum_{i=1}^{L_{m-1}} \Lambda_i^{(m-1)} S_{j-r+m-i} \right) \\ &= \begin{cases} 0 & j = L_r, L_r + 1, \ldots, r-1 \\ \Delta_r - \Delta_r \Delta_m^{-1} \Delta_m = 0 & j = r. \end{cases} \end{aligned}$$

Hence the linear recursion $(\Lambda^{(r)}(x), L_r)$ produces $S_1, \ldots, S_r$. In particular, $(\Lambda^{(2t)}(x), L_{2t})$ produces $S_1, \ldots, S_{2t}$, and the lemma is proved. □

7.3 Decoding of binary BCH codes

Most decoding algorithms for BCH codes can be used for codes over any finite field. When the field is $GF(2)$, however, it is necessary only to find the error locations; the error magnitude is always one (but the decoder might compute the error magnitude as a check). As an example, Table 7.1 shows the computations of the Berlekamp–Massey algorithm used to decode a noisy codeword of the $(15, 5, 7)$ triple-error-correcting binary BCH code. The calculations can be followed by passing six times around the main loop of Figure 7.5 with the aid of a table of the arithmetic of $GF(16)$ (Table 6.2).

The possibility of simplification is suggested by the example of Table 7.1. In this example, Δ_r is always zero on even-numbered iterations. If this were always the case for binary codes, then even-numbered iterations can be skipped. The proof of this is based on the fact that, over $GF(2)$, the even-numbered syndromes are easily determined from the odd-numbered syndromes by the formula

$$S_{2j} = v(\omega^{2j}) = [v(\omega^j)]^2 = S_j^2,$$

as follows from Theorem 5.3.3. Let us calculate algebraic expressions for the coefficients of $\Lambda^{(r)}(x)$ for the first several values of r. Tracing the algorithm through the flow chart of Figure 7.5, and using the fact that $S_4 = S_2^2 = S_1^4$ for all binary codes, gives

$$\begin{aligned}
\Delta_1 &= S_1 & \Lambda^{(1)}(x) &= S_1 x + 1 \\
\Delta_2 &= S_2 + S_1^2 = 0 & \Lambda^{(2)}(x) &= S_1 x + 1 \\
\Delta_3 &= S_3 + S_1 S_2 & \Lambda^{(3)}(x) &= \left(S_1^{-1} S_3 + S_2\right) x^2 + S_1 x + 1 \\
\Delta_4 &= S_4 + S_1 S_3 + S_1^{-1} S_2 S_3 + S_2^2 = 0.
\end{aligned}$$

This shows that, for any binary BCH code, Δ_2 and Δ_4 must always be zero. It is impractical to carry out the calculation in this explicit way to show that Δ_r is zero for all even r. Instead, we will formulate an indirect argument to show the general case.

Theorem 7.3.1. *In a field of characteristic 2, for any sequence $S_1, S_2, \ldots, S_{2\nu-1}$ satisfying $S_{2j} = S_j^2$ for $j \le \nu$, suppose that connection polynomial $\Lambda(x)$ of degree ν satisfies*

$$S_j = \sum_{i=1}^{\nu} \Lambda_i S_{j-i} \quad j = \nu, \ldots, 2\nu - 1.$$

Then the next term of the sequence,

$$S_{2\nu} = \sum_{i=1}^{n-1} \Lambda_i S_{2\nu-i},$$

satisfies

$$S_{2\nu} = S_\nu^2.$$

Table 7.1. *Sample Berlekamp–Massey computation*

BCH $(15, 5)$ $t = 3$ code:

$$
\begin{aligned}
g(x) &= x^{10} + x^8 + x^5 + x^4 + x^2 + x + 1 \\
a(x) &= 0 \\
v(x) &= x^7 + x^5 + x^2 = e(x) \\
S_1 &= \alpha^7 + \alpha^5 + \alpha^2 = \alpha^{14} \\
S_2 &= \alpha^{14} + \alpha^{10} + \alpha^4 = \alpha^{13} \\
S_3 &= \alpha^{21} + \alpha^{15} + \alpha^6 = 1 \\
S_4 &= \alpha^{28} + \alpha^{20} + \alpha^8 = \alpha^{11} \\
S_5 &= \alpha^{35} + \alpha^{25} + \alpha^{10} = \alpha^5 \\
S_6 &= \alpha^{42} + \alpha^{30} + \alpha^{12} = 1
\end{aligned}
$$

r	Δ_r	$B(x)$	$\Lambda(x)$	L
0		1	1	0
1	α^{14}	α	$1+\alpha^{14}x$	1
2	0	αx	$1+\alpha^{14}x$	1
3	α^{11}	$\alpha^4+\alpha^3 x$	$1+\alpha^{14}x+\alpha^{12}x^2$	2
4	0	$\alpha^4 x+\alpha^3 x^2$	$1+\alpha^{14}x+\alpha^{12}x^2$	2
5	α^{11}	$\alpha^4+\alpha^3 x+\alpha x^2$	$1+\alpha^{14}x+\alpha^{11}x^2+\alpha^{14}x^3$	3
6	0	$\alpha^4 x+\alpha^3 x^2+\alpha x^3$	$1+\alpha^{14}x+\alpha^{11}x^2+\alpha^{14}x^3$	3

$$
\begin{aligned}
\Lambda(x) &= 1 + \alpha^{14}x + \alpha^{11}x^2 + \alpha^{14}x^3 \\
&= (1 + \alpha^7 x)(1 + \alpha^5 x)(1 + \alpha^2 x)
\end{aligned}
$$

Proof: The proof consists of finding identical expressions for S_ν^2 and $S_{2\nu}$. First, we have

$$
S_\nu^2 = \left(\sum_{i=1}^{n-1} \Lambda_i S_{\nu-i} \right)^2 = \sum_{i=1}^{n-1} \Lambda_i^2 S_{\nu-i}^2 = \sum_{i=1}^{n-1} \Lambda_i^2 S_{2\nu-2i}.
$$

We also have

$$
S_{2\nu} = \sum_{k=1}^{n-1} \Lambda_k S_{2\nu-k} = \sum_{k=1}^{n-1} \sum_{i=1}^{n-1} \Lambda_k \Lambda_i S_{2\nu-k-i}.
$$

By symmetry, every term in the double sum with $i \neq k$ appears twice, and in a field of characteristic two, the two terms add to zero. This means that only the diagonal terms with $i = k$ contribute, so

$$
S_{2\nu} = \sum_{i=1}^{n-1} \Lambda_i^2 S_{2\nu-2i},
$$

which agrees with the expression for S_ν^2, and thus proves the theorem. □

We conclude that Δ_r is zero for even r, and we can analytically combine two iterations to give for odd r:

$$\Lambda^{(r)}(x) = \Lambda^{(r-2)}(x) - \Delta_r x^2 B^{(r-2)}(x),$$
$$B^{(r)}(x) = \delta_r \Delta_r^{-1} \Lambda^{(r-2)}(x) + (1 - \delta_r) x^2 B^{(r-2)}(x).$$

Using these formulas, iterations with even r can be skipped. This gives a faster decoder for binary codes. Notice that this improvement can be used even though the error pattern might contain more than t errors because only the conjugacy relations of the syndromes were used in the proof of the theorem – nothing was assumed about the weight of the binary error pattern. Therefore subsequent tests for more than t errors are still valid.

7.4 Decoding of nonbinary BCH codes

The Berlekamp–Massey algorithm can be used to compute the error-locator polynomial $\Lambda(x)$ from the $2t$ syndromes $S_1, \ldots, S_{2t}$ for any nonbinary BCH code such as a Reed–Solomon code. A sample computation of the error-locator polynomial for a (15, 9, 7) Reed–Solomon triple-error-correcting code is given in Table 7.2. The calculations should be checked by tracing through the six iterations using the flow diagram of Figure 7.5.

