C. Britton Rorabaugh

ERROR CODING COOKBOOK

Practical C/C++ Routines and Recipes for Error Detection and Correction

C. Britton Ro

ERRO
CODI
COOK

ON RETURN
PLEASE CHECK FOR
CD TAPE
DISK

Practical C
Routines an
for Error De
and Correct

Error Coding Cookbook

Practical C/C++ Routines and Recipes for Error Detection and Correction

C. Britton Rorabaugh

McGraw-Hill

New York San Francisco Washington, D.C. Auckland Bogotá
Caracas Lisbon London Madrid Mexico City Milan
Montreal New Delhi San Juan Singapore
Sydney Tokyo Toronto

McGraw-Hill

A Division of The McGraw-Hill Companies

hc 1 2 3 4 5 6 7 8 9 DOC/DOC 9 0 0 9 8 7 6

Library of Congress Cataloging-in-Publication Data
Rorabaugh, C. Britton.
Error coding cookbook : practical C/C++ routines and recipes for error detection and correction / by C. Britton Rorabaugh.
p. cm.
Includes index.
ISBN 0-07-911720-1 (h)
1. C (Computer program language) 2. Debugging in computer science. I. Title.
QA76.73.C15R67 1995
005.7'2—dc20 95-22892
CIP

McGraw-Hill books are available at special quantity discounts to use as premiums and sales promotions, or for use in corporate training programs. For more information, please write to the Director of Special Sales, McGraw-Hill, 11 West 19th Street, New York, NY 10011. Or contact your local bookstore.

Acquisitions editor: Steve Chapman
Editorial team: Lori Flaherty, Executive Editor
Andrew Yoder, Book Editor
Production team: Katherine G. Brown, Director
Lisa M. Mellott, Coding
Wanda S. Ditch, Desktop Operator
Joann Woy, Indexer
Design team: Jaclyn J. Boone, Designer
Katherine Lukaszewicz, Associate Designer

9117201
EL3

Contents

Introduction *xi*

1 Strategic Issues *1*
- 1.1 Why Code? *1*
- 1.2 Data Errors *1*
  - 1.2.1 Binary Symmetric Channel *1*
  - 1.2.2 Burst Channel *1*
  - 1.2.3 Binary Erasure Channel *2*
- 1.3 Software Notes *2*

2 Mathematical Tools for Coding *5*
- 2.1 Modulo Arithmetic *5*
  - 2.1.1 Modulo-2 Arithmetic *6*
- 2.2 Finite Fields *9*
  - 2.2.1 Groups *9*
  - 2.2.2 Fields *11*
- 2.3 Introduction to Polynomials *13*
- 2.4 Extension Fields *16*
- 2.5 Binary Extension Fields: Manual Construction *20*
  - 2.5.1 Constructing the Field *22*
  - 2.5.2 Representing the Field Elements *24*
- 2.6 Computer Methods for Galois Fields *26*
  - 2.6.1 Representing Field Elements *27*
- 2.7 More About Polynomials *30*
- 2.8 Cyclotomic Cosets *33*
- 2.9 Finding Minimal Polynomials *36*

3 Block and Cyclic Codes *41*
- 3.1 Introduction *41*
- 3.2 Linear Block Codes *44*

3.2.1 Encoding for Linear Block Codes *44*
3.2.2 Constructing the Generator Matrix *47*
3.2.3 Parity-Check Matrix *47*
3.3 Cyclic Codes *49*
3.4 Manual Encoding Methods for Cyclic Codes *51*
3.5 Modifications to Cyclic Codes *52*
3.5.1 Extended Codes *53*
3.5.2 Punctured Codes *54*
3.5.3 Expurgated Codes *54*
3.5.4 Augmented Codes *55*
3.5.5 Shortened Codes *56*
3.5.6 Lengthened Codes *56*
3.6 Hamming Codes *56*
3.6.1 Shortened Hamming Codes *58*

4 BCH and Reed-Solomon Codes *61*
4.1 BCH Codes *61*
4.1.1 Genesis of the Codes *62*
4.1.2 Types of BCH Codes *63*
4.1.3 Critical Features *63*
4.1.4 Constructing the Generator Polynomial *68*
4.1.5 Algorithmic Approach *71*
4.1.6 Computer Approach *73*
4.2 Nonbinary BCH Codes *79*
4.3 Reed-Solomon Codes *82*

5 Encoders and Decoders *85*
5.1 Division Method for Encoding Cyclic Codes *85*
5.1.1 Hardware Implementation *85*
5.1.2 Software Implementation *88*
5.2 Standard Array *89*
5.2.1 Hazards in Standard Array Decoding *92*
5.3 Syndromes for BCH Codes *93*
5.4 Peterson-Berlekamp Method *97*
5.5 Error Location *102*

6 Convolutional Codes *105*
6.1 Canonical Example *105*
6.1.1 Convolutional Encoder Viewed as a Moore Machine *106*

6.1.2 Convolutional Encoder Viewed as a Mealy Machine *106*
6.1.3 Moore Machines vs. Mealy Machines *109*
6.1.4 Notation and Terminology *112*
6.2 Tree Representation of a Convolutional Encoder *116*
6.3 Trellis Representation of a Convolutional Encoder *119*
6.4 Distance Measures *123*

7 Viterbi Decoding *127*
7.1 Introduction to Viterbi Decoding *127*
7.2 Viterbi Decoding Failures *141*
7.2.1 Minimum Free Distance *146*
7.2.2 Weight Distribution *147*
7.2.3 Information Sequence Weight *147*
7.3 Viterbi Decoding with Soft Decisions *147*
7.4 Practical Issues *152*
7.4.1 Decoding Depth *153*

8 Sequential Decoding *155*
8.1 Stack Decoding Algorithm *155*
8.1.1 Fano Metric *161*
8.2 Software for Stack Decoding *168*
8.2.1 Implementing the Stack *171*
8.2.2 Received Symbol Buffering *177*
8.2.3 State Table *179*
8.3 Fano Decoding Algorithm *180*

Appendices

A Minimal Polynomials of Elements in GF(2^m) for Chapter 2 *191*

B Stack Tables for Example 8.1 *197*

C Stack Tables for Example 8.2 *207*

D Stack Tables for Example 8.3 *215*

E Software *227*

Index *245*

About the Author *251*

Disk Warranty *260*

Introduction

Error-correction coding has traditionally been one of the most mathematically challenging of the various specialized disciplines that come together to make modern data communication and data storage possible. A good bit of the difficulty no doubt stems from the use of a type of arithmetic that is different from the usual "standard" arithmetic that we all use every day. This book is an attempt to change all that. After Chapter 1 provides a brief introduction to the strategic issues involved in error correction coding, Chapter 2 demystifies those elements of abstract algebra and Galois field arithmetic that are essential for successful implementation of encoders and decoders. Some basic software modules for generating Galois fields and performing arithmetic within these fields are also developed in Chapter 2. Chapter 3 uses techniques from Chapter 2 to introduce the basics of block and cyclic codes. Chapter 4 covers the specific details of BCH codes and Reed-Solomon codes, which together account for perhaps 80% of all block codes used in practical applications. In Chapter 5, the details of encoders and decoders for block and cyclic codes are introduced and particularly efficient decoding techniques for BCH codes are covered in some detail. The major emphasis is on software implementations rather than hardware implementations, but digital hardware designers will be readily able to "port" the various software constructions into the corresponding hardware constructions. In addition to Block and Cyclic codes, there is another large class of codes called *convolutional codes*, which are also widely used in practical applications. Chapter 6 introduces convolutional codes and the notation and terminology commonly associated with them. Viterbi decoding of convolutional codes is covered in Chapter 7. This coverage includes the development of software modules for performing the Viterbi algorithm. Although it enjoys widespread use, Viterbi decoding does have some limitations. Because of these limitations, alternative decoding techniques, such as stack decoding or Fano decoding, must often be used. These al-

ternative techniques, along with software for performing them, are covered in Chapter 8.

The subject of error correction coding is vast, and this book covers only a small part of it. However, the codes and techniques covered are the most widely used in practical applications. This, after all, accomplishes this book's original motive, which is to place powerful coding techniques as well as the knowledge needed to implement these techniques in the hands of individuals who need to use error-correction coding—even though these individuals are not coding experts.

1 Strategic Issues

1.1 Why Code?

Coding is performed for several different reasons. *Cryptographic* codes are used to protect information from unauthorized reception and to *authenticate* that a message was in fact sent by the party listed in the message as the originator. *Data compression* codes are used to reduce the amount of bandwidth needed to transmit data or to reduce the amount of memory needed to store a message. *Error control codes*, which are the subject of this book, are used to detect and/or correct errors that might be introduced into digital data when it is transmitted or stored.

1.2 Data Errors

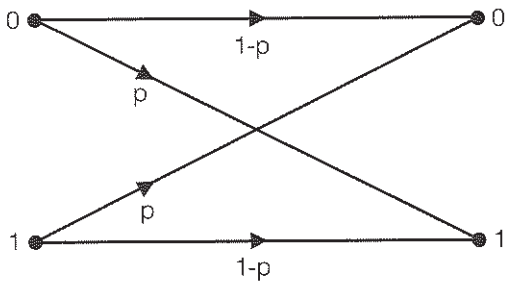
There are several different mechanisms by which errors are introduced into a data signal. The strategy for coding the signal to protect it against errors is often tailored to the particular error mechanism known or assumed to be operating in application of interest.

1.2.1 Binary Symmetric Channel

As shown in Fig. 1-1, in a *binary symmetric channel (BSC)*, the probability of a 1 to 0 error and the probability of a 0 to 1 error are equal. This mechanism is most often assumed to be the dominant error-producing mechanism for errors produced by additive Gaussian noise. If the probability of error is independent from bit-to-bit, the BSC can be further characterized as being a *discrete memoryless channel (DMC)*.

1.2.2 Burst Channel

In many situations, the noise in a channel will vary over time, being weak during some situations and strong during other intervals. Errors will be more likely to occur in bursts that coincide with intervals of strong noise. Such a channel is referred to as a *burst channel*, or a *channel with memory*.



■ **1-1** *Transition probabilities in the binary symmetric channel.*

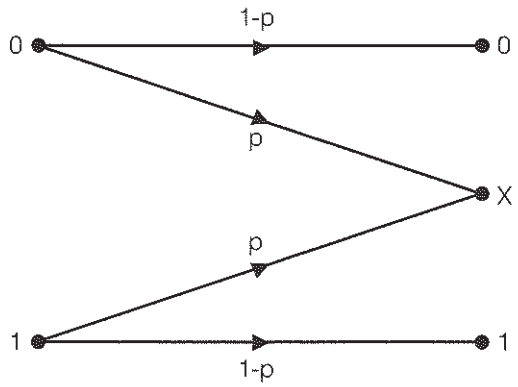
1.2.3 Binary Erasure Channel

Consider a baseband pulse transmission system in which a pulse of –1 volt is transmitted for a 0 and a pulse of +1 volt is transmitted for a 1. At the receiver, a pulse voltage less than –0.7 volt is interpreted as a 0. A pulse voltage greater than +0.7 is interpreted as a 1. The voltage range between –0.7 and +0.7 is considered as a "no-man's land" or *deadband*. Pulse voltages received in this range are deemed "too close to call" and are not interpreted as either 0 or 1. Such decisions not to decide is called *erasures*, and a channel that produces erasures is called a *binary erasure channel (BEC)*. The probability of a transmitted 0 being so corrupted that the received voltage exceeds 0.7 is so small that it is assumed to be impossible. Likewise for the probability of a transmitted 1 being so corrupted that the received voltage falls below –0.7. The probabilities of a 0-to-erasure failure or 1-to-erasure failure are not negligible, and special coding strategies are often used for correcting erasures. The BEC is depicted in Fig. 1-2, with erasures denoted as x.

1.3 Software Notes

The programs in this book were constructed as a set of *modules*. Modules have been described as "poor folks' objects"[1] and they are about as close as we can conveniently get to "true" objects without stepping up from **C** to **C++**. Modules support a subset of object-oriented programming that has been dubbed *object-based programming*. Modules don't support inheritance, but they do support abstraction and encapsulation. Modules are implemented in **C** as separate source files. This means that the programs in this book

1 McConnell: *Code Complete*, Microsoft Press, 1993.



■ **1-2** *Transition probabilities in the binary erasure channel.*

are organized into modules that make sense in the context of what we're trying to accomplish, but which do not necessarily match up with the way chapter contents are organized.

Each file contains both data and functions which can be declared static to make them visible only within the file. *Module variables* are static variables, which are global **within the file**. Not every function in a module makes use of every module variable, so as each function is presented in the text, in the way that it interacts with module variables is also covered. The software accompanying this book is described in Appendix D.

2 Mathematical Tools for Coding

Many of the calculations used to perform encoding, decoding, and performance analysis for the codes in this book make use of peculiar arithmetic and other mathematical techniques that are a bit out of the ordinary. These techniques, which are essential for the work in subsequent chapters, include modulo-p arithmetic, extension fields, and polynomial arithmetic. These topics sound a lot more difficult than they really are, and this chapter attempts to introduce and explain this material in a way that is relatively painless. This goal is consistent with the hands-on "cookbook" approach chosen for this book. However, presenting these techniques as isolated, unconnected rote procedures will prove somewhat unsatisfying to many readers and might actually make it more difficult for these readers to embrace these new techniques and begin using them. Therefore, this chapter is a compromise between extremes. The individual sections present the necessary foundation material embedded in a broader mathematical fabric for interested readers. However, wherever it is appropriate, the sections begin with a box entitled "The Basic Idea." The contents of this box briefly explains how the subsequent material in the section fits into the overall study of coding with an emphasis on why it is important for practical coding applications. Most sections end with a box entitled "The Bottom Line." This box summarizes the information that the reader needs to become familiar with and carry forward into subsequent sections and chapters.

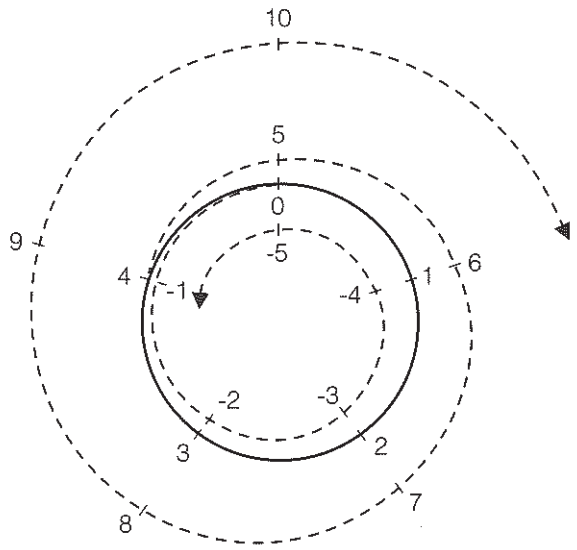
2.1 Modulo Arithmetic

The Basic Idea

Many error detection and correction (EDAC) codes are based on peculiar types of arithmetic that make use of mathematical structures called *fields*, which is covered in Section 2.2.

Two specific types of fields—*prime fields* and *extension fields*—both make extensive use of modulo arithmetic. Therefore, this section reviews modulo arithmetic as preparation for exploring fields.

Modulo arithmetic can be viewed as the result of "wrapping" the integers around a circle (Fig. 2-1) for the specific case of modulo-5 arithmetic. This wrapping causes 5 to be "the same as" or *congruent* to 0. This forces 6 to be congruent to 1, 7 to be congruent to 2, 10 to be congruent to 0, and so on. The congruences need not be limited to just positive integers. As shown in the figure, –1 is congruent to 4, –3 is congruent to 2, etc. The usual mathematical notation for congruence is a triple-bar equals sign, "≡." Thus "1 ≡ 16" is read as "1 is congruent to 16."



■ **2-1** *The modulo-5 number system can be viewed as the set of integers "wrapped" around a circle to establish congruences between 0 and 5, 1 and 6, –1 and 4, etc.*

2.1.1 Modulo-2 Arithmetic

The vast majority of the computations used to perform encoding and decoding involve modulo-2 arithmetic. This is due primarily to the close relationship between modulo-2 arithmetic and the *Boolean algebras* commonly used by designers of digital logic circuits.

Boolean Algebras

Encoders and decoders (not to mention computers) are constructed from logic circuits of various types. The logical behavior of these circuits can be modeled and analyzed using a particular type of algebra called *Boolean algebra* (which is named in honor of its inventor George Boole). Strictly speaking, Boolean algebra includes several different algebras, each of which uses different sets of logical operations, such as {AND, OR, INVERT} or {NAND, NOR, INVERT}. In the particular Boolean algebra of interest here, there are two *elements* or *values* as well as two logical *operations*. The values are TRUE or FALSE, and the operations are AND, and XOR (exclusive or). The results of these operations are listed in Tables 2-1 and 2-2.

Table 2-1 The logical AND operation of Boolean algebra.

AND	F	T
F	F	F
T	F	T

Table 2-2 The logical XOR operation of Boolean algebra.

XOR	F	T
F	F	T
T	T	F

Modulo-2 Arithmetic

A mathematical analog or *isomorphism* to the logical-valued Boolean algebra of Tables 2-1 and 2-2 uses values of 0 and 1, and operations of modulo-2 multiplication and addition, as shown in Tables 2-3 and 2-4. The value 0 corresponds to FALSE, and the value 1 corresponds to TRUE. Modulo-2 multiplication corresponds to the logical AND operation. The result of (a AND b) is true only if a and b are both TRUE.

Similarly, the result of $a \times b$ is 1 only if a and b both equal 1. Modulo-2 addition corresponds to the logical XOR operation. The result of (a XOR b) is 1 only if either a or b (but not both) is TRUE. The result of $a + b$ is 1 only if either a or b (but not both) is 1.

Table 2-3
Binary multiplication.

×	**0**	**1**
0	0	0
1	0	1

Table 2-4
Modulo-2 addition.

+	**0**	**1**
0	0	1
1	1	0

Programming Considerations Modulo arithmetic can be implemented in **C** for arbitrary moduli using the *remainder* operator. The expression x%y returns the remainder produced by dividing x by y. Assuming that x and y are both "legal" values modulo-q (*i.e.* $0 \leq x < q$ and $0 \leq x < q$) then modulo-q addition and multiplication can be easily implemented, as shown in the following code fragment.


```
int x, y, q, sum, product
/***********************************/
/*  Note: if q==0, a divide-by-zero  */
/*  error will occur.                */

/*  modulo-q addition  */
sum = (x+y) %q;

/*  modulo-q multiplication  */
product = (x*y) %q;

/**  end of fragment  **/
```

Modulo-2 arithmetic occurs so frequently that it might lead to greater program efficiency if we use Boolean operations in place of the remainder operation for modulo-2 arithmetic. As shown in the following code fragment, there are a number of possibilities.

```
int x, y, q, sum1, sum2, sum3, prod1, prod2

/*  modulo-2 addition  */

sum1 = x ^ y;      /*  uses bitwise XOR  */
sum2 = x != y;     /*  uses NOT EQUALS  */
sum3 = !(x && y); /*  uses logical NOT with logical AND
                   to implement logical NAND  */
```

```
/*  modulo-2 multiplication  */
prod1 = x & y     /*  uses bitwise AND  */
prod2 = x && y    /*  uses logical AND  */

/**  end of fragment  **/
```

2.2 Finite Fields

The Basic Idea

As so eloquently stated by R. J. McEliece,[1] "A field is a place where you can add, subtract, multiply and divide." Ordinary arithmetic takes place in an *infinite field*, so named because it has an infinite number of elements. Coding applications make use of arithmetic in a *finite field*. This section examines just what it takes for a collection of elements and operations to constitute a field. The related concept of a *group* is presented first because a field is most easily defined in terms of groups.

