Therefore, if we are concerned with each legal code word and each invalid code word consisting of one error, we have $n + 1$ bit patterns associated with each code word (1 legal word and n illegal words). Since each code word consists of n bits, where $n = m + r$, there are 2^n total bit patterns possible. This results in the following inequality:

$$(n + 1) \times 2^m \leq 2^n$$

where $n + 1$ is the number of bit patterns per code word, 2^m is the number of legal code words, and 2^n is the total number of bit patterns possible. Because $n = m + r$, we can rewrite the inequality as:

$$(m + r + 1) \times 2^m \leq 2^{m+r}$$

or

$$(m + r + 1) \leq 2^r$$

This inequality is important because it specifies the lower limit on the number of check bits required (we always use as few check bits as possible) to construct a code with m data bits and r check bits that corrects all single-bit errors.

Suppose we have data words of length $m = 4$. Then:

$$(4 + r + 1) \leq 2^r$$

which implies r must be greater than or equal to 3. We choose $r = 3$. This means to build a code with data words of 4 bits that should correct single-bit errors, we must add 3 check bits.

The Hamming algorithm provides a straightforward method for designing codes to correct single-bit errors. To construct error correcting codes for any size memory word, we follow these steps:

1. Determine the number of check bits, r, necessary for the code and then number the n bits (where $n = m + r$), right to left, starting with 1 (not 0).
2. Each bit whose bit number is a power of 2 is a parity bit—the others are data bits.
3. Assign parity bits to check bit positions as follows: Bit b is checked by those parity bits $b_1, b_2, \ldots, b_j$ such that $b_1 + b_2 + \ldots + b_j = b$ (where "+" indicates the modulo 2 sum).

We now present an example to illustrate these steps and the actual process of error correction.

EXAMPLE 2.33 Using the Hamming code just described and even parity, encode the 8-bit ASCII character K. (The high-order bit will be zero.) Induce a single-bit error and then indicate how to locate the error.

We first determine the code word for K.

Step 1: Determine the number of necessary check bits, add these bits to the data bits, and number all n bits.

Since $m = 8$, we have: $(8 + r + 1) \leq 2^r$, which implies r must be greater than or equal to 4. We choose $r = 4$.

Step 2: Number the n bits right to left, starting with 1, which results in:

$$\begin{array}{cccccccccccc} \_ & \_ & \_ & \_ & \boxed{} & \_ & \_ & \_ & \boxed{} & \_ & \boxed{} & \boxed{} \\ 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \end{array}$$

The parity bits are marked by boxes.

Step 3: Assign parity bits to check the various bit positions.

To perform this step, we first write all bit positions as sums of those numbers that are powers of 2:

$1 = 1$	$5 = 1 + 4$	$9 = 1 + 8$
$2 = 2$	$6 = 2 + 4$	$10 = 2 + 8$
$3 = 1 + 2$	$7 = 1 + 2 + 4$	$11 = 1 + 2 + 8$
$4 = 4$	$8 = 8$	$12 = 4 + 8$

The number 1 contributes to 1, 3, 5, 7, 9, and 11, so this parity bit will reflect the parity of the bits in these positions. Similarly, 2 contributes to 2, 3, 6, 7, 10, and 11, so the parity bit in position 2 reflects the parity of this set of bits. Bit 4 provides parity for 4, 5, 6, 7, and 12, and bit 8 provides parity for bits 8, 9, 10, 11, and 12. If we write the data bits in the nonboxed blanks, and then add the parity bits, we have the following code word as a result:

$$\begin{array}{cccccccccccc} \underline{0} & \underline{1} & \underline{0} & \underline{0} & \boxed{1} & \underline{1} & \underline{0} & \underline{1} & \boxed{0} & \underline{1} & \boxed{1} & \boxed{0} \\ 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \end{array}$$

Therefore, the code word for K is 010011010110.

Let's introduce an error in bit position b_9, resulting in the code word 010111010110. If we use the parity bits to check the various sets of bits, we find the following:

Bit 1 checks 1, 3, 5, 7, 9, and 11: With even parity, this produces an error.

