2

Binary Codes

The present chapter is an extension of the previous chapter on *number systems*. In the previous chapter, beginning with some of the basic concepts common to all number systems and an outline on the familiar decimal number system, we went on to discuss the binary, the hexadecimal and the octal number systems. While the binary system of representation is the most extensively used one in digital systems, including computers, octal and hexadecimal number systems are commonly used for representing groups of binary digits. The binary coding system, called the straight binary code and discussed in the previous chapter, becomes very cumbersome to handle when used to represent larger decimal numbers. To overcome this shortcoming, and also to perform many other special functions, several binary codes have evolved over the years. Some of the better-known binary codes, including those used efficiently to represent numeric and alphanumeric data, and the codes used to perform special functions, such as detection and correction of errors, will be detailed in this chapter.

2.1 Binary Coded Decimal

The binary coded decimal (BCD) is a type of binary code used to represent a given decimal number in an equivalent binary form. BCD-to-decimal and decimal-to-BCD conversions are very easy and straightforward. It is also far less cumbersome an exercise to represent a given decimal number in an equivalent BCD code than to represent it in the equivalent straight binary form discussed in the previous chapter.

The BCD equivalent of a decimal number is written by replacing each decimal digit in the integer and fractional parts with its four-bit binary equivalent. As an example, the BCD equivalent of $(23.15)_{10}$ is written as $(0010\ 0011.0001\ 0101)_{BCD}$. The BCD code described above is more precisely known as the 8421 BCD code, with 8, 4, 2 and 1 representing the weights of different bits in the four-bit groups, starting from MSB and proceeding towards LSB. This feature makes it a weighted code, which means that each bit in the four-bit group representing a given decimal digit has an assigned

Table 2.1 BCD codes.

Decimal	8421 BCD code	4221 BCD code	5421 BCD code
0	0000	0000	0000
1	0001	0001	0001
2	0010	0010	0010
3	0011	0011	0011
4	0100	1000	0100
5	0101	0111	1000
6	0110	1100	1001
7	0111	1101	1010
8	1000	1110	1011
9	1001	1111	1100

weight. Other weighted BCD codes include the 4221 BCD and 5421 BCD codes. Again, 4, 2, 2 and 1 in the 4221 BCD code and 5, 4, 2 and 1 in the 5421 BCD code represent weights of the relevant bits. Table 2.1 shows a comparison of 8421, 4221 and 5421 BCD codes. As an example, $(98.16)_{10}$ will be written as 1111 1110.0001 1100 in 4221 BCD code and 1100 1011.0001 1001 in 5421 BCD code. Since the 8421 code is the most popular of all the BCD codes, it is simply referred to as the BCD code.

2.1.1 BCD-to-Binary Conversion

A given BCD number can be converted into an equivalent binary number by first writing its decimal equivalent and then converting it into its binary equivalent. The first step is straightforward, and the second step was explained in the previous chapter. As an example, we will find the binary equivalent of the BCD number 0010 1001.0111 0101:

- BCD number: 0010 1001.0111 0101.
- Corresponding decimal number: 29.75.
- The binary equivalent of 29.75 can be determined to be 11101 for the integer part and .11 for the fractional part.
- Therefore, $(0010\ 1001.0111\ 0101)_{\mathrm{BCD}} = (11101.11)_2$.

2.1.2 Binary-to-BCD Conversion

The process of binary-to-BCD conversion is the same as the process of BCD-to-binary conversion executed in reverse order. A given binary number can be converted into an equivalent BCD number by first determining its decimal equivalent and then writing the corresponding BCD equivalent. As an example, we will find the BCD equivalent of the binary number 10101011.101:

- The decimal equivalent of this binary number can be determined to be 171.625.
- The BCD equivalent can then be written as 0001 0111 0001.0110 0010 0101.

2.1.3 Higher-Density BCD Encoding

In the regular BCD encoding of decimal numbers, the number of bits needed to represent a given decimal number is always greater than the number of bits required for straight binary encoding of the same. For example, a three-digit decimal number requires 12 bits for representation in conventional BCD format. However, since $2^{10} > 10^3$, if these three decimal digits are encoded together, only 10 bits would be needed to do that. Two such encoding schemes are *Chen-Ho encoding* and the *densely packed decimal*. The latter has the advantage that subsets of the encoding encode two digits in the optimal seven bits and one digit in four bits like regular BCD.

2.1.4 Packed and Unpacked BCD Numbers

In the case of unpacked BCD numbers, each four-bit BCD group corresponding to a decimal digit is stored in a separate register inside the machine. In such a case, if the registers are eight bits or wider, the register space is wasted.

In the case of packed BCD numbers, two BCD digits are stored in a single eight-bit register. The process of combining two BCD digits so that they are stored in one eight-bit register involves shifting the number in the upper register to the left 4 times and then adding the numbers in the upper and lower registers. The process is illustrated by showing the storage of decimal digits '5' and '7':

- Decimal digit 5 is initially stored in the eight-bit register as: 0000 0101.
- Decimal digit 7 is initially stored in the eight-bit register as: 0000 0111.
- After shifting to the left 4 times, the digit 5 register reads: 0101 0000.
- The addition of the contents of the digit 5 and digit 7 registers now reads: 0101 0111.

Example 2.1

How many bits would be required to encode decimal numbers 0 to 9999 in straight binary and BCD codes? What would be the BCD equivalent of decimal 27 in 16-bit representation?

Solution

- Total number of decimals to be represented = 10 000 = $10^4 = 2^{13.29}$.
- Therefore, the number of bits required for straight binary encoding = 14.
- The number of bits required for BCD encoding = 16.
- The BCD equivalent of 27 in 16-bit representation = 0000000000100111.

2.2 Excess-3 Code

The excess-3 code is another important BCD code. It is particularly significant for arithmetic operations as it overcomes the shortcomings encountered while using the 8421 BCD code to add two decimal digits whose sum exceeds 9. The excess-3 code has no such limitation, and it considerably simplifies arithmetic operations. Table 2.2 lists the excess-3 code for the decimal numbers 0–9.

The excess-3 code for a given decimal number is determined by adding '3' to each decimal digit in the given number and then replacing each digit of the newly found decimal number by

Table 2.2 Excess-3 code equivalent of decimal numbers.

Decimal number	Excess-3 code	Decimal number	Excess-3 code
0	0011	5	1000
1	0100	6	1001
2	0101	7	1010
3	0110	8	1011
4	0111	9	1100

its four-bit binary equivalent. It may be mentioned here that, if the addition of '3' to a digit produces a carry, as is the case with the digits 7, 8 and 9, that carry should not be taken forward. The result of addition should be taken as a single entity and subsequently replaced with its excess-3 code equivalent. As an example, let us find the excess-3 code for the decimal number 597:

- The addition of '3' to each digit yields the three new digits/numbers '8', '12' and '10'.
- The corresponding four-bit binary equivalents are 1000, 1100 and 1010 respectively.
- The excess-3 code for 597 is therefore given by: 1000 1100 1010 = 100011001010.

Also, it is normal practice to represent a given decimal digit or number using the maximum number of digits that the digital system is capable of handling. For example, in four-digit decimal arithmetic, 5 and 37 would be written as 0005 and 0037 respectively. The corresponding 8421 BCD equivalents would be 0000000000000101 and 0000000000110111 and the excess-3 code equivalents would be 0011001100111000 and 0011001101101010.

Corresponding to a given excess-3 code, the equivalent decimal number can be determined by first splitting the number into four-bit groups, starting from the radix point, and then subtracting 0011 from each four-bit group. The new number is the 8421 BCD equivalent of the given excess-3 code, which can subsequently be converted into the equivalent decimal number. As an example, following these steps, the decimal equivalent of excess-3 number 01010110.10001010 would be 23.57.

