Computers and Numbers, and Such

1 Bits, Nibbles, Bytes, *and other things too fierce to mention*

A *bit* is a single **b**(inary) (dig)**it**. There are two, and only two, possible values for a bit: 0 or 1.

A *byte* is a collection of eight[1] adjacent bits. A *byte* is the basic unit of storage on most modern digital computers: *bytes* are manipulated, stored, and transferred – not individual *bits*. The possible values for a byte can be exhaustively enumerated:

Table 1: Possible values for a *byte*.

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1
0	0	0	0	0	0	1	0
0	0	0	0	0	0	1	1
⋮							
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	1

There are $2^8 = 256$ possible values for a byte.

Remembering our lessons from third grade we put *place values* on the individual bits in a byte. In this case, however, the place values are powers of 2 and not powers of 10:

Table 2: Places values for a *byte*.

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

Thus, the byte 00101101 is interpreted as

$$\begin{aligned} & 0 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ & = 32 + 8 + 4 + 1 \\ & = 45 \end{aligned} \tag{1}$$

Using this numbering scheme, the 256 possible states for a byte are indexed[2] from 0 to 255.

[1] Historically, the number of *bits* in a *byte* has been somewhat ambiguous. Over the last twenty-or-so years, however, the usage has stabilized and, for the most part, settled on eight bits per byte.

[2] Note that the largest possible value for a byte is 11111111 which equal $2^8 - 1 = 255$ and not 256. Yes, there are 256 possible values for a byte, but we start the counting at 0 and not at 1, so the largest is 255.

A *nibble* is a collection of four bits, or half a byte. Equivalently, there are two *nibbles* per byte. This naming, i.e. a *nibble*, is far too cute for my taste, and its usage is relatively uncommon in popular literature. Nonetheless, a collection of four bits is a convenient piece because it highlights the relationship between bits, bytes, and *hexadecimal*.

A hexadecimal digit takes values from 0 to 15, and are commonly written as: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. One hexadecimal digit is equivalent to 4 bits, i.e. a nibble. As such, a byte like 00101101 is more compactly written in hexadecimal as 2D. This follows from the observations that 0010 in binary equals 2 in hexadecimal, and 1101 in binary equals D in hexadecimal. In summary,

$$00101101_{\mathtt{b}} = 45_{\mathtt{d}} = 2\mathsf{D}_{\mathtt{h}} \tag{2}$$

where the subscripts $\mathtt{b}$, $\mathtt{d}$, and $\mathtt{h}$ represent "binary" (base 2), "decimal" (base 10), and "hexadecimal" (base 16). In hexadecimal[3], a byte may take values from $00_{\mathtt{h}}$ to $\mathsf{FF}_{\mathtt{h}}$. To simplify, we will leave the subscript off when the base is clear from context.

2 Byte Multiples: *too much time on their hands?*

You have all heard of *kilobytes*, *megabytes*, *gigabytes*, and maybe even *terabytes*. From your training in SI (*Le Systeme International d'Unites*[4]) you know the definitions given in Table 3.

Table 3: Some well-known standard SI prefixes.

kilo	$1,000^1$
mega	$1,000^2$
giga	$1,000^3$
tera	$1,000^4$

Due to the inherent binary nature of digital computers, the meanings of these standard prefixes were warped when speaking of computers to mean something slightly different, as shown in Table 4.

Table 4: Geek-speak warping of some standard SI prefixes.

kilo	$2^{10} = 1,024^1$
mega	$2^{20} = 1,024^2$
giga	$2^{30} = 1,024^3$
tera	$2^{40} = 1,024^4$

[3]A common alternate notation used to identify a hexadecimal number is a prefixed "0x". For example, 0x2D is the same as $2\mathsf{D}_{\mathtt{h}}$. The alternate notation has the advantage of not needing to use a subscript.

[4]See `http://en.wikipedia.org/wiki/SI`.

This warping of meanings really bothered some people. In 1998, the *International Bureau of Weights and Measures*[5] formally complained to the international community by publishing a brochere. In 2002, the *Institute of Electrical and Electronics Engineers* convened a formal panel and produced a formal standard: IEEE 1541-2002. In 2008 the *International Organization for Standardization* produced an international standard: ISO/IEC IEC 80000-13:2008.

The outcome is that we "should no longer use the prefixes kilo, mega, giga, and tera when speaking of powers of 2." As an alternative, the *International Organization for Standardization* created a suite of new prefixes for powers of 2, as shown in Table 5.

