Although Unicode has yet to become the exclusive alphabet of American computers, most manufacturers are including at least some limited support for it in their systems. Unicode is currently the default character set of the Java programming language. Ultimately, the acceptance of Unicode by all manufacturers will depend on how aggressively they wish to position themselves as international players and how inexpensively disk drives can be produced to support an alphabet with double the storage requirements of ASCII or EBCDIC.

2.7 ERROR DETECTION AND CORRECTION

No communications channel or storage medium can be completely error-free. It is a physical impossibility. As transmission rates are increased, bit timing gets tighter. As more bits are packed per square millimeter of storage, magnetic flux densities increase. Error rates increase in direct proportion to the number of bits per second transmitted, or the number of bits per square millimeter of magnetic storage.

In Section 2.6.3, we mentioned that a parity bit could be added to an ASCII byte to help determine whether any of the bits had become corrupted during transmission. This method of error detection is limited in its effectiveness: Simple parity can detect only an odd number of errors per byte. If two errors occur, we are helpless to detect a problem. Nonsense could pass for good data. If such errors occur in sending financial information or program code, the effects can be disastrous.

As you read the sections that follow, you should keep in mind that just as it is impossible to create an error-free medium, it is also impossible to detect or correct 100% of all errors that *could* occur in a medium. Error detection and correction is yet another study in the trade-offs that one must make in designing computer systems. The well-constructed error control system is therefore a system where a "reasonable" number of the "reasonably" expected errors can be detected or corrected within the bounds of "reasonable" economics. (Note: The word *reasonable* is implementation-dependent.)

2.7.1 Cyclic Redundancy Check

Checksums are used in a wide variety of coding systems, from bar codes to International Standard Book Numbers. These are self-checking codes that will quickly indicate whether the preceding digits have been misread. A **cyclic redundancy check (CRC)** is a type of checksum used primarily in data communications that determines whether an error has occurred within a large block or stream of information bytes. The larger the block to be checked, the larger the checksum must be to provide adequate protection. Checksums and CRCs are a type of **systematic error detection** scheme, meaning that the error-checking bits are appended to the original information byte. The group of error-checking bits is called a **syndrome**. The original information byte is unchanged by the addition of the error-checking bits.

The word *cyclic* in cyclic redundancy check refers to the abstract mathematical theory behind this error control system. Although a discussion of this theory is

beyond the scope of this text, we can demonstrate how the method works to aid in your understanding of its power to economically detect transmission errors.

Arithmetic Modulo 2

You may be familiar with integer arithmetic taken over a modulus. Twelve-hour clock arithmetic is a modulo 12 system that you use every day to tell time. When we add 2 hours to 11:00, we get 1:00. Arithmetic modulo 2 uses two binary operands with no borrows or carries. The result is likewise binary and is also a member of the modulus 2 system. Because of this closure under addition, and the existence of identity elements, mathematicians say that this modulo 2 system forms an **algebraic field**.

The addition rules are as follows:

$$0 + 0 = 0$$
$$0 + 1 = 1$$
$$1 + 0 = 1$$
$$1 + 1 = 0$$

≡ **EXAMPLE 2.29** Find the sum of 1011_2 and 110_2 modulo 2.

$$\begin{array}{r} 1011 \\ +110 \\ \hline \end{array}$$

$1101_2 \pmod 2$

This sum makes sense only in modulo 2.

Modulo 2 division operates through a series of partial sums using the modulo 2 addition rules. Example 2.30 illustrates the process.

≡ **EXAMPLE 2.30** Find the quotient and remainder when 1001011_2 is divided by 1011_2.

$1011\overline{)1001011}$	1. Write the divisor directly beneath the first bit of the dividend.
$\underline{1011}$	
0010	2. Add these numbers using modulo 2.
	3. Bring down bits from the dividend so that the first 1 of the difference can align with the first 1 of the divisor.
001001	
$\underline{1011}$	4. Copy the divisor as in Step 1.
0010	5. Add as in Step 2.
00101	6. Bring down another bit.
	7. 101_2 is not divisible by 1011_2, so this is the remainder.

The quotient is 1010_2.

