Data representation

Table of Contents

This is about how different kinds of data can be represented in digital systems.

Two-state devices

A *two-state device* is something which can only has two states - like a basic switch which is either on or off. In static RAM (SRAM) each two-state device is made of a set of transistors, and in dynamic RAM (DRAM) it is a capacitor, storing a charge (1) or not (0). Flash memory is similar to SRAM but is non-volatile - it retains state without power. On a magnetic disk drive, it is magnetisation of metal -north-south or south-north. And so on.

Two-state devices are used because they are very small, very cheap, and very reliable.

All digital systems are made of two-state devices. Study the section on digital logic.

So everything - *all code and all data* - has to be in binary form - made of 1s and 0s.

Data representation means finding ways of representing data in binary form.

Suppose we look in memory and see 1001 1101 1100 1001. Does this represent a number? Or some characters? Or a pixel in an image? There is no way to tell, except for the context - the situation. Everything is in binary. We need the context to know how to make sense of it - what it represents.

Number bases

There is a difference between a *number* and the way it is *represented*. The number 3 can be represented as III in Roman numerals, or 11 in base 2, or 3 in base 10. III and 11 and 3 are three different representations of the same number.

A *digit* is one symbol - like 3 or 7 or 9. A number is made of one or more digits - like 12.34.

Base 10

'Ordinary' numbers are written in base 10 (or decimal or denary). That means we use a set of digits, with different places having different *place values*, like this:

Place value	Thousands	Hundreds	Tens	Units
	10^3	10^2	10^1	10^0
Digits	3	2	4	6

So the number 3246 means 3 thousands, 2 hundreds, 4 tens and 6 units.

Notice the place values go up in powers of 10, and we have 10 digits to use - 0 1 2 3 4 5 6 7 8 and 9.

The pattern continues to the right of the 'decimal point', like this:

Place value	Tens	Units	Tenths	Hundredths
	10^1	10^0	10^{-1}	10^{-2}
Digits	7	2	. 1	6

So 72.16 is 7 tens, 2 units, 1 tenth and 6 hundredths.

Base 2

We can use the same idea in other bases. In base 2 the place values are powers of 2 - units, 2, 4 8 16 and so on. We only have the digits 0 and 1 to use:

Place value	Eights	Fours	Twos	Units
	2^3	2^2	2^1	2^0
Digits	1	1	0	1

So 1101 is 1 eight, 1 four, 0 twos and 1 unit = 8 + 4 + 1 = 13.

When we are writing in different number bases, we use a subscript to show which base it is in. So $1101_2 = 13_{10}$

Here is counting in base 2:

Base 2	Base 10
0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8
1001	9
1010	10

To change a number from base 2 to decimal, just add up the place values where there is a 1 - for example:
$10010_2 = 16 + 2 = 20_{10}$

To change decimal to binary, for small numbers, pick out powers of 2 to total the number. For example

$19_{10} = 16 + 3 = 16 + 2 + 1 = 10011_2$

For larger numbers, repeatedly divide by 2 and track the remainders. For example to change 328 into binary:

2 into 328 = 164 remainder 0

2 into 164 = 82 remainder 0

2 into 82 = 41 remainder 0

2 into 41 = 20 remainder 1

2 into 20 = 10 remainder 0

2 into 10 = 5 remainder 0

2 into 5 = 2 remainder 1

2 into 2 = 1 remainder 0

so reading the remainders back:

$328_{10} = 101001000_2$

In effect we are finding powers of 2 as before but more methodically. In practice it is faster and

To convert a decimal fraction to binary:

1. Multiply by 2

2. The whole number part is the next bit. Ignore it when you..

3. Repeat step 1, until you get to zero, or a repeating pattern.

Example: 0.75 - multiply by 2:

1.50 First bit is 1. Now use 0.50

1.00 Second bit is 1, Got to .00, so end

So $0.75_{10}=0.11_2$ (since 0.75 = 1/2 + 1/4)

Example 0.625

1.250 first bit 1, use .250

0.50 second bit 0

1.00 third bit 1, ended

So $0.625_{10}=0.101_2$

Example

0.1

0.2 first bit 0

0.4

0.8

1.6

1.2 (but we've had .2 before..)

0.4

0.8

1.6

1.2

0.4

0.8

1.6

1.2 ..

so 0.1 decimal = binary .000110011001100110011..