Another significant feature that makes this code attractive for performing arithmetic operations is that the complement of the excess-3 code of a given decimal number yields the excess-3 code for 9's complement of the decimal number. As adding 9's complement of a decimal number B to a decimal number A achieves A – B, the excess-3 code can be used effectively for both addition and subtraction of decimal numbers.

Example 2.3

Find (a) the excess-3 equivalent of $(237.75)_{10}$ and (b) the decimal equivalent of the excess-3 number 110010100011.01110101.

Solution

(a) Integer part = 237. The excess-3 code for $(237)_{10}$ is obtained by replacing 2, 3 and 7 with the four-bit binary equivalents of 5, 6 and 10 respectively. This gives the excess-3 code for $(237)_{10}$ as: 0101 0110 1010 = 010101101010.

Fractional part = .75. The excess-3 code for $(.75)_{10}$ is obtained by replacing 7 and 5 with the four-bit binary equivalents of 10 and 8 respectively. That is, the excess-3 code for $(.75)_{10}$ = .10101000.

Combining the results of the integral and fractional parts, the excess-3 code for $(237.75)_{10}$ = 010101101010.10101000.

(b) The excess-3 code = 110010100011.01110101 = 1100 1010 0011.0111 0101.

Subtracting 0011 from each four-bit group, we obtain the new number as: 1001 0111 0000.0100 0010.

Therefore, the decimal equivalent = $(970.42)_{10}$.

2.3 Gray Code

The Gray code was designed by Frank Gray at Bell Labs and patented in 1953. It is an unweighted binary code in which two successive values differ only by 1 bit. Owing to this feature, the maximum error that can creep into a system using the binary Gray code to encode data is much less than the worst-case error encountered in the case of straight binary encoding. Table 2.3 lists the binary and Gray code equivalents of decimal numbers 0–15. An examination of the four-bit Gray code numbers, as listed in Table 2.3, shows that the last entry rolls over to the first entry. That is, the last and the first entry also differ by only 1 bit. This is known as the *cyclic property* of the Gray code. Although there can be more than one Gray code for a given word length, the term was first applied to a specific binary code for non-negative integers and called the *binary-reflected Gray code* or simply the Gray code.

There are various ways by which Gray codes with a given number of bits can be remembered. One such way is to remember that the least significant bit follows a repetitive pattern of ‘2’ (11, 00, 11, . . .), the next higher adjacent bit follows a pattern of ‘4’ (1111, 0000, 1111, . . .) and so on. We can also generate the n-bit Gray code recursively by prefixing a ‘0’ to the Gray code for $n-1$ bits to obtain the first 2^{n-1} numbers, and then prefixing ‘1’ to the reflected Gray code for $n-1$ bits to obtain the remaining 2^{n-1} numbers. The reflected Gray code is nothing but the code written in reverse order. The process of generation of higher-bit Gray codes using the reflect-and-prefix method is illustrated in Table 2.4. The columns of bits between those representing the Gray codes give the intermediate step of writing the code followed by the same written in reverse order.

Table 2.3 Gray code.

Decimal	Binary	Gray	Decimal	Binary	Gray
0	0000	0000	8	1000	1100
1	0001	0001	9	1001	1101
2	0010	0011	10	1010	1111
3	0011	0010	11	1011	1110
4	0100	0110	12	1100	1010
5	0101	0111	13	1101	1011
6	0110	0101	14	1110	1001
7	0111	0100	15	1111	1000

Table 2.4 Generation of higher-bit Gray code numbers.

One-bit Gray code	Two-bit Gray code		Three-bit Gray code		Four-bit Gray code	
0	0	00	00	000	000	0000
1	1	01	01	001	001	0001
	1	11	11	011	011	0011
	0	10	10	010	010	0010
			10	110	110	0110
			11	111	111	0111
			01	101	101	0101
			00	100	100	0100
					100	1100
					101	1101
					111	1111
					110	1110
					010	1010
					011	1011
					001	1001
					000	1000

2.3.1 *Binary–Gray Code Conversion*

A given binary number can be converted into its Gray code equivalent by going through the following steps:

1. Begin with the most significant bit (MSB) of the binary number. The MSB of the Gray code equivalent is the same as the MSB of the given binary number.
2. The second most significant bit, adjacent to the MSB, in the Gray code number is obtained by adding the MSB and the second MSB of the binary number and ignoring the carry, if any. That is, if the MSB and the bit adjacent to it are both ‘1’, then the corresponding Gray code bit would be a ‘0’.
3. The third most significant bit, adjacent to the second MSB, in the Gray code number is obtained by adding the second MSB and the third MSB in the binary number and ignoring the carry, if any.
4. The process continues until we obtain the LSB of the Gray code number by the addition of the LSB and the next higher adjacent bit of the binary number.

The conversion process is further illustrated with the help of an example showing step-by-step conversion of $(1011)_2$ into its Gray code equivalent:

Binary	1011
Gray code	1- - -
Binary	1011
Gray code	11- -
Binary	1011
Gray code	111-
Binary	1011
Gray code	1110

2.3.2 Gray Code–Binary Conversion

A given Gray code number can be converted into its binary equivalent by going through the following steps:

1. Begin with the most significant bit (MSB). The MSB of the binary number is the same as the MSB of the Gray code number.
2. The bit next to the MSB (the second MSB) in the binary number is obtained by adding the MSB in the binary number to the second MSB in the Gray code number and disregarding the carry, if any.
3. The third MSB in the binary number is obtained by adding the second MSB in the binary number to the third MSB in the Gray code number. Again, carry, if any, is to be ignored.
4. The process continues until we obtain the LSB of the binary number.

The conversion process is further illustrated with the help of an example showing step-by-step conversion of the Gray code number 1110 into its binary equivalent:

Gray code	1110
Binary	1- - -
Gray code	1110
Binary	10 - -
Gray code	1110
Binary	101
Gray code	1110
Binary	1011

2.3.3 n-ary Gray Code

The binary-reflected Gray code described above is invariably referred to as the ‘Gray code’. However, over the years, mathematicians have discovered other types of Gray code. One such code is the n-ary Gray code, also called the non-Boolean Gray code owing to the use of non-Boolean symbols for encoding. The generalized representation of the code is the (n, k)-Gray code, where n is the number of independent digits used and k is the word length. A ternary Gray code ($n = 3$) uses the values 0, 1 and 2, and the sequence of numbers in the two-digit word length would be (00, 01, 02, 12, 11, 10, 20, 21, 22). In the quaternary ($n = 4$) code, using 0, 1, 2 and 3 as independent digits and a two-digit word length, the sequence of numbers would be (00, 01, 02, 03, 13, 12, 11, 10, 20, 21, 22, 23, 33, 32, 31, 30). It is important to note here that an (n, k)-Gray code with an odd n does not exhibit the cyclic property of the binary Gray code, while in case of an even n it does have the cyclic property.

The (n, k)-Gray code may be constructed recursively, like the binary-reflected Gray code, or may be constructed iteratively. The process of generating larger word-length ternary Gray codes is illustrated in Table 2.5. The columns between those representing the ternary Gray codes give the intermediate steps.

2.3.4 Applications

1. The Gray code is used in the transmission of digital signals as it minimizes the occurrence of errors.
2. The Gray code is preferred over the straight binary code in angle-measuring devices. Use of the Gray code almost eliminates the possibility of an angle misread, which is likely if the

Table 2.5 Generation of a larger word-length ternary Gray code.

