Designers were aware of the conversion problem well before the incident occurred. However, deploying new software under wartime conditions is anything but trivial. Although the new software would have fixed the bug, field personnel could have simply rebooted the systems at specific intervals to keep the clock value small enough so that 24-bit precision would have been sufficient.

One of the most famous examples of a floating-point numeric disaster is the explosion of the Ariane 5 rocket. On June 4, 1996 the unmanned Ariane 5 was launched by the European Space Agency. Forty seconds after liftoff, the rocket exploded, scattering a $500 million cargo across parts of French Guiana. Investigation revealed perhaps one of the most devastatingly careless but efficient software bugs in the annals of computer science—a floating-point conversion error. The rocket's inertial reference system converted a 64-bit floating-point number (dealing with the horizontal velocity of the rocket) to a 16-bit signed integer. However, the particular 64-bit floating-point number to be converted was larger than 32,767 (the largest integer that can be stored in 16-bit signed representation), so the conversion process failed. The rocket tried to make an abrupt course correction for a wrong turn that it had never taken, and the guidance system shut down. Ironically, when the guidance system shut down, control reverted to a backup unit installed in the rocket in case of just such a failure, but the backup system was running the same flawed software.

It seems obvious that a 64-bit floating-point number could be much larger than 32,767, so how did the rocket programmers make such a glaring error? They decided the velocity value would never get large enough to be a problem. Their reasoning? It had never gotten too large before. Unfortunately, this rocket was faster than all previous rockets, resulting in a larger velocity value than the programmers expected. One of the most serious mistakes a programmer can make is to accept the old adage "but we've always done it that way."

Computers are everywhere—in our washing machines, our televisions, our microwaves—even our cars. We certainly hope the programmers that work on computer software for our cars don't make such hasty assumptions. With approximately 15 to 60 microprocessors in all new cars that roll off the assembly line and innumerable processors in commercial aircraft and medical equipment, a deep understanding of floating-point anomalies can quite literally be a lifesaver.

2.6 CHARACTER CODES

We have seen how digital computers use the binary system to represent and manipulate numeric values. We have yet to consider how these internal values can be converted to a form that is meaningful to humans. The manner in which this is done depends on both the coding system used by the computer and how the values are stored and retrieved.

2.6.1 Binary-Coded Decimal

For many applications, we need the exact binary equivalent of the decimal system, which means we need an encoding for individual decimal digits. This is pre-

cisely the case in many business applications that deal with money—we can't afford the rounding errors that occur when we convert real numbers to floating point when making financial transactions!

Binary coded decimal (BCD) is very common in electronics, particularly those that display numerical data, such as alarm clocks and calculators. BCD encodes each digit of a decimal number into a 4-bit binary form. Each decimal digit is individually converted to its binary equivalent, as seen in Table 2.5. For example, to encode 146, the decimal digits are replaced by 0001, 0100, and 0110 respectively.

Because most computers use bytes as the smallest unit of access, most values are stored in eight bits, not four. That gives us two choices for storing 4-bit BCD digits. We can ignore the cost of extra bits and pad the high-order nibbles with zeros (or ones), forcing each decimal digit to be replaced by 8 bits. Using this approach, padding with zeros, 146 would be stored as 00000001 00000100 00000110. Clearly, this approach is quite wasteful. The second approach, called **packed BCD**, stores two digits per byte. Packed decimal format allows numbers to be signed, but instead of putting the sign at the beginning, the sign is stored at the end. The standard values for this "sign digit" are 1100 for +, 1101 for −, and 1111 to indicate the value is unsigned (see Table 2.5). Using packed decimal format, +146 would be stored as 00010100 01101100. Padding would still be required for an even number of digits. Note that if a number has a decimal point (as with monetary values), this is not stored in the BCD representation of the number and must be retained by the application program.

Another variation of BCD is **zoned decimal format**. Zoned decimal representation stores a decimal digit in the low order nibble of each byte, which is exactly the same as unpacked decimal format. However, instead of padding the high-order nibbles with zeros, a specific pattern is used. There are two choices for the high-order nibble, called the numeric **zone**. **EBCDIC zoned decimal format** requires the zone to be all ones (hexadecimal F). **ASCII zoned decimal format** requires the zone to be 0011 (hexadecimal 3). (See the next two sections for detailed explanations of EBCDIC and ASCII.) Both formats allow for signed numbers (using the sign digits found in Table 2.5) and typically expect the sign to be located in the high-order nibble of the least significant byte (although the sign could be a completely separate byte). For example, +146 in EBCDIC zoned decimal format is 11110001 11110100 11000110 (note that the high-order nibble of the last byte is the sign). In ASCII zoned decimal format, +146 is 00110001 00110100 11000110.

Note from Table 2.5 that six of the possible binary values are not used—1010 through 1111. Although it may appear that nearly 40% of our values are going to waste, we are gaining a considerable advantage in accuracy. For example, the number 0.3 is a repeating decimal when stored in binary. Truncated to an 8-bit fraction, it converts back to 0.296875, giving us an error of approximately 1.05%. In EBCDIC zoned decimal BCD, the number is stored directly as 1111 0011 (we are assuming the decimal point is implied by the data format), giving no error at all.

