UNIT 1.
INFORMATION REPRESENTATION

Computer systems
CFGS DAW

Alfredo Oltra

alfredo.oltra@ceedcv.es

2019/2020

Versión:220915.1154

Licencia

Nomenclatura

A lo largo de este tema se utilizarán distintos símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:

 Importante

 Atención

 Interesante

ÍNDICE DE CONTENIDO

UD01. INFORMATION REPRESENTATION

1. INTRODUCTION

1.1 Piece of information and information

Computers (or more correctly information systems) are machines designed for processing information or, in other words, to get results from the application of operations on a data set. But, what is information? What is a piece of information? And what is an operation?. Take an example:

The temperature is 30º

- Piece of information: formal representation of a concept, in this case: “*30*”
- Information: the result of the interpretation of the data: “*It's hot*”
- Operation: rule applied to get information: “A*s the temperature is higher than 23, it's hot*”

1.2 Data internal representation

Therefore, we need to store and to handle in computers data and operations. And for that, they need to use the binary code.

All kind of data, both numbers or letters, are stored using this system.

This system is based on the use of only two digits, 0 and 1, unlike the decimal system that uses ten (0, 1, 2, 3, 4, 5, 6, 7, 8, 9). This is because computers only know these two numerical values resulting from the detection or not of some potential, of a number of volts. Thus, a computer knows there's a 0 when the potential measured in an inner member has a value close to 0 volts. Otherwise, it detects a 1.

In electrical terms, the potential could be assimilated to the strength in which the electric current passes through a wire.

In general, values of 1 usually correspond to potential around 3 or 5 volts.

All computer elements handle this numbering and interpretation system of information. It might be said that computers actually know nothing at all. They only know about 0's and 1's and how to perform some basic operations with them (+,-, *...), although faster.

2. NUMERAL SYSTEMS

A numeral system is a set of **sorted symbols** used to represent quantities. The number of symbols is called **system base**.

In the real world, we are used to use decimal system (base 10) which set of sorted symbols are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Any number, represented in any numeral system, can be split in digits. For instance, 128 can be split in 1, 2, 8 or 34,76 in 3, 4, 7, 6. From these digits and with their position and the system base is possible to get again the number:

$$128 = 1 * 10^2 + 2 * 10^1 + 8 * 10^0$$

$$34{,}76 = 3 * 10^1 + 4 * 10^0 + 7 * 10^{-1} + 6 * 10^{-2}$$

We can see that a decimal number can be represented as additions of powers of 10 (the decimal system base).

If we generalize, a number **N** expressed in a numeral system *B* would be like:

$$N = a_{n-1}\ a_{n-2} \dots a_1\ a_0\ ,\ a_{-1}\ a_{-2}\ \dots.\ a_{-p+1}\quad a_{-p}$$

where:

N: number to represent

a: the symbols that our numeral system includes (integers from 0 to B-1)

The digits before the comma (,)[1] are the integer part.

The digits after the comma (,) are the fractional part.

2.1 Binary code

The binary code is a numeral system which system base is 2 and its symbols are 0 and 1.

 Each digit of a binary number is called **bit** and it is the smallest unit of information, in other words, it is the least that can be represented

 To avoid confusion, it is usual to indicate the base system number to be represented by a subscript to the right. For example $101_{(10}$ or $101_{(2}$

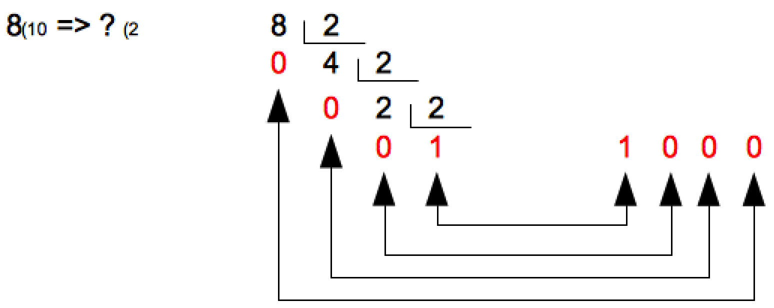
2.1.1 How to convert a decimal number into a binary number

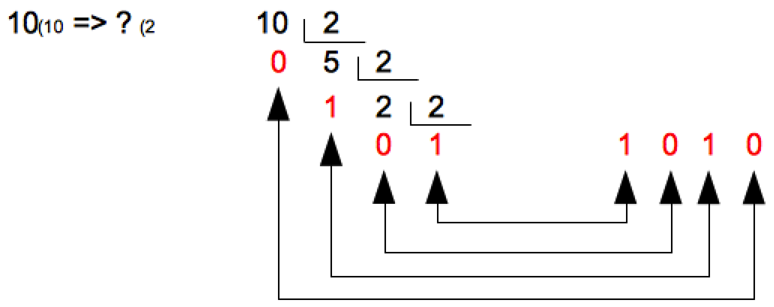
In general, to convert a decimal number into another base, we have to perform successive divisions of the number by the base. At the end, we have to get the