So 0.1 is an example to show that *some values have infinite binary fraction expansions* - just like 1/3 is 0.3333.. in base 10. We will see later that computer arithmetic with numbers other than integers has only limited accuracy.

Hexadecimal

We often use numbers in base 16, known as hexadecimal or hex.
Here the place values are powers of 16. For example:

Place value	4096's	256's	Sixteens	Units
	16^3	16^2	16^1	16^0
Digits	2	0	7	1

So 2071_{16} = 2 X 4096 + 7 X 16 + 1 X 1 = 8305_{10}

The big difference is that we need 16 different digits, and we only have 10, as 0 to 9. The problem is solved by using A to F as 10 to 15. So counting in hex (and binary) is like this:

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111
10	16	00010000
11	17	00010001

The reason that hex is used so often is that it is very easy to convert from binary to hex, but hex notation is much shorter than binary. You change hex to binary by replacing each hex digit by the corresponding 4 bits - like

$A2B3_{16}$ = 1010 0010 1011 0011_2

and in reverse for binary to hex. Most current processors are 64 bit, meaning they handle 64 bits or 8 bytes at a time. So a value a processor might process might be

1111 1010 1000 0011 0000 1110 1111 1010 1000 0101 1100 0011 1111 1010 1000 0011

which is almost impossible to write down without making a mistake. The corresponding hex version is not quite so bad:

F A 8 3 0 E F A 8 5 C 3 F A 8 3

One byte is 2 hex digits or 8 bits.

Octal

This is base 8. Place values are powers of 8, and we use the digits 0 to 7. This is like hex, but using groups of 3 bits:

Octal	Decimal	Binary
0	0	000
1	1	001
2	2	010
3	3	011
4	4	100
5	5	101
6	6	110
7	7	111
10	8	001000
11	9	001001

Test

1. Change decimal 1234 into binary and hex

2. Change hex 1234 into binary and decimal.

Bits and bytes and multiples

A bit is a binary digit - so 0 or 1.

A *byte* is a group of 8 bits - for example 0110 1011

On current digital devices, *each byte in RAM has a different address*. We can think of RAM as a set of boxes, each containing a byte, and each labelled with a unique address.

We often use multiples of bytes, like *kilobytes* and *megabytes*.

Different people use kilo and mega in slightly different ways:

Science-style base 10 multiples: science uses powers of 10 in groups of 3. So we have kilo=10^3, mega=10^6, giga = 10^9. So a kilometer km = 1000 meters. In computing this is sometimes used - so a kilobyte (kB) is 1000 bytes, a megabyte MB is a million bytes, gigabyte GB =one thousand million. The IEC standard uses this.

Base 2 binary multiples : this uses powers of 2 in steps of 10 = 2^{10} =1024. So 1 *kibibyte* = 1024 bytes. 1 *mebibyte* = 1024^2 - this is about a million, but slightly more. A gibibyte = 1024^3, and a tebibyte = 1024^4.