2.2.1 Groups

Modulo-2 addition is an example of a mathematical structure called a *group*, and we can use modulo-2 addition to illustrate a number of important properties that are shared by all groups:

1. The set of integers {1,0} is said to be *closed* under the operation of modulo-2 addition, because the modulo-2 addition of any two elements in the set produces a result that is also in the set.
2. The modulo-2 addition operation is *associative*, meaning that
$$(a + b) + c = a + (b + c)$$
for all a, b, and c which are elements of {0,1}.
3. The element 0 is called the *identity element* and has the property that
$$a + 0 = 0 + a = a$$
for all a, which are elements of {0,1}.
4. For each element a in {0,1}, there is an element $-a$ that when added to a produces a result of 0:
$$0 + 0 = 0 \qquad 1 + 1 = 0$$
The element $-a$ is called the *additive inverse of* a. (In modulo-2 addition, each element is its own inverse, but in general, this will not be true of other groups.)

1 R. J. McEliece: *Encyclopedia of Mathematics and Its Applications, Vol. 3: The Theory of Information and Coding,* Addison-Wesley, Reading MA, 1977.

The group formed by modulo-2 addition is further identified as a *commutative group* or *abelian group* because $a + b = b + a$ for all a and b, which are elements of {0,1}.

It turns out that modulo-q addition forms a commutative group for any value of q = 2,3,4,...

Example 2.1 Show that modulo-3 addition forms a commutative group.

Solution Modulo-3 addition results are summarized in Table 2-5.

Table 2-5
Modulo-3 addition.

+	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

Each of the five prerequisites for a commutative group are satisfied:

- ☐ Inspection of Table 2-5 reveals that modulo-3 addition is closed. All results are elements of the set {0,1,2}.
- ☐ Modulo-3 addition is associative.
- ☐ The identity element for modulo-3 addition is 0.
- ☐ Inspection of Table 2-5 reveals that each element has an additive inverse. Specifically,

  $$-0 = 0 \qquad -1 = 2 \qquad -2 = 1$$

  \end{array}
- ☐ Table 2-5 is symmetric about its main diagonal; therefore, the operation is commutative.

Modulo-3 **multiplication** also forms a commutative group over the nonzero elements {1, 2}. The results for modulo-3 multiplication are summarized in Table 2-6.

Table 2-6
Modulo-3 multiplication.

×	0	1	2
0	0	0	0
1	0	1	2
2	0	2	1

The five prerequisites (some of them reworded slightly for multiplication) are satisfied:

- ☐ Inspection of the table reveals that modulo-3 multiplication is closed over {1, 2}.
- ☐ Modulo-3 multiplication is associative.
- ☐ The identity element for multiplication (or *multiplicative identity*) is 1.
- ☐ Inspection of Table 2-6 reveals that each element in {1, 2} has a multiplicative inverse. Specifically,

  $$1 \times 1 = 1 \qquad 2 \times 2 = 1$$

- ☐ In algebraic form, the multiplicative inverse of a is denoted as a^{-1}.

  (Note: In general, each element will not be its own multiplicative inverse as it is here. For example, in modulo-5 multiplication, the multiplicative inverse of 2 is 3.)

It turns out that the set of **nonzero** elements $\{1, 2, \ldots, p-1\}$ modulo-p with multiplication forms a group over for all prime values of p. However, for composite (*i.e.,* nonprime) values of q, modulo-q multiplication does not form a group over the set $\{1, 2, \ldots, q-1\}$.

Example 2-2 Show that modulo-4 multiplication does not form a group over the elements {1, 2, 3}.

Solution Modulo-4 multiplication results are summarized in Table 2-7. The set of nonzero values {1, 2, 3} is not closed under modulo-4 multiplication because examination of the table reveals that $2 \times 2 = 0$. Furthermore, there is no multiplicative inverse for the element 2.

Table 2-7
Modulo-4 multiplication.

×	**0**	**1**	**2**	**3**
0	0	0	0	0
1	0	1	2	3
2	0	2	0	2
3	0	3	2	1

2.2.2 Fields

A set F, in conjunction with two operations (one "addition-like" called addition and one "multiplication-like" called multiplication)

defined over the elements of F forms a mathematical structure called a *field* if and only if the following conditions are satisfied:

1. The elements of F form a **commutative group** under the operation of addition.
2. The nonzero elements of F (that is all elements except the additive inverse) form a **commutative group** under the operation of multiplication.
3. The multiplication operation **distributes** over the addition operation.

The number of elements in a field can be either finite or infinite. The set R of real numbers, in conjunction with ordinary addition and multiplication, is an example of an infinite field. In coding theory, we will be concerned exclusively with **finite** fields.

Definition 2.1 The number of elements in a field is called the *order* of the field.

Finite fields are also called *Galois fields* in honor of Évariste Galois who outlined the connection between groups and polynomial equations in a letter to his friend Auguste Chevalier written on the eve of a duel on May 30, 1832 in which Galois was fatally wounded.[2] A Galois field of order q is denoted as GF(q). The field specified by Tables 2-3 and 2-4 is thus denoted as GF(2). This field has 2 elements and makes use of modulo-2 addition and modulo-2 multiplication. It is possible to extend this idea to a field with p elements using modulo-p arithmetic, provided that p is prime. Such a field is called a *prime field*. As you will see in Section 2.4, it is also possible to construct fields GF(q), where $q = p^m$, $m = 1, 2,...$

The Bottom Line

- ☐ Groups are a set of elements with an operation defined for these elements such that:
  - ~ The operation is **closed** over the set of elements.
  - ~ The operation is **associative.**
  - ~ The set contains an **identity element.**
  - ~ For each element in the set, there is an **inverse** also contained in the set.
- ☐ If a group's operator is commutative, the group is called a **commutative group** or **abelian group.**
- ☐ Modulo-q addition forms a commutative group for all $q \geq 2$.

2 I. Stewart: *Galois Theory,* 2nd ed., Chapman & Hall, London, 1989.

- Modulo-p multiplication forms a commutative group over the nonzero elements $\{1, 2, \ldots, p-1\}$ for all **prime** values of p.
- Modulo-q multiplication does not form a group for nonprime q.
- Fields are a set of element F with two operations—"addition" and "multiplication"—defined for these elements such that:
  - The elements of F form a commutative group under addition.
  - The nonzero elements of F form a commutative group under multiplication.
  - Multiplication distributes over addition.
- Finite fields are also called **Galois fields** and are therefore denoted as GF(p).

2.3 Introduction to Polynomials

The Basic Idea

This section presents a review of polynomial arithmetic for polynomials whose coefficients are drawn from prime fields and which require only modulo arithmetic for operating on these coefficients. This background is needed before we can explore the construction of extension fields in Section 2.4. Ultimately, we will be concerned with polynomials whose coefficients are drawn from extension fields, and which therefore require extension-field arithmetic for operating on these coefficients. This second type of polynomial is covered in Section 2.7.

Much of coding theory (including the construction of extension fields to be covered shortly) is based on *polynomial arithmetic*. A *polynomial in* x is simply a weighted sum of powers of x. Some examples of polynomials in x are

$$f_1(x) = 5x^2 + 3x + 1$$
$$f_2(x) = x^4 + x^3 + 1$$
$$f_3(x) = x^3 - 2$$

The degree of the polynomial is the highest power of x appearing in the polynomial. The degree of $f_1(x)$ is 2; the degree of $f_2(x)$ is 4; and the degree of $f_3(x)$ is 3.

In coding theory, we will often be interested in polynomials in which the coefficient of each term is either 0 or 1. In the context of coding theory, such a polynomial is called a *polynomial over* GF(2). Of the three example polynomials shown, $f_2(x)$ is the only polynomial over GF(2). When performing arithmetic on polynomials over GF(2), powers of x can be multiplied in the usual way; that is,

$$x^2 \bullet x^3 = x^5$$
$$x \bullet x^7 = x^8$$
$$x \bullet x = x^2$$

However, the addition of coefficients is performed using modulo-2 addition. For example,

$$x^2 + x^2 = 1 \bullet x^2 + 1 \bullet x^2 = (1 + 1) \bullet x^2 = 0 \bullet x^2 = 0,$$

$$\begin{aligned}(x + 1) + (x^2 + x) &= 1 \bullet x + 1 + 1 \bullet x^2 + 1 \bullet x \\ &= 1 \bullet x^2 + (1 + 1) \bullet x + 1 \\ &= 1 \bullet x^2 + 0 \bullet x + 1 = x^2 + 1\end{aligned}$$

and

$$x^5 + x^5 + x^5 = (1 + 1 + 1) \bullet x^5 = 1 \bullet x^5 = x^5$$

Because each element in GF(2) is its own inverse, subtraction of polynomials over GF(2) is the same as addition:

$$\begin{aligned}x^3 + x + 1 &= x^3 + x + 1 \\ -(x^3 + x^2 + 1) &= \frac{+(x^3 + x^2 + 1)}{x^2 + x + 1}\end{aligned}$$

However, this is not true for polynomials over GF(p) $p > 2$. Consider the subtraction of two polynomials over GF(3):

$$\begin{aligned}x^3 + x + 1 &= x^3 + x + 1 \\ -(2x^3 + x^2 + 2) &= \frac{+(x^3 + 2x^2 + 1)}{2x^3 + 2x^2 + x + 2}\end{aligned}$$

[Note that in GF(3), $-2 = 1, -1 = 2$.]

A general polynomial of degree N can be written as

$$f(x) = b_N x^N + b_{N-1} x^{N-1} + \ldots + b_2 x^2 + b_1 x + b_0 \qquad (2.1)$$

where some of the b_n might be zero for $n < N$. (If $b_N = 0$, the degree of the polynomial is not N.) A more compact notation is

$$f(x) = \sum_{n=0}^{N} b_n x^n \qquad (2.2)$$

Programming Considerations How can a polynomial such as Eqs. 2.1 or 2.2 be represented in software? We can store the coefficients of $f(x)$ in an array, for example coef[], with the coefficient for x^n in location n. When the polynomial is represented this way, it is easy to multiply or divide the polynomial by a power of x. To multiply $f(x)$ by x^k, for $k \geq 0$, simply shift the coefficients upward by k locations in the array, as shown in the following code fragment:

```
/**  fragment to multiply polynomial by x**k  **/

/*  shift original N coefficients  */

for( n=N; n>=0; n--)

     {
     coef[n+k]  =  coef[n];
     }

/*  store zeros for the k lowest-order new coefficients  */

for( n=k-1; n>=0; n--)
     {
     coef[n]  =  0;
     }

/*  increase degree of polynomial by k  */

N  += k;

/**  end of fragment  **/
```

To divide $f(x)$ by x^k for $k \geq 0$, shift the coefficients downward by k locations in the array as shown in the following program fragment. The k lowest-order terms in the original polynomial will become the remainder of the division operation.

```
/**  fragment to divide polynomial by x**k  **/

/*  save remainder  */

for( n=0; n<k; n++)
     {
     remainder[n]  =  coef[n];
     }

/*  downshift the higher-order coefficients by k places  */

for( n=0; n<=(N-k); n++)
     {
     coef[n]  =  coef[n+k];
     }

/*  store zeros in locations originally occupied by k
higher-order terms  */
```

```
for( n=(N-k+1); n<=N; n++)
     {
     coef[n]  =  0;
     }

/**  end of fragment  **/
```

2.4 Extension Fields

The Basic Idea

Finite fields based solely on modulo arithmetic are restricted to have prime numbers of elements. This section develops a specific *extension field*, which has $2^3 = 8$ elements. The principles used in this section are generalized in Section 2.5 to include extension fields having 2^m elements where m is any positive integer. Extension fields, especially those having 2^m elements, are widely used in coding applications.

As demonstrated in Section 2.1, it is a simple matter to construct a field GF(p) for any prime p by using modulo-p addition and multiplication. In working with various codes, it turns out to be very useful if all of the necessary manipulations can be performed within a field having 2^m elements, with each element being an m-bit binary value. But so far, we have seen only fields in which the number of elements is a prime number; and of course, 2^m is not prime for $m > 1$. What happens if we try to construct a field with 2^m elements using modulo-2^m arithmetic?

Example 2.3 Consider the specific case of $2^m = 2^3 = 8$. The operations of modulo-8 addition and modulo-8 multiplication are defined in Tables 2-8 and 2-9. Examination of Table 2-8 reveals that modulo-8 addition does form a commutative group (see Section 2.2.2). However, examination of Table 2-9 reveals that there are no multiplicative inverses for the even numbers 2, 4, and 6. This means that the integers 1, 2, 3, 4, 5, 6, and 7 do not form a group under modulo-8 multiplication; therefore, modulo-8 arithmetic cannot be used to construct a field GF(2^m).

Although it is not possible to construct a Galois field GF(2^m) using modulo-2^m multiplication, it is possible to construct such a field using *polynomial arithmetic*.

Table 2-8 Modulo-8 addition.

+	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	0
2	2	3	4	5	6	7	0	1
3	3	4	5	6	7	0	1	2
4	4	5	6	7	0	1	2	3
5	5	6	7	0	1	2	3	4
6	6	7	0	1	2	3	4	5
7	7	0	1	2	3	4	5	6

Table 2-9 Modulo-8 multiplication.

×	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	0	2	4	6
3	0	3	6	1	0	3	6	1
4	0	4	0	4	0	4	0	4
5	0	5	2	7	4	1	6	3
6	0	6	4	2	0	6	4	2
7	0	7	6	5	4	3	2	1

Another Way to Create Fields

So far we have considered fields in which the elements are integers and the operations are modulo-p addition and modulo-p multiplication. Except for the "wrap-around" that occurs between $p - 1$ and 0, these operations are closely related to the usual arithmetic operations of addition and multiplication. In order to construct fields having a composite (nonprime) number of elements, we must resort to operations that are not so closely related to the operations of ordinary arithmetic. Despite their unfamiliarity, these operations are not difficult; and virtually all of coding theory is based on fields that are built upon a few simple rules.

Our principal applications for error-correction coding will involve digital values in binary form. Therefore, we will focus on finding a

way to construct finite fields having 2^m elements with each element being an m-bit binary value.

Consider a commutative group based on elements that are 3-bit binary values and the operation of bit-by-bit modulo-2 addition (or equivalently, bit-by-bit logical exclusive OR). The results of this operation are listed in Table 2-10. All of the requirements for a group are satisfied:

- ☐ The operation is associative.
- ☐ The identity element is 000.
- ☐ Each element is its own inverse.

■ Table 2-10 Bit-by-bit modulo-2 addition of 3-bit values.

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

Assume that this is the additive group for our desired field. We must now find a "multiplication-like" operation that forms a commutative group over the nonzero elements 001, 010, 011, 100, 101, 110, and 111. By "multiplication-like," we mean that whatever this operation turns out to be, it must distribute over the bit-by-bit modulo-2 addition defined by Table 2-10. An operation that meets all of the requirements is defined by Table 2-11. What are the rules used to define this table? None of the logical operations—AND, OR, NAND, NOR, etc.—fits the bill. To find suitable rules, we need to turn to polynomial arithmetic. Let each value in Table 2-11 represent a polynomial in α. Each bit within a value represents the coefficient of a particular power of α. Specifically,

111 represents $\alpha^2 + \alpha + 1$
110 represents $\alpha^2 + \alpha$
101 represents $\alpha^2 + 1$
100 represents α^2
011 represents $\alpha + 1$
010 represents α
001 represents 1
000 represents 0

■ Table 2-11 Multiplication-like operation for 3-bit values.

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

We can try defining a multiplicative operation over this set of polynomials using the ordinary rules of algebraic multiplication. For example

$$\alpha \bullet (\alpha + 1) = \alpha^2 + \alpha \Leftrightarrow (010) \bullet (011) = (110)$$

However, multiplication defined in this way is not closed over the set of 3-bit elements. For example

$$\alpha \bullet (\alpha^2 + 1) = \alpha^3 + \alpha^2$$

and we have no way to represent α^3 in our 3-bit scheme. In much the same way that we define $p \equiv 0$ to force closure in modulo-p arithmetic, we can define $\alpha^3 \equiv \alpha + 1$. This relationship can be used to express α^3, α^4, α^5,... in terms of 1, α, and α^2. For example, we can write

$$\begin{aligned}
\alpha^3 &= \alpha + 1 \text{ (by definition)} \\
\alpha^4 &= \alpha \bullet \alpha^3 = \alpha(\alpha + 1) = \alpha^2 + \alpha \\
\alpha^5 &= \alpha \bullet \alpha^4 = \alpha(\alpha^2 + \alpha) = \alpha^3 + \alpha^2 \\
&= (\alpha + 1) + \alpha^2 = \alpha^2 + \alpha + 1 \\
\alpha^6 &= \alpha \bullet \alpha^5 = \alpha(\alpha^2 + \alpha + 1) = \alpha^3 + \alpha^2 + \alpha \\
&= (\alpha + 1) + \alpha^2 + \alpha = \alpha^2 + 1
\end{aligned}$$

Table 2-12 lists elements α^0 through α^{10} expressed in terms of 1, α, and α^2. Looking at the table, we see that $\alpha^7 = \alpha^0$, $\alpha^8 = \alpha^1$, $\alpha^9 = \alpha^2$, etc. Similarly to the way that defining $5 \equiv 0$ created a closed finite subset of the integers used to construct GF(5), defining $\alpha^3 = \alpha + 1$ has created a closed finite subset of the powers of α. Therefore, we can conclude that the operation at work in Table 2-11 is the special form of polynomial multiplication, based upon the relationship $\alpha^3 = \alpha + 1$. Tables 2-10 and 2-11 comprise exactly the sort of thing we said we needed—a field containing 2^m elements where each element has an m-bit binary value. Such fields are called *extension fields* of GF(2), and the field GF(2) is called the *ground field* or *prime field* of GF(2^m).

■ Table 2-12 Three different representations for GF(2^3).

Power form	Polynomial form	3-tuple form
0	0	000
1	1	001
α	α	010
α^2	α^2	100
α^3	$\alpha + 1$	011
α^4	$\alpha^2 + \alpha$	110
α^5	$\alpha^2 + \alpha + 1$	111
α^6	$\alpha^2 + 1$	101
α^7	1	001
α^8	α	010
α^9	α^2	100
α^{10}	$\alpha + 1$	011

The Bottom Line

Using only modulo arithmetic, it is not possible to construct fields having nonprime numbers of elements. However, this section has shown that it is possible to use a special form of polynomial arithmetic to build the extension field GF(2^3), which has 2^3 elements. In Section 2.5, we tackle the problem of how to extend our results for GF(2^3) to the general case of GF(2^m).

2.5 Binary Extension Fields: Manual Construction

The Basic Idea

Making use of *primitive polynomials*, it is possible to define an extension field GF(2^m) for any integer $m \geq 2$. Such fields will be used extensively throughout the rest of this book. This might be one of the most difficult sections in the book, but it is also one of the most important!

If q is not prime, the set of elements $\{1,2,...,q-1\}$ does not form a group under modulo-q multiplication; therefore, modulo-q multiplication cannot be used to define a field GF(q). However, in Section 2.4, we found that it is possible to define a field GF(2^3). In fact, it is possible for **any** prime p to define fields having p^m elements by using polynomial arithmetic in place of modulo arith-

metic. However, in this book, we are primarily interested in extension fields of the form GF(2^m) having 2^m elements. A particular type of polynomial called a *primitive polynomial*, plays a role in the construction of extension fields that is similar to the role played by the modulus in construction of prime fields.

Definition 2.2 Let p(x) be a polynomial of degree m over GF(2). If p(x) is not divisible by any polynomial over GF(2) of degree one through $m - 1$, then p(x) is *irreducible* over GF(2).

Reality Check

A polynomial of degree m over GF(2) is a polynomial of degree m in which each coefficient is an element of GF(2). The polynomial $p(x) = x^3 + x + 1$ is *irreducible* over GF(2) because p(x) is not divisible by any of the polynomials of degree 2 over GF(2)—x^2, $x^2 + 1$, $x^2 + x$, $x^2 + x + 1$—nor by any of the polynomials of degree 1 over GF(2)—x, $x + 1$. On the other hand, the polynomial $x^3 + x$ is not irreducible because it is factorable into $(x^2 + x)(x + 1)$.

Result 2.1 Any irreducible polynomial over GF(2) of degree m divides $x^{2m} - 1 + 1$.

