MIT 6.02 **DRAFT** *Lecture Notes*
Last update: February 28, 2012
Comments, questions or bug reports?
Please contact hari at mit.edu

CHAPTER 6
Linear Block Codes: Encoding and Syndrome Decoding

The previous chapter defined some properties of linear block codes and discussed two examples of linear block codes (rectangular parity and the Hamming code), but the approaches presented for decoding them were specific to those codes. Here, we will describe a general strategy for encoding and decoding linear block codes. The decoding procedure we describe is **syndrome decoding**, which uses the syndrome bits introduced in the previous chapter. We will show how to perform syndrome decoding efficiently for any linear block code, highlighting the primary reason why linear (block) codes are attractive: the ability to decode them efficiently.

We also discuss how to use a linear block code that works over relatively small block sizes to protect a packet (or message) made up of a much larger number of bits. Finally, we discuss how to cope with burst error patterns, which are different from the BSC model assumed thus far. A packet protected with one or more coded blocks needs a way for the receiver to *detect* errors *after* the error correction steps have done their job, because all errors may not be corrected. This task is done by an *error detection code*, which is generally distinct from the correction code. For completeness, we describe the *cyclic redundancy check (CRC)*, a popular method for error detection.

■ 6.1 Encoding Linear Block Codes

Recall that a linear block code takes k-bit message blocks and converts each such block into n-bit coded blocks. The rate of the code is k/n. The conversion in a linear block code involves only linear operations over the message bits to produce codewords. For concreteness, let's restrict ourselves to codes over $\mathbb{F}_2$, so all the linear operations are additive parity computations.

If the code is in systematic form, each codeword consists of the k message bits $D_1 D_2 \ldots D_k$ followed by the $n-k$ parity bits $P_1 P_2 \ldots P_{n-k}$, where each P_i is some linear combination of the D_i's.

Because the transformation from message bits to codewords is linear, one can represent

each message-to-codeword transformation succinctly using matrix notation:

$$D \cdot G = C, \tag{6.1}$$

where D is a $k \times 1$ matrix of message bits $D_1 D_2 \ldots D_k$, C is the n-bit codeword $C_1 C_2 \ldots C_n$, G is the $k \times n$ **generator matrix** that completely characterizes the linear block code, and $\cdot$ is the standard matrix multiplication operation. For a code over $\mathbb{F}_2$, each element of the three matrices in the above equation is 0 or 1, and all additions are modulo 2.

If the code is in systematic form, C has the form $D_1 D_2 \ldots D_k P_1 P_2 \ldots P_{n-k}$. Substituting this form into Equation 6.1, we see that G is decomposed into a $k \times k$ **identity** matrix "concatenated" horizontally with a $k \times (n-k)$ matrix of values that defines the code.

The encoding procedure for any linear block code is straightforward: given the generator matrix G, which completely characterizes the code, and a sequence of k message bits D, use Equation 6.1 to produce the desired n-bit codeword. The straightforward way of doing this matrix multiplication involves k multiplications and $k-1$ additions for each codeword bit, but for a code in systematic form, the first k codeword bits are simply the message bits themselves and can be produced with no work. Hence, we need $O(k)$ operations for each of $n-k$ parity bits in C, giving an overall encoding complexity of $O(nk)$ operations.

■ 6.1.1 Examples

To illustrate Equation 6.1, let's look at some examples. First, consider the simple linear parity code, which is a $(k+1, k)$ code. What is G in this case? The equation for the parity bit is $P = D_1 + D_2 + \ldots D_k$, so the codeword is just $D_1 D_2 \ldots D_k P$. Hence,

$$G = I_{k \times k} | 1^T, \tag{6.2}$$

where $I_{k \times k}$ is the $k \times k$ identity matrix and 1^T is a k-bit column vector of all ones (the superscript ^{T} refers to matrix transposition, i.e., make all the rows into columns and vice versa). For example, when $k = 3$,

$$G = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}.$$

Now consider the rectangular parity code from the last chapter. Suppose it has $r = 2$ rows and $c = 3$ columns, so $k = rc = 6$. The number of parity bits = $r + c = 5$, so this rectangular parity code is a $(11, 6, 3)$ linear block code. If the data bits are $D_1 D_2 D_3 D_4 D_5 D_6$ organized with the first three in the first row and the last three in the second row, the parity equations are

$$\begin{aligned} P_1 &= D_1 + D_2 + D_3 \\ P_2 &= D_4 + D_5 + D_6 \\ P_3 &= D_1 + D_4 \\ P_4 &= D_2 + D_5 \\ P_5 &= D_3 + D_6 \end{aligned}$$

Fitting these equations into Equation (6.1), we find that

$$G = \left(\begin{array}{cccccc|ccccc} 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \end{array}\right).$$

G is a $k \times n$ (here, 6×11) matrix; you can see the $k \times k$ identity matrix, followed by the remaining $k \times (n-k)$ part (we have shown the two parts separated with a vertical line). Each of the right-most $n-k$ columns corresponds one-to-one with a parity bit, and there is a "1" for each entry where the data bit of the row contributes to the corresponding parity equation. This property makes it easy to write G given the parity equations; conversely, given G for a code, it is easy to write the parity equations for the code.

