

blasts the data at the poor helpless phone until it is completely swamped. Even if the transmission is error free, the receiver may be unable to handle the frames as fast as they arrive and will lose some.

Clearly, something has to be done to prevent this situation. Two approaches are commonly used. In the first one, **feedback-based flow control**, the receiver sends back information to the sender giving it permission to send more data, or at least telling the sender how the receiver is doing. In the second one, **rate-based flow control**, the protocol has a built-in mechanism that limits the rate at which senders may transmit data, without using feedback from the receiver.

In this chapter we will study feedback-based flow control schemes, primarily because rate-based schemes are only seen as part of the transport layer (Chap. 5). Feedback-based schemes are seen at both the link layer and higher layers. The latter is more common these days, in which case the link layer hardware is designed to run fast enough that it does not cause loss. For example, hardware implementations of the link layer as **NICs (Network Interface Cards)** are sometimes said to run at “wire speed,” meaning that they can handle frames as fast as they can arrive on the link. Any overruns are then not a link problem, so they are handled by higher layers.

Various feedback-based flow control schemes are known, but most of them use the same basic principle. The protocol contains well-defined rules about when a sender may transmit the next frame. These rules often prohibit frames from being sent until the receiver has granted permission, either implicitly or explicitly. For example, when a connection is set up the receiver might say: “You may send me n frames now, but after they have been sent, do not send any more until I have told you to continue.” We will examine the details shortly.

3.2 ERROR DETECTION AND CORRECTION

We saw in Chap. 2 that communication channels have a range of characteristics. Some channels, like optical fiber in telecommunications networks, have tiny error rates so that transmission errors are a rare occurrence. But other channels, especially wireless links and aging local loops, have error rates that are orders of magnitude larger. For these links, transmission errors are the norm. They cannot be avoided at a reasonable expense or cost in terms of performance. The conclusion is that transmission errors are here to stay. We have to learn how to deal with them.

Network designers have developed two basic strategies for dealing with errors. Both add redundant information to the data that is sent. One strategy is to include enough redundant information to enable the receiver to deduce what the transmitted data must have been. The other is to include only enough redundancy to allow the receiver to deduce that an error has occurred (but not which error)

and have it request a retransmission. The former strategy uses **error-correcting codes** and the latter uses **error-detecting codes**. The use of error-correcting codes is often referred to as **FEC (Forward Error Correction)**.

Each of these techniques occupies a different ecological niche. On channels that are highly reliable, such as fiber, it is cheaper to use an error-detecting code and just retransmit the occasional block found to be faulty. However, on channels such as wireless links that make many errors, it is better to add redundancy to each block so that the receiver is able to figure out what the originally transmitted block was. FEC is used on noisy channels because retransmissions are just as likely to be in error as the first transmission.

A key consideration for these codes is the type of errors that are likely to occur. Neither error-correcting codes nor error-detecting codes can handle all possible errors since the redundant bits that offer protection are as likely to be received in error as the data bits (which can compromise their protection). It would be nice if the channel treated redundant bits differently than data bits, but it does not. They are all just bits to the channel. This means that to avoid undetected errors the code must be strong enough to handle the expected errors.

One model is that errors are caused by extreme values of thermal noise that overwhelm the signal briefly and occasionally, giving rise to isolated single-bit errors. Another model is that errors tend to come in bursts rather than singly. This model follows from the physical processes that generate them—such as a deep fade on a wireless channel or transient electrical interference on a wired channel/

Both models matter in practice, and they have different trade-offs. Having the errors come in bursts has both advantages and disadvantages over isolated single-bit errors. On the advantage side, computer data are always sent in blocks of bits. Suppose that the block size was 1000 bits and the error rate was 0.001 per bit. If errors were independent, most blocks would contain an error. If the errors came in bursts of 100, however, only one block in 100 would be affected, on average. The disadvantage of burst errors is that when they do occur they are much harder to correct than isolated errors.

Other types of errors also exist. Sometimes, the location of an error will be known, perhaps because the physical layer received an analog signal that was far from the expected value for a 0 or 1 and declared the bit to be lost. This situation is called an **erasure channel**. It is easier to correct errors in erasure channels than in channels that flip bits because even if the value of the bit has been lost, at least we know which bit is in error. However, we often do not have the benefit of erasures.

We will examine both error-correcting codes and error-detecting codes next. Please keep two points in mind, though. First, we cover these codes in the link layer because this is the first place that we have run up against the problem of reliably transmitting groups of bits. However, the codes are widely used because reliability is an overall concern. Error-correcting codes are also seen in the physical layer, particularly for noisy channels, and in higher layers, particularly for

real-time media and content distribution. Error-detecting codes are commonly used in link, network, and transport layers.