Reality Check

We said that $x^3 + x + 1$ is an irreducible polynomial of degree 3 over GF(2), so it should divide $x^{2^4-1} + 1 = x^7 + 1$. And if we attempt the division, we find that

$$\frac{x^7 + 1}{x^3 + x + 1} = x^4 + x^2 + x + 1$$

Definition 2.3 We know from Result 2.1 that if $p(x)$ is an irreducible polynomial over GF(2) of degree m, then $p(x)$ divides $x^n + 1$ for $n = 2^m - 1$. If this value of n is the smallest positive integer for which $p(x)$ divides $x^n + 1$, then $p(x)$ is a *primitive polynomial of degree* m over GF(2).

Reality Check

The polynomial $p_1(x) = x^3 + x + 1$ is a *primitive polynomial of degree* 3 over GF(2), because $p_1(x)$ divides $x^7 + 1$ and it does not divide $x^6 + 1$, $x^5 + 1$, $x^4 + 1$, or $x^3 + 1$. On the other hand, the polynomial $p_2(x) = x^4 + x^3 + x^2 + x + 1$ is an irreducible polynomial of degree 4 over GF(2) because $p_2(x)$ divides $x^{2^4-1} + 1 = x^{15} + 1$. However, $p_2(x)$ is not a primitive polynomial because it also divides $x^{10} + 1$ and $x^5 + 1$.

Primitive polynomials over GF(2) for $2 \leq m \leq 30$ are listed in Appendix C.

2.5.1 Constructing the Field

For the case of binary coefficients, there are 2^m different polynomials of degree $\geq m-1$. The field GF(2^m) will have 2^m elements, each of which will be one of the possible 2^m polynomials of degree $m-1$. In order to preserve the properties of addition and multiplication that are needed in order to have a field, each of the 2^m polynomials must be assigned to the elements of GF(2^m) in a particular order. Recipe 2.1 provides a means for making this assignment correctly.

Recipe 2.1 Constructing the binary extension field GF(2^m). The complete extension field will consist of 2^m elements whose power forms are $0, 1, \alpha, \alpha^2, \alpha^3, \ldots, \alpha^{2m-2}$. The purpose of this recipe is to generate the polynomial forms for each of these elements.

1. Each of the elements $0, 1, \alpha, \alpha^2, \ldots, \alpha^{m-1}$ has a polynomial form that is exactly the same as its power form.
2. Select a primitive polynomial of degree m over GF(2). (In other words, select a primitive polynomial of degree m having coefficient values of 0 or 1.) Let $p(x)$ denote the primitive polynomial selected:
$$p(x) = x^m + f_{m-1}x^{m-1} + f_{m-2}x^{m-2} + \ldots + f_2x^2 + f_1x + f_0$$
3. Substitute α for x, set $p(\alpha) = 0$ and solve for α^m
$$\alpha^m = f_{m-1}\alpha^{m-1} + f_{m-2}\alpha^{m-2} + \ldots + f_2\alpha^2 + f_1\alpha + f_0 \qquad (2.3)$$
4. For $k = m, m+1, m+2, \ldots, 2^m - 3$, we can obtain a polynomial expression for α^{k+1} by multiplying the polynomial for α^k by α:
$$\alpha^{k+1} = \alpha \bullet \alpha^k$$

Sometimes the result will contain a term for α^m and sometimes the degree of the highest-degree term will be $m-1$ or less. When the result does not contain a term for α^m, the result is the correct polynomial form for the α^{k+1} element. When the result does contain a term for α^m, substitute the polynomial form for α^m (Eq. 2.3) and simplify to obtain the polynomial form for α^{k+1}.

Recipe 2.1 can be used directly to construct an extension field GF(2^m) for any m, but it can also be used as the theoretical basis for a more easily mechanized procedure that will be presented in

Section 2.6. We can see now that the relationship $\alpha^3 = \alpha + 1$ that we used in Section 2.4 is derived from $p(x) = x^3 + x + 1$, which is a primitive polynomial of degree 3 over GF(2).

Example 2.4 Construct an extension field GF(2^4).

Solution

1. The first five elements are 0, 1, α, α^2, and α^3.
2. A primitive polynomial of degree 4 over GF(2) is $p(x) = x^4 + x + 1$.
3. $p(\alpha) = \alpha^4 + \alpha + 1 = 0 \Rightarrow \alpha^4 = \alpha + 1$

   A. $\alpha^5 = \alpha \cdot \alpha^4 = \alpha(\alpha + 1) = \alpha^2 + \alpha$

   B. $\alpha^6 = \alpha \cdot \alpha^5 = \alpha(\alpha^2 + \alpha) = \alpha^3 + \alpha^2$

   C. $\alpha^7 = \alpha \cdot \alpha^6 = \alpha(\alpha^3 + \alpha^2) = \alpha^4 + \alpha^3$
   substituting $\alpha^4 = \alpha + 1$ yields
   $\alpha^7 = \alpha^3 + \alpha + 1$

   D. $\alpha^8 = \alpha \cdot \alpha^7 = \alpha(\alpha^3 + \alpha + 1) = \alpha^4 + \alpha^2 + \alpha$
   substituting $\alpha^4 = \alpha + 1$ yields
   $\alpha^8 = \alpha^2 + 1$

   E. $\alpha^9 = \alpha \cdot \alpha^8 = \alpha(\alpha^2 + 1) = \alpha^3 + \alpha$

   F. $\alpha^{10} = \alpha \cdot \alpha^9 = \alpha(\alpha^3 + \alpha) = \alpha^4 + \alpha^2$
   substituting $\alpha^4 = \alpha + 1$ yields
   $\alpha^{10} = \alpha^2 + \alpha + 1$

   G. $\alpha^{11} = \alpha \cdot \alpha^{10} = \alpha(\alpha^2 + \alpha + 1) = \alpha^3 + \alpha^2 + \alpha$

   H. $\alpha^{12} = \alpha \cdot \alpha^{11} = \alpha(\alpha^3 + \alpha^2 + \alpha) = \alpha^4 + \alpha^3 + \alpha^2$
   substituting $\alpha^4 = \alpha + 1$ yields
   $\alpha^{12} = \alpha^3 + \alpha^2 + \alpha + 1$

   I. $\alpha^{13} = \alpha \cdot \alpha^{12} = \alpha(\alpha^3 + \alpha^2 + \alpha + 1) = \alpha^4 + \alpha^3 + \alpha^2 + \alpha$
   substituting $\alpha^4 = \alpha + 1$ yields
   $\alpha^{13} = \alpha^3 + \alpha^2 + 1$

   J. $\alpha^{14} = \alpha \cdot \alpha^{13} = \alpha(\alpha^3 + \alpha^2 + 1) = \alpha^4 + \alpha^3 + \alpha$
   substituting $\alpha^4 = \alpha + 1$ yields
   $\alpha^{14} = \alpha^3 + 1$

These results are summarized in Table 2-13.

■ Table 2-13 Three different representations for GF(2^4).

Power form	Polynomial form	4-tuple form
0	0	0000
1	1	0001
α	α	0010
α^2	α^2	0100
α^3	α^3	1000
α^4	$\alpha + 1$	0011
α^5	$\alpha^2 + \alpha$	0110
α^6	$\alpha^3 + \alpha^2$	1100
α^7	$\alpha^3 + \alpha + 1$	1011
α^8	$\alpha^2 + 1$	0101
α^9	$\alpha^3 + \alpha$	1010
α^{10}	$\alpha^2 + \alpha + 1$	0111
α^{11}	$\alpha^3 + \alpha^2 + \alpha$	1110
α^{12}	$\alpha^3 + \alpha^2 + \alpha + 1$	1111
α^{13}	$\alpha^3 + \alpha^2 + 1$	1101
α^{14}	$\alpha^3 + 1$	1001

2.5.2 Representing the Field Elements

Each element of the extension field GF(p^m) can be represented by an m-digit base-p value. For the specific case of GF(2^m), this means that each element is represented by an m-bit binary value. Each bit within a value corresponds to a coefficient in a polynomial of degree $m - 1$. Obviously, for such a representation to be unambiguous, the bits must correspond to the coefficients of the polynomials when the terms of the polynomials are in some specified order. Four obvious schemes can be used for this representation:

1. The polynomial is arranged from lowest-degree term to highest-degree term with the leftmost bit assigned to the α^0 term. For $m = 4$, this approach yields:

$$1 + \alpha + \alpha^2 + \alpha^3 \leftrightarrow 1111$$
$$1 + \alpha \leftrightarrow 1100$$
$$1 \leftrightarrow 1000$$
$$\alpha + \alpha^3 \leftrightarrow 0101$$

   This approach is used by Rhee,[3] Lin and Costello,[4] and by MacWilliams and Sloane.[5]

3 M. Y. Rhee: *Error Correcting Coding Theory,* McGraw-Hill, New York, 1989.

4 S. Lin and D. J. Costello: *Error Control Coding: Fundamentals and Applications,* Prentice-Hall, Englewood Cliffs, NJ, 1983.

5 F. J. MacWilliams and J. J. A. Sloane: *The Theory of Error-Correcting Codes,* North Holland, Amsterdam, 1977.

2. The polynomial is arranged from lowest-degree term to highest-degree term with the rightmost bit assigned to the α^0 term. For $m = 4$, this approach yields:

$$1 + \alpha + \alpha^2 + \alpha^3 \leftrightarrow 1111$$
$$1 + \alpha \leftrightarrow 0011$$
$$1 \leftrightarrow 0001$$
$$\alpha + \alpha^3 \leftrightarrow 1010$$

This approach is used by Berlekamp.[6]

3. The polynomial is arranged from highest-degree term to lowest-degree term with the rightmost bit assigned to the α^0 term. For $m = 4$, this approach yields:

$$\alpha^3 + \alpha^2 + \alpha + 1 \leftrightarrow 1111$$
$$\alpha + 1 \leftrightarrow 0011$$
$$1 \leftrightarrow 0001$$
$$\alpha^3 + \alpha \leftrightarrow 1010$$

This approach is used by Peterson and Weldon.[7]

4. The polynomial is arranged from highest-degree term to lowest-degree term with the leftmost bit assigned to the α^0 term. For $m = 4$, this approach yields:

$$\alpha^3 + \alpha^2 + \alpha + 1 \leftrightarrow 1111$$
$$\alpha + 1 \leftrightarrow 1100$$
$$1 \leftrightarrow 1000$$
$$\alpha^3 + \alpha \leftrightarrow 0101$$

Although it is not as frequently used as approach 1, approach 3 is the most computationally convenient and is therefore used in the remainder of this book.

The Bottom Line

Recipe 2.1 can be used to construct the binary extension field GF(2^m) provided that we have the following things:

- ☐ a primitive polynomial $p(x)$ of degree m with coefficients drawn from GF(2).
- ☐ an ability to compute $xf(x)$ given $f(x)$ a polynomial in x.
- ☐ an ability to use the primitive polynomial to reduce the degree of $xf(x)$ whenever the result has a degree higher than $m - 1$.

6 E. R. Berlekamp: *Algebraic Coding Theory,* McGraw-Hill, New York, 1968.

7 W. W. Peterson and E. J. Weldon, Jr.: *Error-Correcting Codes,* 2nd ed., MIT Press, Cambridge, MA, 1972.

2.6 Computer Methods for Galois Fields

The Basic Idea

Our present goal is to implement Recipe 2.1 in software. This involves defining a format for storing coefficients, and designing software to generate the elements of GF(2^m). The generation of these elements comprises two major sub-tasks:

1. Given a field element expressed as a polynomial $f(\alpha)$, compute $\alpha f(\alpha)$.
2. Use a primitive polynomial to reduce the degree of $\alpha f(\alpha)$ whenever the result of 1 has a degree higher than $m - 1$.

Programming Considerations Now let's examine the results of Example 2.4 with an eye towards computer realization of Recipe 2.1:

1. The first element is 0000.
2. The second element is 0001.
3. The third element can be obtained by shifting the second element one place to the left to obtain
$$\alpha^1 \leftrightarrow 0010$$
4. Likewise, the fourth and fifth element can be obtained by left-shifting the third and fourth elements to obtain:
$$\alpha^2 \leftrightarrow 0100$$
$$\alpha^3 \leftrightarrow 1000$$
5. If we try to obtain α^4 by a simple left shift of α^3, we will obtain the 5-bit value of 10000, which is incorrect. Instead, the sixth element α^4 is defined by the minimal polynomial selected in step 3 of Example 2.4:
$$\alpha^4 = \alpha + 1 \leftrightarrow 0011$$
6. The binary values for α^5 and α^6 can be obtained via left-shifting of α^4 and α^5:
$$\alpha^5 \leftrightarrow 0110$$
$$\alpha^6 \leftrightarrow 1100$$
7. If we shift the binary value for α^6 one bit to the left, we obtain 11000, which is a 5-bit value. The correct value for α^7 is 1011. The trick is to keep only the rightmost 4 bits of 11000 and add 0011. What is this really doing? It's simply making use of $\alpha^4 = \alpha + 1$. Removing the leftmost bit of 11000 corresponds to subtracting α^4 from the polynomial, and adding 0011 corresponds to adding $\alpha + 1$ to the polynomial.

These observations can be generalized into the following recipe which can be used directly to obtain the m-bit binary representations of the elements in GF(2^m).

Recipe 2.2 Constructing the binary extension field GF(2^m).

1. The first element is the all-zero value.
2. The second element has the least significant bit (LSB) set to 1, with all remaining bits set to 0.
3. Shift all bits one position to the left, shifting a zero into the LSB. If this shifting operation causes a 1 to be shifted out of the leftmost bit of the rightmost m bits, then mask away all but the rightmost m bits and exclusive-or (XOR) the result with the m-bit binary representation corresponding to α^m.
4. Repeat step 3 a total of $2^m - 2$ times.

2.6.1 Representing Field Elements

Elements of GF(2^m) can be represented in a computer in one of two different ways. Consider the extension field GF(2^4) shown in Table 2-13. The computer representation of each element can be either an integer 0, 1, 2,...,($2^m - 2$) that equals the exponent of the element when expressed as a power of α, or the element can be expressed as an m-bit value directly corresponding to the m-tuple representation of the element. Each form has advantages and disadvantages. As shown previously, multiplication is very easy with the exponential form; the exponents are simply added:

$$\alpha^m \bullet \alpha^n = \alpha^{(m+n)}$$

However, addition is easier in the m-tuple form. In the approach used for manual computations, the multiplications are performed in exponential form, and additions are performed in m-tuple form. One form is easily converted to the other by referring to a table of the elements in the particular extension field of interest. This approach can also be implemented in a computer for relatively small values of m. Large values of m require large amounts of memory to hold the table of field elements. The table containing the elements for GF(2^m) will require 2^m locations with each location containing at least m bits. For m = 16, the table will need 131,072 bytes of memory. For some computer systems this might not be a problem; but for many small systems, an alternative approach might be needed. The ideal solution would be an algorithmic means for converting directly between the exponential form and m-tuple form without using a table lookup. Unfortunately, a simple direct conversion algorithm does not exist. However, there are several algo-

rithmic conversion strategies that can be used in lieu of a memory-hogging lookup table.

Brute Force

A brute force approach would entail generating the desired field $GF(2^m)$ one element at a time without permanently saving the result. As each element is generated, it is examined to see if it is the element needed for the conversion being performed. Each element in succession is generated, examined, and discarded until the required element is obtained. Once the required element is obtained, the conversion is performed and the process is finished. This approach is very thrifty with memory, but **on average**, it will require 2^{m-1} iterations to reach the field element required for a conversion.

Modified Method

The brute force approach can be modified to reduce somewhat the number of iterations needed to find the required field element when converting from exponential form to m-tuple form. Simply generate the field elements using the methods of Section 2.5, but only store every k-th element. This will create an array of m-tuple values for elements α^j with $j = 1, 1 + k, 1 + 2k, 1 + 3k,...$ To convert an exponent, for example e, into m-tuple form we find the j-th element in the table such that $j \leq e < (j + k)$. Then, starting with the m-tuple representation for α^j, we use the method of Section 2.5 to generate the m-tuple representations for α^{j+1}, α^{j+2}, α^{j+3},... stopping when the element α^e is reached. For $m = 16$, the brute force method will take, on average, 32,768 iterations to perform the conversion. For $m = 16$ and $k = 1{,}024$, the modified method will take on average only 512 iterations to perform this conversion. For this particular case, the table needed for the brute force method will occupy only 128 bytes of memory. The speed-for-memory trade can be tailored to any particular computer system by an appropriate selection for k. Several different combinations of table size and iterations per conversion for the $m = 16$ case are listed in Table 2-14.

■ Table 2-14 Table sizes and iteration counts for converting forms of GF(2^k).

k	Table size (bytes)	Avg. iterations per conversion
1	131,072	0
2	65,536	1
4	32,768	2
8	16,384	4
16	8,192	8
32	4,096	16
64	2,048	32
128	1,024	64
256	512	128
512	256	256
1,024	128	512
2,048	64	1,024
4,096	32	2,048
8,192	16	4,096
16,384	8	8,192
32,768	4	16,384

Software Implementation

Listing 2-1 contains the BuildExtensionField() function that builds a table of field elements for small values of m. The required input alpha_m can be obtained by setting a primitive polynomial $p(\alpha)$ of degree m over GF(2) equal to zero and then solving for α^m. The variables Galois_Field[] and Num_Field_Elements are module variables that are shared with other functions in module gfield.c. This module contains a function GetAlphaM[] that accesses a data file primpoly.dat to obtain a value of α^m in the proper format to use as an input to BuildExtensionField(). As shown, BuildExtensionField() uses unsigned integers in order to represent field elements in m-tuple form for $m \leq 16$. Before BuildExtensionField() can be used to construct GF(2^m) for $m > 9$, the dimensions of Galois_Field[] must be increased to a value greater than or equal to 2^m. The Software Appendix discusses number of utility functions that are provided for performing arithmetic and format conversions on field elements.

Listing 2-1

```
static unsigned int Galois_Field[512];
static unsigned int Num_Field_Elements;
```

```
/*****************************/
/*                           */
/*   BuildExtensionField()   */
/*                           */
/*****************************/

void BuildExtensionField( int m, int alpha_m)
{
unsigned int shifted_out_mask, keeper_mask, i;

shifted_out_mask = IntPower(2, m);
keeper_mask = shifted_out_mask-1;
Galois_Field[0]=1;

for(i=1; i<keeper_mask; i++) {
  Galois_Field[i] = Galois_Field[i-1]<<1;
  if(Galois_Field[i] >= shifted_out_mask) {
    Galois_Field[i] &= keeper_mask;
    Galois_Field[i] ^= alpha_m;
    }
  }
Num_Field_Elements = IntPower(2,m);
return;
}
```

2.7 More About Polynomials

The Basic Idea

Polynomials with coefficients drawn from finite fields play an important role in the analysis and application of error correcting codes. We will frequently have need to represent various such polynomials in a "computer-friendly" form. Polynomials whose coefficients are drawn from a prime field GF(p) (see Section 2.2) are easy to deal with—use ordinary integer operations on the coefficients and reduce the result modulo p. Polynomials whose coefficients are drawn from an extension field GF(p^m) are a bit more difficult. This section explores several methods for representing such polynomials in software.

Assume that we have a polynomial in x with coefficients drawn from GF(2^m):

$$f(x) = \sum_{n=0}^{N} \beta_n x^n \tag{2.4}$$

$$\beta_n \in \{0, 1, \alpha, \alpha^2, \alpha^3, \ldots, \alpha^{2^m-2}\}$$

How can we represent such a polynomial in software? We could store the coefficients of $f(x)$ in an array, for example beta[], with the coefficient for x^n stored in location n. However, we are faced with a choice between storing the coefficients in m-tuple form or as the exponents of their power form.

Reality Check

Let's say $\beta_n = \alpha^4$, where α^4 is an element of GF(2^4) as defined by Table 2-13. To use the m-tuple representation for β_n, we would store $(0011)_2$ in beta[n]. To use the exponent of the power representation for β_n, we would store $(4)_{10} = (100)_2$ in beta[n].