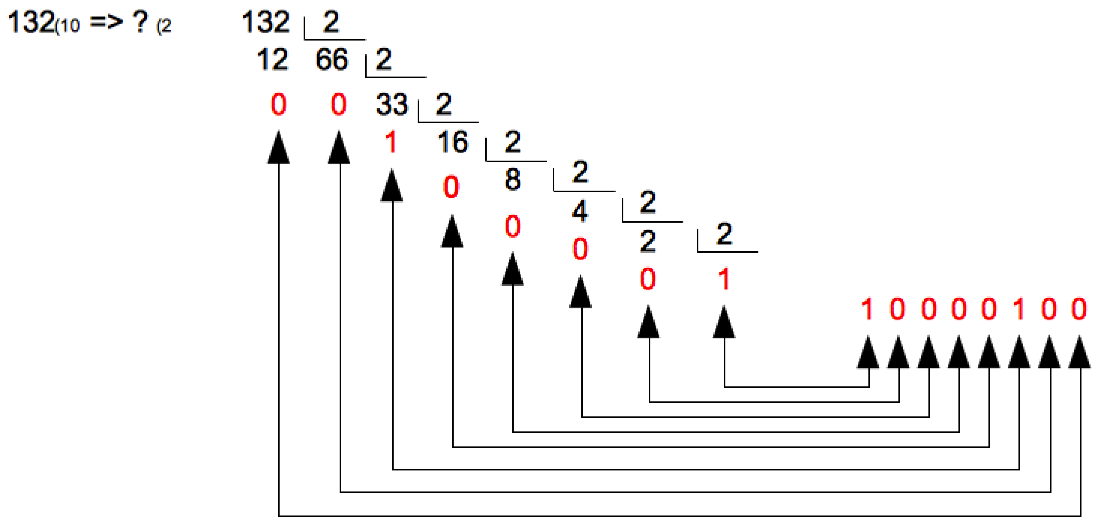
1 In the English culture, the separation between the decimal part and fractional part is a decimal point (.)

remainders and the last quotient and sorted them in the opposite direction.

Consider the case of convert a decimal into binary with some examples:

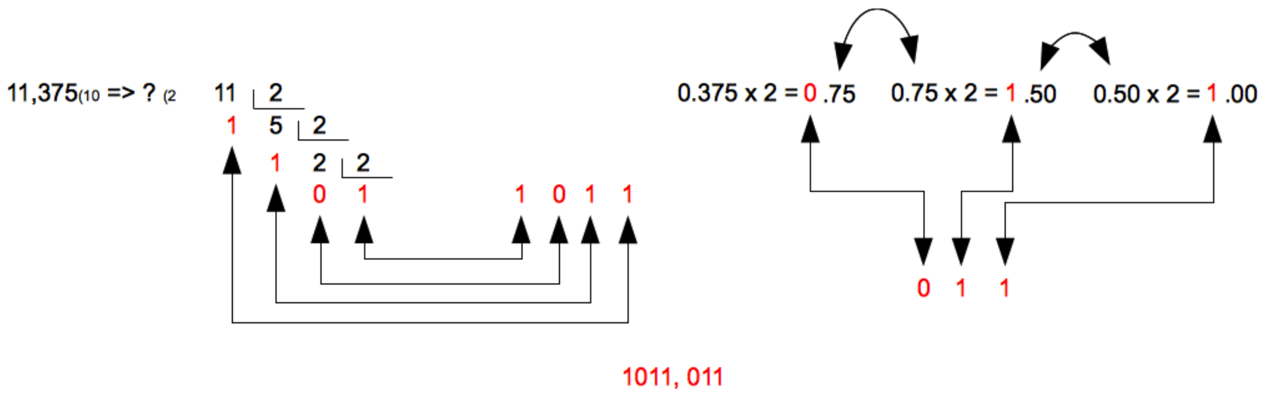






In numbers with fractional part, the process is the same for the integer part, but the fractional part is calculated multiplying by 2 successively and to take the integer part (in this case in right order).

The leftmost bit it is called most significant bit (MSB) and the rightmost bit it is called least significant bit (LSB).