For nonbinary codes, it is not enough to compute the error locations; the error values must be computed as well. There are a great many ways of organizing these computations. An alternative to recursive extension, and to the Gorenstein–Zierler algorithm, is to find the error locations from the zeros of $\Lambda(x)$ and to find the error values from yet another polynomial $\Gamma(x)$, called the *evaluator polynomial* or the *error-evaluator polynomial*, which is introduced for this purpose. Recall from Section 7.1 that the discrepancy polynomial $\Delta(x) = \Lambda(x)S(x)$ can always be decomposed as

$$\Lambda(x)S(x) = \Gamma(x) + x^{\nu+1} O(x) + x^{2t+1} \Theta(x)$$

where $O(x) = 0$ because $(\Lambda(x), \nu)$ produces the sequence $S_1, \ldots, S_{2t}$. Hence

$$\Lambda(x)S(x) = \Gamma(x) + x^{2t+1} \Theta(x)$$

and $\deg \Gamma(x) \leq \nu$. The polynomial $\Gamma(x)$ is the error-evaluator polynomial.

To decode, one computes the error-locator polynomial $\Lambda(x)$, possibly the error-evaluator polynomial $\Gamma(x)$, and the error-spectrum polynomial $E(x)$ or the error polynomial $e(x)$. An overview of selected alternatives is shown in Table 7.3. Some

Table 7.2. *Sample Berlekamp–Massey computation*

Reed–Solomon (15, 9) $t = 3$ code:

$$
\begin{aligned}
g(x) &= (x+\alpha)(x+\alpha^2)(x+\alpha^3)(x+\alpha^4)(x+\alpha^5)(x+\alpha^6) \\
&= x^6 + \alpha^{10}x^5 + \alpha^{14}x^4 + \alpha^4 x^3 + \alpha^6 x^2 + \alpha^9 x + \alpha^6 \\
a(x) &= 0 \\
v(x) &= \alpha x^7 + \alpha^5 x^5 + \alpha^{11}x^2 = e(x) \\
S_1 &= \alpha\alpha^7 + \alpha^5\alpha^5 + \alpha^{11}\alpha^2 = \alpha^{12} \\
S_2 &= \alpha\alpha^{14} + \alpha^5\alpha^{10} + \alpha^{11}\alpha^4 = 1 \\
S_3 &= \alpha\alpha^{21} + \alpha^5\alpha^{15} + \alpha^{11}\alpha^6 = \alpha^{14} \\
S_4 &= \alpha\alpha^{28} + \alpha^5\alpha^{20} + \alpha^{11}\alpha^8 = \alpha^{13} \\
S_5 &= \alpha\alpha^{35} + \alpha^5\alpha^{25} + \alpha^{11}\alpha^{10} = 1 \\
S_6 &= \alpha\alpha^{42} + \alpha^5\alpha^{30} + \alpha^{11}\alpha^{12} = \alpha^{11}
\end{aligned}
$$

r	Δ_r	$B(x)$	$\Lambda(x)$	L
0		1	1	0
1	α^{12}	α^3	$1+\alpha^{12}x$	1
2	α^7	$\alpha^3 x$	$1+\alpha^3 x$	1
3	1	$1+\alpha^3 x$	$1+\alpha^3 x+\alpha^3 x^2$	2
4	1	$x+\alpha^3 x^2$	$1+\alpha^{14}x$	2
5	α^{11}	$\alpha^4+\alpha^3 x$	$1+\alpha^{14}x+\alpha^{11}x^2+\alpha^{14}x^3$	3
6	0	$\alpha^4 x+\alpha^3 x^2$	$1+\alpha^{14}x+\alpha^{11}x^2+\alpha^{14}x^3$	3

$$
\begin{aligned}
\Lambda(x) &= 1+\alpha^{14}x+\alpha^{11}x^2+\alpha^{14}x^3 \\
&= (1+\alpha^7 x)(1+\alpha^5 x)(1+\alpha^2 x)
\end{aligned}
$$

of the algorithms listed there have been described previously, and some will be described in due course.

More directly, the error-evaluator polynomial $\Gamma(x)$ is defined from the error-locator polynomial for $S(x)$ as:

$$\Gamma(x) = \Lambda(x)S(x) \pmod{x^{2t+1}}.$$

Whereas the error-locator polynomial depends on the locations but not on the values of the errors, we shall see that the error-evaluator polynomial also depends on the error values.

The introduction of the error-evaluator polynomial leads to an alternative statement regarding the linear recursion that produces the error spectrum. The locator polynomial $\Lambda(x)$ for $\boldsymbol{e}$ satisfies

$$E(x)\Lambda(x) = 0 \pmod{x^n - 1},$$

Table 7.3. *A menu of algorithms*

Computation of error-locator polynomial $\Lambda(x)$
. Peterson algorithm
. Berlekamp–Massey algorithm
. Sugiyama algorithm
. Hybrid methods
Computation of error polynomial $e(x)$
. Gorenstein–Zierler algorithm
. Forney algorithm/Chien search
. Recursive extension
Computation of error-evaluator polynomial $\Gamma(x)$
. Polynomial multiplication $\Lambda(x)S(x)$
. Berlekamp algorithm
. Sugiyama algorithm

so the locator polynomial $\Lambda(x)$ is the connection polynomial for the periodic repetition of the sequence $E_0, E_1, \ldots, E_{n-1}$. Therefore,

$$E(x)\Lambda(x) = \Gamma(x)(x^n - 1).$$

The polynomial $\Gamma(x)$ that satisfies this equation is the error-evaluator polynomial, and its degree clearly satisfies $\deg \Gamma(x) \leq \deg \Lambda(x)$.

We can reformulate the problem to portray the error-evaluator polynomial in a more central role. This formulation asks for the smallest integer $\nu \leq t$ and for vectors $\mathbf{\Lambda} = (\Lambda_1, \ldots, \Lambda_\nu)$ and $\mathbf{\Gamma} = (\Gamma_1, \ldots, \Gamma_\nu)$ that satisfy the $2t$ equations

$$\Gamma_k = S_k + \sum_{j=1}^{\nu} \Lambda_j S_{k-j} \quad k = 1, \ldots, 2t$$

where $S_j = 0$ for $j \leq 0$. Notice that k now runs over $2t$ values, so there are $2t$ equations and $2t$ unknowns. The solution is described by the two polynomials $\Lambda(x)$ and $\Gamma(x)$.

Theorem 7.4.1. *The error-evaluator polynomial can be written*

$$\Gamma(x) = \sum_{i=1}^{\nu} x Y_i X_i \prod_{j \neq i} (1 - X_j x).$$

Proof: Recall that the (frequency-domain) syndrome polynomial is given by

$$S(x) = \sum_{j=1}^{2t} S_j x^j = \sum_{j=1}^{2t} \sum_{\ell=1}^{\nu} Y_\ell X_\ell^j x^j$$

where $X_\ell = \omega^{i_\ell}$ is the error-location number of the ℓth error, and $Y_\ell = e_{i_\ell}$ is the value of the ℓth error. By definition of the terms in $\Gamma(x)$,

$$\begin{aligned}\Gamma(x) &= \left[\sum_{j=1}^{2t}\sum_{i=1}^{\nu} Y_i X_i^j x^j\right]\left[\prod_{\ell=1}^{\nu}(1 - X_\ell x)\right] \pmod{x^{2t+1}} \\ &= \sum_{i=1}^{\nu} x Y_i X_i \left[(1 - X_i x)\sum_{j=1}^{2t}(X_i x)^{j-1}\right]\prod_{\ell\neq i}(1 - X_\ell x) \pmod{x^{2t+1}}.\end{aligned}$$

The term in square brackets is a factorization of $(1 - X_i^{2t}x^{2t})$. Therefore

$$\begin{aligned}\Gamma(x) &= \sum_{i=1}^{\nu} x Y_i X_i \left(1 - X_i^{2t}x^{2t}\right)\prod_{\ell\neq i}(1 - X_\ell x) \pmod{x^{2t+1}} \\ &= \sum_{i=1}^{\nu} x Y_i X_i \prod_{\ell\neq i}(1 - X_\ell x),\end{aligned}$$

which is the expression that was to be proved. □

Now we are ready to give an expression for the error values that is much simpler than matrix inversion. It uses $\Lambda'(x)$, the formal derivative of $\Lambda(x)$.