One-digit ternary code	Two-digit ternary code		Three-digit ternary code	
0	0	00	00	000
1	1	01	01	001
2	2	02	02	002
	2	12	12	012
	1	11	11	011
	0	10	10	010
	0	20	20	020
	1	21	21	021
	2	22	22	022
			22	122
			21	121
			20	120
			10	110
			11	111
			12	112
			02	102
			01	101
			00	100
			00	200
			01	201
			02	202
			12	212
			11	211
			10	210
			20	220
			21	221
			22	222

angle is represented in straight binary. The cyclic property of the Gray code is a plus in this application.

3. The Gray code is used for labelling the axes of Karnaugh maps, a graphical technique used for minimization of Boolean expressions.
4. The use of Gray codes to address program memory in computers minimizes power consumption. This is due to fewer address lines changing state with advances in the program counter.
5. Gray codes are also very useful in genetic algorithms since mutations in the code allow for mostly incremental changes. However, occasionally a one-bit change can result in a big leap, thus leading to new properties.

Example 2.4

Find (a) the Gray code equivalent of decimal 13 and (b) the binary equivalent of Gray code number 1111.

Solution

(a) The binary equivalent of decimal 13 is 1101.
Binary–Gray conversion

Binary 1101
Gray 1- - -
Binary 1101
Gray 10 - -
Binary 1101
Gray 101 –
Binary 1101
Gray 1011

(b) *Gray–binary conversion*

Gray 1111
Binary 1- - -
Gray 1111
Binary 10- -
Gray 1111
Binary 101-
Gray 1111
Binary 1010

Example 2.5

Given the sequence of three-bit Gray code as (000, 001, 011, 010, 110, 111, 101, 100), write the next three numbers in the four-bit Gray code sequence after 0101.

Solution

The first eight of the 16 Gray code numbers of the four-bit Gray code can be written by appending '0' to the eight three-bit Gray code numbers. The remaining eight can be determined by appending '1' to the eight three-bit numbers written in reverse order. Following this procedure, we can write the next three numbers after 0101 as 0100, 1100 and 1101.

2.4 Alphanumeric Codes

Alphanumeric codes, also called character codes, are binary codes used to represent alphanumeric data. The codes write alphanumeric data, including letters of the alphabet, numbers, mathematical symbols and punctuation marks, in a form that is understandable and processable by a computer. These codes enable us to interface input–output devices such as keyboards, printers, VDUs, etc., with the computer. One of the better-known alphanumeric codes in the early days of evolution of computers, when punched cards used to be the medium of inputting and outputting data, is the 12-bit Hollerith code. The Hollerith code was used in those days to encode alphanumeric data on punched cards. The code has, however, been rendered obsolete, with the punched card medium having completely vanished from the scene. Two widely used alphanumeric codes include the ASCII and the EBCDIC codes. While the former is popular with microcomputers and is used on nearly all personal computers and workstations, the latter is mainly used with larger systems.

Traditional character encodings such as ASCII, EBCDIC and their variants have a limitation in terms of the number of characters they can encode. In fact, no single encoding contains enough characters so as to cover all the languages of the European Union. As a result, these encodings do not permit multilingual computer processing. Unicode, developed jointly by the Unicode Consortium and the International Standards Organization (ISO), is the most complete character encoding scheme that allows text of all forms and languages to be encoded for use by computers. Different codes are described in the following.

2.4.1 ASCII code

The ASCII (American Standard Code for Information Interchange), pronounced 'ask-ee', is strictly a seven-bit code based on the English alphabet. ASCII codes are used to represent alphanumeric data in computers, communications equipment and other related devices. The code was first published as a standard in 1967. It was subsequently updated and published as ANSI X3.4-1968, then as ANSI X3.4-1977 and finally as ANSI X3.4-1986. Since it is a seven-bit code, it can at the most represent 128 characters. It currently defines 95 printable characters including 26 upper-case letters (A to Z), 26 lower-case letters (a to z), 10 numerals (0 to 9) and 33 special characters including mathematical symbols, punctuation marks and space character. In addition, it defines codes for 33 nonprinting, mostly obsolete control characters that affect how text is processed. With the exception of 'carriage return' and/or 'line feed', all other characters have been rendered obsolete by modern mark-up languages and communication protocols, the shift from text-based devices to graphical devices and the elimination of teleprinters, punch cards and paper tapes. An eight-bit version of the ASCII code, known as US ASCII-8 or ASCII-8, has also been developed. The eight-bit version can represent a maximum of 256 characters.

Table 2.6 lists the ASCII codes for all 128 characters. When the ASCII code was introduced, many computers dealt with eight-bit groups (or bytes) as the smallest unit of information. The eighth bit was commonly used as a parity bit for error detection on communication lines and other device-specific functions. Machines that did not use the parity bit typically set the eighth bit to '0'.

Table 2.6 ASCII code.

Decimal	Hex	Binary	Code	Code description
0	00	0000 0000	NUL	Null character
1	01	0000 0001	SOH	Start of header
2	02	0000 0010	STX	Start of text
3	03	0000 0011	ETX	End of text
4	04	0000 0100	EOT	End of transmission
5	05	0000 0101	ENQ	Enquiry
6	06	0000 0110	ACK	Acknowledgement
7	07	0000 0111	BEL	Bell
8	08	0000 1000	BS	Backspace
9	09	0000 1001	HT	Horizontal tab
10	0A	0000 1010	LF	Line feed
11	0B	0000 1011	VT	Vertical tab
12	0C	0000 1100	FF	Form feed
13	0D	0000 1101	CR	Carriage return
14	0E	0000 1110	SO	Shift out
15	0F	0000 1111	SI	Shift in
16	10	0001 0000	DLE	Data link escape
17	11	0001 0001	DC1	Device control 1 (XON)

Table 2.6 (*continued*).

Decimal	Hex	Binary	Code	Code description
18	12	0001 0010	DC2	Device control 2
19	13	0001 0011	DC3	Device control 3 (XOFF)
20	14	0001 0100	DC4	Device control 4
21	15	0001 0101	NAK	Negative acknowledgement
22	16	0001 0110	SYN	Synchronous idle
23	17	0001 0111	ETB	End of transmission block
24	18	0001 1000	CAN	Cancel
25	19	0001 1001	EM	End of medium
26	1A	0001 1010	SUB	Substitute
27	1B	0001 1011	ESC	Escape
28	1C	0001 1100	FS	File separator
29	1D	0001 1101	GS	Group separator
30	1E	0001 1110	RS	Record separator
31	1F	0001 1111	US	Unit separator
32	20	0010 0000	SP	Space
33	21	0010 0001	!	Exclamation point
34	22	0010 0010	"	Quotation mark
35	23	0010 0011	#	Number sign, octothorp, pound
36	24	0010 0100	$	Dollar sign
37	25	0010 0101	%	Percent
38	26	0010 0110	&	Ampersand
39	27	0010 0111	’	Apostrophe, prime
40	28	0010 1000	(	Left parenthesis
41	29	0010 1001	)	Right parenthesis
42	2A	0010 1010	*	Asterisk, ‘star’
43	2B	0010 1011	+	Plus sign
44	2C	0010 1100	,	Comma
45	2D	0010 1101	-	Hyphen, minus sign
46	2E	0010 1110	.	Period, decimal Point, ‘dot’
47	2F	0010 1111	/	Slash, virgule
48	30	0011 0000	0	0
49	31	0011 0001	1	1
50	32	0011 0010	2	2
51	33	0011 0011	3	3
52	34	0011 0100	4	4
53	35	0011 0101	5	5
54	36	0011 0110	6	6
55	37	0011 0111	7	7
56	38	0011 1000	8	8
57	39	0011 1001	9	9
58	3A	0011 1010	:	Colon
59	3B	0011 1011	;	Semicolon
60	3C	0011 1100	<	Less-than sign
61	3D	0011 1101	=	Equals sign
62	3E	0011 1110	>	Greater-than sign
63	3F	0011 1111	?	Question mark
64	40	0100 0000	@	At sign
65	41	0100 0001	A	A

(*continued overleaf*)

Table 2.6 (*continued*).