Before making a choice between the m-tuple or exponential representations, we should take a close look at the implications of option. The m-tuple representation has the following properties:

- ☐ Addition of two coefficients in m-tuple form is accomplished as the bit-by-bit exclusive-or (XOR) of the coefficients.

Reality Check

Add α^4 and α^5 as follows:

$$\begin{aligned} \alpha^4 &= 0011 \\ \alpha^5 &= \underline{0110} \\ &\quad 0101 = \alpha^8 \end{aligned}$$

- ☐ Storage of the element 0 presents no special problems.
- ☐ Multiplication of two coefficients is not straightforward.

The exponential representation has the following properties:

- ☐ Multiplication of two coefficients is accomplished by adding the two exponents and reducing the sum modulo $(2^m - 1)$.

Reality Check

$\alpha^4\alpha^5 = \alpha^9 \qquad \alpha^9\alpha^8 = \alpha^2$

- ☐ The exponential form for the element 0 is $\alpha^{-\infty}$. This form might require special handling. (The value of 0 is reserved for representing α^0.) Two obvious candidates for representing $\alpha^{-\infty}$ are -1 and $2^m - 1$. Any positive value (including $2^m - 1$) is probably a bad idea because there might be times when we need these values to temporarily store the result of an operation before reducing the exponent modulo $(2^m - 1)$.
- ☐ Addition of two coefficients is not straightforward.

Back to our original problem; we wish to multiply a polynomial $p(x)$ by $(x + \alpha^k)$. Assume that m-bit binary representations of the

coefficients for $p(x)$ are stored in an array beta[], with the coefficient for the x^n term stored in beta[n]. We further assume that the highest degree term in $p(x)$ is x^N (or, in other words, beta[i] = 0 for i = $N + 1, N + 2, N + 3,...$). We begin the design of our polynomial multiplication algorithm by noting that

$$(x + \alpha^k)\, p(x) = xp(x) + \alpha^k p(x)$$

Implementation of the term $xp(x)$ from this expansion is very easy—we simply shift the contents of location n into location $n + 1$ for $n = N, N - 1, N - 2,..., 0$. Implementation of the term $\alpha^k p(x)$ is straightforward; simply multiply each location of beta[] by α^k. Of course, this multiplication must be performed using the rules of multiplication defined for GF(2^m), and the result must be added to the location-shifted version of beta[] using the rules of addition defined for GF(2^m). The tricky part is to efficiently combine both operations so that the coefficient array can be updated "in place." The following fragment of C code makes use of function MultExponFieldElems() and AddExponFieldElems() to accomplish the desired arithmetic over GF(2^m). Code to multiply a polynomial over GF(2^m) by a binomial over GF(2^m) is also incorporated in the BuildOntoPolynomial() function presented in Section 2.9 for constructing minimal polynomials. In that code, the operations embedded within MultExponFieldElems() and AddExponFieldElems() are expanded inline to illustrate an alternative approach.

Program Fragment

```
/* The following fragment computes (x + alpha) * p(x)
where p(x)
is a polynomial in x of degree nn originally and of
degree nn+1 after the multiplication.  The coefficients
of p(x) are stored in beta[], and the values of these
coefficients are elements of the symbol field GF(2**m).
*/

for( n=nn; n>=0; n--){
     beta[n+1] = AddExponFieldElems( beta[n],
MultExponFieldElems(alpha, beta[n]));
     }
nn++;

/*  end of fragment  */
```

2.8 Cyclotomic Cosets

The Basic Idea

Cyclotomic cosets are an essential element in an algorithm for finding minimal polynomials which in turn play a crucial role in the construction of BCH codes and Reed-Solomon codes. This section defines cyclotomic cosets and develops a program for partitioning the elements of GF(2^m) into cyclotomic cosets.

Assume that we are interested in the extension field GF(2^m). As shown in Section 2.3, the nonzero elements of GF(2^m) can be written in exponential form as $\alpha^0, \alpha^1, \alpha^2, \ldots, \alpha^{2m-2}$. This set of elements can be just as easily represented by the set of their exponents {0, 1, 2,..., $2^m - 2$}. We could, if we wished, partition this set of exponents into subsets in any number of ways. For coding work, it turns out to be very useful to partition the set of exponents in one particular way. The partitioning technique is easier to understand if we begin with a specific example. Consider the set of exponents corresponding to the nonzero elements of GF(2^4):

$$C = \{0, 1, 2, \ldots, 14\}$$

The exponent 0 always goes into a subset by itself:

$$C_0 = \{0\}$$

Begin the next subset with the element 1:

$$C_1 = \{1,$$

To add another element to C_1, multiply the newest element by 2 and reduce the result modulo $2^4 - 1 = 15$:

$$C_1 = \{1, 2,$$

Continue adding elements in this fashion until the new element to be added is 1:

$$\begin{aligned} C_1 &= \{1, 2 \\ &\downarrow \\ 2 \times 2 &= 4 \pmod{15} \\ &\downarrow \\ C_1 &= \{1, 2, 4 \\ &\downarrow \\ 2 \times 4 &= 8 \pmod{15} \\ &\downarrow \\ C_1 &= \{1, 2, 4, 8 \\ &\downarrow \\ 2 \times 8 &= 1 \pmod{15} \\ C_1 &= \{1, 2, 4, 8\} \end{aligned}$$

The next subset is started with the smallest element in C that has not already been placed in a subset. In this particular example, we have

$$C_3 = \{3,$$

We continue adding elements as we did for C_1, but this time we stop when the new element to be added is 3:

$$C_3 = \{3$$
$$\downarrow$$
$$2 \times 3 = 6$$
$$\downarrow$$
$$C_3 = \{3, 6$$
$$\downarrow$$
$$2 \times 6 = 12 \pmod{15}$$
$$\downarrow$$
$$C_3 = \{3, 6, 12$$
$$\downarrow$$
$$2 \times 12 = 9 \pmod{15}$$
$$\downarrow$$
$$C_3 = \{3, 6, 12, 9$$
$$\downarrow$$
$$2 \times 9 = 3 \pmod{15}$$
$$C_3 = \{3, 6, 12, 9\}$$

The next unused element is 5, so we have

$$C_5 = \{5$$
$$\downarrow$$
$$2 \times 5 = 10$$
$$\downarrow$$
$$C_5 = \{5, 10$$
$$\downarrow$$
$$2 \times 10 = 5 \pmod{15}$$
$$C_5 = \{5,10\}$$

Continuing in a similar fashion, we find

$$C_7 = \{7, 14, 13, 11\}$$

Now each element of C has been placed into one and only one subset:

$$C_0 = \{0\}$$
$$C_1 = \{1, 2, 4, 8\}$$
$$C_3 = \{3, 6, 12, 9\}$$
$$C_5 = \{5, 10\}$$
$$C_7 = \{7, 14, 13, 11\}$$

Subsets formed in this manner are called *cyclotomic cosets* mod $2^m - 1$. The concept of cyclotomic cosets can be generalized to any set of integers mod $p^m - 1$, with each coset consisting of

$$\{s, ps, p^2s, p^3s, \ldots, p^{n-1}s\}$$

where each element is reduced modulo $p^m - 1$ and n is the smallest integer for which $p^n s \equiv s \pmod{p^n - 1}$.

Software Notes Listing 2-2 contains the BuildCyclotomicCosets() function that separates the integers $\{1, 2, \ldots, p^m - 2\}$ into cyclotomic cosets. The variables Coset_Array[][] and Num_Cosets are module variables that are shared with other functions in module gfield.c.

Listing 2-2

```
static exponent Coset_Array[32][MAX_ELEMENTS];
static int Num_Cosets;

/*********************************/
/*                               */
/*  BuildCyclotomicCosets        */
/*                               */
/*********************************/
void BuildCyclotomicCosets( int p,
                            int m)

/*-------------------------------------------------*/
/*  This routine partitions the integers           */
/*  {1,2,3,...((2**m)-2) } into cyclotomic         */
/*  cosets and places these cosets in the          */
/*  module variable Coset_Array[].  For cosets     */
/*  produced by this routine, the first element    */
/*  in each coset will be the smallest element     */
/*  in that coset.  The first elements will        */
/*  also be odd.  Therefore, without fear of       */
/*  conflict, we can place the coset with          */
/*  first element k into row (k+1)/2 of the        */
/*  module variable Coset_Array[].  The first      */
/*  location in each row (i.e. Coset_Array[k][0]) */
/*  contains the number of elements in that row's */
/*  coset.                                         */

{
int modulus, i ,j, first_element,coset_index;
int working_element;
int element_count, total_count;
logical match;

modulus = IntPower(p,m)-1;
match = FALSE;
coset_index=0;
total_count = 0;
first_element=-1;
```

```
Coset_Array[1][0]=0;

do {
  /*  keep checking successive odd integers until
      one is found that is not already in one of
      the completed cosets. */
  do {
    first_element +=2;
    match = FALSE;
    for(i=1; i<=coset_index; i++) {
      for(j=1; j<=(signed)Coset_Array[i][0]; j++) {
        /*  if a candidate element is already
        in a coset, skip the row corresponding
        to this coset in Coset_Array[] and move
        on to the next odd candidate first element.*/

        if(first_element == (signed)Coset_Array[i][j]) {
          match = TRUE;
          Coset_Array[(first_element+1)/2][0]=0;
          i=coset_index+1;
          break;
          }
        }
      }
    }
  while(match);
  /* match remains FALSE if the element under
  consideration is suitable for use as the
  first element in a new coset. */
  coset_index++;

  /*  Generate all the elements of the next coset  */

  working_element=first_element;
  element_count=0;
  do {
    total_count++;
    element_count++;

Coset_Array[coset_index][element_count]=(exponent)working_element;
    working_element=(p*working_element) % modulus;
    }
  while (working_element != first_element);

  /*  Element zero holds the number of elements in this coset */

  Coset_Array[coset_index][0]=(exponent)element_count;
  }
while (total_count<(modulus-1));
Num_Cosets = coset_index;
return;
}
```


2.9 Finding Minimal Polynomials

For an element β of GF(2^m), the minimal polynomial $\phi(x)$ is the polynomial of smallest degree with coefficients drawn from GF(2)

such that $\phi(\beta) = 0$. For example, the element α^5 from GF(2^4) has as its minimal polynomial

$$\phi_5(x) = x^2 + x + 1$$

because

$$\begin{aligned}\phi_5(\alpha^5) &= (\alpha^5)^2 + (\alpha^5) + 1 \\ &= \alpha^{10} + \alpha^5 + 1 \\ &= (\alpha^2 + \alpha + 1) + (\alpha^2 + \alpha) + 1 \\ &= 0\end{aligned}$$

Defining and finding minimal polynomials is more than just an academic exercise—minimal polynomials are crucial to the construction of certain types of codes, such as BCH codes and Reed-Solomon codes.

Clearly, the minimal polynomial of the element 0 is always $\phi(x) = x$ for any extension field GF(2^m). The minimal polynomial $\phi_k(x)$ for any nonzero element α^k can be obtained as

$$\phi^k(x) = \prod_{r=0}^{R-1} \left[x + (\alpha^k)^{2^r}\right]$$

where R is the smallest integer such that $(\alpha^k)^{2^R} = \alpha^k$.

Based upon the way the field GF(2^m) is defined, it will always be the case that $(\alpha^k)^{2^m} = \alpha^k$. However, for some values of α^k, it might turn out that $(\alpha^k)^{2^R} = \alpha^k$ for some values of R less than m. Because $\alpha^{2^m - 1} = 1$, we can conclude that

$$(\alpha^k)^{2^R} = \alpha^{k2^R} = \alpha^k$$

whenever

$$k2^R \equiv k \text{ (modulo } 2^m - 1) \tag{2.5}$$

For any given k and m, it is a simple matter to find the smallest R for which Eq. 2.5 is satisfied by successively trying $R = 1$, $R = 2$, and so on up to $R = m$. (Remember that $R = m$ is guaranteed to work.) This leads us directly to the recipe for finding a minimal polynomial.

Recipe 2.3 Generating the minimal polynomial for element α^k of the field GF(2^m).

1. Determine the smallest positive integer R for which
$$k2^R \equiv k \text{ (modulo } 2^m - 1)$$
2. Using the value of R found in step 1, compute the minimal polynomial $\phi_k(x)$ of the element α^k as
$$\phi_k(x) = \prod_{r=0}^{R-1} \left(x + \alpha^{k2^r}\right) \tag{2.6}$$

Example 2.5

Find the minimal polynomial for each element in GF(2^4).

1. $R = 1$ for α^0; $R = 2$ for α^5 and α^{10}; and $R = 4$ for all other elements.
2. Using Eq. 2.6, we obtain

$$\begin{aligned}
\phi_0(x) &= x + \alpha^0 = x + 1 \\
\phi_1(x) &= (x + \alpha)(x + \alpha^2)(x + \alpha^4)(x + \alpha^8) \\
&= (x^2 + \alpha x + \alpha^2 x + \alpha^3)(x^2 + \alpha^8 x + \alpha^4 x + \alpha^{12}) \\
&= x^4 + \alpha^8 x^3 + \alpha^4 x^3 + \alpha^{12} x^2 + \alpha x^3 + \alpha^9 x^2 \\
&\quad + \alpha^5 x^2 + \alpha^{13} x + \alpha^2 x^3 + \alpha^{10} x^2 + \alpha^6 x^2 \\
&\quad + \alpha^{14} x + \alpha^3 x^2 + \alpha^{11} x + \alpha^7 x + \alpha^{15} \\
&= x^4 + x^3(\alpha^8 + \alpha^4 + \alpha + \alpha^2) \\
&\quad + x^2(\alpha^{12} + \alpha^9 + \alpha^5 + \alpha^{10} + \alpha^6 + \alpha^3) \\
&\quad + x(\alpha^{13} + \alpha^{14} + \alpha^{11} + \alpha^7) + 1 \\
&= x^4 + x + 1
\end{aligned}$$

In a similar fashion, minimal polynomials can be obtained for α^2 through α^{14}:

$$\begin{aligned}
\phi_2(x) = \phi_4(x) = \phi_8(x) &= x^4 + x + 1 \\
\phi_3(x) = \phi_6(x) = \phi_9(x) = \phi_{12}(x) &= x^4 + x^3 + x^2 + x + 1 \\
\phi_5(x) = \phi_{10}(x) &= x^2 + x + 1 \\
\phi_7(x) = \phi_{11}(x) = \phi_{13}(x) = \phi_{14}(x) &= x^4 + x^3 + 1
\end{aligned}$$

Tables of minimal polynomials of elements in GF(2^m) for $2 \le m \le 9$ are provided in Appendix.

Software Notes Listing 2-3 contains the BuildMinimalPolynomials() function from the module gfield.c. This function constructs the minimal polynomials for the extension field GF(2^m) that resides in the module variable Galois_Field[]. The polynomial under construction is multiplied by successive binomial factors via calls to BuildOntoPolynomial(). The variables Num_Field_Elements, Coset_Array[][], Minimal_Polynomial[], Degree_Of_Min_Poly[], and Num_Cosets are module variables that are shared with the other functions in module gfield.c.

Listing 2-3

```
static unsigned int Num_Field_Elements;
static exponent Coset_Array[32][MAX_ELEMENTS];
static word Minimal_Polynomial[MAX_ELEMENTS];
static int Degree_Of_Min_Poly[MAX_ELEMENTS];
static int Num_Cosets;
static int Degree_Of_Field;
```

```
/*******************************/
/*                             */
/*  BuildMinimalPolynomials()  */
/*                             */
/*******************************/

void BuildMinimalPolynomials(void)

/*-----------------------------------------------------*/
/*  all of the roots for any particular minimal      */
/*  polynomial will be contained within a single     */
/*  cyclotomic coset.  Therefore, this routine       */
/*  makes use of the module variable Coset_Array[][] */
/*  and assumes that this array has been properly    */
/*  filled with each row containing the elements     */
/*  of one coset and the first location in each row  */
/*  containing the number of elements in the coset.  */
{
element poly_coeff[64];
element root;
int n, num_roots;
int max_degree,coset_index,first_index_in_poly_coeff;
logical high_degree_to_left;
word packed_min_poly;

/*-----------------------------------------------------*/
/* the coefficients of intermediate results will be  */
/* elements of GF(2**m), so the working polynomials  */
/*  are left in unpacked form.                       */

for(n=0;n<64;n++) poly_coeff[n]=0;
for( coset_index=1; coset_index<=Num_Cosets; coset_index++) {
  max_degree = 1;
  num_roots = Coset_Array[coset_index][0];
  poly_coeff[1]=1;
  poly_coeff[0] = ExponToTuple(Coset_Array[coset_index][1]);

  for( n=2; n<=num_roots; n++) {
    root = Coset_Array[coset_index][n];
    BuildOntoPolynomial(poly_coeff, &max_degree, root);
    printf("max_degree set to %d\n",max_degree);
    }

  /*-------------------------------------------------*/
  /*  The coefficients of the final result will be  */
  /*  elements of GF(2), so we can pack the minimal */
  /*  polynomial's coefficients into a single       */
  /*  binary word.                                  */
  high_degree_to_left = TRUE;
  first_index_in_poly_coeff = 0;
  PackBitsLong( poly_coeff,
          &packed_min_poly,
          max_degree + 1,
          first_index_in_poly_coeff,
          high_degree_to_left);
  /*-------------------------------------------------*/
  /* copy packed representation of minimal polynomial into */
  /* each array location corresponding to one of its roots. */
  /* The table so defined will be very useful for decoding. */
```

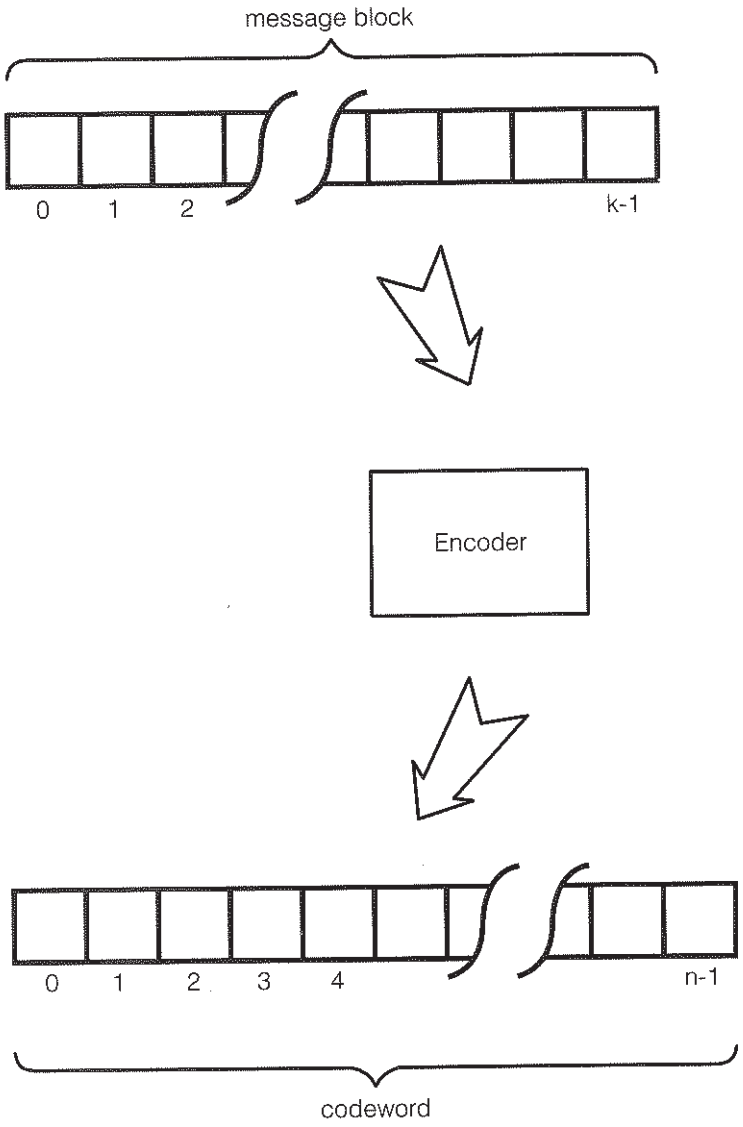
```
  /* The table is held in the module variable minimalPolynomial[],*/
  /* and routines external to this module can access the table */
  /* using the function GetMinPoly().                          */
  for (n=1; n<=num_roots; n++) {

Minimal_Polynomial[(Coset_Array[coset_index][n])]=packed_min_poly;
    Degree_Of_Min_Poly[(Coset_Array[coset_index][n])]=max_degree;
    printf("Degree_Of_Min_Poly[%d] set to %d\n",
            Coset_Array[coset_index][n], max_degree);
    }
  printf("\n %d) M%d = ",coset_index,
Coset_Array[coset_index][1]);
  for(n=max_degree; n>=0; n--) {
    printf("%d",poly_coeff[n]);
    }
  printf(" = %#lx",packed_min_poly);
  }
printf("\n");
return;
}
```

3 Block and Cyclic Codes

3.1 Introduction

A *block code* is a code that operates upon fixed-length blocks of information bits. (This is in contrast to convolutional codes that operate upon a continuous stream of information bits.) For ease of reference, the blocks of information bits are usually called *message blocks*—even though it might take many such blocks to transmit what we would normally think of as a "message" (a telegram, military orders, a computer-to-computer file transfer, etc.). As shown in Fig. 3-1, a block-code encoder takes a message block of k bits and produces an n-bit block, which is referred to as a *code word*, *code vector*, or *code block*. For any code, n will always be greater than k, so the encoder must supply $(n - k)$ additional bits. These additional bits are obtained by applying some particular set of rules to the bits in the k-bit message block. The exact nature of these rules is what defines a specific code: Hamming, Reed-Solomon, BCH, etc. The encoder might or might not produce a code word that explicitly contains the message block somewhere within the code word. Because each message block contains k bits, 2^k possible messages can be conveyed by each block. Each code word contains n bits, so there could be as many as 2^n distinct code words. However, block codes are designed to use only 2^k of the 2^n possible code words—one n-bit code word for each distinct k-bit message. Viewed another way, a block code uses n bits to transmit a message containing only k bits of useful information. Therefore, the code word contains some redundancy. It is this redundancy that is the source of the code's ability to detect and/or correct errors.

