Number Systems

Natural numbers - We use N to denote mathematical notation for natural numbers. Natural numbers are positive integers. Integers are whole numbers. Natural numbers are a subset of integers. The set of natural numbers can also include zero. Thus the set of natural numbers can be:

$$\mathrm{N}= \{1,2,3\ \ldots.\} \text{ Or } \mathrm{N}= \{0,1,2,3\ \ldots., \infty\}$$

Cardinal and ordinal numbers - Natural numbers are used for counting and ordering. More formally we use the terms cardinal and ordinal.

Cardinal numbers are used for counting.

Ordinal numbers are used for ordering. Gives the position of items in a list. For instance in the list A where

A = {"apple","banana","orange"},

The first item in position 1 is "apple", the second is in position 2: "banana" and the third is in position 3: "orange"

Integers - We use Z to denote mathematical notation for the set of integers. Integers are whole numbers include the set of natural numbers in addition to negative numbers. Thus the set of integer numbers is:

$$\mathrm{Z} = \{\ldots, -3,\ -2,\ -1,\ 0,1,2,3\ , \ldots\}$$

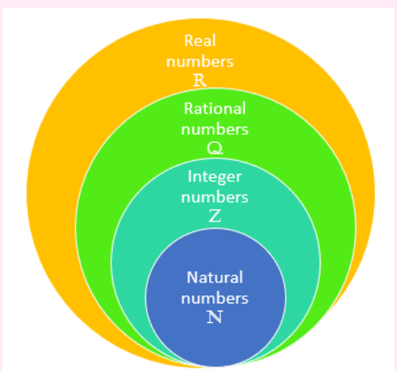
Rational numbers - We use Q to denote mathematical notation for rational numbers. Rational numbers are any number that can be represented by a fraction where the numerator and denominator are integers *a* and *b* $\frac{a}{b}$.Rational numbers can be positive and negative and include integer values. A recurring number such as 0.3333… is a rational number because it can be represented by a fraction, in this case $\frac{1}{3}$

Irrational numbers - Infinite series of non repeating digits andcannot be written as a quotient of integers (ie fractions) and cannot be written as a terminating decimal or repeating decimal. These numbers cannot be represented exactly, although for most practical purposes the values used for these constants are sufficient when rounded. Examples of irrational numbers: π, e, $\sqrt{2}$, $\sqrt{7}$, 0.271982…

Real numbers - The set of real numbers R includes both rational and irrational numbers. Real numbers are uses for the measurement of continuous values. Eg measuring length or weight.

-2, -1, 0 , $\frac{1}{2}$, 1, $\sqrt{2}$, 2, e, 3, π

Relation between number sets



Number bases

Denary (or decimal) is base-10 and is the number system we are most familiar with. We have the columns of units, tens, hundreds, thousands and so on. Base-10 means that we have 10 possible values (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) in each column.

Binary is base-2 and has 2 values, 0 and 1. It requires a greater number of digits in binary to represent a number than denary. This is how data and instructions are stored in a computer.

To calculate the maximum value for a given number of bits we use 2^n-1 where n is the number of bits. For example for 4 bits we have 2^4-1 which is 15.

Bits	Max value binary	Max value denary
1	1_2	1_{10}
2	11_2	3_{10}
3	111_2	7_{10}
4	1111_2	15_{10}
5	11111_2	31_{10}
6	111111_2	63_{10}
7	1111111_2	127_{10}
8	11111111_2	255_{10}

Hexadecimal is base-16. To make up the 16 values we use the ten denary numbers in addition to 6 letters (A, B, C, D, E, F).

Denary	Hex.	Binary
0_{10}	0_{16}	0000_2
1_{10}	1_{16}	0001_2
2_{10}	2_{16}	0010_2
3_{10}	3_{16}	0011_2
4_{10}	4_{16}	0100_2
5_{10}	5_{16}	0101_2
6_{10}	6_{16}	0110_2
7_{10}	7_{16}	0111_2

Denary	Hex.	Binary
8_{10}	8_{16}	1000_2
9_{10}	9_{16}	1001_2
10_{10}	A_{16}	1010_2
11_{10}	B_{16}	1011_2
12_{10}	C_{16}	1100_2
13_{10}	D_{16}	1101_2
14_{10}	E_{16}	1110_2
15_{10}	F_{16}	1111_2

Hexadecimal is used a lot in computing because it much easier to read than binary. There are far fewer characters than binary. So hexadecimal is often used in place of binary as a shorthand to save space. For instance, the hexadecimal number 7BA3D456 (8 digits) is 01111011101000111101010001010110 (32 digits) in binary which is hard to read.