Now consider the $(7,4)$ Hamming code from the previous chapter. Using the parity equations presented there, we leave it as an exercise to verify that for this code,

$$G = \left(\begin{array}{cccc|ccc} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{array}\right). \tag{6.3}$$

As a last example, suppose the parity equations for a $(6,3)$ linear block code are

$$\begin{aligned} P_1 &= D_1 + D_2 \\ P_2 &= D_2 + D_3 \\ P_3 &= D_3 + D_1 \end{aligned}$$

For this code,

$$G = \left(\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{array}\right).$$

We denote the $k \times (n-k)$ sub-matrix of G by A. I.e.,

$$G = I_{k \times k} | A, \tag{6.4}$$

where $|$ represents the horizontal "stacking" (or concatenation) of two matrices with the same number of rows.

6.2 Maximum-Likelihood (ML) Decoding

Given a binary symmetric channel with bit-flip probability ε, our goal is to develop a **maximum-likelihood** (ML) decoder. For a linear block code, an ML decoder takes n received bits as input and returns the most likely k-bit message among the 2^k possible messages.

The simple way to implement an ML decoder is to enumerate all 2^k valid codewords (each n bits in length). Then, compare the received word, r, to each of these valid codewords and find the one with **smallest Hamming distance** to r. If the BSC probability $\varepsilon < 1/2$, then the codeword with smallest Hamming distance is the ML decoding. Note that $\varepsilon < 1/2$ covers all cases of practical interest: if $\varepsilon > 1/2$, then one can simply swap all zeroes and ones and do the decoding, for that would map to a BSC with bit-flip probability $1 - \varepsilon < 1/2$. If $\varepsilon = 1/2$, then each bit is as likely to be correct as wrong, and there is no way to communicate at a non-zero rate. Fortunately, $\varepsilon << 1/2$ in all practically relevant communication channels.

The goal of ML decoding is to maximize the quantity $\mathbb{P}(r|c)$; i.e., to find the codeword c so that the probability that r was received given that c was sent is maximized. Consider any codeword $\tilde{c}$. If r and $\tilde{c}$ differ in d bits (i.e., their Hamming distance is d), then $\mathbb{P}(r|c) = \varepsilon^d(1-\varepsilon)^{N-d}$, where n is the length of the received word (and also the length of each valid codeword). It's more convenient to take the logarithm of this conditional probaility, also termed the *log-likelihood*:[1]

$$\log \mathbb{P}(r|\tilde{c}) = d \log \varepsilon + (N-d)\log(1-\varepsilon) = d \log \frac{\varepsilon}{1-\varepsilon} + N \log(1-\varepsilon). \tag{6.5}$$

If $\varepsilon < 1/2$, which is the practical realm of operation, then $\frac{\varepsilon}{1-\varepsilon} < 1$ and the log term is negative. As a result, maximizing the log likelihood boils down to minimizing d, because the second term on the RHS of Eq. (6.5) is a constant.

ML decoding by comparing a received word, r, with all 2^k possible valid n-bit codewords does work, but has exponential time complexity. What we would like is something a lot faster. Note that this "compare to all valid codewords" method does not take advantage of the linearity of the code. By taking advantage of this property, we can make the decoding a lot faster.

■ 6.3 Syndrome Decoding of Linear Block Codes

Syndrome decoding is an efficient way to decode linear block codes. We will study it in the context of decoding single-bit errors; specifically, providing the following semantics:

> If the received word has 0 or 1 errors, then the decoder will return the correct transmitted message.

If the received word has more than 0 or 1 errors, then the decoder may return the correct message, but it may also not do so (i.e., we make no guarantees). It is not difficult to extend the method described below to both provide ML decoding (i.e., to return the message corresponding to the codeword with smallest Hamming distance to the received word), and to handle block codes that can correct a greater number of errors.