Bit 2 checks 2, 3, 6, 7, 10, and 11: This is ok.

Bit 4 checks 4, 5, 6, 7, and 12: This is ok.

Bit 8 checks 8, 9, 10, 11, and 12: This produces an error.

Parity bits 1 and 8 show errors. These two parity bits both check 9 and 11, so the single bit error must be in either bit 9 or bit 11. However, since bit 2 checks bit 11 and indicates no error has occurred in the subset of bits it checks, the error must occur in bit 9. (We know this because we created the error; however, note that even if we have no clue where the error is, using this method allows us to determine the position of the error and correct it by simply flipping the bit.)

Because of the way the parity bits are positioned, an easier method to detect and correct the error bit is to add the positions of the parity bits that indicate an error. We found that parity bits 1 and 8 produced an error, and $1 + 8 = 9$, which is exactly where the error occurred.

In the next chapter, you will see how easy it is to implement a Hamming code using simple binary circuits. Because of their simplicity, Hamming code protection can be added inexpensively and with minimal impact upon performance.

2.7.3 Reed-Soloman

Hamming codes work well in situations where one can reasonably expect errors to be rare events. Fixed magnetic disk drives have error ratings on the order of 1 bit in 100 million. The 3-bit Hamming code that we just studied will easily correct this type of error. However, Hamming codes are useless in situations where there is a likelihood that multiple adjacent bits will be damaged. These kinds of errors are called **burst errors**. Because of their exposure to mishandling and environmental stresses, burst errors are common on removable media such as magnetic tapes and compact disks.

If we expect errors to occur in blocks, it stands to reason that we should use an error-correcting code that operates at a block level, as opposed to a Hamming code, which operates at the bit level. A **Reed-Soloman (RS)** code can be thought of as a CRC that operates over entire characters instead of only a few bits. RS codes, like CRCs, are systematic: The parity bytes are appended to a block of information bytes. RS(n, k) codes are defined using the following parameters:

- s = The number of bits in a character (or "symbol")
- k = The number of s-bit characters comprising the data block
- n = The number of bits in the code word

RS(n, k) can correct $\frac{(n - k)}{2}$ errors in the k information bytes.

The popular RS(255, 223) code, therefore, uses 223 8-bit information bytes and 32 syndrome bytes to form 255-byte code words. It will correct as many as 16 erroneous bytes in the information block.

The generator polynomial for a Reed-Soloman code is given by a polynomial defined over an abstract mathematical structure called a **Galois field**. (A lucid discussion of Galois mathematics is beyond the scope of this text. See the references at the end of the chapter.) The Reed-Soloman generating polynomial is:

$$g(x) = (x - a^i)(x - a^{i+1}) \ldots (x - a^{i+2t})$$

where $t = n - k$ and x is an entire byte (or symbol) and $g(x)$ operates over the field $GF(2^s)$. (Note: This polynomial expands over the Galois field, which is considerably different from the integer fields used in ordinary algebra.)

The n-byte RS code word is computed using the equation:

$$c(x) = g(x) \times i(x)$$

where $i(x)$ is the information block.

Despite the daunting algebra behind them, Reed-Soloman error-correction algorithms lend themselves well to implementation in computer hardware. They are implemented in high-performance disk drives for mainframe computers as well as compact disks used for music and data storage. These implementations will be described in Chapter 7.

CHAPTER SUMMARY

We have presented the essentials of data representation and numerical operations in digital computers. You should master the techniques described for base conversion and memorize the smaller hexadecimal and binary numbers. This knowledge will be beneficial to you as you study the remainder of this book. Your knowledge of hexadecimal coding will be useful if you are ever required to read a core (memory) dump after a system crash or if you do any serious work in the field of data communications.

You have also seen that floating-point numbers can produce significant errors when small errors are allowed to compound over iterative processes. There are various numerical techniques that can be used to control such errors. These techniques merit detailed study but are beyond the scope of this book.

