Text Representation

Characters are stored in the computer as unique binary numbers. A character is a symbol on the keyboard. **All** the characters that a computer or peripheral recognises is called a **character set**. To ensure that different types of computers communicated it was important that they used the same codes. The **ASCII** (American Standard Code for Information Interchange) system was adopted so that all computers would use the same codes to represent the same characters for American English. ASCII uses the first 7 bits (0 to 127) of the one byte representation. The "8th bit" is used by manufacturers to set up their own distinct characters, this is known as **extended ASCII**.

For example, the letter A is stored as 65 (or **0**1000001 in binary) and the letter Z is stored as 90 (**0**1011010). Other keyboard characters such as the TAB key (**0**0001001) and RETURN key (**0**0001101) are also represented in ASCII. The blank characters can again be used by computer manufacturers for other commands such as turning the printer on and off.

Although the table lists the characters with a unique decimal number, obviously the computer would store them as binary e.g. 65 represents A, the computer would use **0**1000001.

ASCII TABLE FOR 128 STANDARD CHARACTERS

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

Unicode is another method of representing text. It was developed to enable the representation of all languages. Before Unicode there were huge compatibility problems with text as different codes were used for different languages, for example:

American English	ASCII
Japanese	JIS
Indian	ISCII

Unicode uses 16 bits to represent each character. So it can represent 65 536 different characters. Hence Unicode can provide adequate coding for electronic communications in all languages of the world.

Advantages over ASCII

1. Unicode uses 16 bits compared with ASCII's 7 bits. This means that more characters can be represented:

   $2^7 = 128$ $2^{16} = 65\,536$

2. Unicode can therefore represent other languages such as Arabic, Chinese etc. not just English.

Both ASCII and Unicode are **character sets** for representing text, however due to Unicode's wider range of characters it may soon overtake ASCII as the **data standard** for text representation.

Positive Numbers

Decimal Numbers

We use the decimal (base 10) number system for representing numerical data. This system involves grouping quantities into bundles of 10 and requires 10 symbols to represent the quantities 0 to 9 e.g. 1246.12 would be represented as:

thousands	hundreds	tens	units	tenths	hundredths
1	2	4	6	1	2

The number that forms the basis of any system is called the **base**. Our system is so familiar that it is sometimes difficult to imagine any other, but numbers can be represented in any base. The only reason that 10 has emerged as our base is that early civilisation counted on their fingers.

Numbers can be represented in any base ***b***. A numbering system based on ***b*** would have ***b*** symbols to represent quantities from ***0*** to ***b-1***. A place value system would operate from right to left as follows:

	bxbxbxb	**bxbxb**	**b x b**	**b**	**units**
b^n	**b^4**	**b^3**	**b^2**	**b^1**	**b^0**

Decimal numbers can be represented in base ***10***. The decimal system therefore has ***10*** symbols to represent quantities from ***0*** to ***9***. The place value system for decimal would operate from right to left as follows:

10^n	**10^4**	**10^3**	**10^2**	**10^1**	**10^0**
	ten thousands	**thousands**	**hundreds**	**tens**	**units**

Binary Numbers

The lowest number base that can be used to represent numbers is base 2 (called **binary**). This system requires only 2 symbols, 0 and 1.

Binary numbers can be represented in base ***2***. The binary system therefore has ***2*** symbols to represent quantities from ***0*** to ***1***. The place value system for binary would operate from right to left as follows:

2^n	**…....**	**2^{31}**	**…....**	**2^7**	**2^6**	**2^5**	**2^4**	**2^3**	**2^2**	**2^1**	**2^0**
2x2 n times	**…....**	**2147483648**	**…....**	**128**	**64**	**32**	**16**	**8**	**4**	**2**	**1**

To get the place value:

2^0 = 1 2^1= 2 **2^2= 2 x 2 = 4** **2^3= 2 x 2 x 2 = 8** **2^4= 2 x 2 x 2 x 2 = 16**

2^5= 2 x 2 x 2 x 2 x 2 = 32 2^6= 2 x 2 x 2 x 2 x 2 x 2 = 64 and so on

2^{31}=2x2x2x2x2x2x2x2x2x2x2x2x2x2x2x2x2x2x2x2x2x2x2x2x2x2x2x2x2x2x2x2 = 2147483648