The key idea is to take advantage of the linearity of the code. We first give an example, then specify the method in general. Consider the $(7, 4)$ Hamming code whose generator matrix G is given by Equation (6.3). From G, we can write out the parity equations in the

[1] The base of the logarithm doesn't matter.

same form as in the previous chapter:

$$\begin{aligned} P_1 &= D_1 + D_2 + D_4 \\ P_2 &= D_1 + D_3 + D_4 \\ P_3 &= D_2 + D_3 + D_4 \end{aligned} \tag{6.6}$$

(6.7)

Because the arithmetic is over $\mathbb{F}_2$, we can rewrite these equations by moving the P's to the same side as the D's (in modulo-2 arithmetic, there is no difference between a $-$ and a $+$ sign!):

$$\begin{aligned} D_1 + D_2 + D_4 + P_1 &= 0 \\ D_1 + D_3 + D_4 + P_2 &= 0 \\ D_2 + D_3 + D_4 + P_3 &= 0 \end{aligned} \tag{6.8}$$

(6.9)

There are $n - k$ such equations. One can express these equations, in matrix notation using a **parity check matrix**, H, as follows:

$$H \cdot [D_1 D_2 \ldots D_k P_1 P_2 \ldots P_{n-k}]^T = 0. \tag{6.10}$$

H is the horizontal stacking, or concatenation, of two matrices: A^T, where A is the submatrix of the generator matrix of the code from Equation (6.4), and $I_{(n-k)\times(n-k)}$, the identity matrix. I.e.,

$$H = A^T | I_{(n-k)\times(n-k)}, \tag{6.11}$$

where A is given by Equation (6.4).

H has the property that for any valid codeword c (which we represent as a $1 \times n$ matrix),

$$H \cdot c^T = 0. \tag{6.12}$$

Hence, for any received word r without errors, $H \cdot r^T = 0$.

Now suppose a received word r has some errors in it. r may be written as $c + e$, where c is some valid codeword and e is an *error vector*, represented (like c) as a $1 \times n$ matrix. For such an r,

$$H \cdot r^T = H \cdot (c + e)^T = 0 + H \cdot e^T.$$

If r has at most one bit error, then e is made up of all zeroes and at most one "1". In this case, there are $n + 1$ possible values of $H \cdot e^T$; n of these correspond to exactly one bit error, and one of these is a no-error case ($e = 0$), for which $H \cdot e^T = 0$. These $n + 1$ possible vectors are precisely the *syndromes* introduced in the previous chapter: they signify what happens under different error patterns.

Syndrome decoding pre-computes the syndrome corresponding to each error. Assume that the code is in systematic form, so each codeword is of the form $D_1 D_2 \ldots D_k P_1 P_2 \ldots P_{n-k}$. If $e = 100 \ldots 0$, then the syndrome $H \cdot e^T$ is the result when the first data bit, D_1 is in error. In general, if element i of e is 1 and the other elements are 0, the resulting syndrome $H \cdot e^T$ corresponds to the case when bit i in the codeword is

wrong. Under the assumption that there is at most one bit error, we care about storing the syndromes when one of the first k elements of e is 1.

Given a received word, r, the decoder computes $H \cdot r^T$. If it is 0, then there are no single-bit errors, and the receiver returns the first k bits of the received word as the decoded message. If not, then it compares that $(n-k)$-bit value with each of the k stored syndromes. If syndrome j matches, then it means that data bit j in the received word was in error, and the decoder flips that bit and returns the first k bits of the received word as the most likely message that was encoded and transmitted.

If $H \cdot r^T$ is not all zeroes, and if it does not match any stored syndrome, then the decoder concludes that either some parity bit was wrong, or that there were multiple errors. In this case, it might simply return the first k bits of the received word as the message. This method produces the ML decoding if a parity bit was wrong, but may not be the optimal estimate when multiple errors occur. Because we are likely to use single-error correction in cases when the probability of multiple bit errors is extremely low, we can often avoid doing anything more sophisticated than just returning the first k bits of the received word as the decoded message.

The preceding two paragraphs provide the essential steps behind syndrome decoding for single bit errors, producing an ML estimate of the transmitted message in the case when zero or one bit errors affect the codeword.

Correcting multiple errors. It is not hard to expand this syndrome decoding idea to the multiple error case. Suppose we wish to correct all patterns of $\leq t$ errors. In this case, we need to pre-compute more syndromes, corresponding to $0, 1, 2, \ldots t$ bit errors. Each of these should be stored by the decoder. There will be a total of

$$\binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \ldots \binom{n}{t}$$

syndromes to pre-compute and store. If one of these syndromes matches, the decoder knows exactly which bit error pattern produced the syndrome, and it flips those bits and returns the first k bits of the codeword as the decoded message. This method requires the decoder to make $O(n^t)$ syndrome comparisons, and each such comparison involved comparing two $(n-k)$-bit strings with each other.

An example. A detailed example may be useful to understand the encoding and decoding procedures. Consider the (7,4) Hamming code. The G for this linear block code is specified in Equation (6.3). Given any $k = 4$-bit message m, the encoder produces an $n = 7$-bit codeword, c by multiplying $m \cdot G$. (m is a $1 \times k$ matrix, G is a $k \times n$ matrix, and c is a $1 \times n$ matrix.)