The second point to bear in mind is that error codes are applied mathematics. Unless you are particularly adept at Galois fields or the properties of sparse matrices, you should get codes with good properties from a reliable source rather than making up your own. In fact, this is what many protocol standards do, with the same codes coming up again and again. In the material below, we will study a simple code in detail and then briefly describe advanced codes. In this way, we can understand the trade-offs from the simple code and talk about the codes that are used in practice via the advanced codes.

3.2.1 Error-Correcting Codes

We will examine four different error-correcting codes:

1. Hamming codes.
2. Binary convolutional codes.
3. Reed-Solomon codes.
4. Low-Density Parity Check codes.

All of these codes add redundancy to the information that is sent. A frame consists of m data (i.e., message) bits and r redundant (i.e. check) bits. In a **block code**, the r check bits are computed solely as a function of the m data bits with which they are associated, as though the m bits were looked up in a large table to find their corresponding r check bits. In a **systematic code**, the m data bits are sent directly, along with the check bits, rather than being encoded themselves before they are sent. In a **linear code**, the r check bits are computed as a linear function of the m data bits. Exclusive OR (XOR) or modulo 2 addition is a popular choice. This means that encoding can be done with operations such as matrix multiplications or simple logic circuits. The codes we will look at in this section are linear, systematic block codes unless otherwise noted.

Let the total length of a block be n (i.e., $n = m + r$). We will describe this as an (n, m) code. An n -bit unit containing data and check bits is referred to as an n -bit **codeword**. The **code rate**, or simply rate, is the fraction of the codeword that carries information that is not redundant, or m/n . The rates used in practice vary widely. They might be $1/2$ for a noisy channel, in which case half of the received information is redundant, or close to 1 for a high-quality channel, with only a small number of check bits added to a large message.

To understand how errors can be handled, it is necessary to first look closely at what an error really is. Given any two codewords that may be transmitted or received—say, 10001001 and 10110001—it is possible to determine how many

corresponding bits differ. In this case, 3 bits differ. To determine how many bits differ, just XOR the two codewords and count the number of 1 bits in the result. For example:

```

10001001
10110001
-----
00111000

```

The number of bit positions in which two codewords differ is called the **Hamming distance** (Hamming, 1950). Its significance is that if two codewords are a Hamming distance d apart, it will require d single-bit errors to convert one into the other.

Given the algorithm for computing the check bits, it is possible to construct a complete list of the legal codewords, and from this list to find the two codewords with the smallest Hamming distance. This distance is the Hamming distance of the complete code.

In most data transmission applications, all 2^m possible data messages are legal, but due to the way the check bits are computed, not all of the 2^n possible codewords are used. In fact, when there are r check bits, only the small fraction of $2^m/2^n$ or $1/2^r$ of the possible messages will be legal codewords. It is the sparseness with which the message is embedded in the space of codewords that allows the receiver to detect and correct errors.

The error-detecting and error-correcting properties of a block code depend on its Hamming distance. To reliably detect d errors, you need a distance $d + 1$ code because with such a code there is no way that d single-bit errors can change a valid codeword into another valid codeword. When the receiver sees an illegal codeword, it can tell that a transmission error has occurred. Similarly, to correct d errors, you need a distance $2d + 1$ code because that way the legal codewords are so far apart that even with d changes the original codeword is still closer than any other codeword. This means the original codeword can be uniquely determined based on the assumption that a larger number of errors are less likely.

As a simple example of an error-correcting code, consider a code with only four valid codewords:

000000000, 0000011111, 1111100000, and 1111111111

This code has a distance of 5, which means that it can correct double errors or detect quadruple errors. If the codeword 0000000111 arrives and we expect only single- or double-bit errors, the receiver will know that the original must have been 0000011111. If, however, a triple error changes 0000000000 into 0000000111, the error will not be corrected properly. Alternatively, if we expect all of these errors, we can detect them. None of the received codewords are legal codewords so an error must have occurred. It should be apparent that in this example we cannot both correct double errors and detect quadruple errors because this would require us to interpret a received codeword in two different ways.

In our example, the task of decoding by finding the legal codeword that is closest to the received codeword can be done by inspection. Unfortunately, in the most general case where all codewords need to be evaluated as candidates, this task can be a time-consuming search. Instead, practical codes are designed so that they admit shortcuts to find what was likely the original codeword.

Imagine that we want to design a code with m message bits and r check bits that will allow all single errors to be corrected. Each of the 2^m legal messages has n illegal codewords at a distance of 1 from it. These are formed by systematically inverting each of the n bits in the n -bit codeword formed from it. Thus, each of the 2^m legal messages requires $n + 1$ bit patterns dedicated to it. Since the total number of bit patterns is 2^n , we must have $(n + 1)2^m \leq 2^n$. Using $n = m + r$, this requirement becomes

$$(m + r + 1) \leq 2^r \tag{3-1}$$

Given m , this puts a lower limit on the number of check bits needed to correct single errors.