Table 5: The new Geek-speak ISO prefixes.

kibi	$2^{10} = 1,024^1$
mebi	$2^{20} = 1,024^2$
gibi	$2^{30} = 1,024^3$
tebi	$2^{40} = 1,024^4$
pebi	$2^{50} = 1,024^5$
exbi	$2^{60} = 1,024^6$
zebi	$2^{70} = 1,024^7$
yobi	$2^{80} = 1,024^8$

You can be the first on your block to speak of a "500 mebi-byte memory stick", or some such.

3 Characters Welcome

The *American Standard Code for Information Interchange* was first published in 1963, and subsequently modified in 1986. This is a simple substitution-based character encoding scheme for the American English alphabet and friends. The uppercase letters, lowercase letters, numerals, standard punctuation, and a handful of symbols, along with 33 non-printing, mostly obsolete, characters make up the 128 characters in the *ASCII character set*. See Figure 1.

When most digital computers store and manipulate text the rule is: *one byte per character*. The ASCII character set has only 128 characters while one byte can take on 256 unique values. By convention the ASCII characters are associated with values 0 to 127. The higher values, from 128 to 255, are non-uniquely defined by software and hardware manufacturers and various other interested parties. For example, the values from 128 to 255 can hold accented characters used in non-english, European texts.

[5] "The BIPM helps to ensure uniformity of SI weights and measures around the world. It does this with the authority of the Metre Convention, a diplomatic treaty between fifty-one nations (as of August 2008), and it operates through a series of Consultative Committees, whose members are the national metrology laboratories of the Member States of the Convention, and through its own laboratory work." `http://en.wikipedia.org/wiki/International_Bureau_of_Weights_and_Measures`

Figure 1: A table showing the 128 ASCII character set. This table was extracted from `www.ascii.ws`.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

While the ASCII character set permeates almost all character encoding, its limitations were recognized from its inception. Today, a more general character encoding scheme is commonly used. "The Universal Character Set (UCS), defined by the ISO/IEC 10646 International Standard, is a standard set of characters upon which many character encodings are based. The UCS contains nearly a hundred thousand abstract characters, each identified by an unambiguous name and an integer number called its code point."[6].

The variant of the Universal Character Set called **UTF-8**[7] is the most common character encoding in storage and transmission of text: e.g. emails, web pages, and news feeds. Interestingly, the 128 ASCII characters are a special subset of UTF-8.

4 Storing Numbers: *on a cold and frosty morning*

Consider the number[8]: 305,747,332. We can use the ASCII, or ASCII-like, schemes previously described to store this number in the computer and on our portable storage devices (e.g. on a memory stick or your MP3 player):

Table 6: ASCII encoding of the number 305,747,332.

3	33_h
0	30_h
5	35_h
,	$2C_h$
7	37_h
4	34_h
7	37_h
,	$2C_h$
3	33_h
3	33_h
2	32_h

Thus, to store this number we would need to allocate 11 bytes of memory. You may argue that the computer does not need to store the commas, since these are included merely to help human eyes and brains read the numbers when printed on a page or screen. That is, the number could be stored with out the commas, and the commas can be added only when the number is printed. Ok. You won the argument.

Thus, to store this number we would need to allocate 9 bytes of memory (leaving off the commas). This is called *ASCII encoding* of numbers. This is what is done when we create a file that we can open and read in `Notepad`, for example. This strikes me as a bit incestual: we use numbers to encode the characters, and then characters to encode the

[6] `http://en.wikipedia.org/wiki/Universal_Character_Set`

[7] 8-bit UCS/Unicode Transformation Format.

[8] This is the estimated population of the United States at 15:20 GMT (9:20 am in Minneapolis) on Wednesday, 5 February 2009, according to `http://www.census.gov/population/www/popclockus.html`.

numbers. Furthermore, even though this form of encoding is quite convenient because it emulates what we do with pen and paper, it is remarkably inefficient. As discussed below, we can encode the number 305, 747, 332 in a meager four bytes.

5 Integers: *all else is the work of man*

At first glance, the storage of integers[9] with bytes is obvious. The values from 0 to 255 can be stored in a byte using the place-value interpretation previously discussed. Unfortunately, 0 to 255 is a rather limited range.

5.1 Maybe we want to store negative integers?

To store *signed integers* we can take the left-most bit and designate it as the *sign bit*. If the left-most bit is a 0 the number is positive, if the left-most bit is a 1 then the number is negative. Of course, taking up one of the the 8 bits in the byte holding sign information leaves only 7 bit to hold the actual value. In this scheme we can hold values from -127 to $+127$. This scheme is called *sign and magnitude*.