Arithmetic operations over the modulo 2 field have polynomial equivalents that are analogous to polynomials over the field of integers. We have seen how positional number systems represent numbers in increasing powers of a radix, for example,

$$1011_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0.$$

By letting $X = 2$, the binary number 1011_2 becomes shorthand for the polynomial:

$$1 \times X^3 + 0 \times X^2 + 1 \times X^1 + 1 \times X^0.$$

The division performed in Example 2.30 then becomes the polynomial operation:

$$\frac{X^6 + X^3 + X + 1}{X^3 + X + 1}$$

Calculating and Using CRCs

With that lengthy preamble behind us, we can now proceed to show how CRCs are constructed. We will do this by example:

1. Let the information byte $I = 1001011_2$. (Any number of bytes can be used to form a message block.)
2. The sender and receiver agree upon an arbitrary binary pattern, say $P = 1011_2$. (Patterns beginning and ending with 1 work best.)
3. Shift I to the left by one less than the number of bits in P, giving a new $I = 1001011000_2$.
4. Using I as a dividend and P as a divisor, perform the modulo 2 division (as shown in Example 2.30). We ignore the quotient and note the remainder is 100_2. The remainder is the actual CRC checksum.
5. Add the remainder to I, giving the message M:

   $1001011000_2 + 100_2 = 1001011100_2$

6. M is decoded and checked by the message receiver using the reverse process. Only now P divides M exactly:

```
        1010100
    ------------
1011 ) 1001011100
       1011
       ----
       001001
         1011
         ----
         0010
         0000
           1011
           ----
           0000
```

Note: The reverse process would include appending the remainder.

A remainder other than zero indicates that an error has occurred in the transmission of M. This method works best when a large prime polynomial is used. There are four standard polynomials used widely for this purpose:

- CRC-CCITT (ITU-T): $X^{16} + X^{12} + X^5 + 1$
- CRC-12: $X^{12} + X^{11} + X^3 + X^2 + X + 1$
- CRC-16 (ANSI): $X^{16} + X^{15} + X^2 + 1$
- CRC-32: $X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^{10} + X^8 + X^7 + X^5 + X^4 + X + 1$

CRC-CCITT, CRC-12, and CRC-16 operate over pairs of bytes; CRC-32 uses four bytes, which is appropriate for systems operating on 32-bit words. It has been proven that CRCs using these polynomials can detect over 99.8% of all single-bit errors.

CRCs can be implemented effectively using lookup tables as opposed to calculating the remainder with each byte. The remainder generated by each possible input bit pattern can be "burned" directly into communications and storage electronics. The remainder can then be retrieved using a 1-cycle lookup as compared to a 16- or 32-cycle division operation. Clearly, the trade-off is in speed versus the cost of more complex control circuitry.

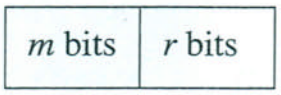
2.7.2 Hamming Codes

Data communications channels are simultaneously more error-prone and more tolerant of errors than disk systems. In data communications, it is sufficient to have only the ability to detect errors. If a communications device determines that a message contains an erroneous bit, all it has to do is request retransmission. Storage systems and memory do not have this luxury. A disk can sometimes be the sole repository of a financial transaction, or other collection of nonreproducible real-time data. Storage devices and memory must therefore have the ability to not only detect but to correct a reasonable number of errors.

Error-recovery coding has been studied intensively over the past century. One of the most effective codes—and the oldest—is the Hamming code. **Hamming codes** are an adaptation of the concept of parity, whereby error detection and correction capabilities are increased in proportion to the number of parity bits added to an information word. Hamming codes are used in situations where random errors are likely to occur. With random errors, we assume each bit failure has a fixed probability of occurrence independent of other bit failures. It is common for computer memory to experience such errors, so in our following discussion, we present Hamming codes in the context of memory bit error detection and correction.

We mentioned that Hamming codes use parity bits, also called **check bits** or **redundant bits**. The memory word itself consists of m bits, but r redundant bits are added to allow for error detection and/or correction. Thus, the final word, called a

code word, is an n-bit unit containing m data bits and r check bits. There exists a unique code word consisting of $n = m + r$ bits for each data word as follows:



The number of bit positions in which two code words differ is called the **Hamming distance** of those two code words. For example, if we have the following two code words:

```
1 0 0 0 1 0 0 1
1 0 1 1 0 0 0 1
    * * *
```

we see that they differ in 3 bit positions, so the Hamming distance of these two code words is 3. (Please note that we have not yet discussed how to create code words; we will do that shortly.)

The Hamming distance between two code words is important in the context of error detection. If two code words are a Hamming distance d apart, d single-bit errors are required to convert one code word to the other, which implies this type of error would not be detected. Therefore, if we wish to create a code that guarantees detection of all single-bit errors (an error in only 1 bit), all pairs of code words must have a Hamming distance of at least 2. If an n-bit word is not recognized as a legal code word, it is considered an error.