Theorem 7.4.2 (Forney Algorithm). *The ℓth error value is given by*

$$Y_\ell = \frac{\Gamma\left(X_\ell^{-1}\right)}{\prod_{j\neq\ell}\left(1 - X_j X_\ell^{-1}\right)} = -X_\ell \frac{\Gamma\left(X_\ell^{-1}\right)}{\Lambda'\left(X_\ell^{-1}\right)}.$$

Proof: Theorem 7.4.1 states that

$$\Gamma(x) = \sum_{i=1}^{\nu} x Y_i X_i \prod_{j\neq i}(1 - X_j x).$$

Set $x = X_\ell^{-1}$. In the sum on i, every term will be zero except the term with $i = \ell$. Therefore

$$\Gamma\left(X_\ell^{-1}\right) = Y_\ell \prod_{j\neq\ell}\left(1 - X_j X_\ell^{-1}\right).$$

and

$$Y_\ell = \frac{\Gamma\left(X_\ell^{-1}\right)}{\prod_{j\neq\ell}\left(1 - X_j X_\ell^{-1}\right)}.$$

Next we express the denominator in terms of the formal derivative of $\Lambda(x)$. Recall that the locator polynomial can be written in two ways

$$\Lambda(x) = \sum_{j=0}^{\nu} \Lambda_j x^j = \prod_{\ell=1}^{\nu}(1 - x X_\ell)$$

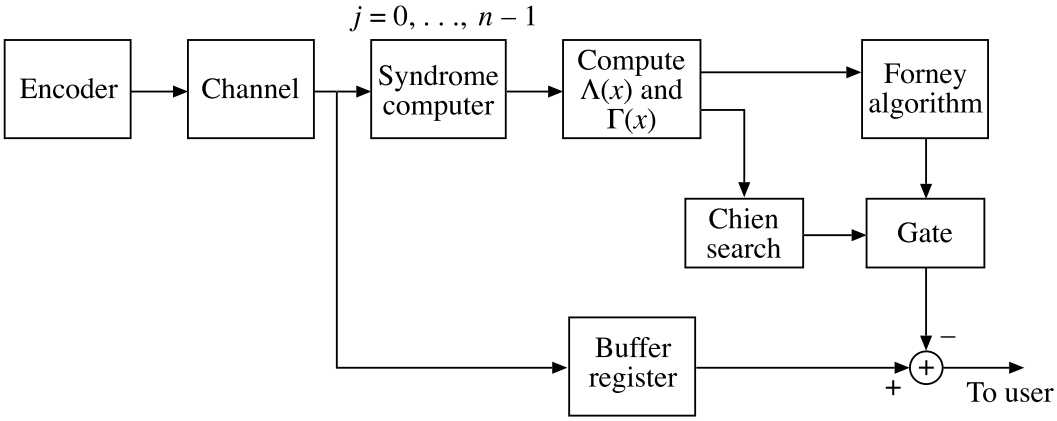


Figure 7.6. Another encoder/decoder scheme for BCH codes

so the formal derivative is

$$\Lambda'(x) = \sum_{j=0}^{v} j\Lambda_j x^{j-1} = -\sum_{\ell=1}^{v} X_i \prod_{j\neq\ell}(1 - xX_\ell).$$

Hence

$$\Lambda'\left(X_\ell^{-1}\right) = -X_\ell \prod_{j\neq\ell}\left(1 - X_j X_\ell^{-1}\right),$$

from which the theorem follows. □

The error-evaluator polynomial can be used with the Forney algorithm to compute the error values, as shown in Figure 7.6. This can be computationally faster than recursive extension, but the structure is not as simple. The Forney algorithm also is simpler computationally than the matrix inversion of the Gorenstein–Zierler algorithm, but it still requires division.

Figure 7.7 shows a flow diagram for a decoder that computes the frequency-domain codeword $\boldsymbol{C}$ from the frequency-domain senseword $\boldsymbol{V}$. The flow diagram must be preceded by a Fourier transform and followed by an inverse Fourier transform to provide a decoder that computes the codeword $\boldsymbol{c}$ from the time-domain senseword $\boldsymbol{v}$. The final $2t$ iterations of the flow diagram provide the recursive extension, but in the slightly modified form of the pair of equations

$$\Delta_r = \sum_{j=0}^{n-1} \Lambda_j S_{r-j}$$

$$S_r \leftarrow S_r - \Delta_r.$$

This restructuring allows the sum to be written with j starting at zero both for $r \leq 2t$ and for $r > 2t$, even though S_j for $j > 2t$ is not set to zero.

The decoder shown in Figure 7.7 has a simple structure for $r > 2t$, but requires $n - 2t$ such iterations. These iterations can be avoided by using the Forney algorithm, as shown in Figure 7.8, but this requires computation of the error-evaluator polynomial.

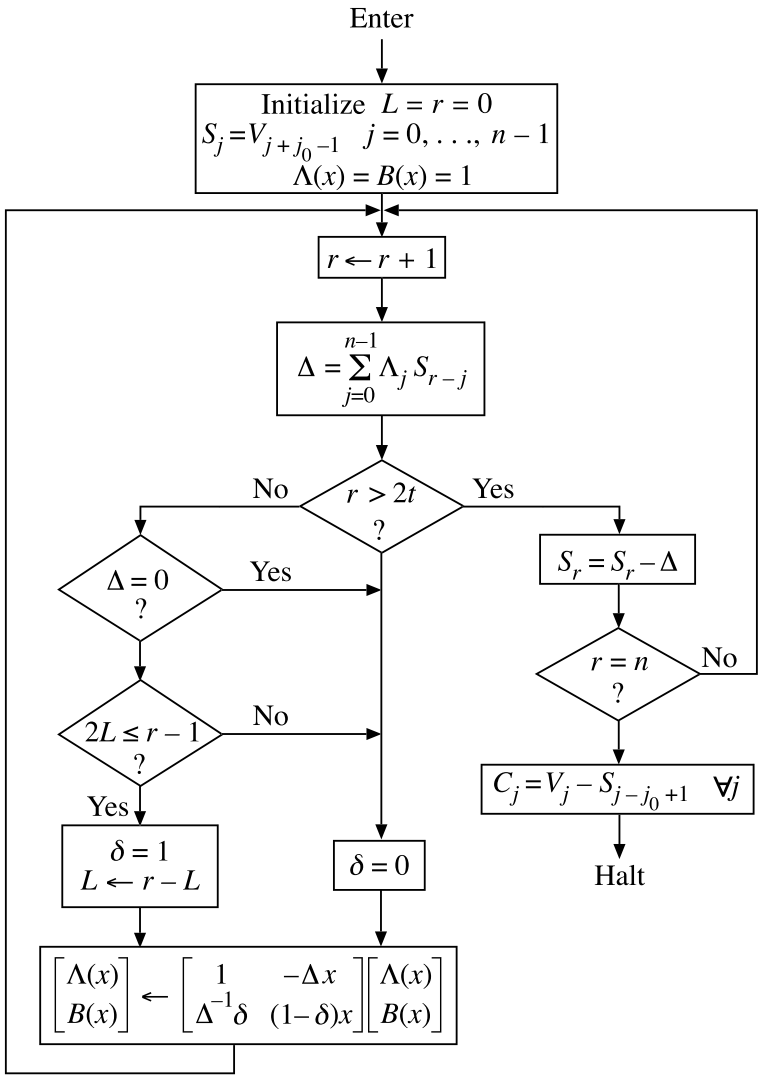


Figure 7.7. A frequency-domain decoder

A variation of the Berlekamp–Massey algorithm, which sometimes may be more attractive than the direct approach, is the *Berlekamp algorithm*. To compute the error-evaluator polynomial $\Gamma(x)$ from the defining equation