Decimal	Hex	Binary	Code	Code description
66	42	0100 0010	B	B
67	43	0100 0011	C	C
68	44	0100 0100	D	D
69	45	0100 0101	E	E
70	46	0100 0110	F	F
71	47	0100 0111	G	G
72	48	0100 1000	H	H
73	49	0100 1001	I	I
74	4A	0100 1010	J	J
75	4B	0100 1011	K	K
76	4C	0100 1100	L	L
77	4D	0100 1101	M	M
78	4E	0100 1110	N	N
79	4F	0100 1111	O	O
80	50	0101 0000	P	P
81	51	0101 0001	Q	Q
82	52	0101 0010	R	R
83	53	0101 0011	S	S
84	54	0101 0100	T	T
85	55	0101 0101	U	U
86	56	0101 0110	V	V
87	57	0101 0111	W	W
88	58	0101 1000	X	X
89	59	0101 1001	Y	Y
90	5A	0101 1010	Z	Z
91	5B	0101 1011	[	Opening bracket
92	5C	0101 1100	\	Reverse slash
93	5D	0101 1101	]	Closing bracket
94	5E	0101 1110	∧	Circumflex, caret
95	5F	0101 1111	_	Underline, underscore
96	60	0110 0000	`	Grave accent
97	61	0110 0001	a	a
98	62	0110 0010	b	b
99	63	0110 0011	c	c
100	64	0110 0100	d	d
101	65	0110 0101	e	e
102	66	0110 0110	f	f
103	67	0110 0111	g	g
104	68	0110 1000	h	h
105	69	0110 1001	i	i
106	6A	0110 1010	j	j
107	6B	0110 1011	k	k
108	6C	0110 1100	l	l
109	6D	0110 1101	m	m
110	6E	0110 1110	n	n
111	6F	0110 1111	o	o
112	70	0111 0000	p	p
113	71	0111 0001	q	q
114	72	0111 0010	r	r

Table 2.6 (*continued*).

Decimal	Hex	Binary	Code	Code description
115	73	0111 0011	s	s
116	74	0111 0100	t	t
117	75	0111 0101	u	u
118	76	0111 0110	v	v
119	77	0111 0111	w	w
120	78	0111 1000	x	x
121	79	0111 1001	y	y
122	7A	0111 1010	z	z
123	7B	0111 1011	{	Opening brace
124	7C	0111 1100	|	Vertical line
125	7D	0111 1101	}	Closing brace
126	7E	0111 1110	~	Tilde
127	7F	0111 1111	DEL	Delete

Looking at the structural features of the code as reflected in Table 2.6, we can see that the digits 0 to 9 are represented with their binary values prefixed with 0011. That is, numerals 0 to 9 are represented by binary sequences from 0011 0000 to 0011 1001 respectively. Also, lower-case and upper-case letters differ in bit pattern by a single bit. While upper-case letters 'A' to 'O' are represented by 0100 0001 to 0100 1111, lower-case letters 'a' to 'o' are represented by 0110 0001 to 0110 1111. Similarly, while upper-case letters 'P' to 'Z' are represented by 0101 0000 to 0101 1010, lower-case letters 'p' to 'z' are represented by 0111 0000 to 0111 1010.

With widespread use of computer technology, many variants of the ASCII code have evolved over the years to facilitate the expression of non-English languages that use a Roman-based alphabet. In some of these variants, all ASCII printable characters are identical to their seven-bit ASCII code representations. For example, the eight-bit standard ISO/IEC 8859 was developed as a true extension of ASCII, leaving the original character mapping intact in the process of inclusion of additional values. This made possible representation of a broader range of languages. In spite of the standard suffering from incompatibilities and limitations, ISO-8859-1, its variant Windows-1252 and the original seven-bit ASCII continue to be the most common character encodings in use today.

2.4.2 EBCDIC code

The EBCDIC (Extended Binary Coded Decimal Interchange Code), pronounced 'eb-si-dik', is another widely used alphanumeric code, mainly popular with larger systems. The code was created by IBM to extend the binary coded decimal that existed at that time. All IBM mainframe computer peripherals and operating systems use EBCDIC code, and their operating systems provide ASCII and Unicode modes to allow translation between different encodings. It may be mentioned here that EBCDIC offers no technical advantage over the ASCII code and its variant ISO-8859 or Unicode. Its importance in the earlier days lay in the fact that it made it relatively easier to enter data into larger machines with punch cards. Since, punch cards are not used on mainframes any more, the code is used in contemporary mainframe machines solely for backwards compatibility.

It is an eight-bit code and thus can accommodate up to 256 characters. Table 2.7 gives the listing of characters in binary as well as hex form in EBCDIC. The arrangement is similar to the one adopted for Table 2.6 for the ASCII code. A single byte in EBCDIC is divided into two four-bit groups called

Table 2.7 EBCDIC code.

Decimal	Hex	Binary	Code	Code description
0	00	0000 0000	NUL	Null character
1	01	0000 0001	SOH	Start of header
2	02	0000 0010	STX	Start of text
3	03	0000 0011	ETX	End of text
4	04	0000 0100	PF	Punch off
5	05	0000 0101	HT	Horizontal tab
6	06	0000 0110	LC	Lower case
7	07	0000 0111	DEL	Delete
8	08	0000 1000		
9	09	0000 1001		
10	0A	0000 1010	SMM	Start of manual message
11	0B	0000 1011	VT	Vertical tab
12	0C	0000 1100	FF	Form feed
13	0D	0000 1101	CR	Carriage return
14	0E	0000 1110	SO	Shift out
15	0F	0000 1111	SI	Shift in
16	10	0001 0000	DLE	Data link escape
17	11	0001 0001	DC1	Device control 1
18	12	0001 0010	DC2	Device control 2
19	13	0001 0011	TM	Tape mark
20	14	0001 0100	RES	Restore
21	15	0001 0101	NL	New line
22	16	0001 0110	BS	Backspace
23	17	0001 0111	IL	Idle
24	18	0001 1000	CAN	Cancel
25	19	0001 1001	EM	End of medium
26	1A	0001 1010	CC	Cursor control
27	1B	0001 1011	CU1	Customer use 1
28	1C	0001 1100	IFS	Interchange file separator
29	1D	0001 1101	IGS	Interchange group separator
30	1E	0001 1110	IRS	Interchange record separator
31	1F	0001 1111	IUS	Interchange unit separator
32	20	0010 0000	DS	Digit select
33	21	0010 0001	SOS	Start of significance
34	22	0010 0010	FS	Field separator
35	23	0010 0011		
36	24	0010 0100	BYP	Bypass
37	25	0010 0101	LF	Line feed
38	26	0010 0110	ETB	End of transmission block
39	27	0010 0111	ESC	Escape
40	28	0010 1000		
41	29	0010 1001		
42	2A	0010 1010	SM	Set mode
43	2B	0010 1011	CU2	Customer use 2
44	2C	0010 1100		
45	2D	0010 1101	ENQ	Enquiry
46	2E	0010 1110	ACK	Acknowledge
47	2F	0010 1111	BEL	Bell
48	30	0011 0000		

Table 2.7 (*continued*).