This theoretical lower limit can, in fact, be achieved using a method due to Hamming (1950). In **Hamming codes** the bits of the codeword are numbered consecutively, starting with bit 1 at the left end, bit 2 to its immediate right, and so on. The bits that are powers of 2 (1, 2, 4, 8, 16, etc.) are check bits. The rest (3, 5, 6, 7, 9, etc.) are filled up with the m data bits. This pattern is shown for an (11,7) Hamming code with 7 data bits and 4 check bits in Fig. 3-6. Each check bit forces the modulo 2 sum, or parity, of some collection of bits, including itself, to be even (or odd). A bit may be included in several check bit computations. To see which check bits the data bit in position k contributes to, rewrite k as a sum of powers of 2. For example, $11 = 1 + 2 + 8$ and $29 = 1 + 4 + 8 + 16$. A bit is checked by just those check bits occurring in its expansion (e.g., bit 11 is checked by bits 1, 2, and 8). In the example, the check bits are computed for even parity sums for a message that is the ASCII letter "A."

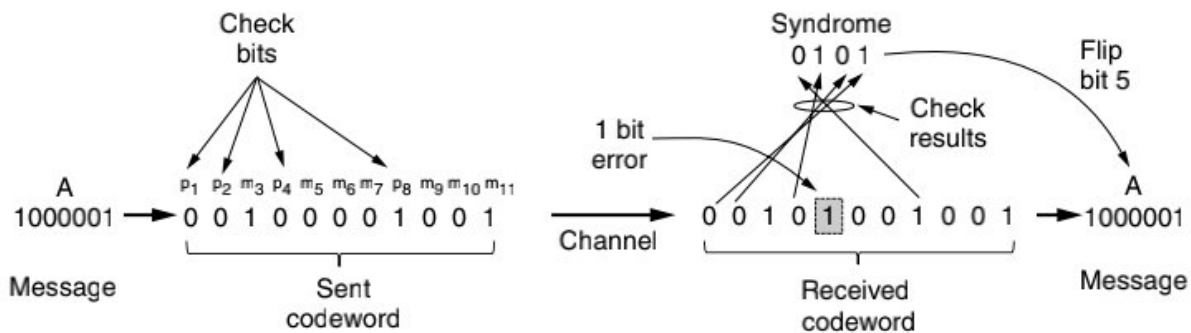


Figure 3-6. Example of an (11, 7) Hamming code correcting a single-bit error.

This construction gives a code with a Hamming distance of 3, which means that it can correct single errors (or detect double errors). The reason for the very careful numbering of message and check bits becomes apparent in the decoding

process. When a codeword arrives, the receiver redoes the check bit computations including the values of the received check bits. We call these the check results. If the check bits are correct then, for even parity sums, each check result should be zero. In this case the codeword is accepted as valid.

If the check results are not all zero, however, an error has been detected. The set of check results forms the **error syndrome** that is used to pinpoint and correct the error. In Fig. 3-6, a single-bit error occurred on the channel so the check results are 0, 1, 0, and 1 for $k = 8, 4, 2,$ and 1, respectively. This gives a syndrome of 0101 or $4 + 1 = 5$. By the design of the scheme, this means that the fifth bit is in error. Flipping the incorrect bit (which might be a check bit or a data bit) and discarding the check bits gives the correct message of an ASCII "A."

Hamming distances are valuable for understanding block codes, and Hamming codes are used in error-correcting memory. However, most networks use stronger codes. The second code we will look at is a **convolutional code**. This code is the only one we will cover that is not a block code. In a convolutional code, an encoder processes a sequence of input bits and generates a sequence of output bits. There is no natural message size or encoding boundary as in a block code. The output depends on the current and previous input bits. That is, the encoder has memory. The number of previous bits on which the output depends is called the **constraint length** of the code. Convolutional codes are specified in terms of their rate and constraint length.

Convolutional codes are widely used in deployed networks, for example, as part of the GSM mobile phone system, in satellite communications, and in 802.11. As an example, a popular convolutional code is shown in Fig. 3-7. This code is known as the NASA convolutional code of $r = 1/2$ and $k = 7$, since it was first used for the Voyager space missions starting in 1977. Since then it has been liberally reused, for example, as part of 802.11.

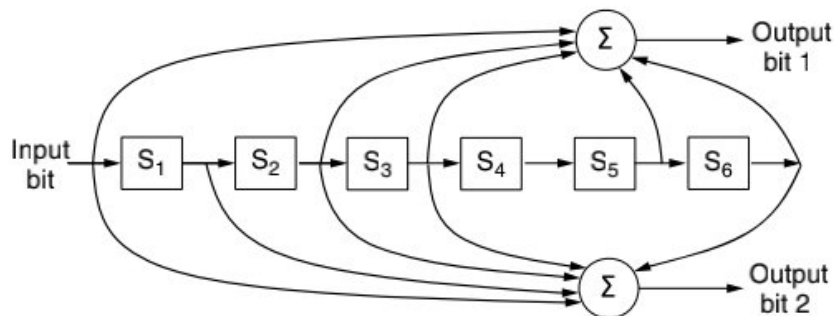


Figure 3-7. The NASA binary convolutional code used in 802.11.