$$\Gamma(x) = \Lambda(x)S(x) \pmod{x^{2t+1}},$$

where

$$S(x) = \sum_{j=1}^{2t} S_j x^j,$$

requires a polynomial product. The Berlekamp algorithm replaces computation of the polynomial product $\Gamma(x) = \Lambda(x)S(x)$ with an iterative computation. It computes $\Gamma(x)$ in lockstep with the computation of $\Lambda(x)$ using a similar iteration. Define the iterate

$$\Gamma^{(r)}(x) = \Lambda^{(r)}(x)S(x) \pmod{x^{r+1}}$$

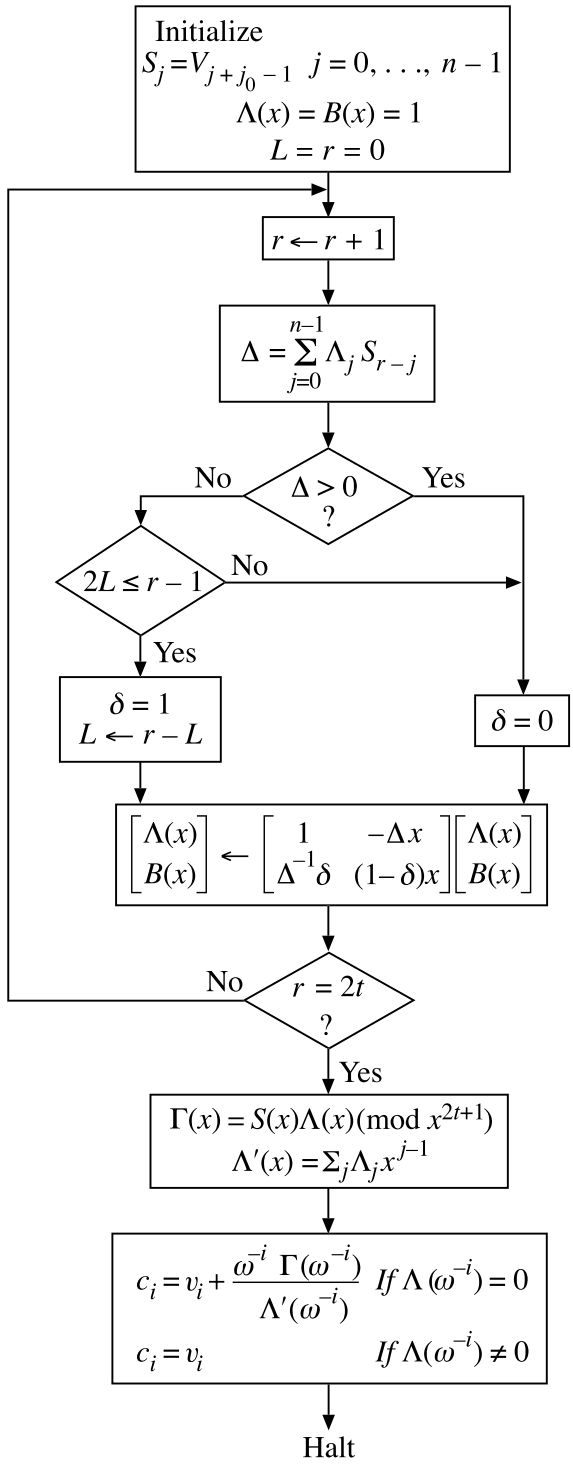


Figure 7.8. A decoder using the Forney algorithm

Clearly, the desired $\Gamma(x)$ is equal to $\Gamma^{(2t)}(x)$. Also define the scratch polynomial

$$A^{(r)}(x) = B^{(r)}(x)S(x) - x^r \pmod{x^{r+1}}.$$

The following algorithm iterates $\Gamma^{(r)}(x)$ and $A^{(r)}(x)$ directly to obtain $\Gamma(x)$ (with Δ_r and δ_r as before).

Algorithm 7.4.3. *Let* $\Gamma^{(0)}(x) = 0$, $A^{(0)}(x) = -1$, *and let*

$$\begin{bmatrix} \Gamma^{(r)}(x) \\ A^{(r)}(x) \end{bmatrix} = \begin{bmatrix} 1 & -\Delta_r x \\ \Delta_r^{-1}\delta_r & (1-\delta_r)x \end{bmatrix} \begin{bmatrix} \Gamma^{(r-1)}(x) \\ A^{(r-1)}(x) \end{bmatrix},$$

then $\Gamma^{(2t)}(x) = \Gamma(x)$.

Proof: Because

$$S(x)\Lambda^{(r)}(x) = \Gamma^{(r)}(x) \pmod{x^{r+1}},$$

we know that

$$S(x)\Lambda^{(r-1)}(x) = \Gamma^{(r-1)}(x) + \Delta x^r \pmod{x^{r+1}}$$

and, by definition of the iterates,

$$\begin{bmatrix} \Gamma^{(r)}(x) \\ A^{(r)}(x) \end{bmatrix} = \begin{bmatrix} S(x)\Lambda^{(r)}(x) \\ S(x)B^{(r)}(x) - x^r \end{bmatrix} \pmod{x^{r+1}}.$$

Using the iteration rule of Algorithm 7.2.1 to expand the right side leads to

$$\begin{aligned} \begin{bmatrix} \Gamma^{(r)}(x) \\ A^{(r)}(x) \end{bmatrix} &= \begin{bmatrix} 1 & -\Delta \\ \Delta^{-1}\delta & \overline{\delta} \end{bmatrix} \begin{bmatrix} S(x)\Lambda^{(r-1)}(x) \\ xS(x)B^{(r-1)}(x) \end{bmatrix} - \begin{bmatrix} 0 \\ x^r \end{bmatrix} \pmod{x^{r+1}} \\ &= \begin{bmatrix} 1 & -\Delta \\ \Delta^{-1}\delta & \overline{\delta} \end{bmatrix} \begin{bmatrix} \Gamma^{(r-1)}(x) + \Delta x^r \\ xA^{(r-1)}(x) + x^r \end{bmatrix} - \begin{bmatrix} 0 \\ x^r \end{bmatrix} \\ &= \begin{bmatrix} 1 & -\Delta \\ \Delta^{-1}\delta & \overline{\delta} \end{bmatrix} \begin{bmatrix} \Gamma^{(r-1)}(x) \\ xA^{(r-1)}(x) \end{bmatrix}, \end{aligned}$$

which is the iteration given in the theorem. It only remains to verify that the initialization gets the iterations off to the right start. The first iteration always gives

$$\begin{bmatrix} \Gamma^{(1)}(x) \\ A^{(1)}(x) \end{bmatrix} = \begin{bmatrix} 1 & -\Delta x \\ \Delta^{-1}\delta & \overline{\delta}x \end{bmatrix} \begin{bmatrix} 0 \\ -1 \end{bmatrix},$$

which, because $S_1 = \Delta_1$, reduces to

$$\Gamma^{(1)}(x) = S_1 x$$

and

$$A^{(1)}(x) = \begin{cases} -x & S_1 = 0 \\ 0 & S_1 \neq 0. \end{cases}$$

On the other hand, by the definition of $\Gamma^{(1)}(x)$ and $A^{(1)}(x)$, the first iteration should compute

$$\begin{aligned} \Gamma^{(1)}(x) &= S(x)(1 - \Delta x) \pmod{x^2} \\ &= S_1 x \end{aligned}$$

and

$$A^{(1)}(x) = \begin{cases} S(x)x - x \pmod{x^2} & \text{if } S_1 = 0 \\ S(x)S_1^{-1} - x \pmod{x^2} & \text{if } S_1 \neq 0 \end{cases}$$
$$= \begin{cases} -x & \text{if } S_1 = 0 \\ 0 & \text{if } S_1 \neq 0. \end{cases}$$

This agrees with the result of the first iteration, so the proof is complete. □

The iterations of Algorithm 7.2.1 and Algorithm 7.4.3 can be combined neatly into the following single statement:

$$\begin{bmatrix} \Lambda(x) & \Gamma(x) \\ B(x) & A(x) \end{bmatrix} \leftarrow \begin{bmatrix} 1 & -\Delta x \\ \Delta^{-1}\delta & \overline{\delta} x \end{bmatrix} \begin{bmatrix} \Lambda(x) & \Gamma(x) \\ B(x) & A(x) \end{bmatrix}.$$

A decoder based on this iteration is shown in Figure 7.9. Clearly both sides of this equation can be multiplied on the right side by any invertible matrix without invalidating the iteration.