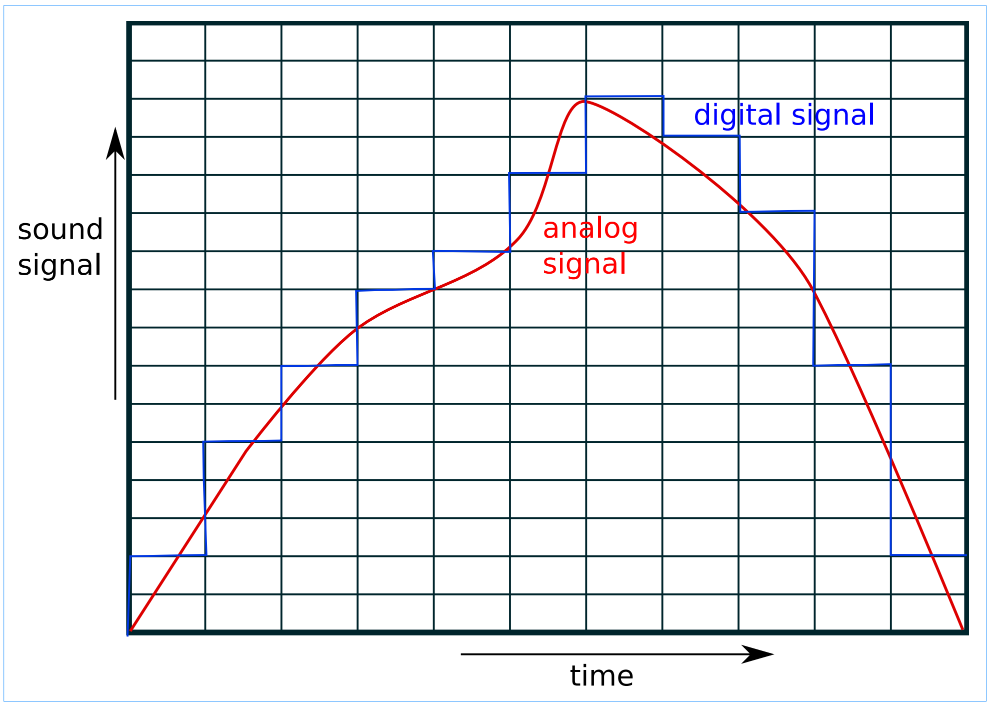
Analog and digital

Real sound is not digital. It is rapidly changing pressure wave travelling through the air. The pressure wave varies continuously - that is, not in steps. A microphone converts the pressure into an electrical signal, which also changes continuously. It is an analog signal. The voltage is an analog of the sound.

Light is also not digital. An old film camera has a lens to focus the light onto film, which carried a chemical which was affected by the light. The light can vary contiuously, not in steps.

Most real world data sources are not digital. Temperature, pressure, force, velocity all vary continuously. We use sensors to convert the data into an electrical analog.

For use in a digital system, the signal must be passed through an *analog-to-digital converter* (ADC).



This illustrates an analog sound signal (directly from a microphone) and a corresponding digital signal. The digital signal is not sampled continuously, but at regular points in time. CD audio, for example, is sampled 44100 times a second.

The digital signal here has 1 of 16 possible levels, coded in binary as 0000 up to 1111. In 4 bits there are 2^4 possible levels. CD audio uses 16 bits, not 4, so has 2^{16} = 65536 possible levels.

Test

If an audio format uses 8 bits, how many possible levels are there?

Numbers

This is about how to represent numbers in digital systems. This is not simply 'in binary', since everything is in the form of binary patterns. Different methods of representations are in use.

Most methods use a fixed amount of memory for all numbers. So we do not use a small number of bytes for small numbers and a lot of bytes for big numbers. Typically we use 16 bits or 32 or 64, for numbers big or small. This avoids the confusion about where one number ends and the next begins.

Some languages (such as Java and Python) support arbitrarily large numbers. A larger number will use more bytes, limited only by the memory available (and arithmetic will be slow). These types are different from the methods described here.

Unsigned integers

An unsigned integer is what it says - an integer without a sign, so we just take it to be positive. These are usually stored simply as their base 2 version.

For example, suppose unsigned integers are stored in 1 byte. The smallest we can have is 0000 0000 = 0. The largest is 1111 1111 = 255 = 2^8-1.

In practice unsigned integers are in 2, 4 or 8 bytes. An 8 byte = 64 bit example would be

0110 1010 1100 1110 0110 1010 1100 1110 0110 1010 1100 1110 0110 1010 1100 1110

and this is why we just use a 1 byte example. The same ideas apply.

We need to be able to do arithmetic with these values. For example to add 24 and 13:

24	0	0	0	1	1	0	0	0
13	0	0	0	0	1	1	0	1
+	0	0	1	0	0	1	0	1
carry			1	1				

so 24 + 13 = 100101_2=37

We are adding 2 bits at a time. The possibilities are 0+0=0, 0+1=1, 1+0=1, and 1+1=0 plus 1 to carry. So in fact we have to add 2 bits, an dmaybe a carry bit from the previous column.

A processor would have instructions to add unsigned integers like this, and would actually do it 'in hardware' - meaning electronics would do it, fast, rather than combinations of other software instructions (slow). The section about logic gates shows how half and full adders are constructed.

The biggest value we can have is 255 (in 1 byte). Suppose we add 240 + 240? The result is 480 and is larger than we can hold. This is arithmetic overflow. The processor would have a flag (a single bit) which the hardware would set (to 1) if overflow occurs. Other instructions can test the overflow flag so overflow can be detected.

Test

In unsigned integers in 8 bits, what is 1001 1100 in decimal?