Decimal	Hex	Binary	Code	Code description
49	31	0011 0001		
50	32	0011 0010	SYN	Synchronous idle
51	33	0011 0011		
52	34	0011 0100	PN	Punch on
53	35	0011 0101	RS	Reader stop
54	36	0011 0110	UC	Upper case
55	37	0011 0111	EOT	End of transmission
56	38	0011 1000		
57	39	0011 1001		
58	3A	0011 1010		
59	3B	0011 1011	CU3	Customer use 3
60	3C	0011 1100	DC4	Device control 4
61	3D	0011 1101	NAK	Negative acknowledge
62	3E	0011 1110		
63	3F	0011 1111	SUB	Substitute
64	40	0100 0000	SP	Space
65	41	0100 0001		
66	42	0100 0010		
67	43	0100 0011		
68	44	0100 0100		
69	45	0100 0101		
70	46	0100 0110		
71	47	0100 0111		
72	48	0100 1000		
73	49	0100 1001		
74	4A	0100 1010	¢	Cent sign
75	4B	0100 1011	.	Period, decimal point
76	4C	0100 1100	<	Less-than sign
77	4D	0100 1101	(	Left parenthesis
78	4E	0100 1110	+	Plus sign
79	4F	0100 1111	|	Logical OR
80	50	0101 0000	&	Ampersand
81	51	0101 0001		
82	52	0101 0010		
83	53	0101 0011		
84	54	0101 0100		
85	55	0101 0101		
86	56	0101 0110		
87	57	0101 0111		
88	58	0101 1000		
89	59	0101 1001		
90	5A	0101 1010	!	Exclamation point
91	5B	0101 1011	$	Dollar sign
92	5C	0101 1100	*	Asterisk
93	5D	0101 1101	)	Right parenthesis
94	5E	0101 1110	;	Semicolon
95	5F	0101 1111	∧	Logical NOT
96	60	0110 0000	-	Hyphen, minus sign

(*continued overleaf*)

Table 2.7 (*continued*).

Decimal	Hex	Binary	Code	Code description
97	61	0110 0001	/	Slash, virgule
98	62	0110 0010		
99	63	0110 0011		
100	64	0110 0100		
101	65	0110 0101		
102	66	0110 0110		
103	67	0110 0111		
104	68	0110 1000		
105	69	0110 1001		
106	6A	0110 1010		
107	6B	0110 1011	,	Comma
108	6C	0110 1100	%	Percent
109	6D	0110 1101	_	Underline, underscore
110	6E	0110 1110	>	Greater-than sign
111	6F	0110 1111	?	Question mark
112	70	0111 0000		
113	71	0111 0001		
114	72	0111 0010		
115	73	0111 0011		
116	74	0111 0100		
117	75	0111 0101		
118	76	0111 0110		
119	77	0111 0111		
120	78	0111 1000		
121	79	0111 1001	‘	Grave accent
122	7A	0111 1010	:	Colon
123	7B	0111 1011	#	Number sign, octothorp, pound
124	7C	0111 1100	@	At sign
125	7D	0111 1101	’	Apostrophe, prime
126	7E	0111 1110	=	Equals sign
127	7F	0111 1111	“	Quotation mark
128	80	1000 0000		
129	81	1000 1001	a	a
130	82	1000 1010	b	b
131	83	1000 1011	c	c
132	84	1000 1100	d	d
133	85	1000 0101	e	e
134	86	1000 0110	f	f
135	87	1000 0111	g	g
136	88	1000 1000	h	h
137	89	1000 1001	i	i
138	8A	1000 1010		
139	8B	1000 1011		
140	8C	1000 1100		
141	8D	1000 1101		
142	8E	1000 1110		
143	8F	1000 1111		
144	90	1001 0000		
145	91	1001 0001	j	j

Table 2.7 (*continued*).

Decimal	Hex	Binary	Code	Code description
146	92	1001 0010	k	k
147	93	1001 0011	l	l
148	94	1001 0100	m	m
149	95	1001 0101	n	n
150	96	1001 0110	o	o
151	97	1001 0111	p	p
152	98	1001 1000	q	q
153	99	1001 1001	r	r
154	9A	1001 1010		
155	9B	1001 1011		
156	9C	1001 1100		
157	9D	1001 1101		
158	9E	1001 1110		
159	9F	1001 1111		
160	A0	1010 0000		
161	A1	1010 0001	~	Tilde
162	A2	1010 0010	s	s
163	A3	1010 0011	t	t
164	A4	1010 0100	u	u
165	A5	1010 0101	v	v
166	A6	1010 0110	w	w
167	A7	1010 0111	x	x
168	A8	1010 1000	y	y
169	A9	1010 1001	z	z
170	AA	1010 1010		
171	AB	1010 1011		
172	AC	1010 1100		
173	AD	1010 1101		
174	AE	1010 1110		
175	AF	1010 1111		
176	B0	1011 0000		
177	B1	1011 0001		
178	B2	1011 0010		
179	B3	1011 0011		
180	B4	1011 0100		
181	B5	1011 0101		
182	B6	1011 0110		
183	B7	1011 0111		
184	B8	1011 1000		
185	B9	1011 1001		
186	BA	1011 1010		
187	BB	1011 1011		
188	BC	1011 1100		
189	BD	1011 1101		
190	BE	1011 1110		
191	BF	1011 1111		
192	C0	1100 0000	{	Opening brace
193	C1	1100 0001	A	A

(*continued overleaf*)

Table 2.7 (*continued*).

Decimal	Hex	Binary	Code	Code description
194	C2	1100 0010	B	B
195	C3	1100 0011	C	C
196	C4	1100 0100	D	D
197	C5	1100 0101	E	E
198	C6	1100 0110	F	F
199	C7	1100 0111	G	G
200	C8	1100 1000	H	H
201	C9	1100 1001	I	I
202	CA	1100 1010		
203	CB	1100 1011		
204	CC	1100 1100		
205	CD	1100 1101		
206	CE	1100 1110		
207	CF	1100 1111		
208	D0	1101 0000	}	Closing brace
209	D1	1101 0001	J	J
210	D2	1101 0010	K	K
211	D3	1101 0011	L	L
212	D4	1101 0100	M	M
213	D5	1101 0101	N	N
214	D6	1101 0110	O	O
215	D7	1101 0111	P	P
216	D8	1101 1000	Q	Q
217	D9	1101 1001	R	R
218	DA	1101 1010		
219	DB	1101 1011		
220	DC	1101 1100		
221	DD	1101 1101		
222	DE	1101 1110		
223	DF	1101 1111		
224	E0	1110 0000	\	Reverse slant
225	E1	1110 0001		
226	E2	1110 0010	S	S
227	E3	1110 0011	T	T
228	E4	1110 0100	U	U
229	E5	1110 0101	V	V
230	E6	1110 0110	W	W
231	E7	1110 0111	X	X
232	E8	1110 1000	Y	Y
233	E9	1110 1001	Z	Z
234	EA	1110 1010		
235	EB	1110 1011		
236	EC	1110 1100		
237	ED	1110 1101		
238	EE	1110 1110		
239	EF	1110 1111		
240	F0	1111 0000	0	0
241	F1	1111 0001	1	1

Table 2.7 (*continued*).

Decimal	Hex	Binary	Code	Code description
242	F2	1111 0010	2	2
243	F3	1111 0011	3	3
244	F4	1111 0100	4	4
245	F5	1111 0101	5	5
246	F6	1111 0110	6	6
247	F7	1111 0111	7	7
248	F8	1111 1000	8	8
249	F9	1111 1001	9	9
250	FA	1111 1010	|	
251	FB	1111 1011		
252	FC	1111 1100		
253	FD	1111 1101		
254	FE	1111 1110		
255	FF	1111 1111	eo	

nibbles. The first four-bit group, called the 'zone', represents the category of the character, while the second group, called the 'digit', identifies the specific character.