7.5 Decoding with erasures and errors

Some channels make both errors and erasures. On such a channel the output alphabet consists of the channel input alphabet and a blank (or an equivalent symbol) denoting an erasure. A senseword is a vector of blocklength n of such symbols. The decoder must correct the errors and fill the erasures. Given a code with minimum distance $d_{\min}$, any pattern of ν errors and ρ erasures can be decoded, provided that

$$d_{\min} \geq 2\nu + 1 + \rho.$$

The largest ν that satisfies this inequality is denoted by t_ρ. To decode a senseword with ρ erasures, it is necessary to find a codeword that differs from the unerased portion of the senseword in at most t_ρ places. Any such codeword will do because, by the definition of t_ρ, such a codeword is unique. We need only to seek any solution because only one solution exists.

One way to decode is to guess the erased symbols and then apply any error-correction procedure. If the procedure finds that less than t_ρ of the unerased symbols are in error (and possibly some of the guesses for the erased symbols), then these errors can be corrected and the codeword restored. If the decoding fails, or if too many errors are found, then a new guess is made for the erased symbols. A trial-and-error procedure is impractical unless the number of combinations that must be tried for the erasures is small.

We will incorporate erasure filling into the decoding algorithm in a more central way. We will only treat BCH codes, and only those with $j_0 = 1$. Let v_i for $i = 0, \ldots, n-1$ be the components of the senseword. Suppose that erasures are made in locations

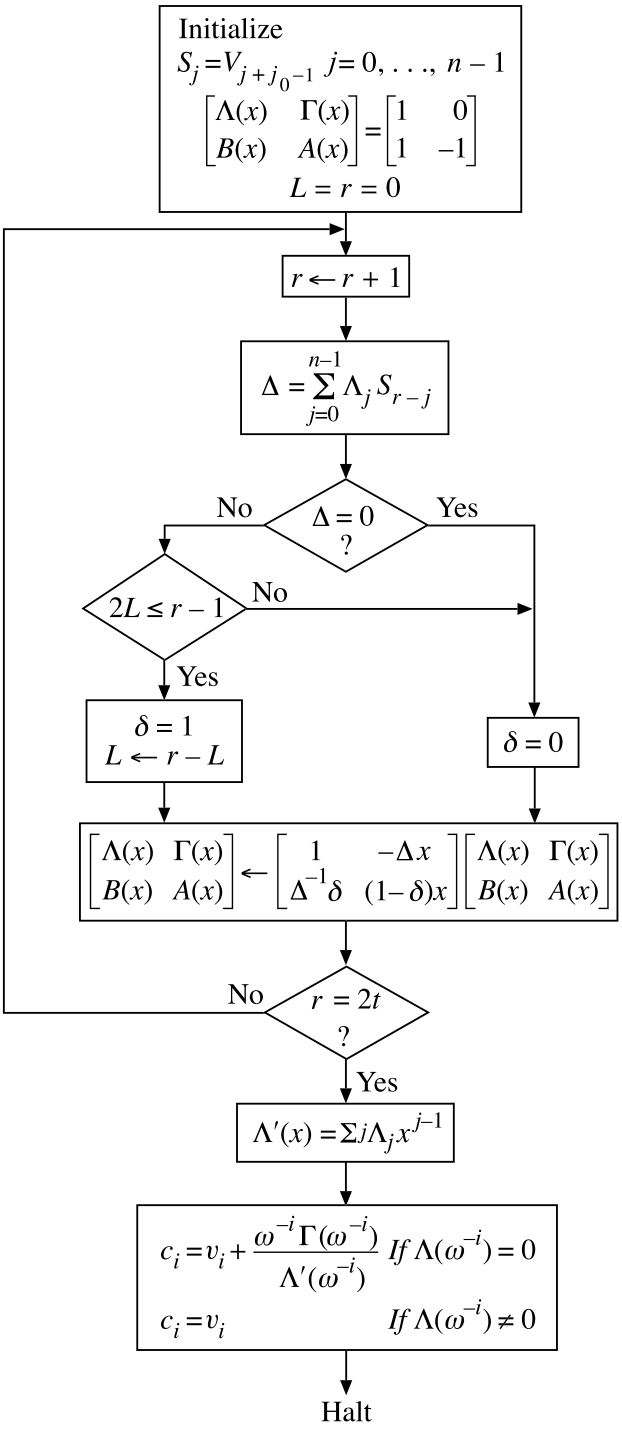


Figure 7.9. Another decoder using the Berlekamp algorithm and the Forney algorithm

$i_1, i_2, \ldots, i_\rho$. At these known locations, the senseword has blanks, which we may fill initially with any arbitrary symbols, but marking each by a special flag. We may choose an estimate of the codeword symbol by any appropriate means to fill the erasure. This symbol then is called a *readable erasure*.

Define the erasure vector as that vector of length n having component f_{i_ℓ} equal to the erased symbol for $\ell = 1, \ldots, \rho$, and in other components, f_i equals zero. Then

$$v_i = c_i + e_i + f_i \quad i = 0, \ldots, n-1.$$

Let ψ be any vector that is zero at every erased location, and otherwise nonzero. Then

$$\psi_i v_i = \psi_i(c_i + e_i + f_i) = \psi_i c_i + \psi_i e_i.$$

Define the modified senseword $v'_i = \psi_i v_i$, the modified error vector $e'_i = \psi_i e_i$, and the modified codeword $c'_i = \psi_i c_i$. The modified error vector $\boldsymbol{e}'$ has errors in the same locations as does $\boldsymbol{e}$. The equation becomes

$$v'_i = c'_i + e'_i.$$

Now the problem is to decode $\boldsymbol{v}'$ to find $\boldsymbol{e}'$, which we know how to do, in principle, if the weight of $\boldsymbol{e}'$ is smaller than $(d_{\min} - \rho)/2$.

The next step in the development is to choose ψ by working in the frequency domain. Let $U_\ell = \omega^{i_\ell}$ for $\ell = 1, \ldots, \rho$ denote the erasure locations. Define the erasure-locator polynomial:

$$\Psi(x) = \sum_{k=0}^{n-1} \Psi_k x^k = \prod_{\ell=1}^{\rho}(1 - xU_\ell).$$

This is defined so that the inverse Fourier transform of the vector $\boldsymbol{\Psi}$ has components ψ_i equal to zero whenever $f_i \neq 0$. Therefore $\psi_i f_i = 0$ for all i. In the frequency domain,

$$\boldsymbol{V}' = (\boldsymbol{\Psi} * \boldsymbol{C}) + \boldsymbol{E}'.$$

But $\boldsymbol{\Psi}$ is nonzero only in a block of length $\rho + 1$, and by the construction of a BCH code, $\boldsymbol{C}$ is zero in a block of length $2t$. Consequently, the convolution $\boldsymbol{\Psi} * \boldsymbol{C}$ is zero in a block of length $2t - \rho$. In this block, define the modified syndrome S'_j by $S'_j = V'_j$. Then

$$S'_j = (\boldsymbol{\Psi} * \boldsymbol{V})_j = E'_j,$$

which has enough syndrome components to compute the correctable patterns of the modified error vector $\boldsymbol{e}'$.