EXAMPLE 2.28 Represent −1265 using packed BCD and EBCDIC zoned decimal.

Digit	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
Zones	
1111	Unsigned
1100	Positive
1101	Negative

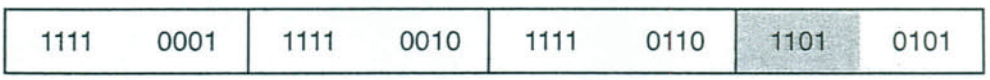
TABLE 2.5 Binary-Coded Decimal

The 4-bit BCD representation for 1265 is:

0001 0010 0110 0101

Adding the sign after the low-order digit and padding the high order bit with 0000, we have:

The EBCDIC zoned decimal representation requires 4 bytes:



The sign bit is shaded in both representations.

2.6.2 EBCDIC

Before the development of the IBM System/360, IBM had used a 6-bit variation of BCD for representing characters and numbers. This code was severely limited in how it could represent and manipulate data; in fact, lowercase letters were not part of its repertoire. The designers of the System/360 needed more information processing capability as well as a uniform manner in which to store both numbers and data. In order to maintain compatibility with earlier computers and peripheral equipment, the IBM engineers decided that it would be best to simply expand BCD from 6 bits to 8 bits. Accordingly, this new code was called **Extended Binary Coded Decimal Interchange Code (EBCDIC)**. IBM continues to use EBCDIC in IBM mainframe and midrange computer systems. The EBCDIC code is shown in Table 2.6 in

Zone \ Digit	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0000	NUL	SOH	STX	ETX	PF	HT	LC	DEL		RLF	SMM	VT	FF	CR	SO	SI
0001	DLE	DC1	DC2	TM	RES	NL	BS	IL	CAN	EM	CC	CU1	IFS	IGS	IRS	IUS
0010	DS	SOS	FS		BYP	LF	ETB	ESC			SM	CU2		ENQ	ACK	BEL
0011			SYN		PN	RS	UC	EOT				CU3	DC4	NAK		SUB
0100	SP										[	.	<	(	+	!
0101	&										]	$	*	)	;	^
0110	–	/									|	,	%	_	>	?
0111										'	:	#	@	'	=	"
1000		a	b	c	d	e	f	g	h	i						
1001		j	k	l	m	n	o	p	q	r						
1010		~	s	t	u	v	w	x	y	z						
1011																
1100	{	A	B	C	D	E	F	G	H	I						
1101	}	J	K	L	M	N	O	P	Q	R						
1110	\		S	T	U	V	W	X	Y	Z						
1111	0	1	2	3	4	5	6	7	8	9						

Abbreviations

NUL	Null
SOH	Start of heading
STX	Start of text
ETX	End of text
PF	Punch off
HT	Horizontal tab
LC	Lowercase
DEL	Delete
RLF	Reverse linefeed
SMM	Start manual message
VT	Vertical tab
FF	Form Feed
CR	Carriage return
SO	Shift out
SI	Shift in
DLE	Data link escape
DC1	Device control 1
DC2	Device control 2
TM	Tape mark
RES	Restore
NL	New line
BS	Backspace
IL	Idle
CAN	Cancel
EM	End of medium
CC	Cursor Control
CU1	Customer use 1
IFS	Interchange file separator
IGS	Interchange group separator
IRS	Interchange record separator
IUS	Interchange unit separator
DS	Digit select
SOS	Start of significance
FS	Field separator
BYP	Bypass
LF	Line feed
ETB	End of transmission block
ESC	Escape
SM	Set mode
CU2	Customer use 2
ENQ	Enquiry
ACK	Acknowledge
BEL	Ring the bell (beep)
SYN	Synchronous idle
PN	Punch on
RS	Record separator
UC	Uppercase
EOT	End of transmission
CU3	Customer use 3
DC4	Device control 4
NAK	Negative acknowledgement
SUB	Substitute
SP	Space

TABLE 2.6 The EBCDIC Code (Values Given in Binary Zone-Digit Format)

zone-digit form. Characters are represented by appending digit bits to zone bits. For example, the character *a* is 1000 0001 and the digit *3* is 1111 0011 in EBCDIC. Note the only difference between upper- and lowercase characters is in bit position 2, making a translation from upper- to lowercase (or vice versa) a simple matter of flipping one bit. Zone bits also make it easier for a programmer to test the validity of input data.

2.6.3 ASCII

While IBM was busy building its iconoclastic System/360, other equipment makers were trying to devise better ways for transmitting data between systems. The **American Standard Code for Information Interchange (ASCII)** is one outcome of these efforts. ASCII is a direct descendant of the coding schemes used for decades by teletype (telex) devices. These devices used a 5-bit (Murray) code that was derived from the Baudot code, which was invented in the 1880s. By the early 1960s, the limitations of the 5-bit codes were becoming apparent. The International Organization for Standardization devised a 7-bit coding scheme that it called International Alphabet Number 5. In 1967, a derivative of this alphabet became the official standard that we now call ASCII.