Assume that noise or other channel impairments corrupt the code word so that one of the unused n-bit patterns appears at the receiver. Because this is an unused or "illegal" pattern, the receiver "knows" that an error must have occurred. The trick is to devise a decoding scheme that will let the receiver decide which of the legal code words was actually transmitted. If too many bits within the block are corrupted, the receiver might make this decision in-



■ **3-1** *Operation of a block encoder.*

correctly. In fact, if enough bits are corrupted, it is possible for a legal (but incorrect) code word to appear at the receiver. In such a situation, the receiver will not even be aware that an error has occurred. In general, the more redundancy there is in the code word [*i.e.*, the larger that $(n - k)$ is], the more bit errors the code will be able to detect and/or correct.

Table 3-1 An example of a simple block code.

Info block	Code word
0 0 0 0	0 0 0 0 0
0 0 0 1	0 0 0 1 1
0 0 1 0	0 0 1 0 1
0 0 1 1	0 0 1 1 0
0 1 0 0	0 1 0 0 1
0 1 0 1	0 1 0 1 0
0 1 1 0	0 1 1 0 0
0 1 1 1	0 1 1 1 1
1 0 0 0	1 0 0 0 1
1 0 0 1	1 0 0 1 0
1 0 1 0	1 0 1 0 0
1 0 1 1	1 0 1 1 1
1 1 0 0	1 1 0 0 0
1 1 0 1	1 1 0 1 1
1 1 1 0	1 1 1 0 1
1 1 1 1	1 1 1 1 0

Example 3.1 A parity scheme that appends one bit to each four-bit nibble in a computer memory is a simple example of a block code that is capable of detecting a single-bit error within each nibble. There are 4 bits in each message block and 5 bits in each code-word. Table 3-1 lists the message blocks and code words for such a scheme using an even-parity rule.

It is fairly common (but not universal) to use n to denote the total number of bits in a code word and to use k to denote the number of information bits. The difference $(n - k)$ is the number of parity bits in the code word. Clearly, there are 2^{n-k} different combinations of the $n - k$ parity bits. Each different k-bit message block will use only one of these 2^{n-k} possible combinations of message bits. [This is not to say that each different k-bit message block will use a different $(n - k)$-bit parity field. Many coding schemes assign the same parity field to two or more different k-bit message blocks. In fact, for codes in which $k > (n - k)$, individual parity fields must be assigned to more than one block because the number of different message blocks (2^k) will be greater than the number of different parity fields (2^{n-k}).]

A large part of coding theory is concerned with identifying "good" ways to assign particular parity fields to message blocks. In fact,

the rules for assigning parity fields to message blocks are what really defines any particular code. Another large part of coding theory is concerned with rules for extracting the original message block from a (possibly corrupted) code block.

3.2 Linear Block Codes

Much of the work done in the area of block codes is confined to *linear* block codes, which are an important subset of all possible block codes. The usual definition for a linear block code is couched in terms of the mathematical structures defined in Chapter 2.

Definition 3.1 A binary block code having 2^k codewords (i.e., k information bits) and n-bits per codeword is a *linear* (n, k) code if and only if the 2^k codewords constitute a k-dimensional subspace of the vector space formed by all the n-tuples over the field GF(2).

This definition can be recast in a more practical, user-oriented form as follows:

Definition 3.2 A binary block code is *linear* if and only if the modulo-2 sum of any two codewords is also a codeword.

3.2.1 Encoding for Linear Block Codes

By Definition 3.1, the code $\mathbf{C}$ is a k-dimensional vector space; therefore, $\mathbf{C}$ can be completely defined by k linearly independent codewords. This can be exploited to form the *generator matrix* $\mathbf{G}$ such that the encoding operation is represented mathematically as

$$\mathbf{v} = \mathbf{u} \bullet \mathbf{G}$$

where $\mathbf{v}$ is a vector of the encoded bits, $\mathbf{u}$ is a vector of k information bits, and $\mathbf{G}$ is the *generator matrix*. Take a more detailed look at the specific case of a block code having a total of n bits per code word, with k of these being information bits and $(n - k)$ being parity bits. For such a code, $\mathbf{v}$ will have n bits, $\mathbf{u}$ will have k bits, and $\mathbf{G}$ will have k rows and n columns:

$$\begin{aligned}
\mathbf{u} &= [u_0 \;\; u_1 \;\; u_2 \;\; \cdots \;\; u_{k-1}], \\
\mathbf{v} &= [v_0 \;\; v_1 \;\; v_2 \;\; \cdots \;\; v_{n-1}], \\
\mathbf{G} &= \begin{bmatrix}
g_{00} & g_{01} & g_{02} & \cdots & g_{0,n-1} \\
g_{10} & g_{11} & g_{12} & \cdots & g_{1,n-1} \\
\vdots & & \ddots & & \vdots \\
g_{k-1,0} & g_{k-1,1} & g_{k-1,2} & \cdots & g_{k-1,n-1}
\end{bmatrix}
\end{aligned}$$

By the standard "row-by-column" definition of matrix multiplication, each component of **v** is given by

$$v_i = \sum_{j=0}^{k-1} u_j g_{ji} \tag{3.1}$$

where the summation is performed using binary addition.

Programming Considerations The following fragment of C code implements Eq. 3.1 for each of the n components in **v**:

```
/****************************************/

for( i=0; i<n; i++){
     v[i] =0;
     for( j=0; j<k; j++ ) {
          v[i] = v[i] ^ (u[j] & g[j][i]);
          }
     }
/*  end of fragment  */
/****************************************/
```

If the codewords within a code can be partitioned into two parts, with one part consisting of the k-bit information word and the other part consisting of the $n - k$ parity digits, the code is said to be a *systematic* code or a code in *systematic form*. The alternative, in which the message digits do not appear directly in the codeword, is described as a *nonsystematic* code. If a code is in systematic form, its generator matrix will be separable into two submatrices such that one of the submatrices is the $k \times k$ identity matrix. This is the part of the generator matrix that "copies" the k-digit message word into the code word. The remaining submatrix calculates the parity digits as linear functions of the message digits. Notationally:

$$\mathbf{G} = [\mathbf{I}_k \; \mathbf{P}]$$

where

$\mathbf{I}_k = k \times k$

$\mathbf{P} = k \times (n - k)$ matrix of parity-check coefficients

Any linear code that is not in systematic form can be converted into systematic form by performing elementary row operations and column transpositions on its generator matrix. Any changes to **G** via elementary row operations leaves the total set of codewords unchanged—even though it changes the particular one-to-one correspondence between k-bit information vectors and n-bit codewords.

Example 3.2 Consider the $(7, 4, d = 3)$ code shown in Table 3-2 and having a generator matrix given by

$$\mathbf{G} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \tag{3.2}$$

Perform row operations as necessary to obtain an equivalent $(7, 4, d = 3)$ code in systematic form.

Table 3-2 A block code in nonsystematic form for Example 3.2.

Info block	Code word
0 0 0 0	0 0 0 0 0 0 0
0 0 0 1	0 0 0 1 0 1 1
0 0 1 0	0 0 1 1 1 0 1
0 0 1 1	0 0 1 0 1 1 0
0 1 0 0	0 1 1 0 0 0 1
0 1 0 1	0 1 1 1 0 1 0
0 1 1 0	0 1 0 1 1 0 0
0 1 1 1	0 1 0 0 1 1 1
1 0 0 0	1 1 0 0 0 1 0
1 0 0 1	1 1 0 1 0 0 1
1 0 1 0	1 1 1 1 1 1 1
1 0 1 1	1 1 1 0 1 0 0
1 1 0 0	1 0 1 0 0 1 1
1 1 0 1	1 0 1 1 0 0 0
1 1 1 0	1 0 0 1 1 1 0
1 1 1 1	1 0 0 0 1 0 1

Solution Starting with the matrix given in Table 3-2, perform the following row operations:

1. Add row 4 to row 3.
2. Add the new row 3 to row 2.
3. Add the new row 2 to row 1.

The resulting matrix is

$$\mathbf{G} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \tag{3.3}$$

The corresponding code is listed in Table 3-3. Notice that the first four columns of the new matrix form a submatrix that is an identity matrix. Comparison of the two tables reveals that **G** from Eq. 3.2 and its systematic form from Eq. 3.3 generate the same sixteen 7-tuples.

Table 3-3 Block code in systematic form for Example 3.2.

Info block	Check block
0 0 0 0	0 0 0
0 0 0 1	0 1 1
0 0 1 0	1 1 0
0 0 1 1	1 0 1
0 1 0 0	1 1 1
0 1 0 1	1 0 0
0 1 1 0	0 0 1
0 1 1 1	0 1 0
1 0 0 0	1 0 1
1 0 0 1	1 1 0
1 0 1 0	0 1 1
1 0 1 1	0 0 0
1 1 0 0	0 1 0
1 1 0 1	0 0 1
1 1 1 0	1 0 0
1 1 1 1	1 1 1

3.2.2 Constructing the Generator Matrix

How do we go about obtaining a generator matrix? As we will see in later chapters, some codes (*e.g.,* Hamming codes) are defined directly in terms of their generator matrices, while most cyclic codes are defined in terms of their generator *polynomial.*

3.2.3 Parity-Check Matrix

For any linear code **C**, it is possible to construct a matrix **H** such that

$$\mathbf{v} \cdot \mathbf{H}^{\perp} = 0 \tag{3.4}$$

if and only if the n-tuple **v** is a codeword in **C**. This matrix **H** is usually called the *parity-check* matrix for code **C**. If a particular n-tuple **r** is received, Eq. 3.4 can be used to determine whether or

not **r** is a legal codeword in **C**. The decoder computes the *syndrome* **s** as

$$\mathbf{s} = \mathbf{r} \cdot \mathbf{H}^{\perp} \tag{3.5}$$

The matrix H has $n - k$ rows and n columns, and the vector **r** has n components. Therefore, based on the definition of matrix multiplication ("row by column"), we know that the syndrome **s** will be a vector having $n - k$ components

$$\mathbf{s} = [s_0, s_1, \ldots, s_{n-k-1}]$$

If $\mathbf{s} = [0, 0, \ldots, 0]$, then **r** is a codeword in **C**. If $\mathbf{s} \neq [0, 0, \ldots, 0]$, then **r** is not a codeword in **C**. The study of decoding is primarily concerned with ways that nonzero values of the syndrome **s** can be exploited to actually locate and correct errors within the received word **r**.

The parity-check matrix **H** can be obtained from the generator matrix **G** via

$$\mathbf{G} = [\mathbf{I}_k \ \mathbf{P}]$$
$$\mathbf{H} = [-\mathbf{P}^{\perp} \ \mathbf{I}_{n-k}] \tag{3.6}$$

where $\mathbf{P}^{\perp}$ is the transpose of **P**. The minus sign on $\mathbf{P}^{\perp}$ is moot in the binary case because $-1 \equiv 1$, but it is included here for consistency in nonbinary applications.

Example 3.3 Using the generator matrix for the (7, 4) systematic code found in Example 3.2, find the parity-check matrix **H**.

Solution We first partition **G** into the form $[\mathbf{I}_k \mathbf{P}]$ as

$$\mathbf{G} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

Making use of Eq. 3.6, we obtain **H** as

$$\mathbf{H} = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Let's take a closer look at what's really happening in Eq. 3.5. Based on the partitioning of **H** given in Eq. 3.6, we can write $\mathbf{H}^{\perp}$ and **s** as

$$\mathbf{H}^{\perp} = \begin{bmatrix} \mathbf{P} \\ \mathbf{I}_{n-k} \end{bmatrix}$$

$$\mathbf{s} = \mathbf{r} \bullet \mathbf{H}^{\perp} = [r_0, r_1, \ldots, r_{n-1}] \begin{bmatrix} \mathbf{P} \\ \mathbf{I}_{n-k} \end{bmatrix}$$

$$= [r_0, r_1, \ldots, r_{n-1}] \begin{bmatrix} p_{00} & p_{01} & \cdots & p_{0,n-k-1} \\ p_{10} & p_{11} & & p_{1,n-k-1} \\ \vdots & & \ddots & \vdots \\ p_{k-1,0} & p_{k-1,1} & \cdots & p_{k-1,n-k-1} \\ 1 & 0 \quad 0 & \cdots & 0 \\ 0 & 1 & & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 \quad 0 & \cdots & 1 \end{bmatrix}$$

The ith component of $\mathbf{s}$ is obtained as the vector multiplication of $\mathbf{r}$ with column i of $\mathbf{H}^{\perp}$

$$s_i = r_0 p_{0i} + r_1 p_{1i} + \ldots + r_{k-1} p_{k-1,i} + r_{k+i}$$
$$= r_{k+i} + \sum_{j=0}^{k-1} r_j p_{ji}$$

The first term, r_{k+i}, is simply the ith parity bit as received at the decoder. The summation is a recomputation of this same parity bit using the parity-check coefficients p and the k received message bits r_0 through r_{k-1}. If the received parity and recomputed parity are equal, the sum of these two will be zero.

Definition 3.3 Let $\mathbf{H}$ be the parity-check matrix of code $\mathbf{C}$. If we use $\mathbf{H}$ as a generator matrix instead of a parity-check matrix, the resulting code $\mathbf{C}_d$ is called the *dual code* of $\mathbf{C}$.

3.3 Cyclic Codes

A special class of linear block codes, called *cyclic* codes, possess mathematical properties that make them easier to encode and decode than linear codes in general. The two most widely used types of linear block codes—BCH codes and Reed-Solomon codes—are cyclic codes.

Given any n-tuple $\mathbf{v} = [v_0, v_1, \ldots, v_{n-1}]$, it is possible to cyclically shift the n-tuple so that each component moves one place to the right and the rightmost component becomes the leftmost component.

$$\mathbf{v}_{\text{shifted}} = [v_{n-1}, v_0, v_1, \ldots, v_{n-2}]$$

This concept can be extended to include shifts of multiple places and shifts in either direction.

Definition 3.4 An (n, k) linear block code $\mathbf{C}$ is a *cyclic code* if every cyclic shift of a codeword in $\mathbf{C}$ is also a codeword in $\mathbf{C}$.

Example 3.4 Consider the code from Example 3.2. We can sort the codewords into groups as shown in Table 3.4 such that the codewords within each group are cyclic shifts of each other. Each group contains all possible shifts of the basic pattern corresponding to that group.

Table 3-4
Codewords for linear (7,4) code sorted into groups such that all codewords within a group are cyclic shifts of each other.

Info block	Check block
0 0 0 0	0 0 0
0 0 0 1	0 1 1
0 0 1 0	1 1 0
0 1 0 1	1 0 0
1 0 1 1	0 0 0
0 1 1 0	0 0 1
1 1 0 0	0 1 0
1 0 0 0	1 0 1
0 1 0 0	1 1 1
1 0 0 1	1 1 0
0 0 1 1	1 0 1
0 1 1 1	0 1 0
1 1 1 0	1 0 0
1 1 0 1	0 0 1
1 0 1 0	0 1 1
1 1 1 1	1 1 1

By definition, if a particular n-tuple is a legal codeword for a cyclic code $\mathbf{C}$, then all cyclic shifts of this n-tuple are also legal codewords. Conversely, if a particular n-tuple is not a legal codeword for a cyclic code $\mathbf{C}$, then none of the cyclic shifts of the n-tuple can be a legal codeword.

Result 3.1 In a cyclic code $\mathbf{C}$, the nonzero code polynomial of minimum degree is unique. (Recall from Section 2.5 that each element of an extension field can be represented in polynomial form. Be-

cause each codeword in a block code can be represented as an element of an extension field, it follows that each codeword can be represented by a polynomial.)

Definition 3.5 For a cyclic code $\mathbf{C}$, the nonzero code polynomial of minimum degree is called the *generator polynomial* of the code. It is customary to denote the generator polynomial by $\mathbf{g}(x)$.

Result 3.2 If the generator polynomial of a cyclic code is given by

$$\mathbf{g}(x) = x^r + g_{r-1}x^{r-1} + g_{r-2}x^{r-2} + \ldots + g_1x + g_0$$

the constant term g_0 will always equal 1.

Result 3.3 In an (n, k) cyclic code $\mathbf{C}$, the degree of the generator polynomial is $n - k$, *i.e.*,

$$\mathbf{g}(x) = x^{n-k} + g_{n-k-1}x^{n-k-1} + g_{n-k-2}x^{n-k-2} + \ldots + g_2x^2 + g_1x + 1$$

Result 3.4 Let $\mathbf{g}(x)$ be the generator polynomial of a cyclic code $\mathbf{C}$. A polynomial of degree $n - 1$ or less with binary coefficients is a code polynomial of $\mathbf{C}$ if and only if the polynomial is divisible by $\mathbf{g}(x)$.

Result 3.5 If $\mathbf{g}(x)$ is the generator polynomial of an (n, k) cyclic code, $\mathbf{g}(x)$ is a factor of $x^n + 1$.

Result 3.6 If $\mathbf{g}(x)$ is a polynomial of degree $n - k$ and is a factor of $x^n + 1$, then $\mathbf{g}(x)$ is the generator polynomial for some (n, k) cyclic code.

This last result says that $\mathbf{g}(x)$ will generate some (n, k) cyclic code—it does not say that every $\mathbf{g}(x)$ that is a factor of $x^n + 1$ produces a good code.

3.4 Manual Encoding Methods for Cyclic Codes

In our explorations of specific codes, it will be useful to have a manual pencil-and-paper method for generating codewords corresponding to specific information fields. Recipe 3.1 provides such a method.

Recipe 3.1 Manual encoding for cyclic codes

1. Obtain the generator polynomial $g(x)$. For a code with k information bits and n total bits per code word, the generator polynomial will be of degree $n - k$.
2. Form the polynomial representation of the k-bit information sequence.
3. Multiply this polynomial by x^{n-k}.