As in the errors-only case, we can find the error-locator polynomial $\Lambda(x)$ from these $2t - \rho$ known values of $\boldsymbol{E}'$, provided the number of errors ν is not more than $(2t - \rho)/2$. Once the error-locator polynomial is known, we can combine it with the erasure-locator polynomial and proceed as in the errors-only case. To do this, first define the *error-and-erasure-locator polynomial.*

$$\overline{\Lambda}(x) = \Psi(x)\Lambda(x).$$

The inverse Fourier transform of $\overline{\boldsymbol{\Lambda}}$ is zero at every error or erasure. That is, $\overline{\lambda}_i = 0$ if $e_i \neq 0$ or $f_i \neq 0$. Therefore $\overline{\lambda}_i(e_i + f_i) = 0$,

$$\overline{\boldsymbol{\Lambda}} * (\boldsymbol{E} + \boldsymbol{F}) = \boldsymbol{0},$$

and $\overline{\boldsymbol{\Lambda}}$ is nonzero in a block of length at most $\nu + \rho + 1$. Hence the $2t$ known values of $\boldsymbol{E} + \boldsymbol{F}$ can be recursively extended to n values by using this convolution equation

and the known value of $\overline{\mathbf{\Lambda}}$. Then

$$C_i = V_i - (E_i + F_i).$$

An inverse Fourier transform completes the decoding.

The step of computing the error-locator polynomial from the modified syndromes can use the Berlekamp–Massey algorithm. Starting with the initial estimates of $\Lambda^{(0)}(x) = 1$ and $B^{(0)}(x) = 1$ and proceeding through $2t - \rho$ iterations, the Berlekamp–Massey algorithm would compute $\Lambda(x)$ by a recursive procedure, using the modified syndromes

$$\Delta_r = \sum_{j=0}^{n-1} \Lambda_j^{(r-1)} S'_{r-j}$$

in the equation for Δ_r. After $2t - \rho$ iterations, the error-locator polynomial $\Lambda(x)$ is obtained.

It is possible to do much better, however, by combining several steps. Multiply the equations of the Berlekamp–Massey algorithm by $\Psi(x)$ to get the new equations

$$\Lambda^{(r)}(x)\Psi(x) = \Lambda^{(r-1)}(x)\Psi(x) - \Delta_r x B^{(r-1)}(x)\Psi(x)$$
$$B^{(r)}(x)\Psi(x) = (1 - \delta_r) x B^{(r-1)}(x)\Psi(x) + \delta_r \Delta_r^{-1} \Lambda^{(r-1)}(x)\Psi(x).$$

Thus if we start instead with the initial values $\Lambda^{(0)}(x) = B^{(0)}(x) = \Psi(x)$, we will iteratively compute the error-and-erasure-locator polynomial $\overline{\Lambda}(x)$, using the iterates $\overline{\Lambda}^{(r)}(x) = \Lambda^{(r)}(x)\Psi(x)$ and $\overline{B}^{(r)}(x) = B^{(r)}(x)\Psi(x)$. Even the computation of Δ_r works out simply because of the associativity property of the convolution. This is

$$\Delta_r = \sum_{j=0}^{n-1} \Lambda_{r-j}^{(r-1)} S'_j = \sum_{k=0}^{n-1} \overline{\Lambda}_{r-k}^{(r-1)} S_k = \sum_{j=0}^{n-1} \left(\sum_{k=0}^{n-1} \Lambda_k^{(r-1)} \Psi_{r-j-k} \right) S_j.$$

Therefore if we initialize the Berlekamp–Massey algorithm with $\Psi(x)$ instead of with one, the modified syndromes are computed implicitly and need not explicitly appear.

We now drop the notation $\overline{\Lambda}(x)$, replacing it with $\Lambda(x)$, which will now be called the error-and-erasure-locator polynomial. The Berlekamp–Massey algorithm initialized with $\Psi(x)$ produces recursively the error-and-erasure-locator polynomial, according to the equations

$$\Lambda^{(r)}(x) = \Lambda^{(r-1)}(x) - \Delta_r x B^{(r-1)}(x)$$
$$B^{(r)}(x) = (1 - \delta_r) x B^{(r-1)}(x) + \delta_r \Delta_r^{-1} \Lambda^{(r-1)}(x)$$
$$\Delta_r = \sum_{j=0}^{n-1} \Lambda_j^{(r-1)} S_{r-j},$$

which are exactly the same as for the case of errors-only decoding. Only the initialization is different.

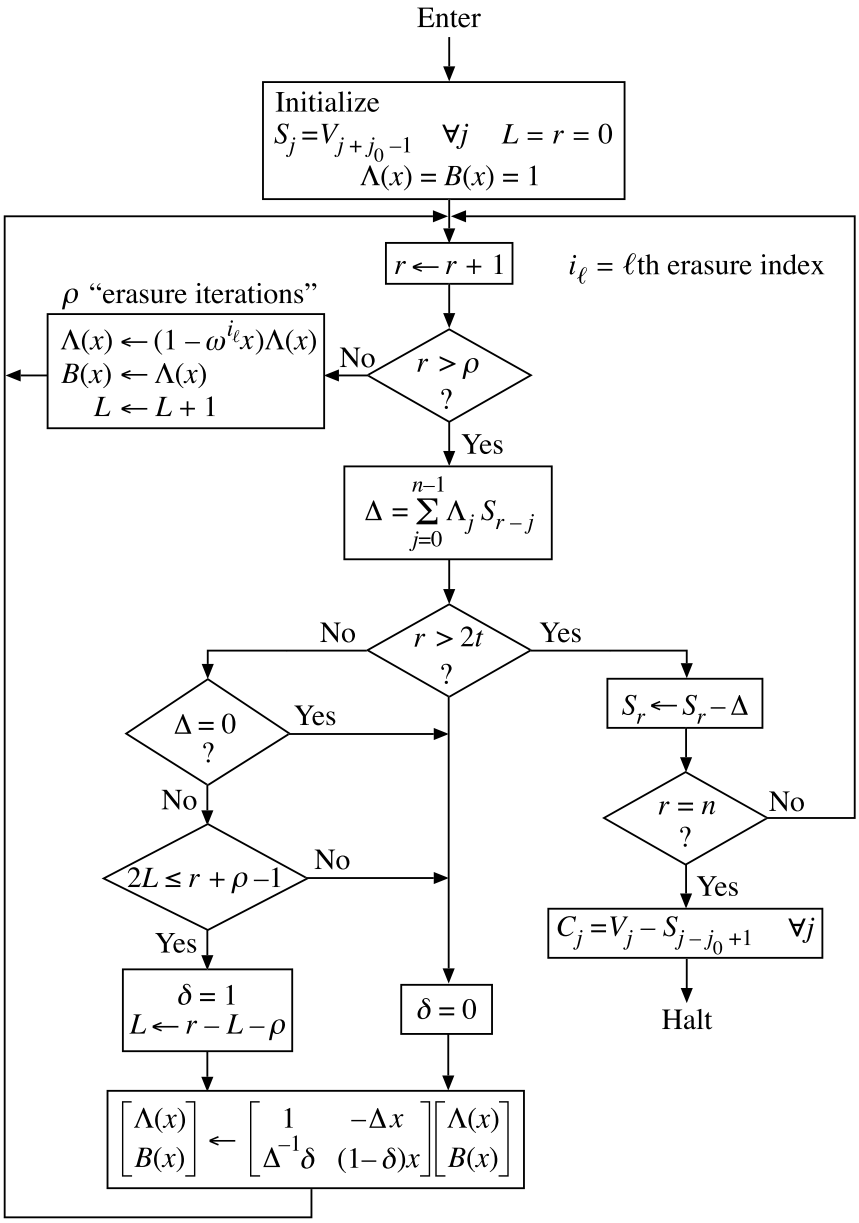


Figure 7.10. An error-and-erasure decoder for BCH codes

The Berlekamp–Massey algorithm, revised to handle erasures as well, is shown in Figure 7.10. This should be compared to Figure 7.7. Only the computation of the erasure-locator polynomial is new, which is a simple computation compared to other decoding computations. This consists of a special loop for the first ρ iterations, as shown in Figure 7.10. The index r counts out the first ρ iterations, which form the erasure polynomial, and then continues to count out the iterations of the Berlekamp–Massey algorithm, stopping when r reaches $2t$. The shift-register length L is increased once for each erasure, and thereafter the length changes according to the procedure of the Berlekamp–Massey algorithm, but modified by replacing r and L by $r - \rho$ and $L - \rho$, respectively.