Binary numbers can be categorized into units:

0 or 1 = 1 bit 8 bits = 1 byte 1024 bytes = 1 Kilobyte (Kb)

1024 Kilobytes = 1 Megabyte (Mb) 1024 Megabytes = 1 Gigabyte (Gb)

1024 Gigabytes = 1 Terabyte (Tb)

Produced by S Lambert, R Simpson and HSDU for The City of Edinburgh, 2004

Positive Numbers

Converting Bits To Bytes etc

These binary units are used to represent storage and memory requirements within the computer. As some file sizes etc. can be very large it is necessary to convert these into appropriate units. For example using measurement of length, 3000000 cm wouldn't be used. This length would be represented as 30 km.

When converting from the *smaller* unit to the *larger* one, division is required.

Examples:

108 bits = 108 ÷ 8 = 13.5 bytes
10987 bytes = 10987 ÷ 1024 = 10.73 Kb
3245676 bytes = 3245676 ÷ 1024 = 3169.61 Kb = 3169.61 ÷ 1024 = 3.1 Mb

When converting from the *larger* unit to the *smaller* one, multiplication is required.

Examples:

4 Terabytes = 4 x 1024 = 4096 Gb
0.03 Megabytes = 0.03 x 1024 = 30.72 Kb
0.0006 Kilobytes = 0.0006 x 1024 = 0.6144 bytes = 0.6144 x 8 = 4.9152 bits

Why Do Computers Use Binary?

Decimal - They could use base 10. For example the symbols 0 to 9 could be represented by 0 volts to 9 volts. However, there are disadvantages in doing this:

- The rules for adding, subtracting, multiplying and dividing each symbol by each of the others, would need to be built into the computer, which would take up a great deal of the processor's instruction set.
- Also any slight degradation in the storing or transferring of data may cause a drop in voltage and thus a change in the data.

Binary - There are corresponding advantages in using the binary system for computers.

- Since there are only two symbols, the rules for add, subtract multiply and divide are very simple and can be built into simple electronic circuits.
- Also, the fact that a unit of information has only two states (0 and 1), it can be represented by 'nothing' and 'something'. Electronically that can mean *0 volts* to represent **0** and *any voltage* (normally 2 - 5 volts) to represent **1**. This means that any slight degradation in the signal level does not change the information.
- All components used in a computer system and all data storage devices have only two states. For example a switch is off or on, transistors conduct or don't conduct, and a signal is a pulse of electricity or no pulse. Binary using the numbers 1 and 0 can be represented by all these states. This ***two-state*** system has parallel advantages when it comes to storing data magnetically and optically: for example magnetic particles in one direction or the other; or a tiny pit or no pit on the surface of a laser read disk.

Produced by S Lambert, R Simpson and HSDU for The City of Edinburgh, 2004

Positive Numbers

Converting Binary Numbers To Decimal

When converting binary numbers to decimal, the place values are used. Here is an example of how to convert the binary number 10011010 (8 bits) to a decimal:

Write binary number below place values →

128	64	32	16	8	4	2	1
1	0	0	1	1	0	1	0

$= 128 + 16 + 8 + 2 = 154_{dec}$

Here is an example of how to convert the binary number 10011010000111001100001000000001 (32 bits) to a decimal:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	0	0	0	0	1	1	1	0	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	1

$= 2^{31} + 2^{28} + 2^{27} + 2^{25} + 2^{20} + 2^{19} + 2^{18} + 2^{15} + 2^{14} + 2^{9} + 2^{0}$