4. Divide this product by $g(x)$. This division yields a quotient $q(x)$ and a remainder $r(x)$. This remainder is the polynomial representation of the check bits.
5. Form the code word by appending the check bits to the information bits.

Example 3.5 Given the generator polynomial $\mathbf{g}(x) = x^3 + x^2 + 1$, use Recipe 3.1 to generate the 7-bit codeword for the 4-bit information sequence 0100.

1. We have $\mathbf{g}(x) = x^3 + x^2 + 1$, with $n = 7$ and $k = 4$.
2. The information sequence 0100 corresponds to the polynomial $\mathbf{u}(x) = x^2$.
3. Multiply $\mathbf{u}(x)$ by x^{n-k},
$$x^{n-k}\mathbf{u}(x) = x^3x^2 = x^5$$
4. Divide x^5 by $x^3 + x^2 + 1$
$$\begin{array}{r} x^2 + x + 1 \\ x^3 + x^2 + 1 \overline{)\, x^5} \\ \underline{x^5 + x^4 \quad + x^2} \\ x^4 \quad + x^2 \\ \underline{x^4 + x^3 \quad + x} \\ x^3 + x^2 + x \\ \underline{x^3 + x^2 \quad + 1} \\ x + 1 \end{array}$$
The remainder shown corresponds to the 3-bit parity sequence 011.
5. The code word is formed by appending the parity bits to the information bits:
$$\underbrace{0100}_{\text{info}} \quad \overbrace{011}^{\text{parity}}$$

3.5 Modifications to Cyclic Codes

The various codes produced in the quest for good error-correction properties might not always be the right size to fit conveniently into real-world applications. This section will present several techniques that are available for adjusting the sizes of n, k, and d for existing cyclic codes.

3.5.1 Extended Codes

One simple way to modify a code is to *extend* the code by including additional check digits. In the binary case, this is frequently accomplished by appending an overall parity bit to an (n, k, d) code to obtain an $(n + 1, k, d')$ code. If the original code's minimum distance d is odd, and the parity rule for the extra bit results in an overall even parity, the extended code's minimum distance will be $d' = d + 1$.

Example 3.6 Consider the $(7, 4, d = 3)$ code listed in Table 3-5. An overall even parity bit can be appended to yield the $(8, 4, d = 4)$ extended code listed in Table 3-6.

Table 3-5
A (7, 4, d = 3) block code for modification examples.

Info block	Check block	Weight
0 0 0 0	0 0 0	0
0 0 0 1	1 0 1	3
0 0 1 0	1 1 1	4
0 0 1 1	0 1 0	3
0 1 0 0	0 1 1	3
0 1 0 1	1 1 0	4
0 1 1 0	1 0 0	3
0 1 1 1	0 0 1	4
1 0 0 0	1 1 0	3
1 0 0 1	0 1 1	4
1 0 1 0	0 0 1	3
1 0 1 1	1 0 0	4
1 1 0 0	1 0 1	4
1 1 0 1	0 0 0	3
1 1 1 0	0 1 0	4
1 1 1 1	1 1 1	7

■ Table 3-6 The (8, 4, d = 4) extended code resulting from adding an overall even parity bit to the code of Table 3-5.

Info block	Check block	Weight
0 0 0 0	0 0 0 0	0
0 0 0 1	1 0 1 1	4
0 0 1 0	1 1 1 0	4
0 0 1 1	0 1 0 1	4
0 1 0 0	0 1 1 1	4
0 1 0 1	1 1 0 0	4
0 1 1 0	1 0 0 1	4
0 1 1 1	0 0 1 0	4
1 0 0 0	1 1 0 1	4
1 0 0 1	0 1 1 0	4
1 0 1 0	0 0 1 1	4
1 0 1 1	1 0 0 0	4
1 1 0 0	1 0 1 0	4
1 1 0 1	0 0 0 1	4
1 1 1 0	0 1 0 0	4
1 1 1 1	1 1 1 1	8

3.5.2 Punctured Codes

Puncturing a code decreases the number of check bits and the total number of bits, while leaving the number of information bits and the number of codewords unchanged. This is accomplished by simply deleting bit positions from the check portion of the codeword structure. In most cases, puncturing will also decrease the minimum distance of a code. Puncturing can be viewed as the inverse of extending a code. Suppose that we are starting with a code that has an even minimum distance, such as the one in Table 3-6. Such a code can detect up to $d/2$ errors per block, but its error-correction capability is no better than a code that has an odd minimum distance $d' = d - 1$. Therefore, in situations where we are only concerned with error correction, it might be desirable to decrease the length of a code by eliminating one of the check bits, provided that this can be done without decreasing the number of errors that can be corrected.

3.5.3 Expurgated Codes

Expurgation of a code decreases the number of information bits k without changing the total number of bits n. The number of code-

words is always 2^k, so each time k is decreased by 1, the number of codewords is halved. The most common approach to expurgation deletes all the odd-weight codewords. Deletion of odd-weight codewords is equivalent to replacing the original code's generator polynomial $\mathbf{g}(x)$ with $\mathbf{g}'(x) = (x + 1)\,\mathbf{g}(x)$. When applied to the (7, 4, d = 3) code of Table 3-5, this approach yields the (6, 2, d = 4) code of Table 3-7. In the new code, the boundary between information bits and check bits has been shifted to take one bit away from the information portion and prepend it to the check portion. The remaining information bits cover all 2^3 possible 3-bit messages. Will it always work out this way, or for some expurgated codes will some possible messages be covered twice while others are not covered? In Table 3-5, the codewords can be paired up so that within each pair the first 3 of the 4 information bits are identical. In the general case, it will always be possible to pair up the codewords so that within each pair, the first $k - 1$ of the k information bits are identical. For any code not already having $(x + 1)$ as a factor of its generator polynomial, one codeword of each pair will have odd weight and one codeword will have even weight. Therefore, when we eliminate all the odd-weight codewords, the even-weight members of each pair will survive and thereby "cover" all 2^{k-1} possible combinations of $k - 1$ information bits.

Table 3-7
The (7, 3, d = 4) block code resulting from expurgation.

Info block	Check block	Weight
0 0 0	0 0 0 0	0
0 0 1	0 1 1 1	4
0 1 0	1 1 1 0	4
0 1 1	1 0 0 1	4
1 0 0	1 0 1 1	4
1 0 1	1 1 0 0	4
1 1 0	0 1 0 1	4
1 1 1	0 0 1 0	4

3.5.4 Augmented Codes

Augmentation can be viewed as the inverse of expurgation. Any code having $(x + 1)$ as a factor of its generator polynomial $\mathbf{g}(x)$ can be augmented by replacing the original generator polynomial with $\mathbf{g}'(x) = \mathbf{g}(x)/(x + 1)$. In most practical applications, augmentation has little to offer.

3.5.5 Shortened Codes

In any linear block code, half of the codewords will have 0 in the most significant bit (MSB) of the information segment and half will have 1. We can shorten an (n, k, d) code by discarding all codewords with 1 in the MSB of the information portion and then eliminating this bit from the codeword structure to obtain an $(n-1, k-1, d')$ code. The minimum distance of the shortened code will be at least as large as the minimum distance of the original code (*i.e.*, $d' \geq d$). This shortening process can be extended to multiple bits by retaining only those codewords having all zeros in the z most significant bit positions and then eliminating these z positions from the codeword structure to obtain an $(n-z, k-z, d_z)$ code.

Example 3.7 Consider the (7, 3, d = 4) code shown in Table 3-7. If we retain only those codewords with 0 in the MSB of the information portion, and then eliminate this bit position, we obtain the (6, 2, d = 4) code shown in Table 3-8.

Table 3-8 The (6, 2, d = 4) block code resulting from shortening the (7, 4, d = 4) code of Table 3-7.

Info block	Check block
0 0	0 0 0 0
0 1	0 1 1 1
1 0	1 1 1 0
1 1	1 0 0 1

3.5.6 Lengthened Codes

Lengthening can be viewed as the inverse of shortening, but it is rarely used in practical applications.

3.6 Hamming Codes

Hamming codes were one of the first codes developed specifically for error correction. Like many binary codes, the lengths of Hamming codes are constrained to be of the form

$$n = 2^m - 1$$

However, unlike other codes of similar length, there is a much tighter coupling between the number of check bits and the length of the code—for $n = 2^m - 1$, the number of check bits is m and the

number of information bits is $k = 2^m - m - 1$. The minimum distance is fixed at 3; consequently, the number of correctable errors per block is $t = 1$. The systematic form of the generator matrix is given by

$$\mathbf{G} = [\mathbf{I}_k \quad \mathbf{Q}^{\perp}]$$

where

$\mathbf{I}_k$ = $k \times k$ identity matrix

$\mathbf{Q}^{\perp}$ = $k \times m$ matrix whose rows are the k m-tuples of weight 2 or larger.

The corresponding parity-check matrix is given by

$$\mathbf{H} = [\mathbf{Q} \quad \mathbf{I}_m]$$

Example 3.8 Construct the generator and parity-check matrices for a Hamming code with $m = 4$.

Solution For $m = 4$, we have $n = 2^4 - 1 = 15$ and $k = n - m = 11$. One possible realization of the generator matrix is

$$\mathbf{G} = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 1 & 0 & \cdots & & & & & & & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 & \cdots & & & & & & 0 & 0 & 1 & 1 & 0 \\
0 & \cdots & 0 & 1 & 0 & \cdots & & & & & 0 & 0 & 1 & 1 & 1 \\
0 & \cdots & & 0 & 1 & 0 & \cdots & & & & 0 & 1 & 0 & 0 & 1 \\
0 & \cdots & & & 0 & 1 & 0 & \cdots & & & 0 & 1 & 0 & 1 & 0 \\
0 & \cdots & & & & 0 & 1 & 0 & \cdots & & 0 & 1 & 0 & 1 & 1 \\
0 & \cdots & & & & & 0 & 1 & 0 & \cdots & 0 & 1 & 1 & 0 & 0 \\
0 & \cdots & & & & & & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\
0 & \cdots & & & & & & & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1
\end{bmatrix} \tag{3.7}$$

The corresponding parity-check matrix is given by

$$\mathbf{H} = \begin{bmatrix}
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1
\end{bmatrix} \tag{3.8}$$

It would be impractical to list the $2^{11} = 2{,}048$ possible codewords for this code.

3.6.1 Shortened Hamming Codes

It is possible to shorten an $(n = 2^m - 1, k = 2^m - m - 1, d = 3)$ Hamming code to produce an $(n = 2^{m-1}, k = 2^{m-1} - m, d = 4)$ code. Simply delete all columns of even weight from the submatrix $\mathbf{Q}$ and all rows of even weight from the submatrix $\mathbf{Q}^{\perp}$. A minimum distance of 4 is not large enough to correct more than one error, but it will allow simultaneous correction of one error and detection of two errors if the following decoding strategy is followed:

1. Compute the syndrome from the received word $\mathbf{r}$.
2. If the syndrome is zero, assume that no errors have occurred.
3. If the syndrome is nonzero and has odd weight, assume that a single error has occurred. Perform correction based on the single-error pattern corresponding to the syndrome observed.
4. If the syndrome is nonzero and has even weight, assume that an uncorrectable error pattern has been detected.

Example 3.9 Shorten the $(15, 11, d = 3)$ Hamming code of Example 3.8 to create an $(8, 4, d = 4)$ code.

Solution Delete all rows of even weight from the $\mathbf{Q}^{\perp}$ submatrix shown in Eq. 3.7 and combine with a 4-×-4 identity matrix to obtain

$$\mathbf{G} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Delete all columns of even weight from the $\mathbf{Q}$ submatrix shown in Eq. 3.8 and combine with a 4-×-4 identity matrix to obtain

$$\mathbf{H} = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

The resulting code is listed in Table 3-9.

■ Table 3-9 The (8, 4, d = 4) code obtained by shortening the (15, 11, d = 3) Hamming code of Example 3.8.

Info bits	Check bits	Weight
0 0 0 0	0 0 0 0	0
0 0 0 1	1 1 1 0	4
0 0 1 0	1 1 0 1	4
0 0 1 1	0 0 1 1	4
0 1 0 0	1 0 1 1	4
0 1 0 1	0 1 0 1	4
0 1 1 0	0 1 1 0	4
0 1 1 1	1 0 0 0	4
1 0 0 0	0 1 1 1	4
1 0 0 1	1 0 0 1	4
1 0 1 0	1 0 1 0	4
1 0 1 1	0 1 0 0	4
1 1 0 0	1 1 0 0	4
1 1 0 1	0 0 1 0	4
1 1 1 0	0 0 0 1	4
1 1 1 1	1 1 1 1	8

Convolutional Codes

6

Convolutional codes are a different breed than the block codes that have been covered thus far. Some authors contend that the two types of codes are **very** different, while other authors tend to emphasize their similarities. McEliece[1] shows how convolutional codes can be viewed as a special type of block code. The mathematical basis of convolutional coding is not as developed or rigorous as it is for block codes, and there are some fundamental obstacles that might cause this to always remain true. Nevertheless, convolutional codes are important weapons in the error-correction wars and they are used in many real-world systems.

6.1 Canonical Example

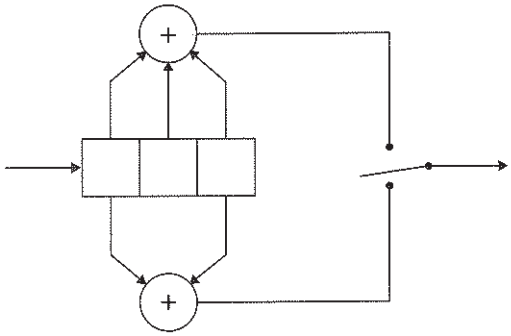
The easiest way to introduce convolutional codes is to start with a specific example.[2] Consider the structure shown in Fig. 6-1. Bits are shifted into the left end of the 3-stage shift register. The switch changes position at twice the input-shifting rate, alternately selecting the output of the top XOR gate and bottom XOR gate. The structure shown is an encoder for a single-input, 2-output, convolutional code with memory of length 3.

It is often useful to analyze a convolutional encoder as a state machine, and to represent the behavior of this machine in the form of a state diagram. The encoder of Fig. 6-1 has a memory of 3 bits, and upon casual observation, it seems as though it would therefore have $2^3 = 8$ possible states. This encoder can, in fact, be represented as a state machine with 8 states as we will demonstrate in Section 6.1.1. However, as you will subsequently discover in Section 6.1.2, the operation of this encoder can be represented using only four states.

1 R. J. McEliece: *Encyclopedia of Mathematics and Its Applications, Vol 3: The Theory of Information and Coding*, Addison-Wesley, Reading MA, 1977.

2 J. H. van Lint has dubbed this particular example the "canonical example" because it is used in almost every introduction to convolutional coding.

J.H. van Lint: *Introduction to Coding Theory*, Springer-Verlag, Berlin, 1992.



■ **6-1** *The so-called "canonical example" of a simple convolutional encoder.*

6.1.1 Convolutional Encoder Viewed as a Moore Machine

Start with the assumption that the encoder does have 8 possible states, with each state being defined by one of the eight possible combinations of 3 bits in the register. It's easy enough to associate a pair of outputs T_U, T_L with each state by realizing that

$$T_U = b_L + b_M + b_R$$
$$T_L = b_L + b_R$$

where

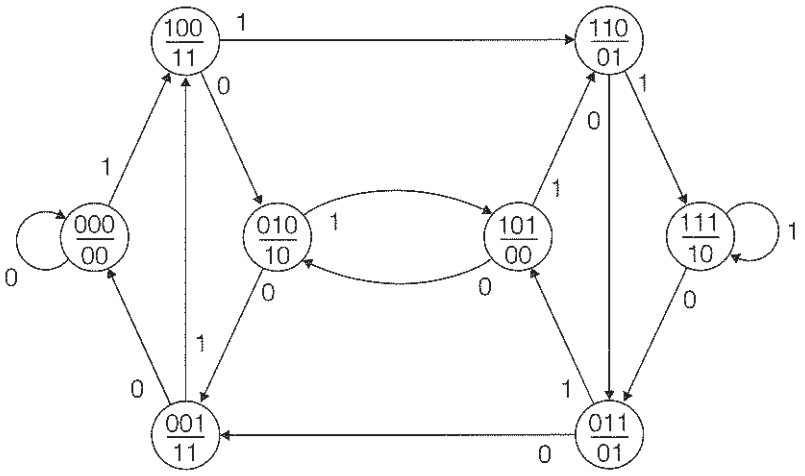
T_U = upper output
T_L = lower output
b_L = left bit in register
b_M = middle bit in register
b_R = right bit in register

Depending on its value, a new input bit will drive the machine from its old state to one of two possible new states. The new output is issued **after** arrival at the new state. The corresponding state diagram is shown in Fig. 6-2. Each state is represented by a circle containing its defining 3-bit combination and its corresponding two bits of output. A machine such as this one, in which the outputs are associated with the states, is called a *Moore machine*.

6.1.2 Convolutional Encoder Viewed as a Mealy Machine

Let's examine the first few steps in the operation of the example encoder to see how it might be possible to eliminate some steps:

1. Assume that at time 0, all three register stages contain zeros. The output bits are 00.



■ **6-2** *State diagram for the encoder of Fig. 6-1 represented as a Moore machine.*

2. The first input bit, u_1, is then shifted into the leftmost stage of the register. The register contents are $(u_1, 0, 0)$. The first output bit v_1 is obtained from the output of the upper summer. Therefore,

$$v_1 = u_1 + 0 + 0 = u_1$$

The second output bit, v_2, is obtained from the output of the lower summer as

$$v_2 = u_1 + 0 = u_1$$

The content of the leftmost stage is shifted into the middle stage, and second input bit, u_2, is shifted into the leftmost stage of the register. The register contents are now $(u_2, u_1, 0)$. The third output bit, v_3, is obtained from the output of the upper summer as

$$v_3 = u_2 + u_1 + 0$$

The fourth output bit, v_4, is obtained from the output of the lower summer as

$$v_4 = u_2 + 0$$

4. The content of the middle stage is shifted into the right stage, and the content of the left stage is shifted into the middle stage. The third input bit, u_3, is shifted into the left stage. The register contents are now (u_3, u_2, u_1). The fifth and sixth outputs are given by

$$v_5 = u_3 + u_2 + u_1$$
$$v_6 = u_3 + u_1$$

5. The content of the right stage is shifted out of existence, the content of the middle stage is shifted into the right stage, and the content of the left stage is shifted into the middle stage. The fourth input bit, u_4, is shifted into the left stage. The register contents are now (u_4, u_3, u_2). The seventh and eighth outputs are given by

$$v_7 = u_4 + u_3 + u_2$$
$$v_8 = u_4 + u_2$$

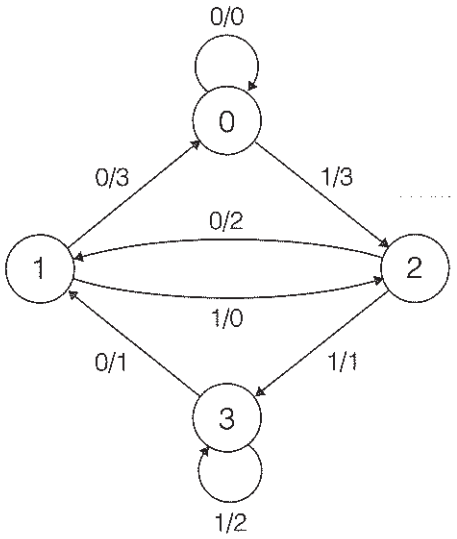
In each case, the outputs are determined by the three stages of the **new** register contents, or equivalently by the value of the input along with the **old** contents of the two leftmost stages of the register **just prior** to insertion of the new input bit. The insertion of the new input bit will push the contents of the rightmost stage completely out of the register. Therefore, two register states which differ only in the rightmost stage will yield the same next state for any given input. It seems as though we could redefine our sequential state machine in terms of just four 2-bit states, which are defined by the contents of the two leftmost stages of the register. The outputs, which are defined in terms of the original 3-bit states, must be mapped into the equivalent combinations of inputs and 2-bit states. This mapping is listed in Table 6-1.