Hexadecimal is better than denary at representing binary because hexadecimal is based on powers of 2.

Converting between number bases

Denary to binary conversion

1) Create a grid:

128	64	32	16	8	4	2	1

2) Add a 1 to the corresponding cell if number contributes to target number and 0 to all the other cells

Worked example: convert 24_{10} to binary.

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

$16_{10} + 8_{10}=24_{10}$
The binary value is $\underline{11000_2}$ (we can ignore the preceding zeros)

Binary to denary conversion

Worked example: Convert 01011001_2 to denary

1) Create the grid:

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

2) Add up the cells that have a corresponding value of 1:
$64 + 16_{10} + 8_{10} + 1= \underline{89_{10}}$

Hexadecimal to binary conversion

1) Find the corresponding 4-bit binary number for the hex digits
2) Concatenate the binary values to give the final binary value

Example: convert $C3_{16}$ to binary
$C_{16} = 12_{10} = 1100_2$
$3_{16}= 3_{10} = 0011_2$
$\underline{11000011_2}$

Binary to hexadecimal conversion

1) Split the binary number into groups of 4 bits: 1110_2 1010_2. If the number of bits is not divisible by 4 prepend the binary number with 1,2 or 3 zeros to make it divisible by 4.
2) Find the corresponding Hex value for each of the 4-bit groups

Worked example: Convert 11101010_2 to hexadecimal
1110_2 |1010_2
$1110_2 = 14_{10} = E_{16}$
$1010_2 = 10_{10} = A_{16}$
$\underline{EA_{16}}$

Hexadecimal to denary conversion general approach

1) Find the corresponding 4-bit binary number for the hex digits
2) Concatenate the binary values to give the final binary value
3) Convert the binary number to denary
4) Add the second value

Worked example: Covert $1AB_{16}$ to denary
$1_{16} = 0001_2$
$A_{16} = 1010_2$
$B_{16} = 1011_2$

Concatenate: $000110101011_2 = \underline{427_{10}}$

Denary to hexadecimal conversion general approach

1) Convert to binary
2) Split the binary number into groups of 4 bits: 11102 10102. If the number of bits is not divisible by 4 prepend the binary number with 1,2 or 3 zeros to make it divisible by 4.
3) Find the corresponding Hex value for each of the 4-bit groups

Worked example: Convert 503_{10} to hex
Convert to binary: 111110111_2
Split into groups of 4 bits: 0001_2 | 1111_2 | 0111_2
$0001_2 = 1_{16}$
$1111_2 = F_{16}$
$0111_2 =7_{16}$
$\underline{1F7_{16}}$

Hexadecimal to denary conversion for two hex digits

1) Convert the two hex values separately to denary value
2) Multiply the first value by 16
3) Add the second value

Worked example: Covert $A3_{16}$ to denary

$A_{16} = 10_{10}$
$3_{16} = 3_{10}$
$(10_{10} \times 16_{10}) + 3_{10} = \underline{163_{10}}$

Denary to hexadecimal conversion for denary values 255 or less

1) Integer divide the denary number by 16
2) Take the modulus 16 of the denary number
3) Convert the two numbers to the corresponding hex values.

Worked example: Convert 189_{10} to hex

$189_{10} / 16_{10} = 11_{10}$ remainder 15_{10}
$11_{10} = B_{16}$
$15_{10} = F_{16}$
$189_{10} = \underline{BF_{16}}$

Units of Information

Bits and Bytes

A bit is the fundamental unit of binary numbers. A bit is a binary digit that can be either 0 or 1. Bit is short for binary digit

A byte contains 8 bits, as a power of 2^3.

The number of different values that can be represented by a sequence of bits is 2^n where n is the number of bits.