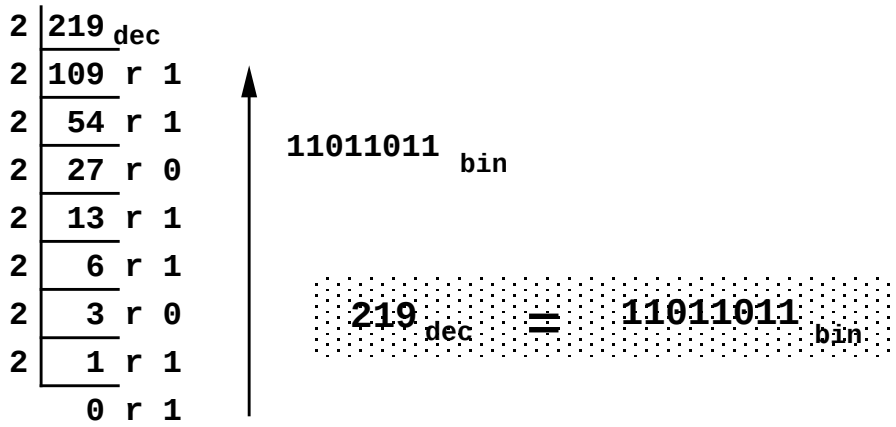
= 2147483648 + 268435456 + 134217728 + 33554432 + 1048576 + 524288 + 262144 + 32768 + 16384 + 512 + 1

$= 2585575937_{dec}$

Converting Decimal Numbers To Binary

To convert from decimal to binary two techniques can be used. One involves continually dividing by 2 (noting remainders), until there is zero left. Then read the remainders from bottom to top.

Example: Converting 219 to a binary number.



The other method involves using place values. If the place value is less than the number then place a **1** under the place value if not place a **0**. Take away the place value from the decimal number and then use the remainder in the same way. Keep going until there is no remainder left. This method is not suitable if the decimal number is very large (unless your arithmetic is excellent).

Example: Converting 1035 to a binary number.

1024	*512*	*256*	*128*	*64*	*32*	*16*	*8*	*4*	*2*	*1*
1	***0***	***0***	***0***	***0***	***0***	***0***	***1***	***0***	***1***	***1***

This is also a good way of checking that your number is correct by converting back again i.e.

1024 + 8 + 2 + 1 = 1035

Produced by S Lambert, R Simpson and HSDU for The City of Edinburgh, 2004

Negative Integers

Negative integers are usually represented by ***Two's Complement.***

The concepts behind this two's complement are that the set of integers should show symmetry about zero and that adding one to any number would produce the next (ignoring carry bits). For example:

BINARY	***INTEGER***
11111101	- 3
11111110	- 2
11111111	- 1
00000000	0
00000001	+ 1
00000010	+ 2
00000011	+3

The simplest way of obtaining the two's complement of a number is to:

1. change all ones to zeros and zeros to ones;
2. add one to this result.

For example to get two's complement of 3:

[start with +3]	00000011
[change 0s to 1s and 1s to 0s]	11111100
[add 1, result is -3]	11111101

To check that the conversion is correct the leftmost bit of the place values can be changed to a negative number instead of a positive one. For example an 8 bit number will have the leftmost bit –128 instead of 128. This bit should always be 1 to indicate a negative number so:

-128	64	32	16	8	4	2	1
1	1	1	1	1	1	0	1

$= \mathbf{-128} + 64 + 32 + 16 + 8 + 4 + 1 = -3_{\mathbf{dec}}$

Adding binary numbers can be tricky, for example:

1 + 1 = 10 1 + 1 + 1 = 11

So:

1	0	1	0	1	1	1	1	+
0	0	0	1	0	1	1	0	
1	1	0	0	0	1	0	1	

Important features of two's complement are:

- taking the two's complement of a number and then taking the two's complement of the result, gets back to the original number;
- arithmetic carried out on two's complement representations of integers produces correct results , for example try adding numbers like -5 and +7 should give +2;
- only 1 value for 0 is represented;
- adding the positive and negative of a number will always give 0.
-

Produced by S Lambert, R Simpson and HSDU for The City of Edinburgh, 2004

Real numbers

Real numbers are stored in a computer as floating point numbers. Floating point is like standard form or scientific notation. Any number can be represented in any number base in the form:

$$\mathbf{m \times b^e}$$

where **m** is called the **mantissa**, **b** is called the **base** and **e** is the **exponent**.