Table 6-1 Mapping from 3-bit states to combinations of 2-bit states and inputs.

3-bit state	Output	2-bit state	Input
0 0 0	0 0	0 0	0
0 0 1	1 1	0 1	0
0 1 0	1 0	1 0	0
0 1 1	0 1	1 1	0
1 0 0	1 1	0 0	1
1 0 1	0 0	0 1	1
1 1 0	0 1	1 0	1
1 1 1	1 0	1 1	1

Each 2-bit state appears in the table twice, and each appearance has a different output. (For example, 2-bit state 01 appears once with output 11 for the 3-bit state 001 and once with output 00 for the 3-bit state 101.) Therefore, the outputs cannot be placed in the state circles as they are in Fig. 6-2. Instead, the outputs must be associated with the transition arrows, which are the only enti-

ties in the diagram that embody both input information and 2-bit state information. (Each transition arrow starts at the 2-bit state and is labeled with the input value that triggers the depicted transition.) A state diagram using 2-bit states is shown in Fig. 6-3. Each state is represented by a circle, labeled with the two leftmost register-bits that define the state. The transitions have two-part labels. The bit before the slash is the value of the input that will cause the indicated transition to occur. The two bits after the slash are the two output bits associated with the indicated transition. A machine, such as this one, in which the outputs are associated with the transition, is called a *Mealy* machine.



■ **6-3** *State diagram for the encoder of Fig. 6-1 represented as a Mealy machine.*

6.1.3 Moore Machines vs. Mealy Machines

It is always possible to convert a Mealy machine into an equivalent Moore machine and vice versa. The number of states in a Moore machine will equal or (usually) exceed the number of states in the equivalent Mealy machine. The Mealy machine of Fig. 6-3 can be converted into the Moore machine of Fig. 6-2 using a few rules of sequential machine design. First, a few definitions must be introduced to support the algorithm to follow.

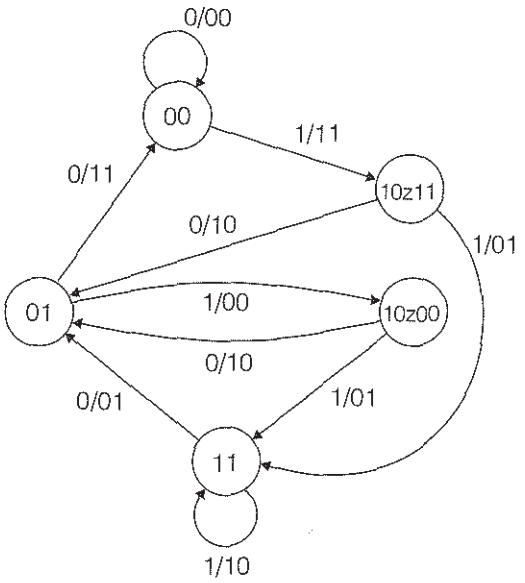
Definition 6.1 A state of a Mealy machine is *z-homogeneous* if the output associated with each transition line incident (*i.e.*, pointing into) to that state is the same; otherwise, it is *z-nonhomogeneous*.

A z-nonhomogeneous state of a Mealy machine can be split into an equivalent set of z-homogeneous states.

Definition 6.2 A Mealy machine is *z-homogeneous* if every state of the machine is z-homogeneous; otherwise it is *z-nonhomogeneous*.

A z-homogeneous Mealy machine is equivalent to a Moore machine. Because the outputs associated with all the transitions incident to a z-homogeneous state are equal, the output can be associated directly with the state.

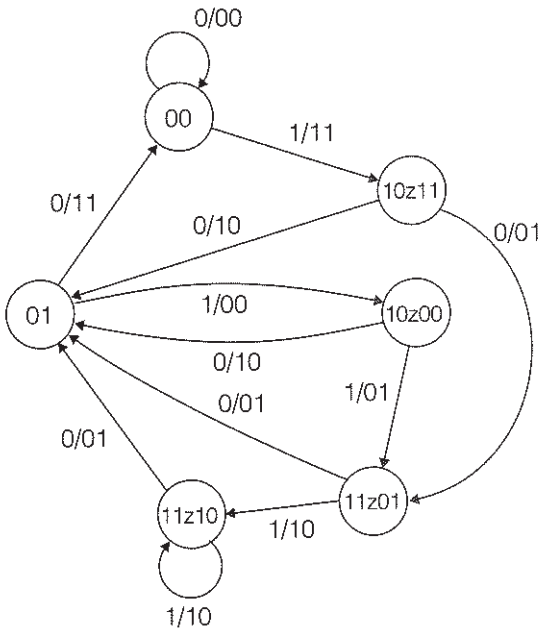
Example 6.6.1 Incident to state 10 in Fig. 6-3 are two transitions—one with output 00 and one with output 11. Hence, state 10 is z-nonhomogeneous. We can split state 10 into two states (for example, state 10z00 and state 10z11) as shown in Fig. 6-4. State 10z00 becomes the target for all transitions incident to state 10 with output 00, and state 10z11 becomes the target for all transitions incident to state 10 with output 11. All transitions out of state 10 must be duplicated for both states 10z00 and 10z11. (The transition labeled 0/10 going from state 10 to state 01 has been replaced with a transition from state 10z00 to state 01 plus a transition from state 10z11 to state 01.)



■ **6-4** *State diagram of a Mealy machine showing how z-nonhomogeneous state 10 can be split into two z-homogeneous states 10z11 and 10z00.*

Splitting states gets just a bit trickier for states that have *slings* (*i.e.*, transitions that start and end at the same state). In these cases, one of the new "post-split" states will retain the sling and there will be a transition from each of the other "post-split" states to the state that retains the sling. The input/output label on each of these transitions will match the input/output label on the sling.

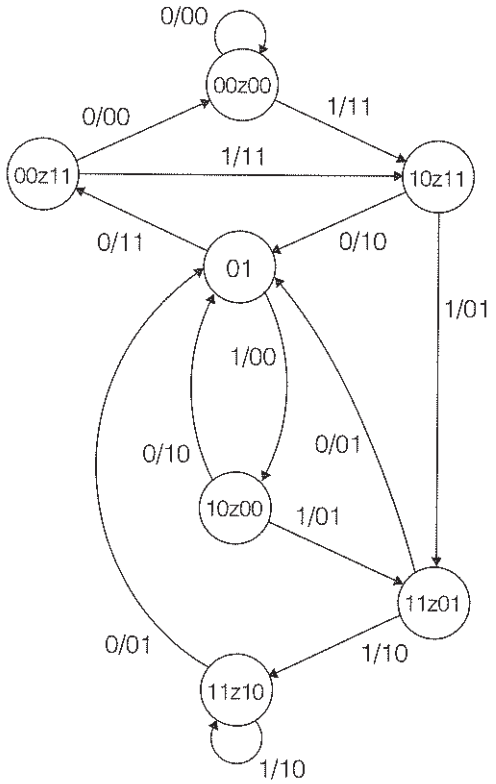
Example 6.6.2 Incident to state 11 in Fig. 6-4 are three transitions—two with output 01 and one (a sling) with output 10. We can split state 11 into two states 11z01 and 11z10 as shown in Fig. 6-5.



■ **6-5** *State diagram of a Mealy machine showing how a z-nonhomogeneous state with a sling (state 11) can be split into two z-homogeneous states 11z01 and 11z10.*

This is Example 1: Example 6.6.1. The results of Example 6.2 can be further extended by splitting state 00 to obtain Fig. 6-6 and then splitting state 01 to obtain Fig. 6-7, which is a z-homogeneous Mealy machine. Comparison of Figs. 6-7 and 6-2 reveals that the two machines are equivalent with the following correspondences between their states:

00z00 = 000
00z11 = 001
10z11 = 100



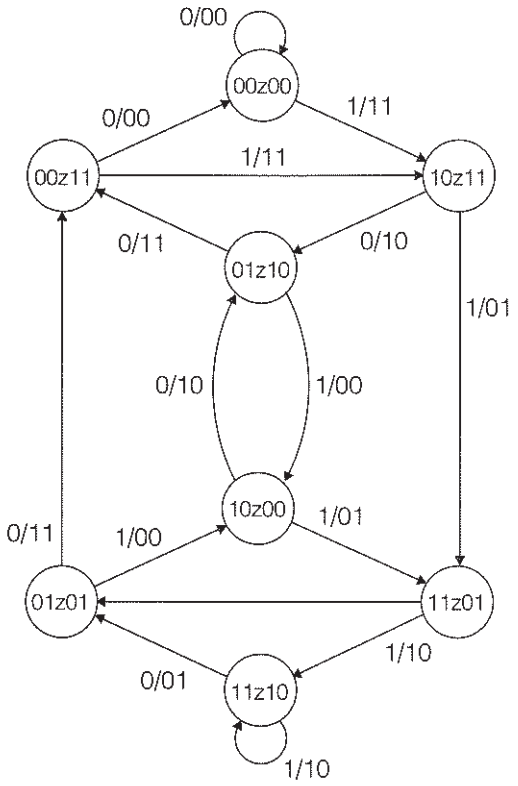
■ **6-6** *Result of splitting state 00 in the Mealy machine of Fig. 6-5.*

10z00 = 101
01z10 = 010
01z01 = 011
11z01 = 110
11z11 = 111

It often seems natural to think of the transitions as transient phenomena, which are short-lived relative to the possibly long dwell time that the machine spends within a state while waiting for an input to trigger the next transition. For this reason, texts on sequential machines often treat the outputs of a Mealy machine as *pulses* and the outputs of a Moore machine as *levels*. For purposes of encoder design, the outputs can be viewed as either pulses or levels.

6.1.4 Notation and Terminology

Notation and terminology for describing convolutional codes is not standard throughout the literature. This can be a source of con-



■ **6-7** *The z-homogeneous Mealy machine obtained by splitting state 01 in the Mealy machine of Fig. 6-6.*

siderable confusion for novices attempting to use multiple sources. Some of the confusion seems related to the choice between the Moore machine and Mealy machine for representing the encoder behavior.

Lin and Costello

The convention followed by Lin & Costello[3] and Rhee[4] allows the input to be "used" without first being shifted into a register. Thus, they would draw the encoder of Fig. 6-1 in the form shown in Fig. 6-8. This approach uses only 2 stages in the shift register and is perhaps more intuitively consistent with the 4-state Mealy machine of Fig. 6-3. Further bolstering this consistency is the fact that the outputs are shown directly as functions of **both** the 2-bit

3 S. Lin and D. J. Costello: *Error Control Coding: Fundamentals and Applications*, Prentice-Hall, Englewood Cliffs, NJ, 1983.

4 M. Y. Rhee: *Error Correcting Coding Theory*, McGraw-Hill, New York, 1989.

register state and the input. Both systems characterize convolutional codes using the ordered triple (n, k, m) where

n = number of outputs
k = number of inputs
m = memory order or memory length

In the code from Lin & Costello and Rhee, the *constraint length* n_A is defined as

$$n_A = n(m + 1)$$

The code implemented by Figs. 6-1 and 6-8 would be classified as (2, 1, 2) code with a constraint length of 6. The connections to the shift register are specified by associating a polynomial with each of the outputs:

upper XOR gate: $g^{(1)} = (1\ 1\ 1)$
lower XOR gate: $g^{(2)} = (1\ 0\ 1)$

Clark and Cain

Clark and Cain[5] characterize a convolutional code in terms of the code rate, and implicitly assume that when the code rate is expressed as a common fraction, m/n, then m is the number of inputs and n is the number of outputs. They denote the number of stages as k and call this quantity the *constraint length*. [For codes with multiple registers, k is the total number of register stages.] This definition of constraint length is different than that of Lin & Costello and Rhee. Using the conventions of Clark and Cain, the code implemented by Fig. 6-1 would be characterized as a rate ½, constraint length 3 code.

Michelson and Levesque

Michelson and Levesque[6] structure their notation and presentation around the idea that each input shift can accept a multi-bit symbol rather than just a single bit. Each symbol shift is called a *cycle* and the following notation is used

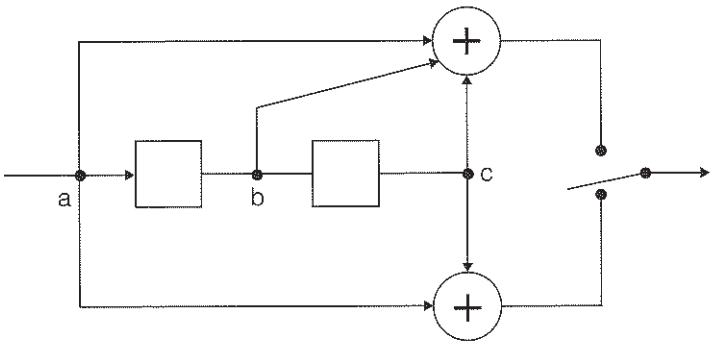
b = bits per cycle
k = number of register stages
V = number of outputs

The number of register stages k is in units of symbol stages, *i.e.*, for $b > 1$, k is given by

$$k = \frac{\textit{total number of register bits}}{b}$$

5 G. C. Clark and J. B. Cain: *Error-Correction Coding for Digital Communications*, Plenum Press, New York, 1981.

6 A. M. Michelson and A. H. Levesque: *Error-Control Techniques for Digital Communication*, Wiley-Interscience, New York, 1985.



■ **6-8** *The canonical example encoder redrawn following the convention used by Lin and Costello.*

Constraint length is defined as the number of symbol registers k. (It should be noted that Michelson and Levesque warn their readers about alternative definitions for constraint length.)

Viterbi

Viterbi[7] defines the *constraint length* as the number of **bit** stages in the encoder shift register. The number of **symbol** stages is denoted by K, and the constraint length is bK, where b is the number of bits per symbol. For the binary case, $b = 1$, and the constraint length is simply K. Viterbi describes an encoder similar to Fig. 6-8 as having a 3-stage shift register, and hence the corresponding code is described as having a constraint length of 3. (The two boxes in Fig. 6-8 should be thought of as delay elements between the stages rather than as the stages of the shift register. The three stages are the three signal nodes labeled A, B, and C in the figure.)

Rather than associating a polynomial with each output, Viterbi associates a polynomial with each stage of the shift register.

node a: $g_0 = (1, 1)$
node b: $g_1 = (1, 0)$
node c: $g_2 = (1, 1)$

Summary

With the possible exception of k, it's not really a big deal which particular letters are used to denote specific parameters. However, it is very common for communications engineers to speak of a "rate ½, $k = 7$" convolutional code. These engineers (many of whom have never seen the Clark and Cain code) don't say "mem-

7 A. J. Viterbi and J. K. Omura: *Principles of Digital Communication and Coding*, McGraw-Hill, New York, 1979.

ory of 7" or "$m = 7$"; they say "$k = 7$." Apparently, because of just common usage, the notation used by Clark and Cain seems to have acquired a certain "preferred" status despite the fact that it disagrees with that of Lin and Costello, which is somewhat of a *de facto* "standard" with regard to other terminology.

In addition to the notation and terminology discussed so far, there are a number of items on which there is general agreement.

Just as for a block code, we can define the *code rate* as the ratio of information bits to total channel bits. Conceptually, the output of a convolutional encoder is semi-infinite, beginning at time zero and continuing forever. Therefore, the code rate R would simply be the ratio of encoder inputs to encoder outputs.

$$R = \frac{k}{n}$$

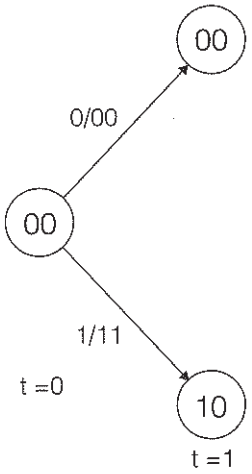
However, in the real world, the sequence of input bits will always be finite in length, and it will be necessary to append several zero bits to the input in order to "flush" the encoder registers and obtain all the output bits associated with the final information bit in the input sequence. This will increase slightly the percentage of noninformation bits in the encoder's output sequence. If the input sequence contains L bits and the encoder has m bits of memory, the *truncated code rate* R_T will be given by

$$R_T = \frac{kL}{n(m+L)} = R\left(\frac{L}{m+L}\right) = R\left(1 - \frac{m}{m+L}\right)$$

The quantity $m/(m+L)$ is sometimes referred to as the *fractional rate loss*.

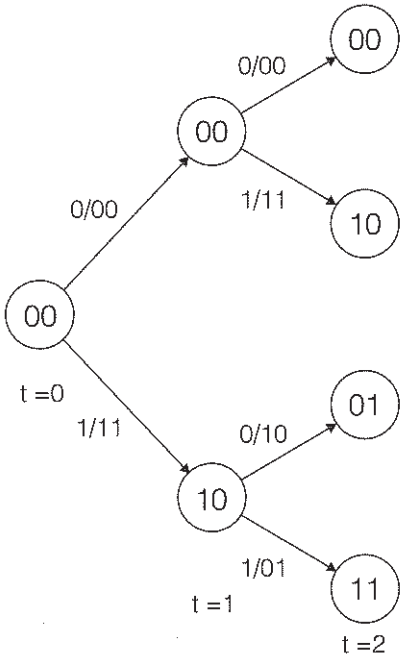
6.2 Tree Representation of a Convolutional Encoder

The state diagram of an encoder is a compact way to depict all of the possible states and transitions between them, but it does not provide a good view of the encoder operation as being sequential in time. Let's try to adapt the state diagram into something which is better able to depict the sequential nature of encoder operation. Consider the state diagram in Fig. 6-3. Assume that at $t = 0$ the encoder is in state 00. For $t = 1$, only two states are possible: the encoder will remain in state 00 if the input is 0, or the encoder will change to state 10 if the input is 1. We can illustrate this fact by drawing just a portion of the state diagram as shown in Fig. 6-9. Notice how the states for $t = 1$ have been placed in a vertical column directly above the notation "$t = 1$." Now for transitions from $t = 1$ to $t = 2$, we must consider two cases. If the encoder is in state

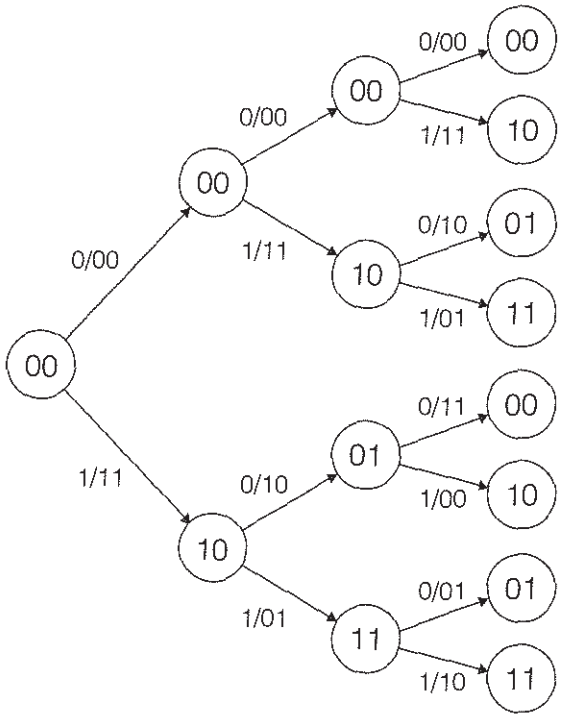


■ **6-9** *Portion of the state diagram used to show possible encoder states for t = 1.*

00 at $t = 1$, it can be in either state 00 or state 10 at $t = 2$. (Which one depends upon the input value.) Similarly, if the encoder is in state 10 at $t = 1$, it can change to either state 01 or state 11 at $t = 2$. We can illustrate these possibilities by adding another column to the diagram of Fig. 6-9 to obtain the diagram shown in Fig. 6-10. We could repeat the process for transitions from $t = 2$ to $t = 3$, and obtain the diagram shown in Fig. 6-11. It's easy to see how this diagram could quickly become unwieldy for even moderate values of



■ **6-10** *Modified state diagram used to show possible encoder states for t = 2.*



■ **6-11** *Modified state diagram used to show possible encoder states for t = 3.*

t. Several simplifications are usually made to make the diagram more compact:

1. In Fig. 6-11 each state is represented by a circle with one transition incident from the left and two transitions leaving from the right. This can be simplified by eliminating the circle and allowing the state to be implicitly represented by the intersection of the three transition lines. (If the input is not binary, there will be more than two transitions leaving from each state.)
2. The transitions leaving to the right of each state can be arranged in order of input value so that it becomes unnecessary to label each transition with the input value that triggers it. In most of the literature, the transition corresponding to a 0 input is drawn above the transition corresponding to a 1 input for binary encoders, but in some cases[8] the order is reversed.