No. of bits	No. of values	Values
1	2 (2^1)	0, 1
2	4 (2^2)	00, 01, 10, 11
3	8 (2^3)	000, 001, 010, 011,100,101,110,111
4	16 (2^4)	0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110 , 1111

Units

- We often state that a kilobyte is 1024 bytes but this is not strictly true because a kilobyte is 1000 bytes.
- Powers of 2 units including kibibytes, mibibytes, gibibytes and tibibytes were introduced to allow us to refer to the number of bytes unambiguously.
- We should therefore use kibibytes to refer to 1024 bytes. This also applies elsewhere where we often use megabytes, gigabytes and terrabytes when we should use mebibytes, gibibytes and tebibytes, respectively.
- There is only a small difference in the number of bytes between kB and KiB, but this increases significantly for larger powers such as between yottabytes (10^{24}) and yobibytes (2^{80}).

Units Powers of 10

UNIT	SYMBOL	NUMBER OF BYTES
Kilobyte	kB	10^3 (1000)
Megabyte	MB	10^6 (1 million)
Gigabyte	GB	10^9 (1 billion)
Terabyte	TB	10^{12} (1 trillion)

Units Powers of 2

UNIT	SYMBOL	NUMBER OF BYTES
KibiByte	KiB	2^{10} (1024)
MebiByte	MiB	2^{20}
GibiByte	GiB	2^{30}
TebiByte	TiB	2^{40}

Binary Number System

Unsigned Binary

Signed Integer – can be used to represent both positive and negative numbers

Unsigned Integer – can only represent positive numbers

For unsigned integers the minimum and maximum values are 0 and 2^n-1 respectively, where n is the number of bits.

FOR UNSIGNED INTEGERS DO NOT GET CONFUSED BETWEEN THE MAXIMUM VALUE (2^n-1) AND THE NUMBER OF VALUES (2^n) THAT CAN BE REPRESENTED FOR A GIVEN NUMBER OF BITS

For signed integers the minimum value is

$$\frac{-2^n}{2}$$

and the maximum value is

$$\frac{2^n}{2} - 1$$

	SIGNED INTEGERS		UNSIGNED INTEGERS	
NO. OF BITS	**Min Value**	**Max Value**	**Min Value**	**Max Value**
2	-2	1	0	3
3	-4	3	0	7
4	-8	7	0	15
5	-16	15	0	31
6	-32	31	0	63
7	-64	63	0	127
8	-128	127	0	255

Add unsigned binary integers

Binary addition rules

$0_2 + 0_2 = 0_2$
$0_2 + 1_2 = 1_2$
$1_2 + 0_2 = 1_2$
$1_2 + 1_2 = 10_2$ (carry 1)
$1_2 + 1_2 + 1_2 = 11_2$ (carry 1)

Example

$$\begin{array}{rr} & 01001001_2 \\ + & \underline{01010101_2} \\ & 10011111_2 \\ \text{carry} & 1 \quad\quad 1 \end{array}$$

Multiply unsigned binary numbers

Multiplication in binary is the same as it is in denary

Binary Multiplication rules

$0_2 \times 0_2 = 0_2$
$0_2 \times 1_2 = 0_2$
$1_2 \times 0_2 = 0_2$
$1_2 \times 1_2 = 1_2$

Example

$$\begin{array}{rrrrrrrrr} & & & & 1 & 1 & 1 & 1 & 0 & 1_2 \\ & & & \times & & & 1 & 0 & 0 & 1_2 \\ \hline & & & & 1 & 1 & 1 & 1 & 0 & 1_2 \\ & & & 0 & 0 & 0 & 0 & 0 & 0 & 0_2 \\ & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0_2 \\ + & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0_2 \\ \hline 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1_2 \end{array}$$

Two's Complement – Technique applied to binary numbers to make then negative

Steps for converting from Denary to Two's Complement

1) Find binary value of the positive denary number
2) Flip all 1 bits to 0 and 0 bits to 1
3) Add 1

Worked Example: Convert -18 to Two's complement binary in 8 bits

1. Find binary value of the positive denary number — $18_{10} = 00010010_2$
2. Flip all 1 bits to 0 and 0 bits to 1 — 11101101_2
3. Add 1 — 11101101_2 + $\underline{00000001_2}$ = $\underline{11101110_2}$

Example of a quick way to convert a negative number to Two's complement binary

1. Find binary value of the positive denary number — $18_{10} = 00010010_2$
2. Flip all 1 bits to 0 and 0 bits to 1, up to the final 1 and copy the remaining bits. This is the same a flipping all the bits and adding 1 — $\underline{11101110_2}$