In Fig. 3-7, each input bit on the left-hand side produces two output bits on the right-hand side that are XOR sums of the input and internal state. Since it deals with bits and performs linear operations, this is a binary, linear convolutional code. Since 1 input bit produces 2 output bits, the code rate is $1/2$. It is not systematic since none of the output bits is simply the input bit.

The internal state is kept in six memory registers. Each time another bit is input the values in the registers are shifted to the right. For example, if 111 is input and the initial state is all zeros, the internal state, written left to right, will become 100000, 110000, and 111000 after the first, second, and third bits have been input. The output bits will be 11, followed by 10, and then 01. It takes seven shifts to flush an input completely so that it does not affect the output. The constraint length of this code is thus $k = 7$.

A convolutional code is decoded by finding the sequence of input bits that is most likely to have produced the observed sequence of output bits (which includes any errors). For small values of k , this is done with a widely used algorithm developed by Viterbi (Forney, 1973). The algorithm walks the observed sequence, keeping for each step and for each possible internal state the input sequence that would have produced the observed sequence with the fewest errors. The input sequence requiring the fewest errors at the end is the most likely message.

Convolutional codes have been popular in practice because it is easy to factor the uncertainty of a bit being a 0 or a 1 into the decoding. For example, suppose $-1V$ is the logical 0 level and $+1V$ is the logical 1 level, we might receive $0.9V$ and $-0.1V$ for 2 bits. Instead of mapping these signals to 1 and 0 right away, we would like to treat $0.9V$ as “very likely a 1” and $-0.1V$ as “maybe a 0” and correct the sequence as a whole. Extensions of the Viterbi algorithm can work with these uncertainties to provide stronger error correction. This approach of working with the uncertainty of a bit is called **soft-decision decoding**. Conversely, deciding whether each bit is a 0 or a 1 before subsequent error correction is called **hard-decision decoding**.

The third kind of error-correcting code we will describe is the **Reed-Solomon code**. Like Hamming codes, Reed-Solomon codes are linear block codes, and they are often systematic too. Unlike Hamming codes, which operate on individual bits, Reed-Solomon codes operate on m bit symbols. Naturally, the mathematics are more involved, so we will describe their operation by analogy.

Reed-Solomon codes are based on the fact that every n degree polynomial is uniquely determined by $n + 1$ points. For example, a line having the form $ax + b$ is determined by two points. Extra points on the same line are redundant, which is helpful for error correction. Imagine that we have two data points that represent a line and we send those two data points plus two check points chosen to lie on the same line. If one of the points is received in error, we can still recover the data points by fitting a line to the received points. Three of the points will lie on the line, and one point, the one in error, will not. By finding the line we have corrected the error.

Reed-Solomon codes are actually defined as polynomials that operate over finite fields, but they work in a similar manner. For m bit symbols, the codewords are $2^m - 1$ symbols long. A popular choice is to make $m = 8$ so that symbols are bytes. A codeword is then 255 bytes long. The (255, 233) code is widely used; it adds 32 redundant symbols to 233 data symbols. Decoding with error correction

is done with an algorithm developed by Berlekamp and Massey that can efficiently perform the fitting task for moderate-length codes (Massey, 1969).

Reed-Solomon codes are widely used in practice because of their strong error-correction properties, particularly for burst errors. They are used for DSL, data over cable, satellite communications, and perhaps most ubiquitously on CDs, DVDs, and Blu-ray discs. Because they are based on m bit symbols, a single-bit error and an m -bit burst error are both treated simply as one symbol error. When $2t$ redundant symbols are added, a Reed-Solomon code is able to correct up to t errors in any of the transmitted symbols. This means, for example, that the (255, 233) code, which has 32 redundant symbols, can correct up to 16 symbol errors. Since the symbols may be consecutive and they are each 8 bits, an error burst of up to 128 bits can be corrected. The situation is even better if the error model is one of erasures (e.g., a scratch on a CD that obliterates some symbols). In this case, up to $2t$ errors can be corrected.

Reed-Solomon codes are often used in combination with other codes such as a convolutional code. The thinking is as follows. Convolutional codes are effective at handling isolated bit errors, but they will fail, likely with a burst of errors, if there are too many errors in the received bit stream. By adding a Reed-Solomon code within the convolutional code, the Reed-Solomon decoding can mop up the error bursts, a task at which it is very good. The overall code then provides good protection against both single and burst errors.