Signed two's complement

The most common way of representing signed integers is called *two's complement.* To form it, follow this process:

If the number is positive, you just change it to base 2.

If it is negative, you change it to binary, invert it (change 0 to 1 and vice versa) and add 1.

For example, -3

1. In binary it is 0000 0011 (in 8 bits)

2. Invert: 1111 1100

3. Add 1:

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

So -3 is 1111 1101

Sometimes you need a carry. For example -4:

1. In binary it is 0000 0100 (in 8 bits)

2. Invert: 1111 1011

3. Add 1:

1	1	1	1	1	0	1	1	
							1	+
1	1	1	1	1	1	0	0	
					1	1		Carry

So -4 is 1111 1100

To interpret a two's complement number (to change from binary to normal) :

If the leading bit is 0, the number is positive - just change from base 2 to base 10.

If it is 1, the number is negative. Form the two's complement, then change to decimal : the value is minus what you get.

For example: 1110 1010

The leading bit is 1, so this is negative:

Invert it : 0001 0101

Add 1:

0	0	0	1	0	1	0	1	
							1	+
0	0	0	1	0	1	1	0	
						1		Carry

10110 = 2 + 4+16 =22
so 1110 1010 = -22

Range of this method

If you are using n bits, the largest positive number you can have is 01111.. (n-1 bits). You can't have 1111111.. since that will be negative (in fact, -1). This is 2^n-1.

The smallest (biggest and negative) is 10000..(n-1 bits). This is -2^n

So in 8 bits the range is from 2^8-1 = 127, to -2^8 which is -128

Test

In 2s complement in 8 bits, how big is 0100 1111?

How big is 1001 1100 ?

Numbers with fractional parts

This means values like 3.46 and -67.23. Two methods are used fixed-point (not common) and floating-point (usual).

Fixed point

Values are stored with the position of the decimal (or binary) point assumed. For example

1	0	0	1	assume point here	1	1	0	0

so what is actually stored is just

1	0	0	1	1	1	0	0

and this represents 1001.1100_2. The integer part is 1001 = 9. The fractional part is .11 in binary, which is 1/2 + 1/4 = 0.75. So this value is 9.75.

The largest value we could have would be

1	1	1	1	1	1	1	1

which is 1111.1111 = 15 + 1/2 + 1/4 + 1/8 +1/16 = 15.9375. The smallest value would be 0.

Floating point

This is much more common than fixed point - in fact 'floating point' is often taken to mean a number with fractional parts.

Mantissa and exponent

For example, the number 283.89. In scientific notation this is written as 2.8389×10^2. This the idea - to store the number in two parts:

28389	The mantissa
2	The exponent

This is the basic idea.

Mantissa and exponent in binary

Since we are using digital systems, everything must be in binary. So the mantissa and exponent are in binary, and the exponent is the power of two, not ten.

They are both fixed width. Suppose we have an 8 bit mantissa and 4 bit exponent (the real thing uses maybe a 53 bit mantissa and at least 15 bits exponent).

What would 13.75 look like? First write it in binary

$13.75_{10} = 1101.11_2$ (the binary fraction values are 1/2 and 1/4, so 0.75 = .11)

Then *adjust the point to before the first digit*

1101.11 = .110111 X 2^4

So the exponent is 4_{10}=100_2, and the mantissa is 0.110111. In our format this would be

0	.	1	1	0	1	1	1	0	0	1	0	0
Mantissa									Exponent			

What is the biggest number we can have? The biggest exponent is 0111 (1000 would be negative, since this is two's complement). The biggest mantissa is 0111 1111. So the biggest value is

0.111 1111 X 2^{0111} = 0.111 1111 X 2^7 = 1111111.0_2 = 127

Real systems, using more bits, might have floating point values up to 10^{308}

An example is to represent 0.1 in binary:

0 : 01111011 : 10011001100110011001101

0.1_{10} = .000110011001100110011.. = 1.10011001100110011... X 2^{-4}

Sign bit 0 : positive number

Exponent = -4+127 = 123 = 1111011

Mantissa corresponds to 1.10011001100110011...

But the point is that the mantissa only holds 23 bits, whereas the exact value has an *infinite* number of bits. So 0.1 *cannot be represented exactly* in this format. The same is true for many values - they *cannot be represented exactly*. In turn arithmetic on them will not be exact. If we use double instead, we get more accuracy - but still not exact.