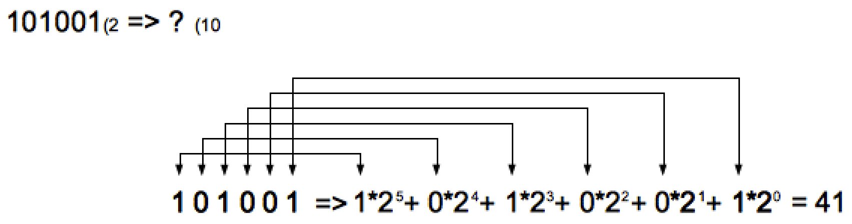
2.1.2 How to convert a binary number into a decimal number

In this case the process is very easy. As explained above, a decimal number can be represented as additions of powers of ten.

On the whole, it can convert the value of a number represented in a numeral system B[2] into decimal system using the next formula:

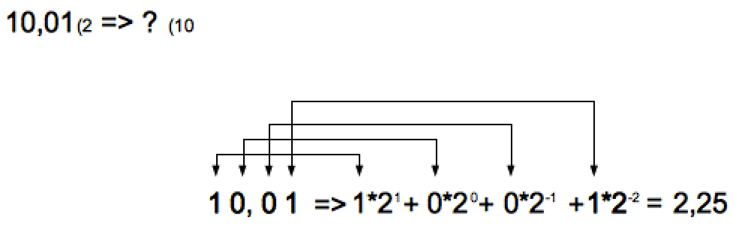
$$N = a_{n-1}B^{n-1}+a_{n-2}B^{n-2}+\ldots+a_1B^1+a_0B^0+a_{-1}B^{-1}+\ldots+a_{-p}B^{-p} = \sum_{i=-p}^{n-1} a_i B^i$$

We are going to use it to convert into base 2



The process involves four steps

1. To write the binary number figures multiplied by 2.
2. To write a plus sign (+) between each of products.
3. To write an exponent in each 2, starting from zero and from the last number of the integer part (on the far right if there is not fractional part) and increasing it one by one to the left and decreasing to the right.
4. To perform the operation



2 As we will see later, B can be any numeral system

2.1.3 Maximum number of values to represent

One of the typical questions when handling a binary number is to know what is the maximum decimal value that can be represented by a certain bits number. The answer is easy: 2^n, where n is the bit number. For instance, with 4 bits we can represent 16 values, from 0 to 15 (0000-1111)

2.1.4 Operations with binary numbers

Binary addition and subtraction follow the next rules:

Addition:

0 + 0 = 0

1 + 0 = 1

0 + 1 = 1

1 + 1 = 0 (carry 1)

Subtraction:

0 - 0 = 0

1 - 0 = 1

0 - 1 = 1 (carry 1 to *subtrahend*)

1 - 1 = 0

The operations result is the same as their related decimal operations, except for the cases where the result don't have a value in the binary system, that is 1+1, which can not be represented by 2 and 0-1, which can not represent by -1. This is where carry-over is important.

Some examples:

```
    11                111111
   10011010             1011
+  01001100         +  111101
   11100110          1001000
```

If we want to add two binary numbers which addition is greater than the maximum number to represent the computer throws an *overflow* warning. For instance, if we have a computer who works with 8 bits, it can represent from $0_{(10}$ to $255_{(10}$. If we want to add $10000000_{(2}$ ($128_{(10)}$) plus $10000000_{(2}$ ($128_{(10)}$) we have a problem because the result is $100000000_{(2}$ ($256_{(10)}$) greater than 255. So an *overflow* occurs.

```
  1
   10000000
+  10000000
  100000000
```

```
   101101              11101
   1                    11
-   10101          -    00111
   011000               10110
```

In the subtraction, the carry-over don't add to the minuend, but subtrahend.

Multiplication:

0 * 0 = 0

1 * 0 = 0

0 * 1 = 0

1 * 1 = 1

Division:

0 / 0 = Undefined

1 / 0 = Boundless

1 / 1 = 1

0 / 1 = 0

Both, multiplication and division, presents no difference from the related operations in decimal, unless the auxiliary operations are performed in binary.

In multiplication, when we add, it may be that we have in the same column more than two 1's. In this case, we perform the additions in groups of two and are going to carry the 1's in the next column.

In division, we start getting in dividend and divisor the same number of figures. If you can not be divided, we try getting one figure more in the dividend.

If the division is possible, then, the divisor can only be contained once in the dividend, that is, the first quotient figure is 1. In this case, the result of multiplying the divisor by 1 is the divisor itself (the value that we will subtract).

```
101010   |110
-110      111
 1001
  1
-1110
 00110
   110
   000
```

2.1.5 Negative numbers

When we need to represent a negative binary number, we have several options although three are the most important. This range indicates that the way to express it should be an agreement between two sides: the one that generates the number and one to read it. If not, the real value to be expressed would be wrong.