There is a problem with this clear but naive scheme: $00000000_{\mathtt{b}}$ and $10000000_{\mathtt{b}}$ are both zero. The first is *positive zero* and the second is apparently *negative zero*. This is somewhat silly, and a waste of space – we are using up two states to hold the same number. Computer scientists have come up with a number of alternate schemes[10] to hold signed integers: two of these schemes are called *ones' complement* and *twos' complement*[11]. Using either of these schemes a single byte can hold values from -128 to $+127$.

5.2 Maybe we want to store larger integers?

To store larger integer we simply need to stack bytes together. Two bytes contain 16 bits, so there $2^{16} = 65,535$ unique possible states. Using the *ones' complement* or *twos' complement* schemes, two bytes can hold signed integers from $-32,768$ to $+32,767$. Similarly, four bytes can hold signed integers from $-2,147,483,648$ to $+2,147,483,647$. This approach can, and is, extended to even large integers, but most PC software use 4-byte integers as their standard, and limit the range of integers accordingly. This is true for both `Excel` and `MatLab`.

There are some issues about how the bytes are ordered in computer memory when storing multi-byte integers[12], but that is beyond the scope of this class.

[9] Leopold Kronecker (1823-1891) is quoted, after translation, as saying: "God created the integers, all else is the work of man." I am not certain what he meant, but it sounds cool. Apparently Stephen Hawking agrees with me, since he titled one of his recent publications *God Created the Integers: The Mathematical Breakthroughs That Changed History*.

[10] See `http://en.wikipedia.org/wiki/Signed_number_representations`.

[11] `http://en.wikipedia.org/wiki/Twos_complement`

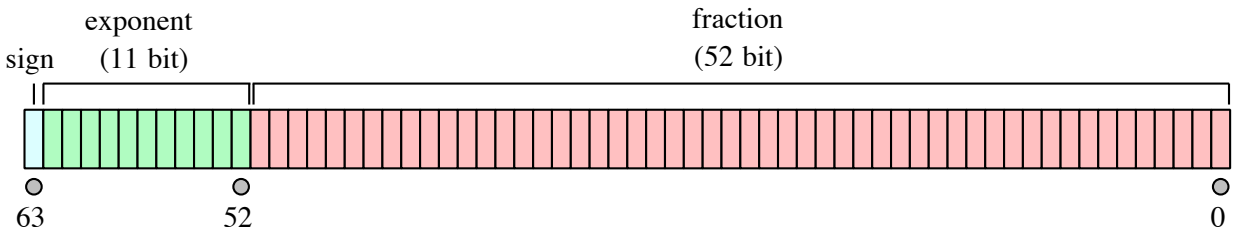
[12] If you have read (the *Wishbone* version of this story did not get into this chapter, sorry) Jonathan Swift's book *Gulliver's Travels* you may enjoy some of the humor surrounding the debate. In computer science the question is whether to store the high-order byte first, or the low-order byte first. In *Gulliver's Travels* the question is which end of a soft-boiled egg should be broken first: the "big end" or the "little end". Computer scientists have demonstrated their humility, and humor, by embracing Swift's vocabulary for their debate: *big-endian* versus *little-endian*. See `http://en.wikipedia.org/wiki/Big_endian` and `http://en.wikipedia.org/wiki/Little-endian`.

6 Floating Point Numbers: *don't worry, we'll all float on, Ok?*

Numbers in a computer, that are not explicitly integers, are stored as using a binary version of *scientific notation*. This is called a *floating point* representation[13]. The standard representation used in most civil engineering applications is called, for historical reasons, a *double precision floating point number*[14]. The format and manipulation is formally defined by the IEEE 754-2008 standard[15].

A double precision floating point number uses 8 consecutive bytes; that is, 64 bits. The representation has a *sign bit*, 11 bits for the *exponent*, and 52 bits for the *mantissa* (i.e. the fraction part), as shown in Figure 2.

Figure 2: The IEEE 754 double precision binary floating-point format which is used in most civil engineering applications. This figure was extracted from `en.wikipedia.org/wiki/Double_precision`.



There are a couple of oddities in this storage format. First the exponent needs both a sign and a magnitude: as engineers we work with both large and small numbers. Rather than allocate a sign bit for the exponent, the exponent is stored in a *biassed format*. To get the real exponent one must take the integer value stored in the 11 bits allocated for the exponent and subtract $3\mathtt{FF}_\mathtt{h} = 1023_\mathtt{d}$.

The second oddity comes from the binary representation. When we write a number in scientific notation, like Avogadro's number, we would write 6.0221415×10^{23}. We would not usually[16] write $0.060221415 \times 10^{25}$ or 6022.1415×10^{20}. We move the decimal point in the mantissa just to the right of the first non-zero number and adjust the exponent accordingly. The same is true with the double precision floating point format: we move the binary radix[17] point just to the right of the first non-zero number and adjust the exponent accordingly. However, since the only possible values for a binary digit are 0 and 1 we know

[13] See `http://en.wikipedia.org/wiki/Floating_point`.