The final error-correcting code we will cover is the **LDPC (Low-Density Parity Check)** code. LDPC codes are linear block codes that were invented by Robert Gallager in his doctoral thesis (Gallager, 1962). Like most theses, they were promptly forgotten, only to be reinvented in 1995 when advances in computing power had made them practical.

In an LDPC code, each output bit is formed from only a fraction of the input bits. This leads to a matrix representation of the code that has a low density of 1s, hence the name for the code. The received codewords are decoded with an approximation algorithm that iteratively improves on a best fit of the received data to a legal codeword. This corrects errors.

LDPC codes are practical for large block sizes and have excellent error-correction abilities that outperform many other codes (including the ones we have looked at) in practice. For this reason they are rapidly being included in new protocols. They are part of the standard for digital video broadcasting, 10 Gbps Ethernet, power-line networks, and the latest version of 802.11. Expect to see more of them in future networks.

3.2.2 Error-Detecting Codes

Error-correcting codes are widely used on wireless links, which are notoriously noisy and error prone when compared to optical fibers. Without error-correcting codes, it would be hard to get anything through. However, over fiber or

high-quality copper, the error rate is much lower, so error detection and retransmission is usually more efficient there for dealing with the occasional error.

We will examine three different error-detecting codes. They are all linear, systematic block codes:

1. Parity.
2. Checksums.
3. Cyclic Redundancy Checks (CRCs).

To see how they can be more efficient than error-correcting codes, consider the first error-detecting code, in which a single **parity bit** is appended to the data. The parity bit is chosen so that the number of 1 bits in the codeword is even (or odd). Doing this is equivalent to computing the (even) parity bit as the modulo 2 sum or XOR of the data bits. For example, when 1011010 is sent in even parity, a bit is added to the end to make it 10110100. With odd parity 1011010 becomes 10110101. A code with a single parity bit has a distance of 2, since any single-bit error produces a codeword with the wrong parity. This means that it can detect single-bit errors.

Consider a channel on which errors are isolated and the error rate is 10^{-6} per bit. This may seem a tiny error rate, but it is at best a fair rate for a long wired cable that is challenging for error detection. Typical LAN links provide bit error rates of 10^{-10} . Let the block size be 1000 bits. To provide error correction for 1000-bit blocks, we know from Eq. (3-1) that 10 check bits are needed. Thus, a megabit of data would require 10,000 check bits. To merely detect a block with a single 1-bit error, one parity bit per block will suffice. Once every 1000 blocks, a block will be found to be in error and an extra block (1001 bits) will have to be transmitted to repair the error. The total overhead for the error detection and retransmission method is only 2001 bits per megabit of data, versus 10,000 bits for a Hamming code.

One difficulty with this scheme is that a single parity bit can only reliably detect a single-bit error in the block. If the block is badly garbled by a long burst error, the probability that the error will be detected is only 0.5, which is hardly acceptable. The odds can be improved considerably if each block to be sent is regarded as a rectangular matrix n bits wide and k bits high. Now, if we compute and send one parity bit for each row, up to k bit errors will be reliably detected as long as there is at most one error per row.

However, there is something else we can do that provides better protection against burst errors: we can compute the parity bits over the data in a different order than the order in which the data bits are transmitted. Doing so is called **interleaving**. In this case, we will compute a parity bit for each of the n columns and send all the data bits as k rows, sending the rows from top to bottom and the bits in each row from left to right in the usual manner. At the last row, we send the n parity bits. This transmission order is shown in Fig. 3-8 for $n = 7$ and $k = 7$.

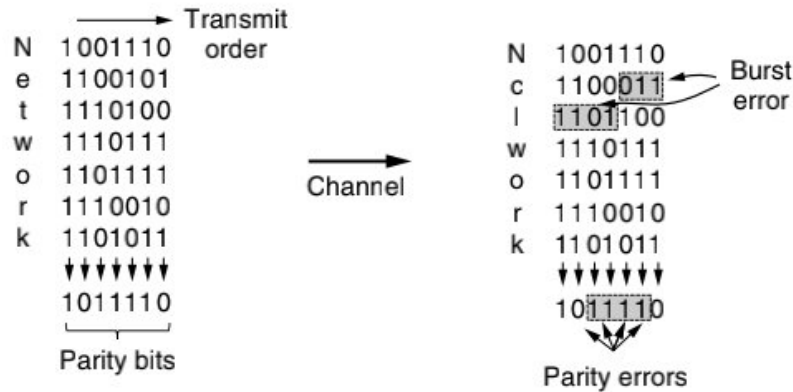


Figure 3-8. Interleaving of parity bits to detect a burst error.