2.4.3 Unicode

As briefly mentioned in the earlier sections, encodings such as ASCII, EBCDIC and their variants do not have a sufficient number of characters to be able to encode alphanumeric data of all forms, scripts and languages. As a result, these encodings do not permit multilingual computer processing. In addition, these encodings suffer from incompatibility. Two different encodings may use the same number for two different characters or different numbers for the same characters. For example, code 4E (in hex) represents the upper-case letter 'N' in ASCII code and the plus sign '+' in the EBCDIC code. Unicode, developed jointly by the Unicode Consortium and the International Organization for Standardization (ISO), is the most complete character encoding scheme that allows text of all forms and languages to be encoded for use by computers. It not only enables the users to handle practically any language and script but also supports a comprehensive set of mathematical and technical symbols, greatly simplifying any scientific information exchange. The Unicode standard has been adopted by such industry leaders as HP, IBM, Microsoft, Apple, Oracle, Unisys, Sun, Sybase, SAP and many more.

Unicode and ISO-10646 Standards

Before we get on to describe salient features of Unicode, it may be mentioned that another standard similar in intent and implementation to Unicode is the ISO-10646. While Unicode is the brainchild of the Unicode Consortium, a consortium of manufacturers (initially mostly US based) of multilingual software, ISO-10646 is the project of the International Organization for Standardization. Although both organizations publish their respective standards independently, they have agreed to maintain compatibility between the code tables of Unicode and ISO-10646 and closely coordinate any further extensions.

The Code Table

The code table defined by both Unicode and ISO-10646 provides a unique number for every character, irrespective of the platform, program and language used. The table contains characters required to represent practically all known languages and scripts. The list includes not only the Greek, Latin, Cyrillic, Arabic, Arabian and Georgian scripts but also Japanese, Chinese and Korean scripts. In addition, the list also includes scripts such as Devanagari, Bengali, Gurmukhi, Gujarati, Oriya, Telugu, Tamil, Kannada, Thai, Tibetan, Ethiopic, Sinhala, Canadian Syllabics, Mongolian, Myanmar and others. Scripts not yet covered will eventually be added. The code table also covers a large number of graphical, typographical, mathematical and scientific symbols.

In the 32-bit version, which is the most recent version, the code table is divided into 2^{16} subsets, with each subset having 2^{16} characters. In the 32-bit representation, elements of different subsets therefore differ only in the 16 least significant bits. Each of these subsets is known as a plane. Plane 0, called the basic multilingual plane (BMP), defined by 00000000 to 0000FFFF, contains all most commonly used characters including all those found in major older encoding standards. Another subset of 2^{16} characters could be defined by 00010000 to 0001FFFF. Further, there are different slots allocated within the BMP to different scripts. For example, the basic Latin character set is encoded in the range 0000 to 007F. Characters added to the code table outside the 16-bit BMP are mostly for specialist applications such as historic scripts and scientific notation. There are indications that there may never be characters assigned outside the code space defined by 00000000 to 0010FFFF, which provides space for a little over 1 million additional characters.

Different characters in Unicode are represented by a hexadecimal number preceded by ‘U+’. For example, ‘A’ and ‘e’ in basic Latin are respectively represented by U+0041 and U+0065. The first 256 code numbers in Unicode are compatible with the seven-bit ASCII-code and its eight-bit variant ISO-8859-1. Unicode characters U+0000 to U+007F (128 characters) are identical to those in the ASCII code, and the Unicode characters in the range U+0000 to U+00FF (256 characters) are identical to ISO-8859-1.

Use of Combining Characters

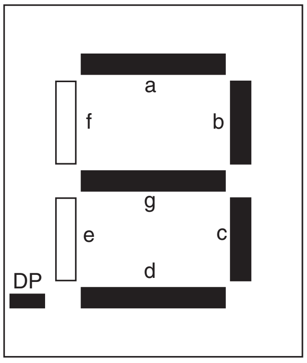
Unicode assigns code numbers to combining characters, which are not full characters by themselves but accents or other diacritical marks added to the previous character. This makes it possible to place any accent on any character. Although Unicode allows the use of combining characters, it also assigns separate codes to commonly used accented characters known as precomposed characters. This is done to ensure backwards compatibility with older encodings. As an example, the character ‘ä’ can be represented as the precomposed character U+00E4. It can also be represented in Unicode as U+0061 (Latin lower-case letter ‘a’) followed by U+00A8 (combining character ‘..’).

Unicode and ISO-10646 Comparison

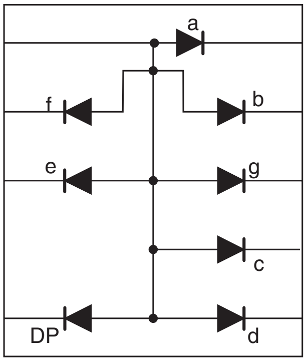
Although Unicode and ISO-10646 have identical code tables, Unicode offers many more features not available with ISO-10646. While the ISO-10646 standard is not much more than a comprehensive character set, the Unicode standard includes a number of other related features such as character properties and algorithms for text normalization and handling of bidirectional text to ensure correct display of mixed texts containing both right-to-left and left-to-right scripts.

2.5 Seven-segment Display Code

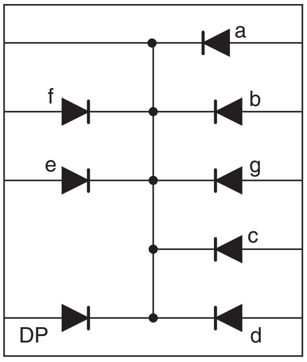
Seven-segment displays [Fig. 2.1(a)] are very common and are found almost everywhere, from pocket calculators, digital clocks and electronic test equipment to petrol pumps. A single seven-segment display or a stack of such displays invariably meets our display requirement. There are both LED and



(a)



(b)



(c)

Figure 2.1 Seven-segment displays.

Table 2.8 Seven-segment display code.

Common cathode type ‘1’ means ON									Common anode type ‘0’ means ON								
	a	b	c	d	e	f	g	DP		a	b	c	d	e	f	g	DP
0	1	1	1	1	1	1	0		0	0	0	0	0	0	0	1	
1	0	1	1	0	0	0	0		1	1	0	0	1	1	1	1	
2	1	1	0	1	1	0	1		2	0	0	1	0	0	1	0	
3	1	1	1	1	0	0	1		3	0	0	0	0	1	1	0	
4	0	1	1	0	0	1	1		4	1	0	0	1	1	0	0	
5	1	0	1	1	0	1	1		5	0	1	0	0	1	0	0	
6	0	0	1	1	1	1	1		6	1	1	0	0	0	0	0	
7	1	1	1	0	0	0	0		7	0	0	0	1	1	1	1	
8	1	1	1	1	1	1	1		8	0	0	0	0	0	0	0	
9	1	1	1	0	0	1	1		9	0	0	0	1	1	0	0	
a	1	1	1	1	1	0	1		a	0	0	0	0	0	1	0	
b	0	0	1	1	1	1	1		b	1	1	0	0	0	0	0	
c	0	0	0	1	1	0	1		c	1	1	1	0	0	1	0	
d	0	1	1	1	1	0	1		d	1	0	0	0	0	1	0	
e	1	1	0	1	1	1	1		e	0	0	1	0	0	0	0	
f	1	0	0	0	1	1	1		f	0	1	1	1	0	0	0	

LCD types of seven-segment display. Furthermore, there are common anode-type LED displays where the arrangement of different diodes, designated *a*, *b, c, d, e, f* and *g*, is as shown in Fig. 2.1(b), and common cathode-type displays where the individual diodes are interconnected as shown in Fig. 2.1(c). Each display unit usually has a dot point (DP).