7.6 Decoding in the time domain

The decoding procedure developed in Section 7.1, when stated in the language of spectral estimation, consists of a Fourier transform (syndrome computer), followed by a spectral analysis (Berlekamp–Massey algorithm), followed by an inverse Fourier transform (Chien search). In this section, we shall show how to transform the decoding procedure into the time domain, rather than transform the data into the frequency domain. This will eliminate both Fourier transforms from the decoder. The idea is to notice that the Berlekamp–Massey iteration is linear in the two updated polynomials. A time-domain counterpart to these equations can be found analytically by taking the inverse Fourier transform of the decoding equations. This gives a set of decoding equations that operates directly on the raw data, and the error correction is completed without the need to compute transforms.

To eliminate the need for computing a Fourier transform of the data, transform all variables in Algorithm 7.2.1. The transform of $\mathbf{\Lambda}$ is the time-domain error-locator vector $\boldsymbol{\lambda} = \{\lambda_i \mid i = 0, \ldots, n-1\}$. The transform of $\boldsymbol{B}$ is the time-domain scratch vector $\boldsymbol{b} = \{b_i \mid i = 0, \ldots, n-1\}$. The Berlekamp–Massey algorithm transforms into the iterative procedure in the time domain, described in the following theorem.

Algorithm 7.6.1. *Let $\boldsymbol{v}$ be at distance at most t from a codeword of a BCH code of designed distance $2t+1$. With initial conditions $\lambda_i^{(0)} = 1$, $b_i^{(0)} = 1$ for all i and $L_0 = 0$, let the following set of equations for $r = 1, \ldots, 2t$ be used iteratively to compute $\lambda_i^{(2t)}$ for $i = 0, \ldots, n-1$:*

$$\Delta_r = \sum_{i=0}^{n-1} \omega^{ir} \left(\lambda_i^{(r-1)} v_i\right)$$

$$L_r = \delta_r (r - L_{r-1}) + \overline{\delta}_r L_{r-1}$$

$$\begin{bmatrix} \lambda_i^{(r)} \\ b_i^{(r)} \end{bmatrix} = \begin{bmatrix} 1 & -\Delta_r \omega^{-i} \\ \Delta_r^{-1} \delta_r & \overline{\delta}_r \omega^{-i} \end{bmatrix} \begin{bmatrix} \lambda_i^{(r-1)} \\ b_i^{(r-1)} \end{bmatrix},$$

$r = 1, \ldots, 2t$, where $\delta_r = (1 - \overline{\delta}_r) = 1$ if both $\Delta_r \neq 0$ and $2L_{r-1} \leq r-1$, and otherwise $\delta_r = 0$. Then $\lambda_i^{(2t)} = 0$ if and only if $e_i \neq 0$.

Proof: Take the Fourier transform of all vector quantities. Then the equations of Algorithm 7.2.1 are obtained, except for the equation

$$\Delta_r = \sum_{j=0}^{n-1} \Lambda_j^{(r-1)} V_{r-j},$$

which has $\boldsymbol{V}$ in place of $\boldsymbol{S}$. But $\Lambda_j^{(r-1)} = 0$ for $j > 2t$, and so $V_j = S_j$ for $j = 1, \ldots, 2t$.

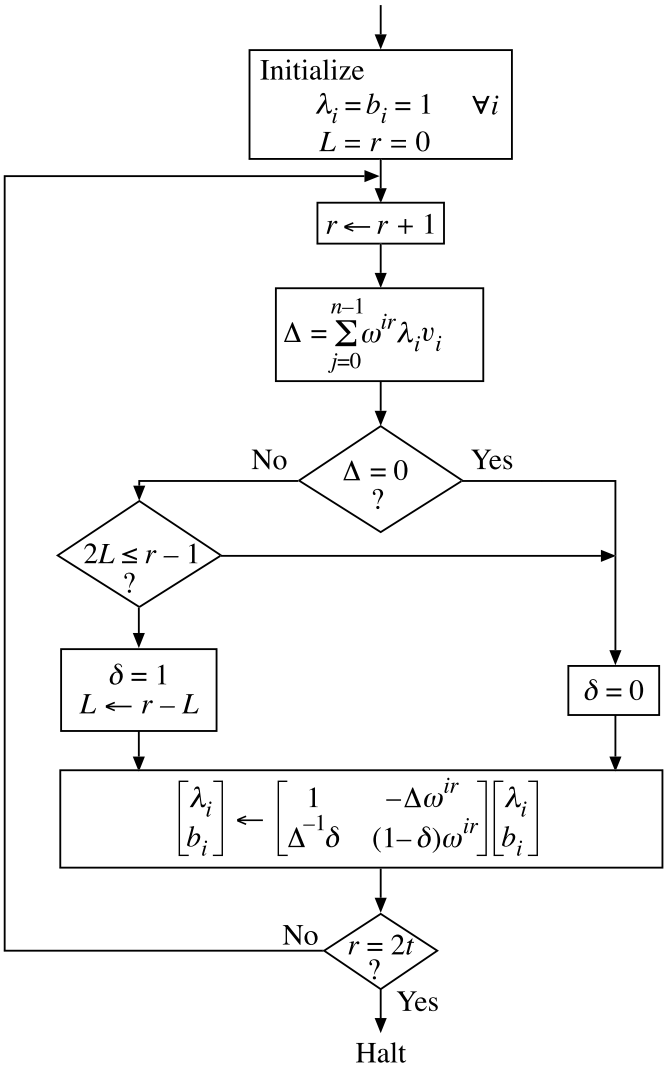


Figure 7.11. Time-domain Berlekamp–Massey algorithm

Therefore

$$\Delta_r = \sum_{j=0}^{n-1} \Lambda_j^{(r-1)} S_{r-j},$$

and the time-domain algorithm reduces exactly to the form of Algorithm 7.2.1. □

We call Algorithm 7.6.1 the *time-domain* (or the *code-domain*) *Berlekamp–Massey algorithm* which is given in flow-chart form in Figure 7.11. It computes the time-domain error-locator vector $\boldsymbol{\lambda}$ directly from the raw data $\boldsymbol{v}$; no Fourier transform is needed. For binary codes, once the vector $\boldsymbol{\lambda}$ is known, correction is nearly complete because the senseword has an error in component i if and only if $\lambda_i = 0$. In Figure 7.12, a time-domain decoder for binary codes is shown.

For nonbinary codes, it does not suffice to compute only the error locations in the time domain. We must also compute the error values. In the frequency domain, all

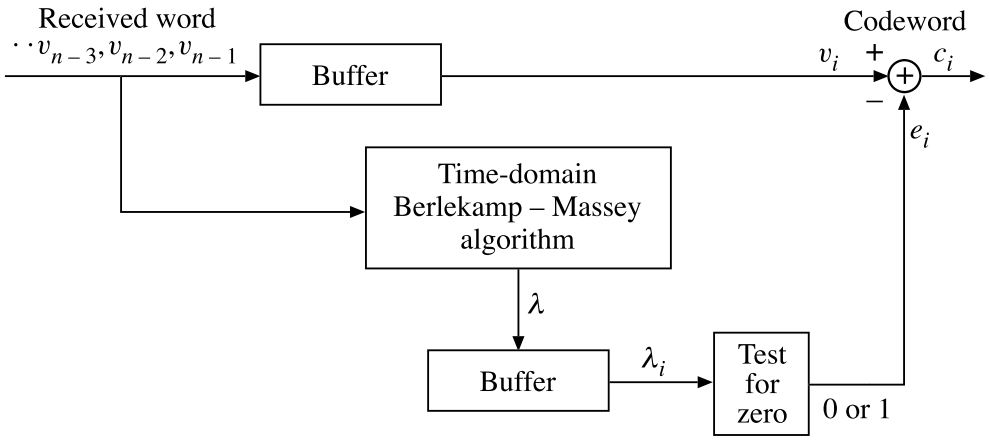


Figure 7.12. A time-domain decoder for a binary BCH code

components of $\boldsymbol{E}$ can be computed by the linear recursion

$$E_k = -\sum_{j=1}^{t} \Lambda_j E_{k-j} \quad k = 2t+1, \ldots, n-1.$$

To write the inverse Fourier transform of this equation, some restructuring is necessary. We will give the corresponding expression in the time domain in the next theorem.