Floating point is not exact

One consequence of this is that floating point types should not be used to represent currency. Customers will not be at all happy with approximate invoices.

Test

A floating point format has a mantissa of 1 byte and an exponent in 4 bits, in 2s complement form.

How would the largest possible value be represented in binary? What is it in base 10?

Normalisation

Suppose a floating point format has a 4 bit mantissa and 4 bit exponent. Look at these values:

Mantissa				Exponent			
0	1	0	0	0	0	0	0
0	0	1	0	0	0	0	1
0	0	0	1	0	0	1	0

The first row is 0.1 X 2^0 (binary) = 1/2 decimal

The second row is 0.01 X2^1 = 0.1 =1/2 decimal

The third is 0.001 X 2^2 = 0.1 = 1/2 decimal

So these are the same value. We are shifting the mantissa to the right, and making the exponent bigger.

Which version is better? The first, because it keeps the most bit places to the right of the first bit. If the mantissa was 0.111, the second version would be 0.011, and we lose a bit. The third is 0.001, and we have lost 2 bits.

So the best format is to have the leading 1 bit in the second bit place, to hold as much data as possible. This is called *normalisation.*

Why not have the leading 1 in the first bit place? So 1000? We might do, but the mantissa is probably *two's complement*, so this is a negative value. 1.000 is decimal -1.

Range

Range means the biggest and smallest values which can be represented. Smallest means most negative.

For example with 4 bit mantissa and exponent, the largest value is

Mantissa				Exponent			
0	1	1	1	0	1	1	1

The mantissa is as large as possible, = 0.111 binary. The exponent is as large as possible, =7. The number is

$0.111 \times 2^7 = 1110000_2 =$ 112 decimal.

The smallest (biggest negative) is

Mantissa				Exponent			
1	0	0	0	0	1	1	1

The mantissa (2s complement) is -1.000. So the value is 10000000 = -128.

So the range of this format is -128 to +112

Precision

The precision is the smallest change we can represent. In decimal with 3 decimal places, we hav evalues like 23.123. The smallest change we can have is in the smallest decimal place, so 0.001. The factor that controls this is how many decimal places we have.

In floating point the precision depends on the number of bits in the mantissa. So in

Mantissa				Exponent			
0	1	1	1	0	1	1	1

the precision is controlled by the 3 bits highlighted.

Absolute and relative errors

Floating point values will not usually be totally accurate. So they will have errors.

Suppose in decimal we have a value 100.1, and the last digit is an error. Then the *absolute error* is 0.1.

The *relative error* is the error as a fraction of the value. So in this example teh relative error is 0.1 in 100, so 1 in a thousand, or 0.001, or a tenth of 1 per cent.

Underflow and overflow

Overflow is an event when the result is larger than the largest value which can be represented.

For example suppose we have unsigned integers in 4 bits, and we add 15 +1:

		Data bits			
15		1	1	1	1
1		0	0	0	1
Sum		0	0	0	0
Carry	1	1	1	1	0

The sum looks like 0000 = 0. This is incorrect because we have overflow.

Underflow is similar but is when a result is too small to be represented

Characters and strings

A character is a symbol used in written text. A *character set* is, as expected, a set of characters, with each character having a unique *code point*, an integer.

Two common character sets at the moment are *ASCII* and *Unicode*.

The original ASCII character set had 128 characters, with code points from 0 to 127. The first 32 of these were control characters used to control printing, which at the time were printers called Teletypes. Code point 10 was 'line feed', which moved the paper through 1 line. 13 was 'carraige return', which moved the print head back to the left edge. Printed characters started at 32. Character 'A' was code point 65, 'B' was 66 and so on.

Extended ASCII has 256 characters, code points 0 to 255. The first 128 of these were the same as orginal ASCII.

ASCII stands for 'American Standard Code for Information Interchange'. Being American it only covered the Western script alphabet A to Z and a to z.

To handle all the scripts required (essential with the Internet) a much wider range of codepoints was required. This was Unicode. This started in 1987, and we are now at version 18, with around 150,000 characters - like ओ and تهجى and N and ∀

An *encoding* is a way of representing a character code point in binary.