The DP could be located either towards the left (as shown) or towards the right of the figure ‘8’ display pattern. This type of display can be used to display numerals from 0 to 9 and letters from A to F. Table 2.8 gives the binary code for displaying different numeric and alphabetic characters for both the common cathode and the common anode type displays. A ‘1’ lights a segment in the common cathode type display, and a ‘0’ lights a segment in the common anode type display.

2.6 Error Detection and Correction Codes

When we talk about digital systems, be it a digital computer or a digital communication set-up, the issue of error detection and correction is of great practical significance. Errors creep into the bit stream owing to noise or other impairments during the course of its transmission from the transmitter to the receiver. Any such error, if not detected and subsequently corrected, can be disastrous, as digital systems are sensitive to errors and tend to malfunction if the bit error rate is more than a certain threshold level. Error detection and correction, as we will see below, involves the addition of extra bits, called check bits, to the information-carrying bit stream to give the resulting bit sequence a unique characteristic that helps in detection and localization of errors. These additional bits are also called redundant bits as they do not carry any information. While the addition of redundant bits helps in achieving the goal of making transmission of information from one place to another error free or reliable, it also makes it inefficient. In this section, we will examine some common error detection and correction codes.

2.6.1 Parity Code

A parity bit is an extra bit added to a string of data bits in order to detect any error that might have crept into it while it was being stored or processed and moved from one place to another in a digital system.

We have an *even parity*, where the added bit is such that the total number of ls in the data bit string becomes even, and an *odd parity*, where the added bit makes the total number of ls in the data bit string odd. This added bit could be a ‘0’ or a ‘1’. As an example, if we have to add an even parity bit to 01000001 (the eight-bit ASCII code for ‘A’), it will be a ‘0’ and the number will become 001000001. If we have to add an odd parity bit to the same number, it will be a ‘l’ and the number will become 101000001. The odd parity bit is a complement of the even parity bit. The most common convention is to use even parity, that is, the total number of 1s in the bit stream, including the parity bit, is even.

The parity check can be made at different points to look for any possible single-bit error, as it would disturb the parity. This simple parity code suffers from two limitations. Firstly, it cannot detect the error if the number of bits having undergone a change is even. Although the number of bits in error being equal to or greater than 4 is a very rare occurrence, the addition of a single parity cannot be used to detect two-bit errors, which is a distinct possibility in data storage media such as magnetic tapes. Secondly, the single-bit parity code cannot be used to localize or identify the error bit even if one bit is in error. There are several codes that provide self-single-bit error detection and correction mechanisms, and these are discussed below.

2.6.2 Repetition Code

The repetition code makes use of repetitive transmission of each data bit in the bit stream. In the case of threefold repetition, ‘1’ and ‘0’ would be transmitted as ‘111’ and ‘000’ respectively. If, in the received data bit stream, bits are examined in groups of three bits, the occurrence of an error can be detected. In the case of single-bit errors, ‘1’ would be received as 011 or 101 or 110 instead of 111, and a ‘0’ would be received as 100 or 010 or 001 instead of 000. In both cases, the code becomes self-correcting if the bit in the majority is taken as the correct bit. There are various forms in which the data are sent using the repetition code. Usually, the data bit stream is broken into blocks of bits, and then each block of data is sent some predetermined number of times. For example, if we want to send eight-bit data given by 11011001, it may be broken into two blocks of four bits each. In the case of threefold repetition, the transmitted data bit stream would be 110111011101100110011001. However, such a repetition code where the bit or block of bits is repeated 3 times is not capable of correcting two-bit errors, although it can detect the occurrence of error. For this, we have to increase the number of times each bit in the bit stream needs to be repeated. For example, by repeating each data bit 5 times, we can detect and correct all two-bit errors. The repetition code is highly inefficient and the information throughput drops rapidly as we increase the number of times each data bit needs to be repeated to build error detection and correction capability.

2.6.3 Cyclic Redundancy Check Code

Cyclic redundancy check (CRC) codes provide a reasonably high level of protection at low redundancy level. The cycle code for a given data word is generated as follows. The data word is first appended by a number of 0s equal to the number of check bits to be added. This new data bit sequence is then divided by a special binary word whose length equals $n+1$, n being the number of check bits to be added. The remainder obtained as a result of modulo-2 division is then added to the dividend bit sequence to get the cyclic code. The code word so generated is completely divisible by the divisor used in the generation of the code. Thus, when the received code word is again divided by the same divisor, an error-free reception should lead to an all '0' remainder. A nonzero remainder is indicative of the presence of errors.

The probability of error detection depends upon the number of check bits, n, used to construct the cyclic code. It is 100 % for single-bit and two-bit errors. It is also 100 % when an odd number of bits are in error and the error bursts have a length less than $n + 1$. The probability of detection reduces to $1 - (1/2)^{n-1}$ for an error burst length equal to $n + 1$, and to $1 - (1/2)^n$ for an error burst length greater than $n + 1$.

2.6.4 Hamming Code

We have seen, in the case of the error detection and correction codes described above, how an increase in the number of redundant bits added to message bits can enhance the capability of the code to detect and correct errors. If we have a sufficient number of redundant bits, and if these bits can be arranged such that different error bits produce different error results, then it should be possible not only to detect the error bit but also to identify its location. In fact, the addition of redundant bits alters the 'distance' code parameter, which has come to be known as the Hamming distance. The Hamming distance is nothing but the number of bit disagreements between two code words. For example, the addition of single-bit parity results in a code with a Hamming distance of at least 2. The smallest Hamming distance in the case of a threefold repetition code would be 3. Hamming noticed that an increase in distance enhanced the code's ability to detect and correct errors. Hamming's code was therefore an attempt at increasing the Hamming distance and at the same time having as high an information throughput rate as possible.

The algorithm for writing the generalized Hamming code is as follows:

1. The generalized form of code is $P_1P_2D_1P_3D_2D_3D_4P_4D_5D_6D_7D_8D_9D_{10}D_{11}P_5\ldots$, where P and D respectively represent parity and data bits.
2. We can see from the generalized form of the code that all bit positions that are powers of 2 (positions 1, 2, 4, 8, 16, . . .) are used as parity bits.
3. All other bit positions (positions 3, 5, 6, 7, 9, 10, 11, . . .) are used to encode data.
4. Each parity bit is allotted a group of bits from the data bits in the code word, and the value of the parity bit (0 or 1) is used to give it certain parity.
5. Groups are formed by first checking $N-1$ bits and then alternately skipping and checking N bits following the parity bit. Here, N is the position of the parity bit; 1 for P_1, 2 for P_2, 4 for P_3, 8 for P_4 and so on. For example, for the generalized form of code given above, various groups of bits formed with different parity bits would be $P_1D_1D_2D_4D_5\ldots$, $P_2D_1D_3D_4D_6D_7\ldots$, $P_3D_2D_3D_4D_8D_9\ldots$, $P_4D_5D_6D_7D_8D_9D_{10}D_{11}\ldots$ and so on. To illustrate the formation of groups further, let us examine the group corresponding to parity bit P_3. Now, the position of P_3 is at number 4. In order to form the group, we check the first three bits ($N-1=3$) and then follow it up by alternately skipping and checking four bits ($N=4$).