The parity check matrix, H, for this code is obtained by applying Equation (6.11):

$$H = \left(\begin{array}{cccc|ccc} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{array}\right). \tag{6.13}$$

Suppose c is sent over the channel and is received by the decoder as r. For concreteness,

suppose $c = 1010001$ and $r = 1110001$ (error in the second bit).

The decoder pre-computes syndromes corresponding to all possible single-bit errors. (It actually needs to pre-compute only k of them, each corresponding to an error in one of the first k bit positions of a codeword.) In our case, the $k = 4$ syndromes of interest are:

$$\begin{aligned} H \cdot [1000000]^T &= [110]^T \\ H \cdot [0100000]^T &= [101]^T \\ H \cdot [0010000]^T &= [011]^T \\ H \cdot [0001000]^T &= [111]^T \end{aligned}$$

For completeness, the syndromes for a single-bit error in one of the parity bits are, not surprisingly:

$$\begin{aligned} H \cdot [0000100]^T &= [100]^T \\ H \cdot [0000010]^T &= [010]^T \\ H \cdot [0000001]^T &= [001]^T \end{aligned}$$

The decoder implements the following steps to correct single-bit errors:

1. Compute $c' = H \cdot r^T$ (remembering to replace each value with its modulo-2 value). In this example, $c' = [101]^T$.

2. If c' is 0, then return the first k bits of r as the message. In this example c' is not 0.

3. If c' is not 0, then compare c' with the n pre-computed syndromes, $H \cdot e_i$, where $e_i = [00 \ldots 1 \ldots 0]$ is a $1 \times n$ matrix with 1 in position i and 0 everywhere else.

4. If there is a match in the previous step for error vector e_ℓ, then bit position ℓ in the received word is in error. Flip that bit and return the first k elements of r (note that we need to perform this check only for the first k error vectors because only one of those may need to be flipped, which is why it is sufficient to only store k single-error syndromes and not n.

   In this example, the syndrome for $H \cdot [0100000]^T = [101]^T$, which matches $c' = H \cdot r^T$. Hence, the decoder flips the second bit in the received word and returns the first $k = 4$ bits of r as the ML decoding. In this example, the returned estimate of the message is [1010].

5. If there is no match, return the first k bits of r. Doing so is not necessarily ML decoding when multiple bit errors occur, *but* if the bit error probability is small, then it is a very good approximation. It is unlikely that doing full-fledged ML decoding in this case is worth the effort in terms of reduced bit error rate of a packet made up of many such coded blocks.

■ 6.4 Protecting Longer Messages with SEC Codes

SEC codes are a good building block, but they correct at most one error in a block of n coded bits. As messages get longer, the solution, of course, is to break up a longer message

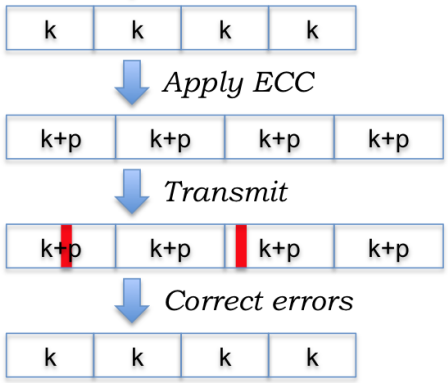

Figure 6-1: Dividing a long message into multiple SEC-protected blocks of k bits each, adding parity bits to each constituent block. The red vertical rectangles refer to bit errors.

into smaller blocks of k bits each, and to protect each one with its own SEC code. The result might look as shown in Figure 6-1. In addition, one would introduce an error detection code (like a CRC) at the end of the packet, as described in Section 6.6.

■ 6.5 Coping with Burst Errors

Over many channels, errors occur in bursts and the BSC error model is invalid. For example, wireless channels suffer from *interference* from other transmitters and from *fading*, caused mainly by *multi-path propagation* when a given signal arrives at the receiver from multiple paths and interferes in complex ways because the different copies of the signal experience different degrees of attenuation and different delays. Another reason for fading is the presence of obstacles on the path between sender and receiver; such fading is called *shadow fading*.

The behavior of a fading channel is complicated and beyond our current scope of discussion, but the impact of fading on communication is that the random process describing the bit error probability is no longer independent and identically distributed from one bit to another. The BSC model needs to be replaced with a more complicated one in which errors may occur in *bursts*. Many such theoretical models guided by empirical data exist, but we won't go into them here. Our goal is to understand how to develop error correction mechanisms when errors occur in bursts.