Interleaving is a general technique to convert a code that detects (or corrects) isolated errors into a code that detects (or corrects) burst errors. In Fig. 3-8, when a burst error of length $n = 7$ occurs, the bits that are in error are spread across different columns. (A burst error does not imply that all the bits are wrong; it just implies that at least the first and last are wrong. In Fig. 3-8, 4 bits were flipped over a range of 7 bits.) At most 1 bit in each of the n columns will be affected, so the parity bits on those columns will detect the error. This method uses n parity bits on blocks of kn data bits to detect a single burst error of length n or less.

A burst of length $n + 1$ will pass undetected, however, if the first bit is inverted, the last bit is inverted, and all the other bits are correct. If the block is badly garbled by a long burst or by multiple shorter bursts, the probability that any of the n columns will have the correct parity by accident is 0.5, so the probability of a bad block being accepted when it should not be is 2^{-n} .

The second kind of error-detecting code, the **checksum**, is closely related to groups of parity bits. The word “checksum” is often used to mean a group of check bits associated with a message, regardless of how are calculated. A group of parity bits is one example of a checksum. However, there are other, stronger checksums based on a running sum of the data bits of the message. The checksum is usually placed at the end of the message, as the complement of the sum function. This way, errors may be detected by summing the entire received codeword, both data bits and checksum. If the result comes out to be zero, no error has been detected.

One example of a checksum is the 16-bit Internet checksum used on all Internet packets as part of the IP protocol (Braden et al., 1988). This checksum is a sum of the message bits divided into 16-bit words. Because this method operates on words rather than on bits, as in parity, errors that leave the parity unchanged can still alter the sum and be detected. For example, if the lowest order bit in two different words is flipped from a 0 to a 1, a parity check across these bits would fail to detect an error. However, two 1s will be added to the 16-bit checksum to produce a different result. The error can then be detected.

The Internet checksum is computed in one's complement arithmetic instead of as the modulo 2^{16} sum. In one's complement arithmetic, a negative number is the bitwise complement of its positive counterpart. Modern computers run two's complement arithmetic, in which a negative number is the one's complement plus one. On a two's complement computer, the one's complement sum is equivalent to taking the sum modulo 2^{16} and adding any overflow of the high order bits back into the low-order bits. This algorithm gives a more uniform coverage of the data by the checksum bits. Otherwise, two high-order bits can be added, overflow, and be lost without changing the sum. There is another benefit, too. One's complement has two representations of zero, all 0s and all 1s. This allows one value (e.g., all 0s) to indicate that there is no checksum, without the need for another field.

For decades, it has always been assumed that frames to be checksummed contain random bits. All analyses of checksum algorithms have been made under this assumption. Inspection of real data by Partridge et al. (1995) has shown this assumption to be quite wrong. As a consequence, undetected errors are in some cases much more common than had been previously thought.

The Internet checksum in particular is efficient and simple but provides weak protection in some cases precisely because it is a simple sum. It does not detect the deletion or addition of zero data, nor swapping parts of the message, and it provides weak protection against message splices in which parts of two packets are put together. These errors may seem very unlikely to occur by random processes, but they are just the sort of errors that can occur with buggy hardware.

A better choice is **Fletcher's checksum** (Fletcher, 1982). It includes a positional component, adding the product of the data and its position to the running sum. This provides stronger detection of changes in the position of data.

Although the two preceding schemes may sometimes be adequate at higher layers, in practice, a third and stronger kind of error-detecting code is in widespread use at the link layer: the **CRC (Cyclic Redundancy Check)**, also known as a **polynomial code**. Polynomial codes are based upon treating bit strings as representations of polynomials with coefficients of 0 and 1 only. A k -bit frame is regarded as the coefficient list for a polynomial with k terms, ranging from x^{k-1} to x^0 . Such a polynomial is said to be of degree $k - 1$. The high-order (leftmost) bit is the coefficient of x^{k-1} , the next bit is the coefficient of x^{k-2} , and so on. For example, 110001 has 6 bits and thus represents a six-term polynomial with coefficients 1, 1, 0, 0, 0, and 1: $1x^5 + 1x^4 + 0x^3 + 0x^2 + 0x^1 + 1x^0$.

Polynomial arithmetic is done modulo 2, according to the rules of algebraic field theory. It does not have carries for addition or borrows for subtraction. Both addition and subtraction are identical to exclusive OR. For example:

$$\begin{array}{r}
 10011011 \\
 + 11001010 \\
 \hline
 01010001
 \end{array}
 \quad
 \begin{array}{r}
 00110011 \\
 + 11001101 \\
 \hline
 11111110
 \end{array}
 \quad
 \begin{array}{r}
 11110000 \\
 - 10100110 \\
 \hline
 01010110
 \end{array}
 \quad
 \begin{array}{r}
 01010101 \\
 - 10101111 \\
 \hline
 11111010
 \end{array}$$

Long division is carried out in exactly the same way as it is in binary except that

the subtraction is again done modulo 2. A divisor is said “to go into” a dividend if the dividend has as many bits as the divisor.