Theorem 7.6.2. *Let $\boldsymbol{v} = \boldsymbol{c} + \boldsymbol{e}$ be a received noisy BCH codeword. Given the time-domain error locator $\boldsymbol{\lambda}$, the following set of equations,*

$$\Delta_r = \sum_{i=0}^{n-1} \omega^{ir} v_i^{(r-1)} \lambda_i$$

$$v_i^{(r)} = v_i^{(r-1)} - \Delta_r \omega^{-ri}$$

for $r = 2t+1, \ldots, n$, results in

$$v_i^{(n)} = e_i \quad i = 0, \ldots, n-1.$$

Proof: The equation of the recursive extension in the frequency domain is rewritten by separating it into two steps, starting with the senseword spectrum $\boldsymbol{V}$ and changing it to $\boldsymbol{E}$ – one component at a time:

$$\Delta_r = V_r - \left(-\sum_{j=1}^{n-1} \Lambda_j V_{r-j} \right) = \sum_{j=0}^{n-1} \Lambda_j V_{r-j}$$

$$V_j^{(r)} = \begin{cases} V_j^{(r-1)} & j \neq r \\ V_j^{(r-1)} - \Delta_r & j = r. \end{cases}$$

Because $V_j^{(2t)} = E_j$ for $j = 1, \ldots, 2t$ and $\Lambda_j = 0$ for $j > t$, the above equations are

equivalent to

$$E_r = -\sum_{j=1}^{n-1} \Lambda_j E_{r-j}.$$

This equivalence proves the theorem. □

A flow chart in a field of characteristic 2 for a complete time-domain decoder, including erasure filling and a single extension symbol, is shown in Figure 7.13. The

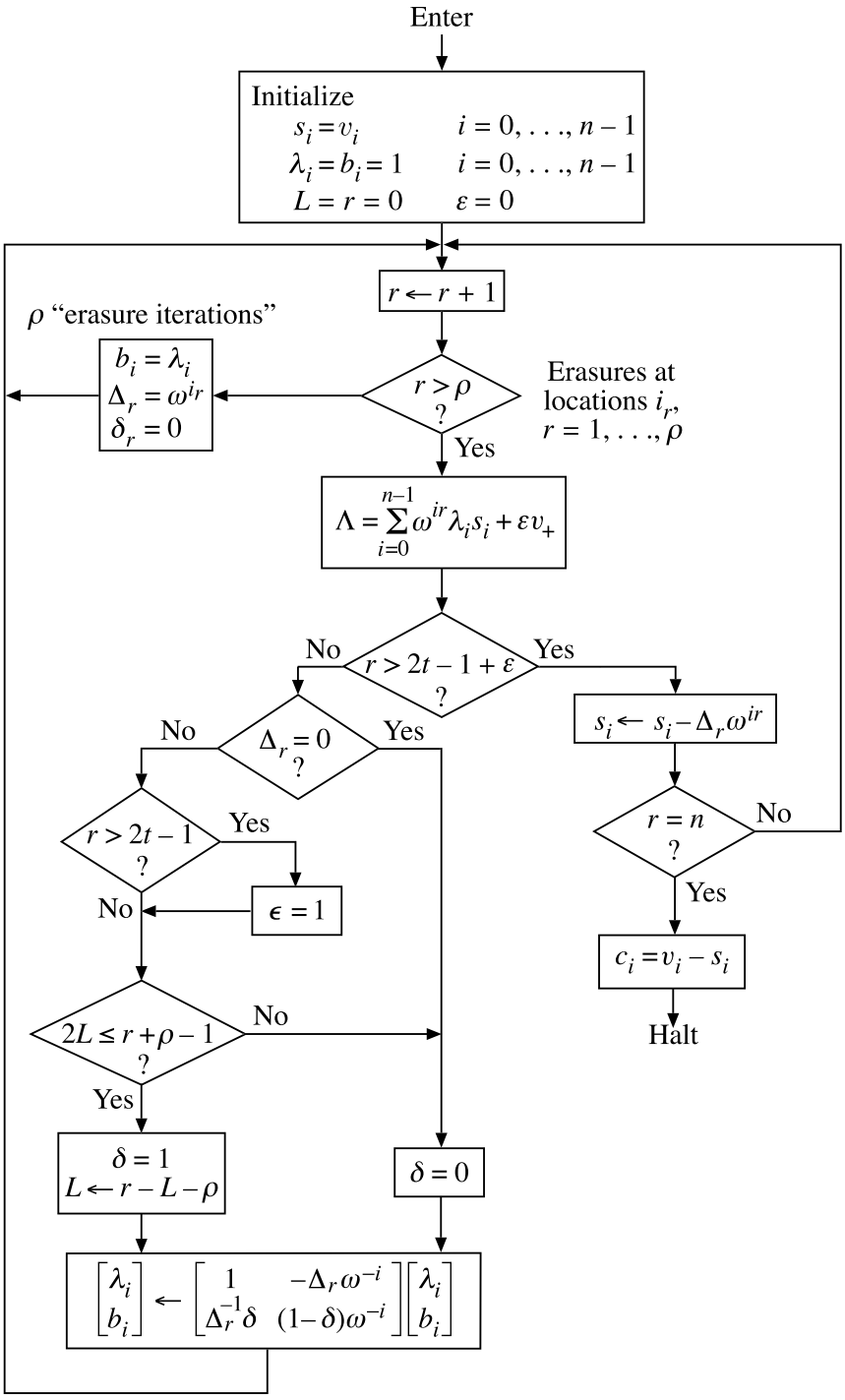


Figure 7.13. A time-domain BCH decoder

left side of the flow diagram is based on Algorithm 7.6.1, and the right side is based on Theorem 7.6.2. During the iterations on the left side of Figure 7.13, the inverse Fourier transform of the error-locator polynomial is being formed. During each pass on the right side, the senseword is changed into one component of the spectrum of the error word. Hence after the last iteration, the senseword has been changed into the error word.

The time-domain decoder can be attractive because there is only one major task in the computation. There is no syndrome computer or Chien search. This means that the decoder has a very simple structure, but the penalty is a longer running time. The number of major clocks required is n, which is independent of the number of errors corrected. During each such clock, vectors of length n are processed, and thus the complexity of the decoder is proportional to n^2. For high-rate codes of large blocklength, decoders with fewer computations can be found, but with a more complex structure. In some operations, structural simplicity may be at least as important as the number of arithmetic operations.

7.7 Decoding within the BCH bound

The decoding algorithms for BCH codes that have been developed in this chapter are bounded-distance decoders that will decode correctly whenever the number of errors does not exceed the packing radius of the code. When the distance from the senseword to the transmitted codeword is larger than the packing radius t, the senseword may lie within a distance t of another codeword, but usually the distance from the senseword to every codeword will be larger than t. Then the error pattern, though uncorrectable by a bounded-distance decoder, can be detected. This detection of an uncorrectable error pattern is called a *decoding failure*. It may be preferable to a *decoding error*, which occurs when the decoder produces the wrong codeword. In this section, we shall first discuss tests for detecting a decoding failure. Then we shall discuss making the decoding spheres smaller in order to decrease the probability of decoding error at the cost of a higher probability of decoding failure.

When the number of errors exceeds the packing radius, the various decoding algorithms can behave in different ways. In general, it is not true that the output of a decoding algorithm must always be a codeword. A particular decoding algorithm may fail to detect uncorrectable but detectable errors if this requirement was not imposed in the design of the decoding algorithm.

The most direct way of detecting uncorrectable errors is to check the decoder output, verifying that it is a true codeword and is within distance t of the decoder input. Alternatively, indirect tests can be embedded within an algorithm.