Steps for converting a negative Two's complement number to denary

1) Flip bits
2) Add 1
3) Convert from binary to denary
4) Add the minus sign (it is really easy to forget)

Worked Example: convert the following negative Two's complement number to denary: 10011010_2

1. Flip bits — 10011010_2 -> 01100101_2
2. Add 1 — 01100110_2
3. Convert to denary — $01100110_2 = 102_{10}$
4. Add the minus sign — $\underline{-102_{10}}$

Steps to Subtract binary numbers

1. Apply two's complement to the second binary number
2. Add the two numbers together

Worked Example: Subtract the following two numbers:

00101011_2 43_{10}
$-\ 00010000_2$ 17_{10}

1. Apply two's complement to the second binary number — 00101011_2 - > 11101111_2
2. Add the two numbers together — 00101011_2 + $\underline{11101111_2}$
   Disregard the overflow 1 — (1)$\underline{00011010_2}$ 26_{10}

Fixed Point Representation

Fixed point binary is one method used to represent fractions in binary. An imaginary point is used to represent the fractional part of a number. The table below shows the corresponding decimal and binary representation of some fractions. We need to introduce an imaginary point to the binary number to separate the whole and fractional part of the number.

	Fraction	Decimal	Binary
2^1	$\frac{2}{1}$	2	10

2^0	$\frac{1}{1}$	1	1
2^{-1}	$\frac{1}{2}$	0.5	0.1
2^{-2}	$\frac{1}{4}$	0.25	0.01
2^{-3}	$\frac{1}{8}$	0.125	0.001
2^{-4}	$\frac{1}{16}$	0.0625	0.0001
2^{-5}	$\frac{1}{32}$	0.03125	0.00001
2^{-6}	$\frac{1}{64}$	0.015625	0.000001
2^{-7}	$\frac{1}{128}$	0.0078125	0.0000001

Worked example: What is binary fraction 1001.1100_2 in denary?

8	4	2	1		$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$
1	0	0	1	.	1	1	0	0

8 + 1 + 0.5 + 0.25 = 9.75

Worked example: What is denary number 3.125_{10} in fixed point binary, represented with 4 bits before and after the imaginary point.

8	4	2	1		$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$
0	0	1	1	.	0	0	1	0

0011.0010

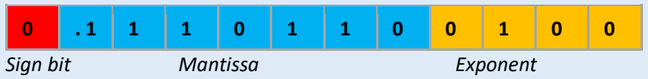
Floating Point Representation

Floating point representation allows us to have a greater range of numbers for a given number of bits than fixed point binary. Floating point numbers are split into two parts:

- **Mantissa** – Represents the significant bits of a number. The larger number of bits used to represent the mantissa the greater the precision.
- **Exponent** – The power to which the mantissa is raised. The larger the number of bits used to represent the exponent the greater the range of numbers.

Example: the denary number 0.001234 that can also be represented as 1.234×10^{-3} where 1.234 is the mantissa and 10^{-3} is the exponent,

Real numbers



Sign bit *Mantissa* *Exponent*

- Example: 8 bits for the mantissa, 4 bits for the exponent
- The sign bit (red left most bit) determines whether the number is positive or negative. If the sign bit is 0 then the number will be positive and if the sign bit is 1 then the number will be negative.
- Likewise the first (leftmost) bit of the exponent determines whether the exponent is positive or negative.
- In this example both the mantissa and exponent are positive.
- The imaginary point is placed directly after the sign bit.

Examples: Convert the following 12 bit floating point numbers where the first 8 bits represent the mantissa and the remaining 4 bits represent the exponent to denary

Positive Mantissa and Positive Exponent

Exponent: $0000100_2 = 4_{10}$
Mantissa: 0.11101100_2
The point of the mantissa moves 4 points to the right thus:
$01110.11000_2 = 14.75_{10}$

Positive Mantissa and Negative Exponent

The exponent is negative so we have to apply two's complement to find the denary equivalent
Exponent: 1 1 1 0 $_2$
Flip bits: 0 0 0 1 $_2$
Add 1: 0 0 1 0 $_2 = -2_{10}$
Mantissa: 0 . 1 1 0 0 0 0 0 $_2$
The point of the mantissa moves 2 points to the *left* thus:
0 . 0 0 1 1 $_2 = 0.1875_{10}$