To convert 34006.8 to standard form involves moving the point so the number is between 1 and 10 and counting how many places you have moved the point. e.g.

$$34\,006.8 = 3.40068 \times 10^4$$

The 4 represents the fact that the point has moved 4 places to the left. The 10 represents the base. The 4 is called the exponent and 3.40068 is called the mantissa.

How The Computer Represents Real Numbers

The computer obviously uses binary and stores only the **mantissa** and **exponent** as the base is always 2. Usually the computer uses four bytes for storing the mantissa and one byte for the exponent this gives nine figure accuracy when converted to decimal and a range of $\mathbf{2^0}$ to $\mathbf{2^{255}}$ which is approximately $\mathbf{10^0}$ to $\mathbf{10^{76}}$.

The number of digits quoted in the mantissa indicates the *accuracy* of the number and the number of digits in the exponent is a measure of the *range of numbers* which can be stored. The number of total bits used to store both mantissa and exponent stays the same, so:

Increase mantissa, exponent must be decreased = increased accuracy, decreased range

Decrease mantissa, exponent must be increased = decreased accuracy, increased range

When converting binary numbers into floating point notation, the point is moved in front of the first 1.

Example one:

$$\mathbf{1101.101_{bin} = .1101101 \times 2^4 = .1101101 \times 2^{100}}$$

The mantissa is 11011010 if 1 byte is used and the exponent 0100 if half a byte is used.

Example two:

$$\mathbf{110100.110111001_{bin} = .110100110111001 \times 2^6 = .110100110111001 \times 2^{110}}$$

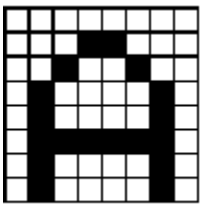
If 4 bytes is used to store the mantissa and 1 byte for the exponent the computer would store the mantissa as 11010011011101000000000000000000 and exponent as 00000110.

Note that both the mantissa and the exponent can be negative. A negative mantissa means a negative number, but a negative exponent simply means a number less than one (or more accurately between 1 and -1).

Produced by S Lambert, R Simpson and HSDU for The City of Edinburgh, 2004

Bit-Mapped Graphics

When drawing a graphic using a bit-mapped package, the computer stores this information as a two-dimensional array of pixels (if it is a black and white image then each pixel is represented by 1 bit in memory – hence bit-mapped). Once drawn, each individual pixel can be edited. Every pixel making up the image has to be stored. This can be wasteful if a page consists mainly of white space



is stored as

00000000
00011000
00100100
01000010
01000010
01111110
01000010
01000010

As each graphic is created by generating a series of pixels the main **disadvantages** are:

- the graphic cannot be resized without redrawing the graphic from scratch (pixelation occurs see below);

- the graphic will always stay at the resolution that it was created in - i.e. printing at 600dpi when the graphic was created at 300dpi. The graphic will not be printed out as 600 dpi. Hence bit-mapped is **resolution dependent**.

Storage Requirements

The storage requirements of bit mapped images is calculated using the resolution, area and bit depth (see below) of the graphic.

Storage (bits) = area (square inches) x resolution (bits) x bit depth (bits)

Often the resolution of the screen or printer resolution is given in dots per inch (dpi) or pixels per inch (ppi).

Produced by S Lambert, R Simpson and HSDU for The City of Edinburgh, 2004

Bit-Mapped Graphics

Working Out The Storage Requirements Of A Bit-Mapped Graphic

Example one:
A resolution of 600 dpi means that a square inch needs 600 x 600 bits of storage. If we had a black and white image which had an area of 2″ x 5″, its storage requirements can be calculated as:

2 x 5 x 600 x 600 x 1 = 3600000 bits = 8437.5 bytes = 8.24 Kilobytes

Example two:
An A4 page (10" x 8") at a resolution of 1200 dpi for black and white would require:

10 x 8 x 1200 x 1200 x 1 = 115200000 bits = 14400000 bytes = 14062.5 Kilobytes = 13.73 Megabytes

Colours And Shades Of Grey

Most images do not use just black and white. Usually graphics require lots of colours to represent images etc. With a black and white (or any two colours) image can be represented by one bit, 0 for one colour and 1 for the other. If a range of colours are available, each pixel needs more than one bit to represent its colour. This is known as the **bit depth** of the graphic. The number of bits needed indicates the number of colours available, for example:

colours (or shades of grey)	bits	
2	1	0 for white and 1 for black
4	2	00, 01, 10 and 11 are available
8	3	000, 001, 010, 100, 011, 101, 110, 111.
16	4	0000, 0001 0010 and so on.
.	.	.
.	.	.
16777216	24	24 bit depth is known as true colour
.	.	.

