APPENDIX

E

CHARACTER CODES AND NUMBER CONVERSION

E.1 CHARACTER CODES

Information storage and processing in computers involves coding the individual items of information by using several binary variables. Positive and negative numbers are represented in some variation of the binary number system. The most usual formats are presented in Chapter 6, where both integer and floating-point numbers are discussed.

In computers used mainly for business data processing, it is useful to represent and process numbers in the base-10 (decimal) format. Table E.1 gives the most usual coding for individual digits, called the binary-coded decimal (BCD) code. This code is simply the first 10 values (0–9) of the 4-bit binary number system. Strings of these 4-bit code values can be used to represent any desired range of positive and negative integers, with an appropriate code used for the sign position.

Alphabetic characters (A–Z), operators, punctuation symbols, control characters (+ – / , : ; LF CR EOT), and numbers must be represented for text storage and editing and for high-level language input, processing, and output operations. Two standard codes for this purpose are the American Standards Committee on Information Interchange (ASCII) code and the Extended Binary Coded Decimal Interchange Code (EBCDIC). The standard ASCII code is a 7-bit code, and the EBCDIC code is an 8-bit code. Tables E.2 and E.3 show the standard ASCII and EBCDIC codes, respectively. The ASCII code is by far the most frequently used.

In many applications, it is preferable to use 8-bit quantities; thus, the basic ASCII code is often extended to 8-bits. A common way of doing this is to set the high-order bit position, bit 7, to zero. Another popular possibility is to use bit 7 as a parity bit for the encoded character.

Some comments about the structure of the ASCII and EBCDIC codes are helpful. Note that in both codes the low-order 4 bits of the decimal character codes (0–9) are the BCD codes of Table E.1. This facilitates two operations. First, two characters that represent decimal digits can be compared to determine which is larger. This can be done with the same type of logic circuits that are used to perform the standard arithmetic

Table E.1 BCD encoding of decimal digits

Decimal digit	BCD code
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Table E.2 The 7-bit ASCII code

Bit positions	Bit positions 654							
3210	000	001	010	011	100	101	110	111
0000	NUL	DLE	SPACE	0	@	P	'	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	/	l	|
1101	CR	GS	−	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	—	o	DEL

NUL	Null/Idle	SI	Shift in
SOH	Start of header	DLE	Data link escape
STX	Start of text	DC1-DC4	Device control
ETX	End of text	NAK	Negative acknowledgment
EOT	End of transmission	SYN	Synchronous idle
ENQ	Enquiry	ETB	End of transmitted block
ACK	Acknowledgment	CAN	Cancel (error in data)
BEL	Audible signal	EM	End of medium
BS	Back space	SUB	Special sequence
HT	Horizontal tab	ESC	Escape
LF	Line feed	FS	File separator
VT	Vertical tab	GS	Group separator
FF	Form feed	RS	Record separator
CR	Carriage return	US	Unit separator
SO	Shift out	DEL	Delete/Idle

Bit positions of code format = [6][5][4][3][2][1][0]

Table E.3 The 8-bit EBCDIC code

Bit positions	Bit positions 7654															
3210	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0000	NULL				SP	&	–									0
0001							/		a	j			A	J		1
0010									b	k	s		B	K	S	2
0011									c	l	t		C	L	T	3
0100	PF	RES	BYP	PN					d	m	u		D	M	U	4
0101	HT	NL	LF	RS					e	n	v		E	N	V	5
0110	LC	BS	EOB	UC					f	o	w		F	O	W	6
0111	DEL	IL	PRE	EOT					g	p	x		G	P	X	7
1000									h	q	y		H	Q	Y	8
1001									i	r	z		I	R	Z	9
1010			SM		¢	!		:								
1011					.	$	,	#								
1100					<	*	%	@								
1101					(	)	–	'								
1110					+	;	>	=								
1111					|	¬	?	"								