[14] `http://en.wikipedia.org/wiki/Double_precision`.

[15] `http://en.wikipedia.org/wiki/IEEE_floating-point_standard`.

[16] If we did write it in one of these aberrant formats, our 9th Grade science teacher would not have given us credit for an otherwise perfectly correct answer. The terminology used to differentiate between these formats is *normalized* (good), and *unnormalized* (bad).

[17] A *decimal point* is the point used to separate the integer part from the fractional part of a base-10 number. *Decimal* means base-10, so it would be silly to call the equivalent point in a base-2 or base-16 representation a *decimal point*. The generic name for the point separating the integer part from the fractional part is *radix point*.

that the digit just to the left of the radix point must be a 1 – since it is not a 0. Since we know that it is a 1 we do not need to store it in the computer – it is implied[18].

Putting these pieces together we can diagram the value of an IEEE 754 double precision floating point number as[19]

$$\boxed{(-1)^{\text{sign}} \times 2^{\text{exponent}-3\text{FF}_\text{h}} \times 1.\text{mantissa}} \tag{3}$$

The complete bit representation for the number 1 in this format is

$$00111111\ 11110000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000 \tag{4}$$

Lets parse (4) and verify my claim.

sign bit	0
exponent	$011\ 1111\ 1111 = 3\text{FF}_\text{h}$
mantissa	0000 00000000 00000000 00000000 00000000 00000000 00000000

Substituting these pieces of (4) into (3) we have

$$(-1)^0 \times 2^{3\text{FF}_\text{h}-3\text{FF}_\text{h}} \times 1.00000000000000000000000000000000000000000000000000 = 1 \tag{5}$$

6.1 Limits and Precision

The largest value that can be represented using the IEEE 754 double precision floating point number representation is

$$0111111111101111111111111111111111111111111111111111111111111111 \approx 1.7976931348623157\times10^{308} \tag{6}$$

The total precision of the representation is 53 binary digits (52 given plus the 1 implied). Translating this into decimal we get $\log_{10}(2^{53}) \approx 15.93$, or just shy of 16 significant decimal digits.

$$\boxed{\text{Computers maintain approximately 16 significant decimal digits.}} \tag{7}$$

6.2 Holes

Were we to discuss the IEEE 754 double precision floating point number representation, outlined above, with Stanley Yelnats[20] he would probably point out that there are holes in our number system. Consider the representation for the number 1 as given in (4). The next

[18] The use of the *implied* leading 1-bit saves one bit of storage, or more accurately, it increases our precision by one bit, given the fixed allocation of 52 bits for the mantissa.

[19] See `http://en.wikipedia.org/wiki/Double_precision`.

[20] After getting out of juvi, Stanley went to college and earned a degree in Civil Engineering, specializing in geomechanics and minoring in mathematics.

highest number that can be represented in this scheme is found by changing the right-most bit from a 0 to a 1:

$$00111111\ 11110000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000 \quad (8)$$
$$00111111\ 11110000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001 \quad (9)$$

The decimal number represented by (8) is 1, the decimal number represented by (9) is

$$1.0000\ 0000\ 0000\ 0002\ 220446049250313080847263336181640625 \quad (10)$$

The next highest number above (9) is

$$00111111\ 11110000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000010 \quad (11)$$

which has the decimal representation

$$1.0000\ 0000\ 0000\ 0004\ 440892098500626161694526672363281250 \quad (12)$$

In summary, we **CANNOT** represent a number between

$$1.0000\ 0000\ 0000\ 0002\ 220446049250313080847263336181640625 \quad (13)$$

and

$$1.0000\ 0000\ 0000\ 0004\ 440892098500626161694526672363281250 \quad (14)$$

using the IEEE 754 double precision floating point number representation. These holes exist between every adjacent pair of representable numbers. Furthermore, as the numbers get larger and larger the absolute sizes of the holes get larger. This is important.

6.3 Machine Epsilon

The machine *epsilon* is defined as the difference between 1 and the smallest representable number greater than one. For the IEEE 754 double precision floating point number representation the machine epsilon is

$$\boxed{\epsilon = 2.220446049250313080847263336181640625 \times 10^{-16}} \quad (15)$$

This is an objective characterization of the machine precision[21] in that is is the upper bound on the relative error due to roundoff during an addition or subtraction.

[21] See `http://www.validlab.com/goldberg/paper.pdf`. This is a remarkably complete and well-written discussion of the topic. It is free. I recommend this document to you. However, most of the material in this document is beyond the scope of CE 3101.