But what do we mean by a "burst"? The simplest model is to model the channel as having two states, a "good" state and a "bad" state. In the "good" state, the bit error probability is p_g and in the "bad" state, it is $p_b > p_g$. Once in the good state, the channel has some probability of remaining there (generally $> 1/2$) and some probability of moving into the "bad" state, and vice versa. It should be easy to see that this simple model has the property that the probability of a bit error depends on whether the previous bit (or previous few bits) are in error or not. The reason is that the odds of being in a "good" state are high if the previous few bits have been correct.

At first sight, it might seem like the block codes that correct one (or a small number of) bit errors are poorly suited for a channel experiencing burst errors. The reason is shown in Figure 6-2 (left), where each block of the message is protected by its SEC parity bits. The

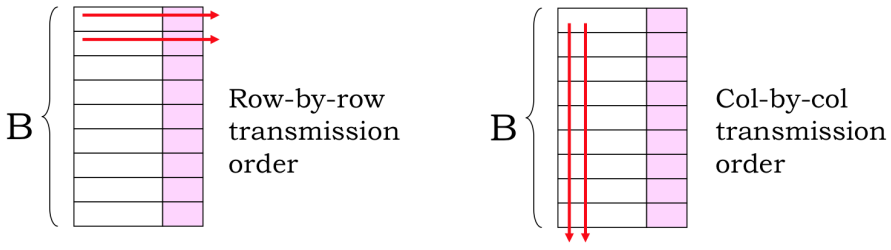

Figure 6-2: Interleaving can help recover from burst errors: code each block row-wise with an SEC, but transmit them in interleaved fashion in columnar order. As long as a set of burst errors corrupts some set of k^{th} bits, the receiver can recover from *all* the errors in the burst.

different blocks are shown as different rows. When a burst error occurs, multiple bits in an SEC block are corrupted, and the SEC can't recover from them.

Interleaving is a commonly used technique to recover from burst errors on a channel even when the individual blocks are protected with a code that, on the face of it, is not suited for burst errors. The idea is simple: code the blocks as before, but transmit them in a "columnar" fashion, as shown in Figure 6-2 (right). That is, send the first bit of block 1, then the first bit of block 2, and so on until all the first bits of each block in a set of some predefined size are sent. Then, send the second bits of each block in sequence, then the third bits, and so on.

What happens on a burst error? Chances are that it corrupts a set of "first" bits, or a set of "second" bits, or a set of "third" bits, etc., because those are the bits sent in order on the channel. As long as only a set of k^{th} bits are corrupted, the receiver can correct *all* the errors. The reason is that each coded block will now have at most one error. Thus, block codes that correct a small number of bit errors per block are still a useful primitive to correct burst errors, when used in concert with interleaving.

■ 6.6 Error Detection

This section is optional reading and is not required for 6.02 in Spring 2012.

The reason why error detection is important is that no practical error correction schemes can perfectly correct all errors in a message. For example, any reasonable error correction scheme that can correct all patterns of t or fewer errors will have some error pattern of t or more errors that cannot be corrected. Our goal is not to eliminate all errors, but to reduce the bit error rate to a low enough value that the occasional corrupted coded message is not a problem: the receiver can just discard such messages and perhaps request a retransmission from the sender (we will study such retransmission protocols later in the term). To decide whether to keep or discard a message, the receiver needs a way to detect any errors that might remain after the error correction and decoding schemes have done their job: this task is done by an error detection scheme.

An error detection scheme works as follows. The sender takes the message and produces a compact *hash* or *digest* of the message; i.e., a function that takes the message as input and produces a unique bit-string. The idea is that commonly occurring corruptions of the message will cause the hash to be different from the correct value. The sender includes

the hash with the message, and then passes that over to the error correcting mechanisms, which code the message. The receiver gets the coded bits, runs the error correction decoding steps, and then obtains the presumptive set of original message bits and the hash. The receiver computes the same hash over the presumptive message bits and compares the result with the presumptive hash it has decoded. If the results disagree, then clearly there has been some unrecoverable error, and the message is discarded. If the results agree, then the receiver believes the message to be correct. Note that if the results agree, the receiver can only *believe* the message to be correct; it is certainly possible (though, for good detection schemes, unlikely) for two different message bit sequences to have the same hash.

The design of an error detection method depends on the errors we anticipate. If the errors are adversarial in nature, e.g., from a malicious party who can change the bits as they are sent over the channel, then the hash function must guard against as many of the enormous number of different error patterns that might occur. This task requires cryptographic protection, and is done in practice using schemes like SHA-1, the secure hash algorithm. We won't study these here, focusing instead on non-malicious, random errors introduced when bits are sent over communication channels. The error detection hash functions in this case are typically called *checksums*: they protect against certain random forms of bit errors, but are by no means the method to use when communicating over an insecure channel.