Extended ASCII has codepoints 0 to 255, and these will fit into 1 byte. So the obvious way to encode ASCII was to simply have 1 byte per character. So if 3 bytes contained 65 - 66 - 67 (in binary) this would be the characters ABC.

Unicode codepoints currently go up to 10FFFF hex = 1,114,111 decimal, so these will not fit into 1 byte. Commonly used encodings are called UTF8, UTF16 and UTF32. Some use 2 bytes, and some use a variable number of bytes.

Test

What is the ASCII code point of 'C'?

What is the ASCII code point of 'c'?

How do you convert ASCII capital letters to lower case?

What is the ASCII code point of the character '0'?

Digits as characters

Note the difference between digits and characters representing digits.

For example, 1 is a digit. The character '1' has ASCII code 49. 2 is a different digit. The character '2' has ASCII code 50.

Images and sound

Image and sound formats

Audio clips are usually large (in terms of number of bytes). As a result they are usually held in a file. Only part is loaded into memory at any time, into a buffer, while the sound is being played.

Movie data is even larger.

So there are two issues -

- How the data file is formatted
- How the data is represented once loaded into memory.

Common sound file formats are mp3, FLAC and wav.

Image files are jpg, gif, png, tiff and so on.

Video formats are ogg, avi, wmv and mpeg.

Bitmapped graphics

A bitmapped graphic is a 2D array of *pixels* - the dots making up the image.

Each dot will usually have red, green and blue components. So a pixel value might be 00 01 11, meaning red=00 = 0, green = 01 = 1, and blue = 11 = 3.

A more common pixel format would be 8 bits for each of red green and blue - so for example in hex 00 80 77. In binary the red component is 0000 0000, green is 1000 0000, and blue is 0111 0111. That gives a 24-bit *colour depth* - a possible 2^{24} different colours.

The terms size and resolution can mean several things.

Size usually means the number of rows and columns in the bitmap. For example, DVD is 720 pixels across and 576 rows. Blu-ray is 1920 by 1080. 4K cinema is 4096 by 2160. IMAX is about 10,000 by 7,000. Some people use the term resolution to also mean this.

But resolution might mean dots (pixels) per inch.

For example, a laptop might display an image on its screen. And it might be connected to a projector which displays the same image on a large surface. There the dots per inch would be much lower, since the image is larger. But the number of pixels would be the same.

For example, Blu-ray might (there are different versions) have a colour depth of 8 bits per colour, so each pixel is 3 bytes. The size is 1920 by 1080, so there are 2073600 pixels in one frame. So 3 times that = 6220800 bytes per frame - around 6MB. Blu-rays show 24 frames per second, so that is about 120 MB of data being processed per second.

Graphics image data usually includes *meta-data*, that is, data *about* the image.

An example would be EXIF data in an image. The camera includes, together with the pixel data, the camera make and model, exposure time, F number, date and time and so on.

Test

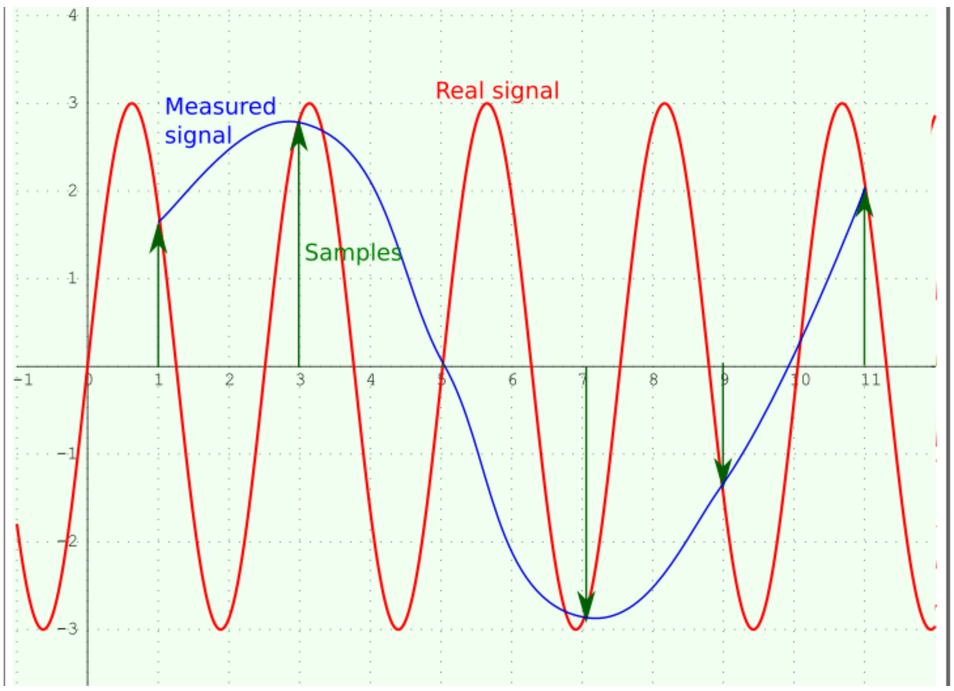
If an image is 300 pixels by 200, with a colour depth of 16 bits, how many bytes does it need (ignoring an metadata)