NULL	Null/Idle	NL	New line	PRE	Prefix
PF	Punch off	BS	Backspace	SM	Set mode
HT	Horizontal tab	IL	Idle	PN	Punch on
LC	Lowercase	BYP	Bypass	RS	Reader stop
DEL	Delete	LF	Line feed	UC	Uppercase
RES	Restore	EOB	End of block	EOT	End of transmission
				SP	Space

Bit positions of code format = | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

operations on binary numbers. This is helpful when strings of decimal numbers must be sorted into numerical order. Second, when it is determined by context that consecutive 7- or 8-bit codes in some input string represent a decimal number that is to be stored and processed as a single entity, then it is sometimes practical to remove the leftmost 3 or 4 bits of each digit code and compress the number being represented into a string of 4-bit BCD digits. This compression or packing of data requires starting and ending delimiters, but it is justified in many situations in which storage space requirements are a concern. Similar comments apply to the codes for the alphabetic characters. The fact that their binary bit patterns are in numerical sequence facilitates alphabetic sorting.

E.2 DECIMAL-TO-BINARY CONVERSION

This section shows how to convert a fixed-point decimal number to its binary equivalent. The value, V, represented by the binary number

$$B = b_n b_{n-1} \cdots b_0.b_{-1} b_{-2} \cdots b_{-m}$$

is given by

$$V(B) = b_n \times 2^n + b_{n-1} \times 2^{n-1} + \cdots + b_0 \times 2^0$$
$$+ b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \cdots + b_{-m} \times 2^{-m}$$

To convert a fixed-point decimal number into binary, the integer and fraction parts are handled separately. First, the integer part is converted as follows. It is divided by 2. The remainder is the least significant bit of the integer part of the binary representation. The quotient is again divided by 2, and the remainder is the next bit of the binary representation. The process is repeated up to and including the step in which the quotient becomes 0.

Second, the fraction part is converted by multiplying it by 2. The part of the product to the left of the decimal point, which is either 0 or 1, is a bit in the binary representation. The fractional part of the product is again multiplied by 2, generating the next bit of the binary representation. The first bit generated is the bit immediately to the right of the binary point. The next bit generated is the second bit to the right, and so on. The process is repeated until the required accuracy is attained.

Figure E.1 shows an example of conversion from $(927.45)_{10}$ to binary. Note that conversion of the integer part is always exact, but the binary fraction for an exact decimal fraction may not be exact. For example, the fraction $(0.45)_{10}$ used in Figure E.1 does not have an exact binary equivalent. This is obvious from the pattern developing in the figure. In such cases, the binary fraction is generated to some desired level of accuracy. In general, the maximum absolute error, e, in generating a k-bit fractional representation is bounded as $e \leq 2^{-k}$. Of course, some decimal fractions have an exact binary representation. For example, $(0.25)_{10}$ equals $(0.01)_2$.

Convert $(927.45)_{10}$

$\frac{927}{2} = 463 + \frac{1}{2}$ → 1 LSB

$\frac{463}{2} = 231 + \frac{1}{2}$ → 1

$\frac{231}{2} = 115 + \frac{1}{2}$ → 1

$\frac{115}{2} = 57 + \frac{1}{2}$ → 1

$\frac{57}{2} = 28 + \frac{1}{2}$ → 1

$\frac{28}{2} = 14 + \frac{0}{2}$ → 0

$\frac{14}{2} = 7 + \frac{0}{2}$ → 0

$\frac{7}{2} = 3 + \frac{1}{2}$ → 1

$\frac{3}{2} = 1 + \frac{1}{2}$ → 1

$\frac{1}{2} = 0 + \frac{1}{2}$ → 1 LSB

$0.45 \times 2 = 0.90$ → 0 MSB

$0.90 \times 2 = 1.80$ → 1

$0.80 \times 2 = 1.60$ → 1

$0.60 \times 2 = 1.20$ → 1

$0.20 \times 2 = 0.40$ → 0

$0.40 \times 2 = 0.80$ → 0

$0.80 \times 2 = 1.60$ → 1 LSB

$$(927.45)_{10} = (1110011111.0111001\ \ldots)_2$$

Figure E.1 Conversion from decimal to binary.

INDEX

A

D

J

K

L

M

Q

R