The Hamming code is capable of correcting single-bit errors on messages of any length. Although the Hamming code can detect two-bit errors, it cannot give the error locations. The number of parity bits required to be transmitted along with the message, however, depends upon the message length, as shown above. The number of parity bits n required to encode m message bits is the smallest integer that satisfies the condition $(2^n - n) > m$.

Table 2.9 Generation of Hamming code.

	P_1	P_2	D_1	P_3	D_2	D_3	D_4
Data bits (without parity)			0		1	1	0
Data bits with parity bit P_1	1		0		1		0
Data bits with parity bit P_2		1	0			1	0
Data bits with parity bit P_3				0	1	1	0
Data bits with parity	1	1	0	0	1	1	0

The most commonly used Hamming code is the one that has a code word length of seven bits with four message bits and three parity bits. It is also referred to as the Hamming (7, 4) code. The code word sequence for this code is written as $P_1P_2D_1P_3D_2D_3D_4$, with P_1, P_2 and P_3 being the parity bits and D_1, D_2, D_3 and D_4 being the data bits. We will illustrate step by step the process of writing the Hamming code for a certain group of message bits and then the process of detection and identification of error bits with the help of an example. We will write the Hamming code for the four-bit message 0110 representing numeral '6'. The process of writing the code is illustrated in Table 2.9, with even parity.

Thus, the Hamming code for 0110 is 1100110. Let us assume that the data bit D_1 gets corrupted in the transmission channel. The received code in that case is 1110110. In order to detect the error, the parity is checked for the three parity relations mentioned above. During the parity check operation at the receiving end, three additional bits X, Y and Z are generated by checking the parity status of $P_1D_1D_2D_4$, $P_2D_1D_3D_4$ and $P_3D_2D_3D_4$ respectively. These bits are a '0' if the parity status is okay, and a '1' if it is disturbed. In that case, ZYX gives the position of the bit that needs correction. The process can be best explained with the help of an example.

Examination of the first parity relation gives $X = 1$ as the even parity is disturbed. The second parity relation yields $Y = 1$ as the even parity is disturbed here too. Examination of the third relation gives $Z = 0$ as the even parity is maintained. Thus, the bit that is in error is positioned at 011 which is the binary equivalent of '3'. This implies that the third bit from the MSB needs to be corrected. After correcting the third bit, the received message becomes 1100110 which is the correct code.

Example 2.6

By writing the parity code (even) and threefold repetition code for all possible four-bit straight binary numbers, prove that the Hamming distance in the two cases is at least 2 in the case of the parity code and 3 in the case of the repetition code.

Solution

The generation of codes is shown in Table 2.10. An examination of the parity code numbers reveals that the number of bit disagreements between any pair of code words is not less than 2. It is either 2 or 4. It is 4, for example, between 00000 and 10111, 00000 and 11011, 00000 and 11101, 00000 and 11110 and 00000 and 01111. In the case of the threefold repetition code, it is either 3, 6, 9 or 12 and therefore not less than 3 under any circumstances.

Example 2.7

It is required to transmit letter 'A' expressed in the seven-bit ASCII code with the help of the Hamming (11, 7) code. Given that the seven-bit ASCII notation for 'A' is 1000001 and that the data word gets

Table 2.10 Example 2.6.

Binary number	Parity code	Three-time repetition Code	Binary number	Parity code	Three-time repetition code
0000	00000	000000000000	1000	11000	100010001000
0001	10001	000100010001	1001	01001	100110011001
0010	10010	001000100010	1010	01010	101010101010
0011	00011	001100110011	1011	11011	101110111011
0100	10100	010001000100	1100	01100	110011001100
0101	00101	010101010101	1101	11101	110111011101
0110	00110	011001100110	1110	11110	111011101110
0111	10111	011101110111	1111	01111	111111111111

corrupted to 1010001 in the transmission channel, show how the Hamming code can be used to identify the error. Use even parity.

Solution

- The generalized form of the Hamming code in this case is $P_1P_2D_1P_3D_2D_3D_4P_4D_5D_6D_7 = P_1P_21P_3000P_4001$.
- The four groups of bits using different parity bits are $P_1D_1D_2D_4D_5D_7$, $P_2D_1D_3D_4D_6D_7$, $P_3D_2D_3D_4$ and $P_4D_5D_6D_7$.
- This gives $P_1 = 0$, $P_2 = 0$, $P_3 = 0$ and $P_4 = 1$.
- Therefore, the transmitted Hamming code for 'A' is 00100001001.
- The received Hamming code is 00100101001.
- Checking the parity for the P_1 group gives '0' as it passes the test.
- Checking the parity for the P_2 group gives '1' as it fails the test.
- Checking the parity for the P_3 group gives '1' as it fails the test.
- Checking the parity for the P_4 group gives '0' as it passes the test.
- The bits resulting from the parity check, written in reverse order, constitute 0110, which is the binary equivalent of '6'. This shows that the bit in error is the sixth from the MSB.
- Therefore, the corrected Hamming code is 00100001001, which is the same as the transmitted code.
- The received data word is 1000001.

Review Questions

1. Distinguish between weighted and unweighted codes. Give two examples each of both types of code.
2. What is an excess-3 BCD code? Which shortcoming of the 8421 BCD code is overcome in the excess-3 BCD code? Illustrate with the help of an example.
3. What is the Gray code? Why is it also known as the binary-reflected Gray code? Briefly outline some of the important applications of the Gray code.
4. Briefly describe salient features of the ASCII and EBCDIC codes in terms of their capability to represent characters and suitability for their use in different platforms.
5. What is the Unicode? Why is it called the most complete character code?

6. What is a parity bit? Define even and odd parity. What is the limitation of the parity code when it comes to detection and correction of bit errors?
7. What is the Hamming distance? What is the role of the Hamming distance in deciding the error detection and correction capability of a code meant for the purpose? How does it influence the information throughput rate?
8. With the help of the generalized form of the Hamming code, explain how the number of parity bits required to transmit a given number of data bits is decided upon.

Problems

1. Write the excess-3 equivalent codes of $(6)_{10}$, $(78)_{10}$ and $(357)_{10}$, all in 16-bit format.

   0011001100111001, 0011001110101011, 0011011010001010
2. Determine the Gray code equivalent of $(10011)_2$ and the binary equivalent of the Gray code number 110011.

   11010, $(100010)_2$
3. A 16-bit data word given by 1001100001110110 is to be transmitted by using a fourfold repetition code. If the data word is broken into four blocks of four bits each, then write the transmitted bit stream.

   1001100110011001100010001000100001110111011101110110011001100110
4. Write (a) the Hamming (7, 4) code for 0000 using even parity and (b) the Hamming (11, 7) code for 1111111 using odd parity.

   (a) 0000000; (b) 00101110111
5. Write the last four of the 16 possible numbers in the two-bit quaternary Gray code with 0, 1, 2 and 3 as its independent digits, beginning with the thirteenth number.

   33, 32, 31, 30

Further Reading

1. Tokheim, R. L. (1994) *Schaum's Outline Series of Digital Principles*, McGraw-Hill Book Companies Inc., USA.
2. Gillam, R. (2002) *Unicode Demystified: A Practical Programmer's Guide to the Encoding Standard*, 1st edition, Addison-Wesley Professional, Boston, MA, USA.
3. MacWilliams, F. J. and Sloane, N. J. A. (2006) *The Theory of Error-Correcting Codes*, North-Holland Mathematical Library, Elsevier Ltd, Oxford, UK.
4. Huffman, W. C. and Pless, V. (2003) *Fundamentals of Error-Correcting Codes*, Cambridge University Press, Cambridge, UK.