Negative Mantissa and Positive Exponent

Exponent: 0 1 0 0 $_2 = 4_{10}$
The mantissa is negative so we have to apply two's complement
Mantissa: 1 . 0 1 0 1 0 0 0 $_2$
Flip bits: 0 . 1 0 1 0 1 1 1 $_2$
Add 1: 0 . 1 0 1 1 0 0 0 $_2$
The point of the mantissa moves 4 points to the *right* thus:
1 0 1 1 . 0 $_2 = -11.0_{10}$

Negative Mantissa and Negative Exponent

The exponent is negative so we have to apply two's complement to find the denary equivalent
Exponent: 1 1 1 0 $_2$
Flip bits: 0 0 0 1 $_2$
Add 1: 0 0 1 0 $_2 = -2_{10}$
The mantissa is negative so we have to apply two's complement to find the denary equivalent
0.1100000_2
The binary point of the mantissa moves 2 points to the **left** thus:
0 . 0 0 1 1 $_2 = -0.1875_{10}$

Range using floating point representation with a 6 bits for the mantissa and 4 bits for the exponent

Largest positive value that can be represented	$0.11111\ 0011_2$
Minimum positive value that can be represented	$0.10000\ 1000_2$
Largest magnitude negative value that can be represented	$1.00000\ 0011_2$
Smallest magnitude negative number that can be represented	$1.01111\ 1000_2$

Rounding Errors

There are fractions that we cannot represent exactly using fixed point or floating point representation. For instance, we cannot represent $\frac{1}{10}$ exactly.

4 bits: $0.001_2 = 0.125_{10} - 0.1_{10} = 0.025_{10}$
6 bits: $0.00011_2 = 0.09375_{10} - 0.1_{10} = -0.00625_{10}$
8 bits: $0.00011010_2 = 0.1015625_{10} - 0.1_{10} = 0.0015625_{10}$
10 bits: $0.0001100011_2 = 0.099609375_{10} - 0.1_{10} = 0.00041063_{10}$

The greater the number of bits the closer we can get to $\frac{1}{10}$ but we can never represent it exactly even if we can get very close. We say that the number of bits is asymptotic to the fraction we are trying to represent.

Normalisation

These are all the same number:

- 32.5191×10^0
- 0.0325191×10^3
- 0.0000325191×10^6
- 32519.1×10^{-3}
- $32519100.0 \times 10^{-6}$

But this is the normalised number: 3.25191×10^1

In scientific notation for a normalised number there is only 1 digit before the decimal point and it must be a significant (non zero) value.

The purpose of normalising is to allow the maximum precision for a given number of bits.

In floating point representation the first two bits of the mantissa will be 0.1_2 (remember 0 is the sign bit) for positive numbers. The mantissa is between 0.5_{10} (0.1_2) and 1 (0.1111111_2).

In floating point representation the first two bits of the mantissa will be 1.0_2 (remember 1 is the sign bit) for negative numbers. The mantissa is between -0.5_{10} and -1_{10}.

Example normalised numbers	*Example of non-normalised numbers*
1.01001 0011	1.10110 0011
0.11101 1001	0.00100 0111

Normalisation is important because we increase the precision of the number we wish to represent

Example: Representing 1_{10}. Using e have the following floating point representation with 8 bits for the mantissa and 4 bits of the exponent.

$0.1111111\ 0000_2 = 0.9921875_{10}$

In binary this is the normalised number because the first bit after the decimal point is a significant value. For a non-normalised number.

$0.0011111\ 0010_2 = 0.96875_{10}$

The normalised number has allowed to represent 1_{10} with greater accuracy.