Digital sound

Sampling rates and Nyquist's Theorem

Sound signals from a microphone, or any other analog input, are not captured continuously. They are sampled at intervals. CD audio, for example, is sampled at a frequency of 44.1 kHz - 44,100 times a second.

The sampling rate cannot be too low, or the measured signal will be very different from the real signal, like this:



Nyquists's Theorem says the sampling rate must be *at least twice* the highest input frequency.

So CD audio at 44.1 kHz can handle sound up to 22.05 kHz. Most humans can only hear sound up to around 20 kHz, so this covers everything we can hear.

CD audio uses 16 bits, so it has 2 bytes per sample. So at a sample rate of around 20 kHz, this is about 40 kB per second data rate.

MIDI

This is Musical Instrument Digital Interface. This is a standard which allows music keyboards and other devices to be connected. A very simple set-up would be a keyboard connected by MIDI as input into a digital audio workstation (DAW).

A MIDI cable can carry up to 16 channels, each of which can be used by 1 instrument. A channel carries data for each note, such as note pitch, length and loudness.

Advantages of MIDI over cables carrying actual sound signals are

1. Its standard, so different manaufacturer's instruments can be interconnected.

2. The data rate is much lower, so file sizes are much smaller.

3. Different sound fonts on a DAW mean one instrument can produce different sounds (like piano, harpsichord, pipe organ or guitar).

Data errors compression and encryption

Error correction

Data can contain errors, especially when sent over a network.

We can

1. Try to *detect the errors*
2. and then *correct them*

As an example, we have a stream of 4 bit data values. Each value has a *parity bit*, and is followed by a *checksum.*

A parity bit is an extra bit added to data, to make the number of 1 bits even (in even parity - we can also use odd parity). So if the data is 0111, the parity bit is 1, so we have an even number of 1s. If the data is 0011, it is already even, so the parity bit is 0.

If we have data 1011 and parity bit 0, there is an error - we have an odd number of 1s. We have detected the error. But we do not know which bit is wrong.

We can also form a checksum by various methods. One is simply to XOR the data bitwise. So if the first bit in 4 data values is 0 0 1 0, if we XOR these we get 1. So the first bit in the checksum is 1.

In this example:

Data bits				**Parity bit**
0	1	0	0	1
0	1	0	1	0
1	0	**1**	1	0
0	0	1	0	1
1	**0**	**1**	**0**	**XOR Checksum**

the third row has odd parity - so it contains an error.

The third column also has an incorrect checksum bit.

We therefore know the third row and third column is wrong. It is a 1 - so it should be a 0.

Error correction is sometimes possible if we have redundant data - extra beyond the actual data. More redundant data enables more error correction.

A related idea is majority voting. This is a general technique for dealing with errors. For digital data, we can simply send it 3 times, and compare corresponding bits. With no errors they will agree - we get 000 (correct value 0) or 111 (correct value 1). If we get 010 there is an error. We have 2 0s, so we assume the 0 is correct. 011 means an error, and correct value is 1.

Of course it is possible the majority is wrong. We could transmit 5 copies. We might receive 00100, and correct this to a 0. The more copies, the more redundancy, the lower the chance of errors.

A related technique is used in safety-critical systems such as aircraft control. Devices are used in triplicate. The devices are from different manufacturers containing different software so they will not have the same fault. If their output differs, the majority version is used.

Test

Original **ASCII** was 7 bits per **character**, with 1 **parity bit,** so filling 1 **byte**. Explain the terms in bold.