You have learned that most computers use ASCII or EBCDIC to represent characters. It is generally of little value to memorize any of these codes in their entirety, but if you work with them frequently, you will find yourself learning a number of "key values" from which you can compute most of the others that you need.

Unicode is the default character set used by Java and recent versions of Windows. It is likely to replace EBCDIC and ASCII as the basic method of character representation in computer systems; however, the older codes will be with us for the foreseeable future, owing both to their economy and their pervasiveness.

Error-detecting and correcting codes are used in virtually all facets of computing technology. Should the need arise, your understanding of the various error control methods will help you to make informed choices among the various options available. The method that you choose will depend on a number of factors including computational overhead and the capacity of the storage and transmission media available to you.

FURTHER READING

A brief account of early mathematics in Western civilization can be found in Bunt et al. (1988).

Knuth (1998) presents a delightful and thorough discussion of the evolution of number systems and computer arithmetic in Volume 2 of his series on computer algorithms. (*Every* computer scientist should own a set of the Knuth books.)

A definitive account of floating-point arithmetic can be found in Goldberg (1991). Schwartz et al. (1999) describe how the IBM System/390 performs floating-point operations in both the older form and the IEEE standard. Soderquist and Leeser (1996) provide an excellent and detailed discussion of the problems surrounding floating-point division and square roots.

Detailed information about Unicode can be found at the Unicode Consortium Web site, www.unicode.org, as well as in *The Unicode Standard, Version 4.0* (2003).

The International Standards Organization Web site can be found at www.iso.ch. You will be amazed at the span of influence of this group. A similar trove of information can be found at the American National Standards Institute Web site: www.ansi.org.

After you have mastered the ideas presented in Chapter 3, you will enjoy reading Arazi's book (1988). This well-written book shows how error detection and correction are achieved using simple digital circuits. The appendix of this book gives a remarkably lucid discussion of the Galois field arithmetic that is used in Reed-Soloman codes.

If you'd prefer a rigorous and exhaustive study of error-correction theory, Pretzel's (1992) book is an excellent place to start. The text is accessible, well-written, and thorough.

Detailed discussions of Galois fields can be found in the (inexpensive!) books by Artin (1998) and Warner (1990). Warner's much larger book is a clearly written and comprehensive introduction to the concepts of abstract algebra. A study of abstract algebra will be helpful to you should you delve into the study of mathematical cryptography, a fast-growing area of interest in computer science.

REFERENCES

Arazi, Benjamin. *A Commonsense Approach to the Theory of Error Correcting Codes.* Cambridge, MA: The MIT Press, 1988.

Artin, Emil. *Galois Theory*. New York: Dover Publications, 1998.

Bunt, Lucas N. H., Jones, Phillip S., & Bedient, Jack D. *The Historical Roots of Elementary Mathematics.* New York: Dover Publications, 1988.

Goldberg, David. "What Every Computer Scientist Should Know About Floating-Point Arithmetic." *ACM Computing Surveys 23*: 1, March 1991, pp. 5–47.

Knuth, Donald E. *The Art of Computer Programming*, 3rd ed. Reading, MA: Addison-Wesley, 1998.

Pretzel, Oliver. *Error-Correcting Codes and Finite Fields.* New York: Oxford University Press, 1992.

Schwartz, Eric M., Smith, Ronald M., & Krygowski, Christopher A. "The S/390 G5 Floating-Point Unit Supporting Hex and Binary Architectures." *IEEE Proceedings from the 14th Symposium on Computer Arithmetic,* 1999, pp. 258–265.

Soderquist, Peter, & Leeser, Miriam. "Area and Performance Tradeoffs in Floating-Point Divide and Square-Root Implementations." *ACM Computing Surveys 28*: 3, September 1996, pp. 518–564.

The Unicode Consortium. *The Unicode Standard, Version 4.0.* Reading, MA: Addison-Wesley, 2003.

Warner, Seth. *Modern Algebra.* New York: Dover Publications, 1990.