Worked Example
Normalise the following positive number with 8 bits for the mantissa and 4 bits for the exponent: $0.0001110\ \ 0101_2$

Move the point 3 places to the right: 0000.1110_2
subtract 3 from the exponent to make it smaller so the exponent becomes 0010_2
Mantissa: 0.1110_2
Exponent: $-3_{10} = 011_2 = 101_2$
$\underline{0.11100\ 0010_2}$

Worked Example
Normalise the following negative number with 8 bits for the mantissa and 4 bits for the exponent: $1.1110100\ 0010_2$

Move the point 3 places to the right so: 1.0100000_{10}
The mantissa is larger so the exponent needs to be smaller.
Subtract -3_{10} from the exponent which gives: $-1_{10} = 1111_2$
$\underline{1.0100000\ 1111_2}$

Absolute and Relative Error

The **absolute error** is the positive difference between the actual value and our floating point value.

$$\text{Absolute error} = |\ \text{actual number} - \text{floating number}\ |$$

The **relative error** is the absolute error divided by the actual value

$$\text{Relative error} = \frac{|\ \text{actual number} - \text{floating number}\ |}{\text{actual number}}$$

Consider a table that is 1m in length and it is measured to be 1.1m. The absolute error is 0.1m, and the relative error is 0.1 or 10 percent.

Consider a building that is 100m in height but it is measured to be 101. The absolute error is 1m (greater that for the table) , but the relative error is now 1% (smaller than for the table).

Worked example
Given 4 bits what is the relative and absolute error if we try to represent the number 0.9_{10} using 4 bits

$0.111_2 = 0.875_{10}$

Absolute error: *actual number – represented number | 0.9 – 0.875 | =* $\underline{0.025}$
Relative error *= absolute error / actual number 0.025 / 0.9 = 0.028 =* $\underline{2.8\%}$

Range and Precision

For floating point representation the larger the number of bits used for the mantissa the greater the precision and the larger the number of bits used for the exponent the greater the range. Of course there is a trade off between the two given a fixed number of bits.

Consider a 12 bit floating point representation with 4 bits for the mantissa and 8 bits for the exponent.

Exponent range: 1000000 = -128 to 0111111 127.
The precision does not allow us to go below increments of 1/8 or 0.125.

Now consider the same 12 bit representation with 8 bits for the mantissa and 4 bits for the exponent
Exponent ranges from 1000=-8 to 0111 7
The precision allows us to go as far as 1/128 or 0.0078125

In fixed point binary the range and precision depends on the position of the implied point. Consider 8 bits:

For 0.0000000 then our range is 0 to 1.99 and with intervals of 0.0078125

For 0000000.0 then our range is 0 to 127 and intervals are 0.5 are less precise

Fixed point binary is quicker than floating point binary and requires fewer operations to perform calculations.

Underflow Number is too small to be represented in a number of bits. For instance you cannot represent the number 0.00390625 using fixed point 8 bits. The number would be represented as 0. Can occur when you divide a small number be a very large number.

Overflow Number is too large to be held. When the result of a sum is too large to be represented by your number system you might run out of space to represent it and end up storing a much smaller number. For instance 1024 would be too large to be held in 8 bits. Can occur when multiplying two very large numbers together.

Converting from denary to normalised floating point representation

Positive numbers
Example: Convert the following denary number 3.25 into a binary real number with the first 5 bits represent the mantissa and the remaining 3 bits represent the exponent
Convert to binary: 11.01
Normalised binary = 0.1101×2^2
5-bit mantissa: 0.1101
Exponent: 2 = 010
$\underline{01101\ 010}$

Negative Numbers
Example: Convert the following denary number -2.75_{10} into a binary real number with first 6 bits represent the mantissa and the remaining 4 bits represent the exponent

Convert to binary: 10.1100_2
Apply two's complement: 01.0100_2
Shift the point two place to the left until you get to the normalised number 1.01010_2
Exponent: is $2_{10} = 0010_2$
$\underline{1.01010\ 0010_2}$

Information Coding Systems

ASCII and Unicode

Character coding schemes allows text to be represented in the computer. One such coding scheme is **ASCII**. ASCII uses 7 bits to represent each character which means that a total of 128 characters can be represented.

Lower case letters	26
Upper case letters	26
Numbers	10
Symbols (e.g. comma, colon)	33
Control characters	33

ASCII encoded values for some characters

A	1000001_2	65_{10}
B	1000010_2	66_{10}
a	1100001_2	97_{10}
b	1100010_2	98_{10}
"0"	0110000_2	48_{10}
"1"	0110001_2	49_{10}

- ASCII has a limited character set (7 bits, 128 characters), but **Unicode** has 16 bits and allows many more (65K) characters.
- Unicode provides a unique character for different languages and different platforms.
- It allows us to represent different alphabets for instance Greek, Mandarin, Japanese, Emojis etc.
- Unicode and ASCII are the same up to 127.