The most common packet-level error detection method used today is the Cyclic Redundancy Check (CRC).[2] A CRC is an example of a block code, but it can operate on blocks of any size. Given a message block of size k bits, it produces a compact digest of size r bits, where r is a constant (typically between 8 and 32 bits in real implementations). Together, the $k + r = n$ bits constitute a **code word**. Every valid code word has a certain minimum Hamming distance from every other valid code word to aid in error detection.

A CRC is an example of a *polynomial code* as well as an example of a *cyclic code*. The idea in a polynomial code is to represent every code word $w = w_{n-1}w_{n-2}w_{n-2}\ldots w_0$ as a polynomial of degree $n-1$. That is, we write

$$w(x) = \sum_{i=0}^{n-1} w_i x^i. \tag{6.14}$$

For example, the code word 11000101 may be represented as the polynomial $1 + x^2 + x^6 + x^7$, plugging the bits into Eq.(6.14) and reading out the bits from right to left. We use the term *code polynomial* to refer to the polynomial corresponding to a code word.

The key idea in a CRC (and, indeed, in any cyclic code) is to ensure that *every valid code polynomial is a multiple of a generator polynomial*, $g(x)$. We will look at the properties of good generator polynomials in a bit, but for now let's look at some properties of codes built with this property. The key idea is that we're going to take a message polynomial and divide it by the generator polynomial; the (coefficients of) the remainder polynomial from the division will correspond to the hash (i.e., the bits of the checksum).

All arithmetic in our CRC will be done in $\mathbb{F}_2$. The normal rules of polynomial addition,

[2]Sometimes, the literature uses "checksums" to mean something different from a "CRC", using checksums for methods that involve the addition of groups of bits to produce the result, and CRCs for methods that involve polynomial division. We use the term "checksum" to include both kinds of functions, which are both applicable to random errors and not to insecure channels (unlike secure hash functions).

subtraction, multiplication, and division apply, except that all coefficients are either 0 or 1 and the coefficients add and multiply using the $\mathbb{F}_2$ rules. In particular, note that all minus signs can be replaced with plus signs, making life quite convenient.

6.6.1 Encoding Step

The CRC encoding step of producing the digest is simple. Given a message, construct the message polynomial $m(x)$ using the same method as Eq.(6.14). Then, our goal is to construct the code polynomial, $w(x)$ by combining $m(x)$ and $g(x)$ so that $g(x)$ divides $w(x)$ (i.e., $w(x)$ is a multiple of $g(x)$).

First, let us multiply $m(x)$ by x^{n-k}. The reason we do this multiplication is to shift the message left by $n-k$ bits, so we can add the redundant check bits ($n-k$ of them) so that the code word is in systematic form. It should be easy to verify that this multiplication produces a polynomial whose coefficients correspond to original message bits followed by all zeroes (for the check bits we're going to add in below).

Then, let's divide $x^{n-k}m(x)$ by $g(x)$. If the remainder from the polynomial division is 0, then we have a valid codeword. Otherwise, we have a remainder. We know that if we subtract this remainder from the polynomial $x^{n-k}m(x)$, we will obtain a new polynomial that will be a multiple of $g(x)$. Remembering that we are in $\mathbb{F}_2$, we can replace the subtraction with an addition, getting:

$$w(x) = x^{n-k}m(x) + x^{n-k}m(x) \bmod g(x), \tag{6.15}$$

where the notation $a(x) \bmod b(x)$ stands for the remainder when $a(x)$ is divided by $b(x)$.

The encoder is now straightforward to define. Take the message, construct the message polynomial, multiply by x^{n-k}, and then divide that by $g(x)$. The remainder forms the check bits, acting as the digest for the entire message. Send these bits appended to the message.

6.6.2 Decoding Step

The decoding step is essentially identical to the encoding step, one of the advantages of using a CRC. Separate each code word received into the message and remainder portions, and verify whether the remainder calculated from the message matches the bits sent together with the message. A mismatch guarantees that an error has occurred; a match suggests a reasonable likelihood of the message being correct, *as long as a suitable generator polynomial is used*.

6.6.3 Mechanics of division

There are several efficient ways to implement the division and remaindering operations needed in a CRC computation. The schemes used in practice essentially mimic the "long division" strategies one learns in elementary school. Figure 6-3 shows an example to refresh your memory!