Colour Bit Depth And File Size

The storage requirements of bit mapped images can be very high when large areas are needed at high resolution and high bit depth.

So if the image described in example one (previous page) was available in 16 colours, then each pixel would require 4 bits to store that colour. Thus storage requirements would be:

2 x 5 x 600 x 600 x 4 = 14400000 bits = 1800000 bytes = 1757.81 Kb = 1.72 Mb

There is a considerable storage difference between using 2 colours compares with 16. In fact 8.24 Kb compared with 1.72 Mb. Think what would happen to the file size of the same graphic if the bit depth with 24 bits with a resolution of 1200 dpi The file size would be:

2 x 5 x 1200 x 1200 x 24 = 345600000 bits = 41.2 Mb

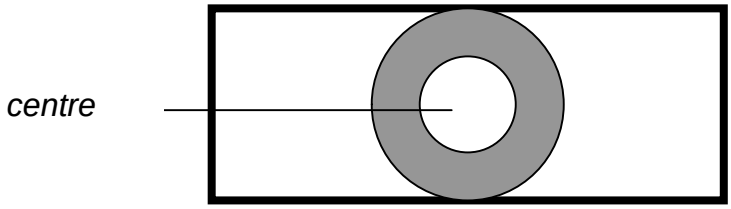
Data Compression
In order to compensate for the large file sizes that bit-mapped graphics create, **data compression** is used. This reduces the file size so that the file is not storage or memory intensive. There are a variety of techniques used to compress files. Some delete information which would not be vital to the graphic while others cut-down the number of colours available. Whatever technique is used the amount of memory and storagethat the graphic takes up is reduced, allowing faster transfer on the Internet or more files to be saved on disk.

Produced by S Lambert, R Simpson and HSDU for The City of Edinburgh, 2004

Vector Graphics

Vector graphics or object-orientated graphics is another method used to store images etc. Images are stored as a collection of objects. The computer stores information about each object by its **attributes**. These attributes are the description of how that object is drawn.

For example:



This image is made up of 4 objects:

2 circles
1 rectangle
text
1 line

For each of the four objects, their attributes would be stored.

For the rectangle it might be:

* start x and y position
* length, breadth and angle of rotation
* thickness and colour of the lines
* colour fill etc.

For one of the circles it might be:

* centre x and y position
* length of radius
* thickness and colour of the lines
* colour fill etc.

This means that any of the objects can be selected at a later time and altered by changing one of its attributes e.g. dragging to a new position etc. It is not possible to change the colour of any individual part of the rectangle though, but it is possible to change the colour of the lines forming the rectangle and the interior fill since it is an attribute. The only data stored is that concerning the actual objects drawn.

Produced by S Lambert, R Simpson and HSDU for The City of Edinburgh, 2004

Graphics Representation

Bit-Mapped Or Vector Graphics

Obviously there are different reasons for choosing which type of graphics to use. Most packages only allow bit-mapped or vector to be stored. So which one do you use?

Advantages of vector graphics:

- Take up less amounts of memory as only data about the actual drawing is stored;
- **Resolution independant**. If you print on a higher resolution printer then the graphic is printed at a higher quality;
- Easy to edit graphics i.e. alter size and shape, move, delete etc.

Disadvantages of vector graphics:

- Can't rub out bits or parts of objects.

Advantages of bit-mapped graphics:

- Very like manual tools such as paint brushes, spray cans etc;
- More accurate graphics can be produced as they can be edited to pixel level;
- Ability to edit individual pixels.

Disadvantages of bit-mapped graphics:

- Use large amounts of memory;
- Don't improve resolution if printed at higher dpi;
- Difficult to edit.

Produced by S Lambert, R Simpson and HSDU for Drummond Community High School Computing Department 2004