We need to be able to differentiate between character code representation and binary representation.

For instance, the binary representation for 48 is $1\ 1\ 0\ 0\ 0\ 0_2$

The ASCII code for '48' is $0\ 1\ 1\ 0\ 1\ 0\ 0\ \ \ 0\ 1\ 1\ 1\ 0\ 0\ 0_2$

Error Checking and Correcting

When data are sent over a network between devices there needs to be a way to check that the bits have not been corrupted by interference during transmission and have been correctly received. Corruption occurs when a bit transmitted as a 0 is converted to a 1 and vice versa.

Transmitted signal	0 1 0 0 1 1 0 1
Received signal	0 1 0 1 1 1 0 1

Parity Bits
Parity bits allow us to check whether there is an error in a sequence of bits.
A parity bit is a check bit that is appended to a sequence of binary digits. The parity bit takes on a value of 0 or 1 according to whether the total number of 1s is odd or even.

Even Parity
For even parity the total number of 1s in the sequence including the parity bit needs to be even. In the following sequence we have 9 bits with a value of 1 therefore the parity bit is 1 for even parity giving is a total of 8 1s which is even.

1	0	1	1	1	1	1	0

Odd Parity
For odd parity the total number of 1s in the sequence including the parity bit needs to be odd. In the following sequence we have 7 bits with a value of 1 therefore the parity bit is 0 for odd parity giving is a total of 7 1s which is odd.

0	0	1	0	1	0	1	0

Finding Errors

If 1 bit in our sequence that was corrupted during transmission then the parity bit will be able to pick this up.

For instance, in the following sequence of transmitted data we use even parity to check the bits. we have 4 bits with a value of 1 and the parity bit a 1. Clearly there is a problem. We have an odd number of 1s for even parity which is not right so one of the bits has been corrupted.

1	1	1	0	1	0	1	0

Limitations of Parity bits

- Parity bits do not work if an even number of bits have been corrupted.
- Therefore only used for short sequences of bits where more that corrupted bit is unlikely.
- We do not know which bit is in error so we cannot fix, so we have to discard the whole sequence.

Majority Voting

With majority voting each bit is transmitted three times. If one of the bits is corrupted we take the majority as being the correct value for the bit.

The sequence: `0 1 0 0 1 1 1`

will be transmitted as: `000 111 000 000 111 111 111`

If one of the three bits is corrupted we take the majority as being the correct value for the bit. The values highlight below in red are the incorrect ones.

`000 101 010 000 111 111 111`

This approach means that 3 times the volume of data needs to be sent.

Check Digit

A check digit is an additional digit at the end of a sequence of digits to check for errors. For instance, a bar code contains 12 digits plus a check digit and the purpose of the check digit here is to ensure the bar code has been input correctly. In the below example we show how to calculate the check digit.

1. Multiply value by the weighting, this will be 1 or 3 and alternate for each digit
2. Add of the multiplication from all 12 numbers together
3. Calculate the modulus of 10 of the result

Worked example

Bar code	9	7	8	0	1	9	8	7	7	6	2	6
Weighting	1	3	1	3	1	3	1	3	1	3	1	3
Multiplication	9	+ 21	+ 8	+ 0	+ 1	+ 27	+ 8	+ 21	+ 7	+ 18	+ 2	+ 18

Add values = 140
140 MOD 10 = **0**

Checksum

A checksum is designed to check for errors in a block of data transmitted between computers over a network. Often when you download a large file you can calculate its checksum and compare this against a separate checksum that is made available alongside the file to ensure the file has not been corrupted or tampered with during transmission. If the checksums are the same then the file's integrity can be ensured, otherwise the file has been corrupted.

A Simple Checksum Algorithm

1. Split message into 8-bit blocks
2. add together
3. apply two's complement

`01110111 10001000 01100110`

	0	1	1	1	0	1	1	1
	1	0	0	0	1	0	0	0
+	0	1	1	0	0	1	1	0
(1)	0	1	1	0	0	1	0	1
	1	**0**	**0**	**1**	**1**	**0**	**1**	**1**

To validate

a) split into 8 bit blocks
b) add together all the sequences including the checksum
c) The result should be zero