8 A. M. Michelson and A. H. Levesque: *Error-Control Techniques for Digital Communication*, Wiley-Interscience, New York, 1985.

The result of these simplifications is called a *tree*. A tree for the encoder of Fig. 6-1 is shown in Fig. 6-12. At each fork, the number of branches equals the number of possible values for a single input symbol. The bits superimposed on each branch are the outputs issued in response to the input values corresponding to the branch.

Example 6.6.3 Assume that the encoder register contains all zeros prior to the arrival of the first input bit. The sequence of input bits is 1, 0, 0, 1, 0. Figure 6-13 shows the tree of Fig. 6-12 with the path corresponding to this input sequence highlighted. Following along the highlighted path we find the output sequence 11, 10, 11, 11, 10.

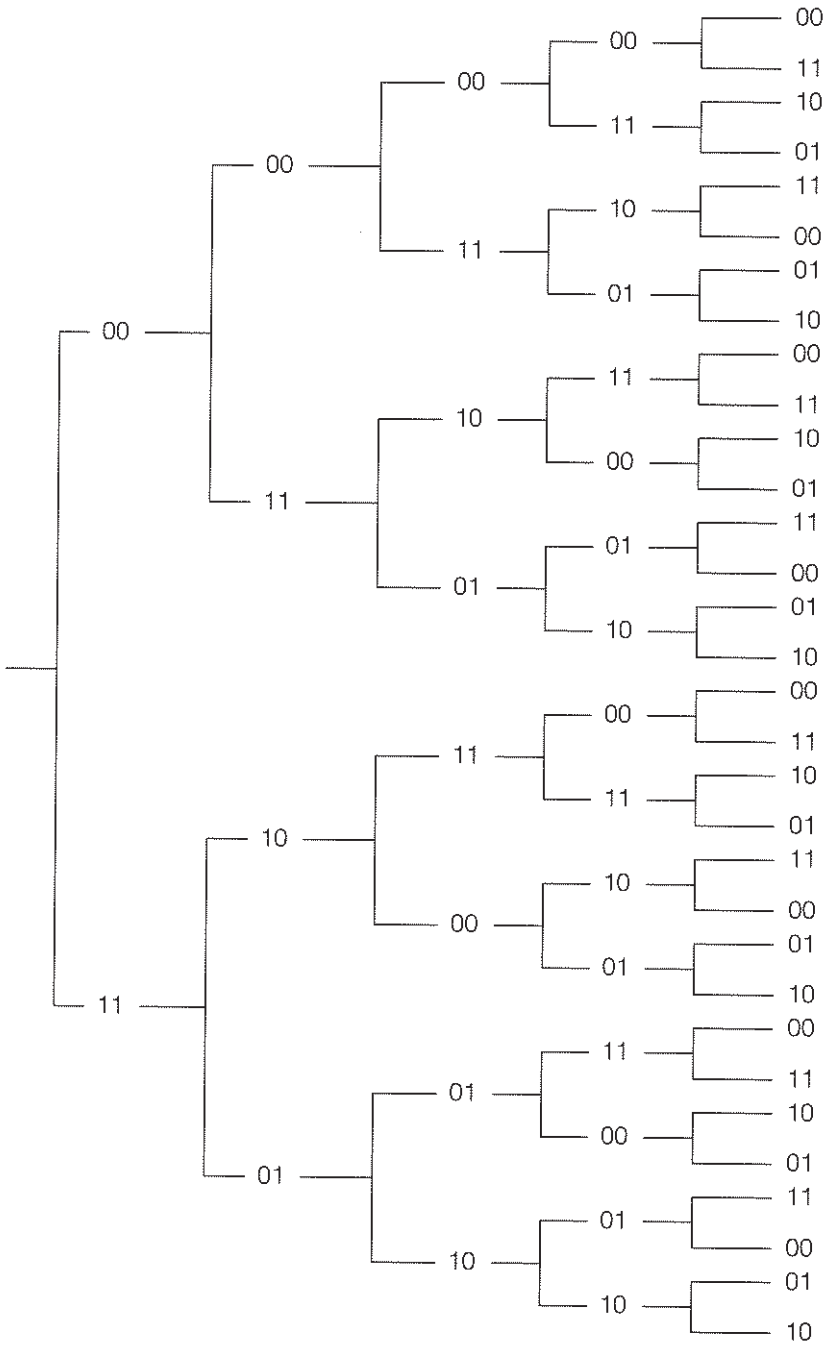
Figure 6-12 shows a tree the way it usually appears in the literature. I have found that it is sometimes easier to keep things straight if each fork is annotated with the encoder state as in Fig. 6-14.

6.3 Trellis Representation of a Convolutional Encoder

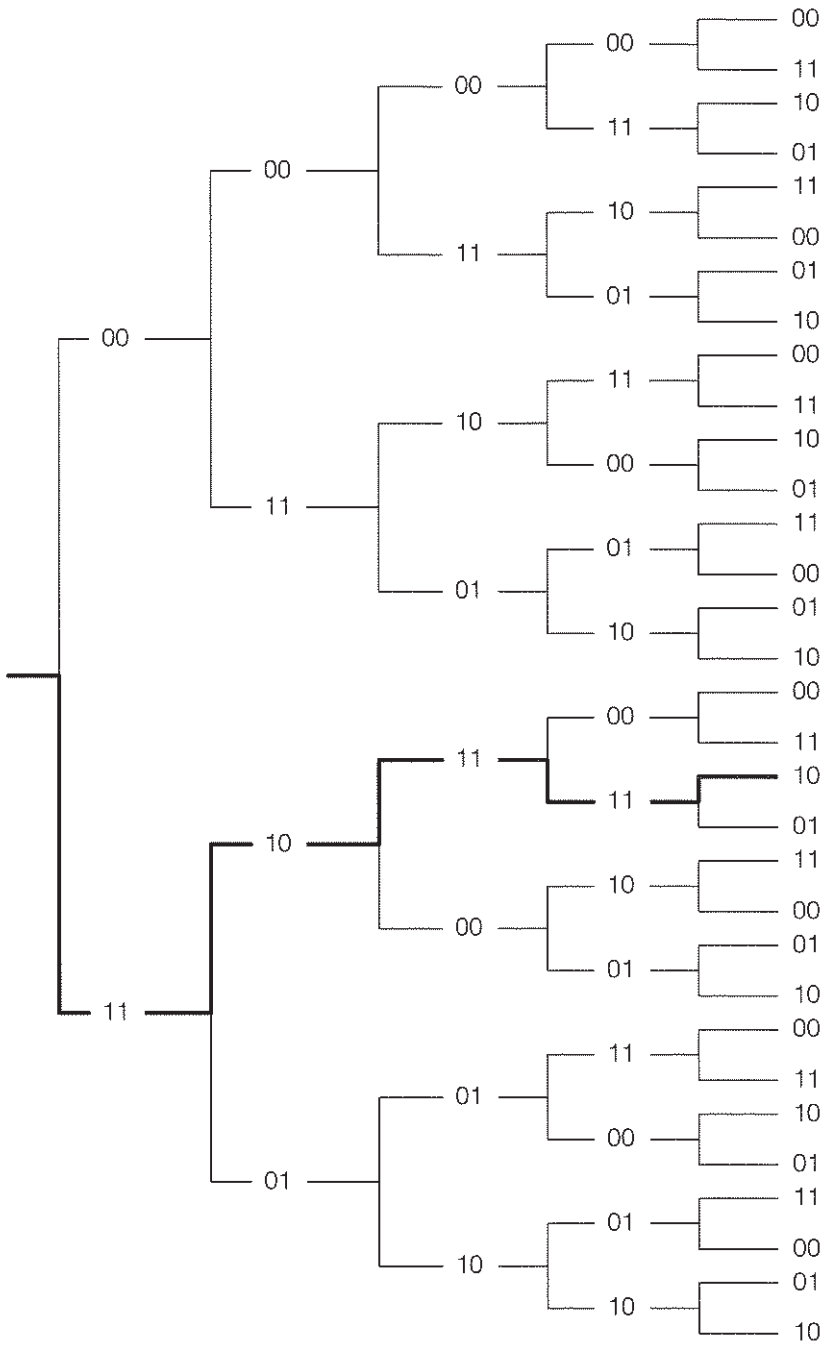
Even with simplifications, a tree can become difficult to draw as t increases. For binary encoders, each state will appear $2^{\tau-2}$ times in the column for $t = \tau$. This amounts to unnecessary repetition because for each appearance of a particular state, the output and next state will always be the same for a given input.

Example 6.6.4 Examination of Fig. 6-12 reveals that:

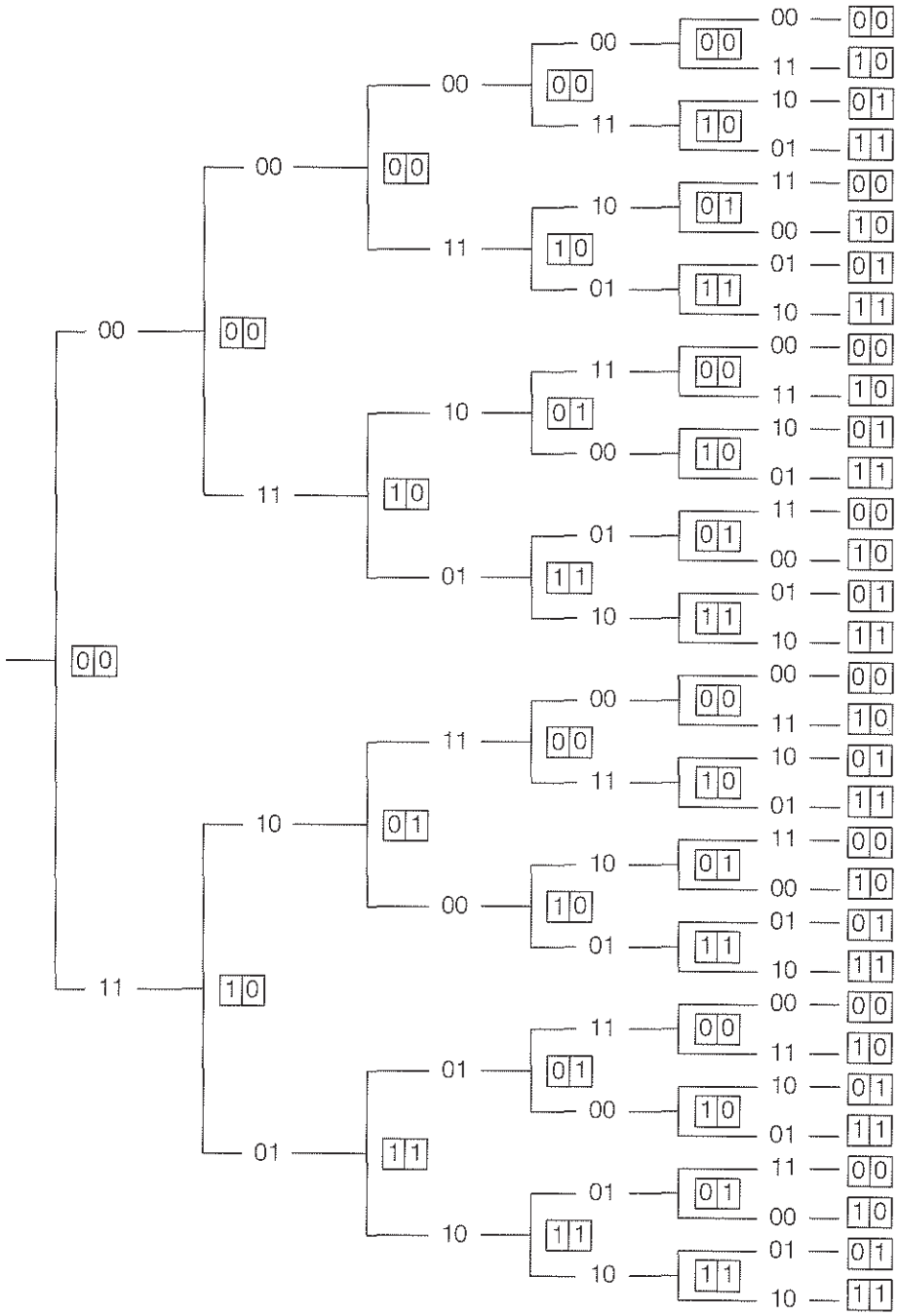
1. State 00 always reacts to an input of 0 by issuing 00 as output and remaining in state 00.
2. State 00 always reacts to an input of 1 by issuing 11 as output and changing to state 10.
3. State 01 always reacts to an input of 0 by issuing 11 as output and changing to state 00.
4. State 01 always reacts to an input of 1 by issuing 00 as output and changing to state 10.
5. State 10 always reacts to an input of 0 by issuing 10 as output and changing to state 01.
6. State 10 always reacts to an input of 1 by issuing 01 as output and changing to state 11.
7. State 11 always reacts to an input of 0 by issuing 01 as output and changing to state 01.
8. State 11 always reacts to an input of 1 by issuing 10 as output and remaining in state 10.



■ **6-12** *A tree used to depict possible sequences of encoder states for the encoder of Fig. 6-1.*



■ **6-13** *An encoder tree with path highlighted for the input sequence 10010.*

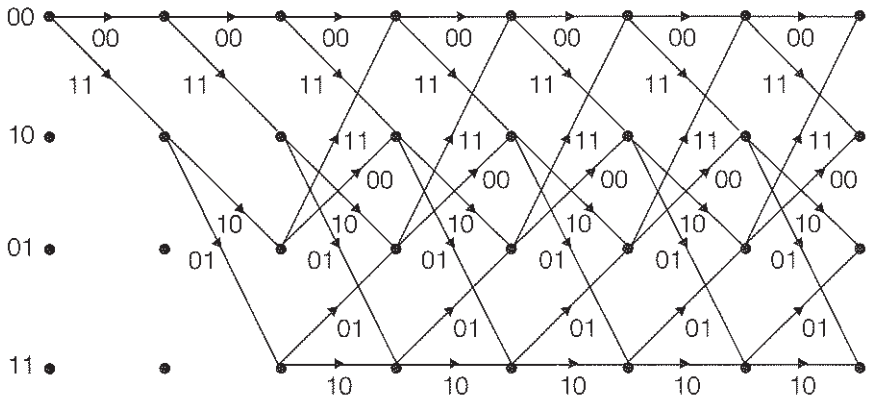


■ **6-14** *Tree with state annotations.*

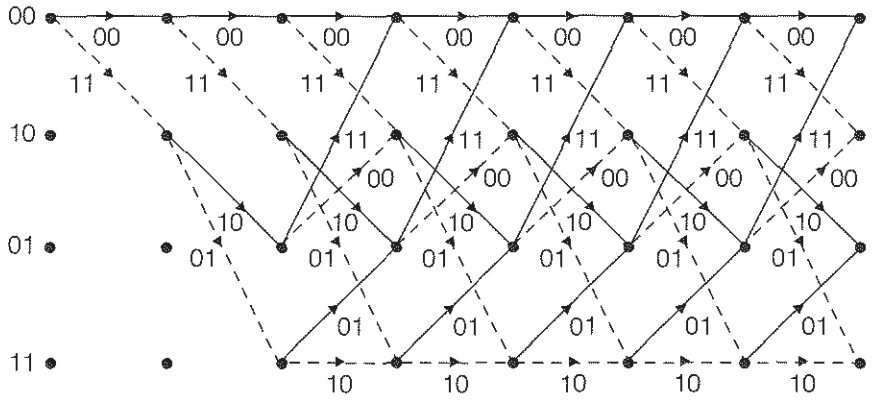
A tree can be turned into a *trellis* by allowing each state to appear only once in each column. Because of the fact that a given input will always produce the same output and next state for each appearance of a given state, it will always be possible to draw the necessary transitions between columns of the trellis without creating conflicts or ambiguities. The trellis corresponding to Fig. 6-12 is shown in Fig. 6-15. There are several different approaches for keeping track of which transitions are caused by which inputs. Figure 6-15 follows the same convention used in drawing trees—the states are arranged within each column so that for the transitions **leaving** a state, the transition caused by a 0 input is drawn above the transition because of a 1 input. For some codes, it might be difficult or impossible to find an order for the states that will allow this convention to be used. For binary codes, many authors use solid lines to draw transitions corresponding to 0 inputs and dashed lines to draw transitions corresponding to 1 inputs as shown in Fig. 6-16. It might be possible to extend this convention to nonbinary codes with small symbol alphabets by using different types of broken lines such as small dashes, long dashes, dashes alternated with dots, etc. Although it leads to more cluttered diagrams, the one approach guaranteed to work for all codes is to simply retain the two-part "input/output" label used in state diagrams, as shown in Fig. 6-17.

6.4 Distance Measures

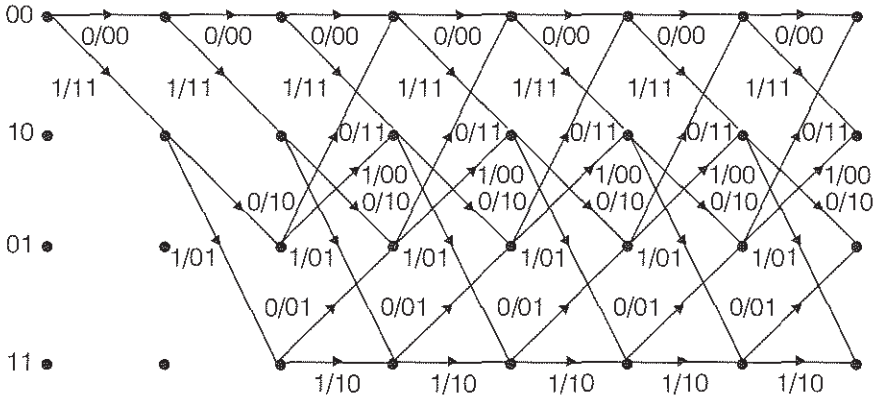
In the analysis of block codes, the minimum distance of a code proves to be an important measure for assessing the performance



■ **6-15** *A trellis used to depict possible sequences of encoder states for the encoder of Fig. 6-1.*



■ **6-16** *Trellis with dashed lines used to draw transitions corresponding to input = 1.*



■ **6-17** *Trellis having each transition labeled with both input and output.*

of the code. The same is true of convolutional codes, but the definition of "minimum distance" needs to be adjusted slightly for the different nature of convolutional codes.

For convolutional codes, the minimum distance between any two codewords is called the *minimum free distance* and is denoted as d_{free}. Because a convolutional code is a linear code, we can take one of the codewords as the all-zero codeword and view the minimum free distance as the weight of the minimum-weight nonzero codeword. Consider the code defined by the state diagram of Fig. 6-3. Because all codewords begin and end with the encoder at state 00, each nonzero codeword corresponds to sequence of state transitions that departs from state 00 for some number of state

times before returning to state 00. Thus, the minimum-weight nonzero codeword corresponds to the minimum-weight path that departs from and returns to state 00. The encoder defined by Fig. 6-3 is simple enough that we can find such a minimum-weight path by inspection. The only way out of state 00 is via the transition to state 10. This transition has a weight of 2. Similarly, the only way back into state 00 is via the transition from state 01. This transition also has a weight of 2. The transition from state 10 to state 01 has a weight of 1. The minimum-weight path that departs from and returns to state 00 goes through states 10 and 01 and has a total weight of $2 + 1 + 2 = 5$.