Compression

This means processing data into a form which uses less space - fewer bytes. This is common for sound and audio and movie files, and other data types.

This has the advantages of

- less storage space used
- less traffic on a network
- faster upload and download

But it usually means the data must be de-compressed before use.

Lossless compression means no data is lost - it can be expanded back to the original.

Lossy compression means some data is lost.

Run-length encoding

Run-length encoding (RLE) is a common lossless compression techniques. For example:

4E4A6F1B5Z

is a compressed form of

EEEEAAAAFFFFFFBZZZZZ

We have replaced each run of the same value (like EEEE) by a count followed by the data (like 4E).

This works well for image files. For example part of an image might be a patch of blue sky, with several rows of hundreds of pixels, all simply blue. We replace them with 100blue.

Test

Some RLE compressed text is 3A5B4C. De-compress it

Dictionary compression

Suppose we have a small graphics image, 100 pixels by 100, and have a total colour depth of 8 bits, so 1 byte per pixel. So that is 10,000 bytes.

But suppose the image only has 4 colours - black, white, red and green.

In the image file we start with a table:

index	colour
00	black
01	white
10	red
11	green

Then we just need 2 bits per pixel, and the filesize is 20,000 bits = 2500 bytes. This is a quarter of the size.

gif file format uses this technique. It calls the table a 'palette'. The number of colours can be up to 256, so the index is 1 byte in size. But a line drawing might be just black and white, so the palette could have just 2 colours, with 1 bit per pixel.

This is an example of dictionary compression. Instead of storing the original data, we have a dictionary containing all the different data values needed, and store indexes into the dictionary.

Test

A gif file has 16 different colours. How many bits are needed for each pixel?

Encryption

All data and all software in a digital system is in binary. In this sense it is all encoded, in that some method of representing it as sets of binary patterns is used. For example, characters might be encoded as ASCII code points.

Encryption is different. It means taking some message and turning it into a form which cannot be read by normal means. It can then be sent to someone who can reverse the process, so that they can read it.

The message is called *plaintext*.

The encoded version is called *cyphertext*.

The method of encoding is called the *cipher*. Also spelt *cypher.*

We *encrypt* the plaintext to produce the cyphertext. We *decrypt* that to get back to the palintext.

One simple method is called the *Caesar cypher.* It simply shifts letters through the alphabet. For example we might replace a letter by the letter 5 plces further on. As A is replaced by F. B is replaced by G. C is replaced by H. If we reach the end of the alphabet, we go back to A. So X is replaced by C.

So HELLO in plaintext becomes

MJQQT

The Caesar cypher is easily broken. This is because letters are used with different frequencies. E is used most often. Then T, then A, O, I, N and so on. So we take the cyphertext and see which letter is most common. That will probably match E, and we have an idea what the shift is. The second most common will probably be T, and this will confirm the shift.

Test

Write a program (any language) to show a Caesar cypher. The program should start with a string, output it, encrypt it and output that, and decrypt it and output that.

Another method is the *Vernam cypher*, also called a *one-time pad*.

Imagine a pad of paper, with the top page covered in a table of letters. The person who will receive the message has the same pad, with the same letters. The message is encrypted as follows:

1. Take the first message letter and the first letter on the pad.

2. Merge them - say by shifting by that amount. So if the pad letter is A, shift by 1. If its B, shift by 2

3. This gives the encrypted letter. Continue for the whole message (so the pad text must be at least as long as the message)

The receiver just reverses this. They

1. Take the first message letter and the first letter on the pad.

2. De-merge them - say by shifting backwards instead of forwards

3. This gives the encrypted letter. Continue for the whole message

The pad page is *only used once* - so its a one-time pad. Next message we use the next page.

The pad letters must be completely random.

This has been proved to be completely unbreakable (by Shannon, published 1949).

But it is not widely used. One reason is a copy of the pad must be sent to the receiver, but must not fall into the hands of the enemy.

For a computerised version, the merging can be done by a bitwise XOR. If this is done twice, you get back to the original. For example

1	0	1	1	0	1	plaintext data	original
0	0	1	1	1	0	OTP data	encode
0	0	0	0	1	1	XOR result	cyphertext
0	0	1	1	1	0	OTP again	decode
1	1	0	0	0	1	XOR again	original

One problem is that perfectly random sequences are impossible on a computer.