When the polynomial code method is employed, the sender and receiver must agree upon a **generator polynomial**, $G(x)$, in advance. Both the high- and low-order bits of the generator must be 1. To compute the CRC for some frame with m bits corresponding to the polynomial $M(x)$, the frame must be longer than the generator polynomial. The idea is to append a CRC to the end of the frame in such a way that the polynomial represented by the checksummed frame is divisible by $G(x)$. When the receiver gets the checksummed frame, it tries dividing it by $G(x)$. If there is a remainder, there has been a transmission error.

The algorithm for computing the CRC is as follows:

1. Let r be the degree of $G(x)$. Append r zero bits to the low-order end of the frame so it now contains $m + r$ bits and corresponds to the polynomial $x^r M(x)$.
2. Divide the bit string corresponding to $G(x)$ into the bit string corresponding to $x^r M(x)$, using modulo 2 division.
3. Subtract the remainder (which is always r or fewer bits) from the bit string corresponding to $x^r M(x)$ using modulo 2 subtraction. The result is the checksummed frame to be transmitted. Call its polynomial $T(x)$.

Figure 3-9 illustrates the calculation for a frame 1101011111 using the generator $G(x) = x^4 + x + 1$.

It should be clear that $T(x)$ is divisible (modulo 2) by $G(x)$. In any division problem, if you diminish the dividend by the remainder, what is left over is divisible by the divisor. For example, in base 10, if you divide 210,278 by 10,941, the remainder is 2399. If you then subtract 2399 from 210,278, what is left over (207,879) is divisible by 10,941.

Now let us analyze the power of this method. What kinds of errors will be detected? Imagine that a transmission error occurs, so that instead of the bit string for $T(x)$ arriving, $T(x) + E(x)$ arrives. Each 1 bit in $E(x)$ corresponds to a bit that has been inverted. If there are k 1 bits in $E(x)$, k single-bit errors have occurred. A single burst error is characterized by an initial 1, a mixture of 0s and 1s, and a final 1, with all other bits being 0.

Upon receiving the checksummed frame, the receiver divides it by $G(x)$; that is, it computes $[T(x) + E(x)]/G(x)$. $T(x)/G(x)$ is 0, so the result of the computation is simply $E(x)/G(x)$. Those errors that happen to correspond to polynomials containing $G(x)$ as a factor will slip by; all other errors will be caught.

If there has been a single-bit error, $E(x) = x^i$, where i determines which bit is in error. If $G(x)$ contains two or more terms, it will never divide into $E(x)$, so all single-bit errors will be detected.