As you can see in Table 2.7, ASCII defines codes for 32 control characters, 10 digits, 52 letters (upper- and lowercase), 32 special characters (such as $ and #), and the space character. The high-order (eighth) bit was intended to be used for parity.

Parity is the most basic of all error detection schemes. It is easy to implement in simple devices like teletypes. A parity bit is turned "on" or "off" depending on whether the sum of the other bits in the byte is even or odd. For example, if we decide to use even parity and we are sending an ASCII *A*, the lower 7 bits are 100 0001. Because the sum of the bits is even, the parity bit would be set to off and we would transmit 0100 0001. Similarly, if we transmit an ASCII *C*, 100 0011, the parity bit would be set to on before we sent the 8-bit byte, 1100 0011. Parity can be used to detect only single-bit errors. We will discuss more sophisticated error detection methods in Section 2.7.

To allow compatibility with telecommunications equipment, computer manufacturers gravitated toward the ASCII code. As computer hardware became more reliable, however, the need for a parity bit began to fade. In the early 1980s, microcomputer and microcomputer-peripheral makers began to use the parity bit to provide an "extended" character set for values between 128_{10} and 255_{10}.

Depending on the manufacturer, the higher-valued characters could be anything from mathematical symbols to characters that form the sides of boxes to foreign-language characters such as ñ. Unfortunately, no amount of clever tricks can make ASCII a truly international interchange code.

0	NUL	16	DLE	32		48	0	64	@	80	P	96	`	112	p
1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
4	EOT	20	DC4	36	$	52	4	68	D	84	T	100	d	116	t
5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
7	BEL	23	ETB	39	'	55	7	71	G	87	W	103	g	119	w
8	BS	24	CAN	40	(	56	8	72	H	88	X	104	h	120	x
9	HT	25	EM	41	)	57	9	73	I	89	Y	105	i	121	y
10	LF	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
11	VT	27	ESC	43	+	59	;	75	K	91	[	107	k	123	{
12	FF	28	FS	44	,	60	<	76	L	92	\	108	l	124	|
13	CR	29	GS	45	-	61	=	77	M	93	]	109	m	125	}
14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	~
15	SI	31	US	47	/	63	?	79	O	95	_	111	o	127	DEL

Abbreviations

NUL	Null
SOH	Start of heading
STX	Start of text
ETX	End of text
EOT	End of transmission
ENQ	Enquiry
ACK	Acknowledge
BEL	Bell (beep)
BS	Backspace
HT	Horizontal tab
LF	Line feed, new line
VT	Vertical tab
FF	Form feed, new page
CR	Carriage return
SO	Shift out
SI	Shift in
DLE	Data link escape
DC1	Device control 1
DC2	Device control 2
DC3	Device control 3
DC4	Device control 4
NAK	Negative acknowledge
SYN	Synchronous idle
ETB	End of transmission block
CAN	Cancel
EM	End of medium
SUB	Substitute
ESC	Escape
FS	File separator
GS	Group separator
RS	Record separator
US	Unit separator
DEL	Delete/Idle

TABLE 2.7 The ASCII Code (Values Given in Decimal)

2.6.4 Unicode

Both EBCDIC and ASCII were built around the Latin alphabet. As such, they are restricted in their abilities to provide data representation for the non-Latin alphabets used by the majority of the world's population. As all countries began using computers, each was devising codes that would most effectively represent their native languages. None of these were necessarily compatible with any others, placing yet another barrier in the way of the emerging global economy.

In 1991, before things got too far out of hand, a consortium of industry and public leaders was formed to establish a new international information exchange code called Unicode. This group is appropriately called the Unicode Consortium.

Unicode is a 16-bit alphabet that is downward compatible with ASCII and the Latin-1 character set. It is conformant with the ISO/IEC 10646-1 international alphabet. Because the base coding of Unicode is 16 bits, it has the capacity to encode the majority of characters used in every language of the world. If this weren't enough, Unicode also defines an extension mechanism that will allow for the coding of an additional million characters. This is sufficient to provide codes for every written language in the history of civilization.

The Unicode codespace consists of five parts, as shown in Table 2.8. A full Unicode-compliant system will also allow formation of composite characters from the individual codes, such as the combination of ´ and A to form Á. The algorithms used for these composite characters, as well as the Unicode extensions, can be found in the references at the end of this chapter.

Character Types	Character Set Description	Number of Characters	Hexadecimal Values
Alphabets	Latin, Cyrillic, Greek, etc.	8192	0000 to 1FFF
Symbols	Dingbats, Mathematical, etc.	4096	2000 to 2FFF
CJK	Chinese, Japanese, and Korean phonetic symbols and punctuation	4096	3000 to 3FFF
Han	Unified Chinese, Japanese, and Korean	40,960	4000 to DFFF
	Expansion or spillover from Han	4096	E000 to EFFF
User defined		4095	F000 to FFFE

TABLE 2.8 Unicode Codespace