Given an algorithm for computing check bits, it is possible to construct a complete list of legal code words. The smallest Hamming distance found among all pairs of the code words in this code is called the **minimum Hamming distance** for the code. The minimum Hamming distance of a code, often signified by the notation **D(min)**, determines its error detecting and correcting capability. Stated succinctly, for any code word X to be received as another valid code word Y, at least $D(min)$ errors must occur in X. So, to detect k (or fewer) single-bit errors, the code must have a Hamming distance of $D(min) = k + 1$. Hamming codes can always detect $D(min) - 1$ errors and correct $\lfloor (D(min) - 1)/2 \rfloor$ errors.[1] Accordingly, the Hamming distance of a code must be at least $2k + 1$ in order for it to be able to correct k errors.

Code words are constructed from information words using r parity bits. Before we continue the discussion of error detection and correction, let's consider a simple example. The most common error detection uses a single parity bit appended to the data (recall the discussion on ASCII character representation). A single-bit error in any bit of the code word produces the wrong parity.

[1]The $\lfloor\ \rfloor$ brackets denote the integer floor function, which is the largest integer that is smaller than or equal to the enclosed quantity. For example, $\lfloor 8.3 \rfloor = 8$ and $\lfloor 8.9 \rfloor = 8$.

EXAMPLE 2.31 Assume a memory with 2 data bits and 1 parity bit (appended at the end of the code word) that uses even parity (so the number of 1s in the codeword must be even). With 2 data bits, we have a total of 4 possible words. We list here the data word, its corresponding parity bit, and the resulting code word for each of these 4 possible words:

Data Word	Parity Bit	Code Word
00	0	000
01	1	011
10	1	101
11	0	110

The resulting code words have 3 bits. However, using 3 bits allows for 8 different bit patterns, as follows (valid code words are marked with an *):

000*	100
001	101*
010	110*
011*	111

If the code word 001 is encountered, it is invalid and thus indicates an error has occurred somewhere in the code word. For example, suppose the correct code word to be stored in memory is 011, but an error produces 001. This error can be detected, but it cannot be corrected. It is impossible to determine exactly how many bits have been flipped and exactly which ones are in error. Error-correcting codes require more than a single parity bit, as we see in the following discussion.

What happens in the above example if a valid code word is subject to two-bit errors? For example, suppose the code word 011 is converted into 000. This error is not detected. If you examine the code in the above example, you will see that *D(min)* is 2, which implies this code is guaranteed to detect only single-bit errors.

We have already stated that the error detecting and correcting capabilities of a code are dependent on *D(min)*, and, from an error detection point of view, we have seen this relationship exhibited in Example 2.31. Error correction requires the code to contain additional redundant bits to ensure a minimum Hamming distance $D(min) = 2k + 1$ if the code is to detect and correct k errors. This Hamming distance guarantees that all legal code words are far enough apart that even with k changes, the original invalid code word is closer to one unique valid code word. This is important, because the method used in error correction is to change

the invalid code word into the valid code word that differs in the fewest number of bits. This idea is illustrated in Example 2.32.

EXAMPLE 2.32 Suppose we have the following code (do not worry at this time about how this code was generated; we will address this issue shortly):

0 0 0 0 0
0 1 0 1 1
1 0 1 1 0
1 1 1 0 1

First, let's determine *D(min)*. By examining all possible pairs of code words, we discover that the minimum Hamming distance *D(min)* = 3. Thus, this code can detect up to two errors and correct one single bit error. How is correction handled? Suppose we read the invalid code word 10000. There must be at least one error because this does not match any of the valid code words. We now determine the Hamming distance between the observed code word and each legal code word: it differs in 1 bit from the first code word, 4 from the second, 2 from the third, and 3 from the last, resulting in a **difference vector** of [1,4,2,3]. To make the correction using this code, we automatically correct to the legal code word closest to the observed word, resulting in a correction to 00000. Note that this "correction" is not necessarily correct! We are assuming the minimum number of possible errors has occurred, namely 1. It is possible that the original code word was supposed to be 10110 and was changed to 10000 when two errors occurred.

Suppose two errors really did occur. For example, assume we read the invalid code word 11000. If we calculate the distance vector of [2,3,3,2], we see there is no "closest" code word, and we are unable to make the correction. The minimum Hamming distance of three permits correction of one error only, and cannot ensure correction, as evidenced in this example, if more than one error occurs.

In our discussion up to this point, we have simply presented you with various codes, but have not given any specifics as to how the codes are generated. There are many methods that are used for code generation; perhaps one of the more intuitive is the Hamming algorithm for code design, which we now present. Before explaining the actual steps in the algorithm, we provide some background material.

Suppose we wish to design a code with words consisting of m data bits and r check bits, which allows for single-bit errors to be corrected. This implies there are 2^m legal code words, each with a unique combination of check bits. Since we are focused on single-bit errors, let's examine the set of invalid code words that are a distance of 1 from all legal code words.

Each valid code word has n bits, and an error could occur in any of these n positions. Thus, each valid code word has n illegal code words at a distance of 1.