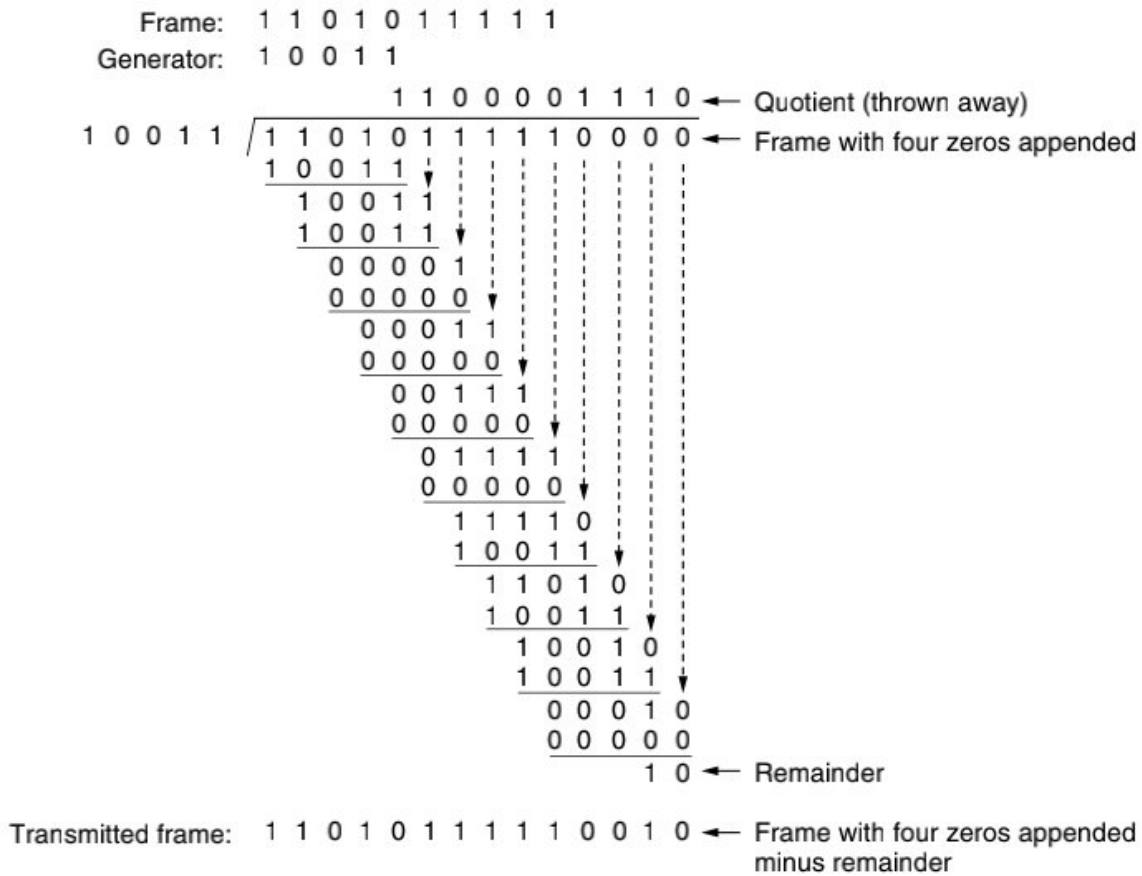


Figure 3-9. Example calculation of the CRC.

If there have been two isolated single-bit errors, $E(x) = x^i + x^j$, where $i > j$. Alternatively, this can be written as $E(x) = x^j(x^{i-j} + 1)$. If we assume that $G(x)$ is not divisible by x , a sufficient condition for all double errors to be detected is that $G(x)$ does not divide $x^k + 1$ for any k up to the maximum value of $i - j$ (i.e., up to the maximum frame length). Simple, low-degree polynomials that give protection to long frames are known. For example, $x^{15} + x^{14} + 1$ will not divide $x^k + 1$ for any value of k below 32,768.

If there are an odd number of bits in error, $E(x)$ contains an odd number of terms (e.g., $x^5 + x^2 + 1$, but not $x^2 + 1$). Interestingly, no polynomial with an odd number of terms has $x + 1$ as a factor in the modulo 2 system. By making $x + 1$ a factor of $G(x)$, we can catch all errors with an odd number of inverted bits.

Finally, and importantly, a polynomial code with r check bits will detect all burst errors of length $\leq r$. A burst error of length k can be represented by $x^i(x^{k-1} + \dots + 1)$, where i determines how far from the right-hand end of the received frame the burst is located. If $G(x)$ contains an x^0 term, it will not have x^i as a factor, so if the degree of the parenthesized expression is less than the degree of $G(x)$, the remainder can never be zero.

If the burst length is $r + 1$, the remainder of the division by $G(x)$ will be zero if and only if the burst is identical to $G(x)$. By definition of a burst, the first and last bits must be 1, so whether it matches depends on the $r - 1$ intermediate bits. If all combinations are regarded as equally likely, the probability of such an incorrect frame being accepted as valid is $1/2^{r-1}$.

It can also be shown that when an error burst longer than $r + 1$ bits occurs or when several shorter bursts occur, the probability of a bad frame getting through unnoticed is $1/2^r$, assuming that all bit patterns are equally likely.

Certain polynomials have become international standards. The one used in IEEE 802 followed the example of Ethernet and is

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1$$

Among other desirable properties, it has the property that it detects all bursts of length 32 or less and all bursts affecting an odd number of bits. It has been used widely since the 1980s. However, this does not mean it is the best choice. Using an exhaustive computational search, Castagnoli et al. (1993) and Koopman (2002) found the best CRCs. These CRCs have a Hamming distance of 6 for typical message sizes, while the IEEE standard CRC-32 has a Hamming distance of only 4.

Although the calculation required to compute the CRC may seem complicated, it is easy to compute and verify CRCs in hardware with simple shift register circuits (Peterson and Brown, 1961). In practice, this hardware is nearly always used. Dozens of networking standards include various CRCs, including virtually all LANs (e.g., Ethernet, 802.11) and point-to-point links (e.g., packets over SONET).

3.3 ELEMENTARY DATA LINK PROTOCOLS

To introduce the subject of protocols, we will begin by looking at three protocols of increasing complexity. For interested readers, a simulator for these and subsequent protocols is available via the Web (see the preface). Before we look at the protocols, it is useful to make explicit some of the assumptions underlying the model of communication.

To start with, we assume that the physical layer, data link layer, and network layer are independent processes that communicate by passing messages back and forth. A common implementation is shown in Fig. 3-10. The physical layer process and some of the data link layer process run on dedicated hardware called a **NIC (Network Interface Card)**. The rest of the link layer process and the network layer process run on the main CPU as part of the operating system, with the software for the link layer process often taking the form of a **device driver**. However, other implementations are also possible (e.g., three processes offloaded to dedicated hardware called a **network accelerator**, or three processes running on the