6.6.4 Good Generator Polynomials

So how should one pick good generator polynomials? There is no magic prescription here, but ny observing what commonly occuring error patterns do to the received code

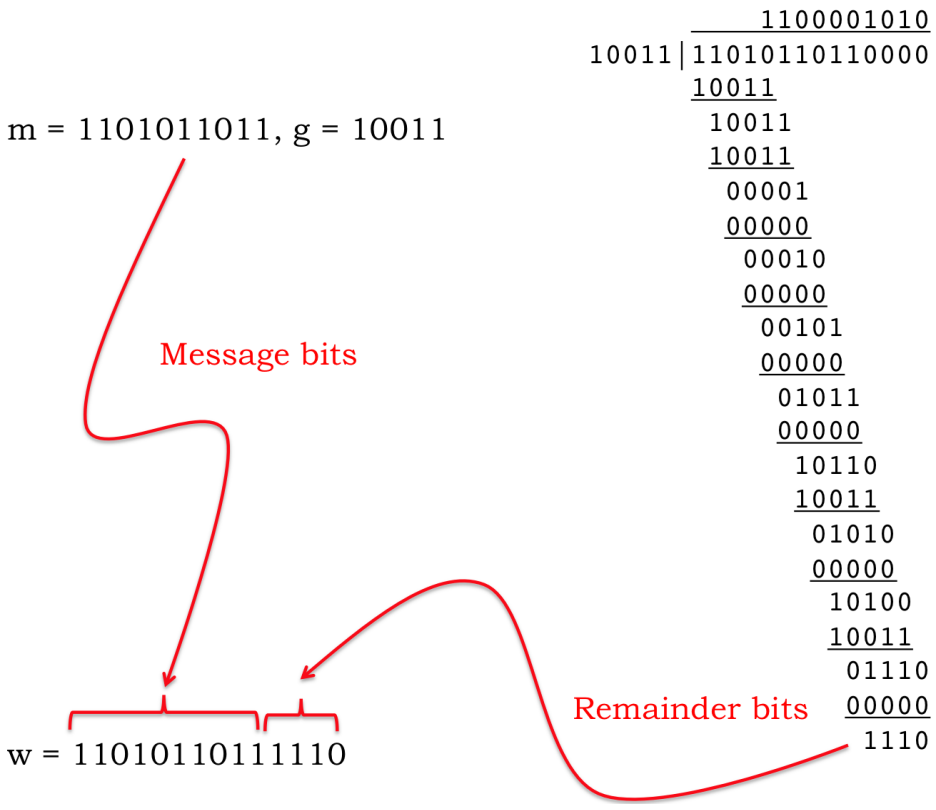

Figure 6-3: CRC computations using "long division".

words, we can form some guidelines. To develop suitable properties for $g(x)$, first observe that if the receiver gets a bit sequence, we can think of it as the code word sent added to a sequence of zero or more errors. That is, take the bits obtained by the receiver and construct a received polynomial, $r(x)$, from it. We can think of $r(x)$ as being the sum of $w(x)$, which is what the sender sent (the receiver doesn't know what the real w was) and an *error polynomial*, $e(x)$. Figure 6-4 shows an example of a message with two bit errors and the corresponding error polynomial. Here's the key point: If $r(x) = w(x) + e(x)$ is *not* a multiple of $g(x)$, then the receiver is *guaranteed* to detect the error. Because $w(x)$ is constructed as a multiple of $g(x)$, this statement is the same as saying that if $e(x)$ is not a multiple of $g(x)$, the receiver is guaranteed to detect the error. On the other hand, if $r(x)$, and therefore $e(x)$, *is* a multiple of $g(x)$, then we either have no errors, or we have an error that we cannot detect (i.e., an erroneous reception that we falsely identify as correct). Our goal is to ensure that this situation does not happen for commonly occurring error patterns.

1. First, note that for single error patterns, $e(x) = x^i$ for some i. That means we must ensure that $g(x)$ has at least two terms.

2. Suppose we want to be able to detect all error patterns with two errors. That error pattern may be written as $x^i + x^j = x^i(1 + x^{j-i})$, for some i and $j > i$. If $g(x)$ does not divide this term, then the resulting CRC can detect all double errors.

3. Now suppose we want to detect all odd numbers of errors. If $(1 + x)$ is a factor of

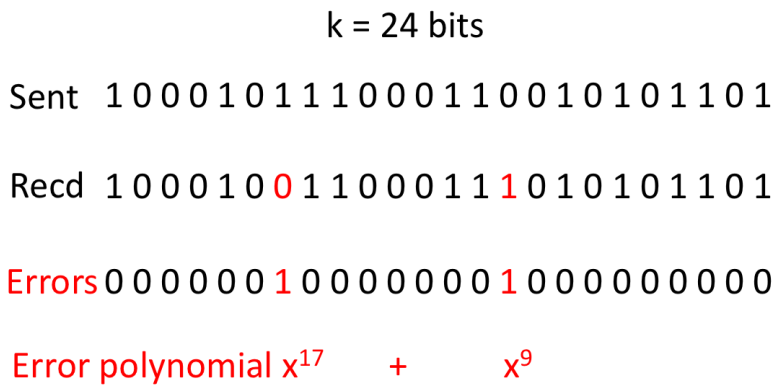

Figure 6-4: Error polynomial example with two bit errors; the polynomial has two non-zero terms corresponding to the locations where the errors have occurred.

$g(x)$, then $g(x)$ must have an *even number of terms*. The reason is that any polynomial with coefficients in $\mathbb{F}_2$ of the form $(1+x)h(x)$ must evaluate to 0 when we set x to 1. If we expand $(1+x)h(x)$, if the answer must be 0 when $x=1$, the expansion must have an even number of terms. Therefore, if we make $1+x$ a factor of $g(x)$, the resulting CRC will be *able to detect all error patterns with an odd number of errors*. Note, however, that the converse statement is not true: a CRC may be able to detect an odd number of errors even when its $g(x)$ is not a multiple of $(1+x)$. But all CRCs used in practice do have $(1+x)$ as a factor because its the simplest way to achieve this goal.

4. Another guideline used by some CRC schemes in practice is the ability to detect *burst errors*. Let us define a burst error pattern of length b as a sequence of bits $1\varepsilon_{b-2}\varepsilon_{b-3}\ldots\varepsilon_1 1$: that is, the number of bits is b, the first and last bits are both 1, and the bits ε_i in the middle could be either 0 or 1. The minimum burst length is 2, corresponding to the pattern "11".

   Suppose we would like our CRC to detect all such error patterns, where $e(x) = x^s(1 \cdot x^{b-1} + \sum_{i=1}^{b-2}\varepsilon_i x^i + 1)$. This polynomial represents a burst error pattern of size b starting s bits to the left from the end of the packet. If we pick $g(x)$ to be a polynomial of degree b, and if $g(x)$ does not have x as a factor, then any error pattern of length $\leq b$ is guaranteed to be detected, because $g(x)$ will not divide a polynomial of degree smaller than its own. Moreover, there is exactly one error pattern of length $b+1$—corresponding to the case when the burst error pattern matches the coefficients of $g(x)$ itself—that will not be detected. All other error patterns of length $b+1$ will be detected by this CRC.

   If fact, such a CRC is quite good at detecting longer burst errors as well, though it cannot detect all of them.

CRCs are *cyclic* codes, which have the property that if c is a code word, then any cyclic shift (rotation) of c is another valid code word. Hence, referring to Eq.(6.14), we find that

CRC-1: $x + 1$ (parity bit)
CRC-5-EPC: $x^5 + x^3 + 1$ (Gen 2 RFID)
CRC-8-WCDMA: $x^8 + x^7 + x^4 + x^3 + x + 1$
CRC-15-CAN: $x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1$ (Controller Area Network in vehicles)
CRC-16-ANSI: $x^{16} + x^{15} + x^2 + 1$ (USB, etc.)
CRC-16-CCITT: $x^{16} + x^{12} + x^5 + 1$ (Bluetooth, etc.)
CRC-16-DECT: $x^{16} + x^{10} + x^8 + x^7 + x^3 + 1$ (cordless phones)
CRC-32-IEEE: $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$ (Ethernet, WiFi, POSIX cksum, etc.)

Figure 6-5: Commonly used CRC generator polynomials, $g(x)$. From Wikipedia.

one can represent the polynomial corresponding to one cyclic left shift of w as

$$w^{(1)}(x) = w_{n-1} + w_0 x + w_1 x^2 + \ldots w_{n-2} x^{n-1} \quad (6.16)$$
$$= xw(x) + (1 + x^n) w_{n-1} \quad (6.17)$$

Now, because $w^{(1)}(x)$ must also be a valid code word, it must be a multiple of $g(x)$, which means that $g(x)$ must divide $1 + x^n$. Note that $1 + x^n$ corresponds to a double error pattern; what this observation implies is that the CRC scheme using cyclic code polynomials can detect the errors we want to detect (such as all double bit errors) as long as $g(x)$ is picked so that the smallest n for which $1 + x^n$ is a multiple of $g(x)$ is quite large. For example, in practice, a common 16-bit CRC has a $g(x)$ for which the smallest such value of n is $2^{15} - 1 = 32767$, which means that it's quite effective for all messages of length smaller than that.

6.6.5 CRCs in practice

CRCs are used in essentially all communication systems. The table in Figure 6-5, culled from Wikipedia, has a list of common CRCs and practical systems in which they are used. You can see that they all have an even number of terms, and verify (if you wish) that $1 + x$ divides most of them.

6.7 Summary

This chapter described syndrome decoding of linear block codes, described how to divide a packet into one or more blocks and protect each block using an error correction code, and described how interleaving can handle some burst error patterns. We then showed how error detection using CRCs can be done.

The next two chapters describe the encoding and decoding of convolutional codes, a different kind of error correction code that does not require fixed-length blocks.