Signed magnitude

Perhaps it is the easiest approach to understand. The idea is to keep the MSB to indicate the sign of the number: 0 positive, 1 negative. The remaining bits indicate the number value in absolute value. For example:

Decimal	**Binary**	**Positive Binary**	**Negative Binary**
5	101	*0*101	*1*101

As can be seen, we need a bit to indicate the sign, so that what in normal representation values would be 0 to 15 in this case, to use the sign, becomes of -7 to +7 (1111 - 0111) .

This system is simple to understand but complex to use when performing mathematical operations. Besides it has a problem: there are two ways to define the $0_{(10}$: $0000_{(2}$ and $1000_{(2}$

Ones' complement

The second option also uses the first bit as sign indicator, but in this case the negative number is achieved complemented positive number (changing ones' by zeros and vice versa).

Decimal	**Binary**	**Positive Binary**	**Negative Binary**
5	0101	*0*101	*1010*

In this option is required to give the number of bits to encode, in such a way that if in the previous example we use 8 bits to encode:

Decimal	**Binary**	**Positive Binary**	**Negative Binary**
5	101	*0*0000101	*1*1111010

This method has the same problem that signed magnitude: there are two ways to define the $0_{(10}$: $0000_{(2}$ and $1111_{(2}$

Two's complement

Although ones' complement simplifies the mathematical operations, they do much more with the use of two's complement. That is why it is the most used method.

Two's complement consists in to apply a ones' complement and then, to add 1. For instance, two's complement of 5 encoded with 8 bits is:

$5_{(10} \rightarrow 101_{(2} \rightarrow$ (encoded in 8 bits) $00000101_{(2} \rightarrow$ (1's complement) $1111010_{(2} \rightarrow$ (+1) 11111011

Decimal	Binary	Positive Binary	Negative Binary
5	101	*0*0000101	*1*1111011

What decimal number represents a number in two's complement?. Easy. We have to preform the same process:

$11111011_{(2} \rightarrow$ (1's complement) $\rightarrow 00000100_{(2} \rightarrow$ (+1) $00000101_{(2} \rightarrow 5_{(10}$

The great advantage of the two's complement method is that it allows subtraction as if they were adds. This is because subtract two binary numbers is the same as adding to the minuend the complement of the subtrahend..

$101101_{(2}$ ($45_{(10}$) – $010101_{(2}$ ($21_{(10}$) ⇨ $010101_{(2}$ (1's complement) $\rightarrow 101010_{(2} \rightarrow$

$\rightarrow$ (+1) 101011 ⇨ 101101 + 101011

```
        101101
+       101011
      1011000(2   (24(10)  the last carry-over 1 is rejected
```

Excess-K or offset binary

Depending on the number of bits available, mid-range is dedicated for negative numbers and the other half (minus 1) to the positives (the zero value is in the middle). The new range will be [-K,K-1], where we can calculate by K = 2^{n-1} [3] . Once we have the permissible range, the smallest number is who has all its bits to 0. Let us see an example:

We have 3 bits for representing the number so we can represent 2^3 numbers, the range [0,7]. In this case, K will be $2^{3-1} = 2^2 = 4$, so the range with negative numbers will be [-4,3]. The smallest number -4 will be 000 and the biggest 3 will be 111. The complete board will be:

-4	-3	-2	-1	0	1	2	3
000	001	010	011	100	101	110	111

If we have a number in *Excess-K* and we know its decimal value, we need to subtract the value of the excess to the decimal value. For instance, if n = 8 and is K = 2^{n-1} = 128,

11001100 → 204 ; 204 - 128 = 76   or   00111100 → 60 ; 60 - 128 = -68

2.1.6 Real numbers

When we write a real number in a paper we use a decimal comma (or decimal point, it depends of the culture) to distinguish between integer part and fractional part. In a computer, the space to represent this kind of numbers is

3 There is another version of this method with K = $2^{n-1}-1$

divided in two areas: one for the integer part and one for the fractional part. There are two ways to denote the size of this areas (fields), and therefore, the comma position: *fixed point* and *floating point*.

Fixed point

In this notation, we assign a fixed size to the integer part and the fractional part of the number, in other words, a fixed place to the comma.

The advantage is that the process to perform basic operations is the same that integers numbers. However, this method does not take advantage of the capacity of representation format used. For instance, a computer with 8 bits to represent numbers, could use 5 bits for integer part and 3 for fractional part $b_7b_6b_5b_4b_3,b_2b_1b_0$.

In this case the maximum number to represent will be 01111,111 and the minimum (positive) 00000,001. If the decimal comma would be in a floating position, the range of positive numbers tu represent would be 011111111 – 0,0000001

Floating point

The range of numbers that can be represented in the fixed point format is insufficient for many applications, particularly for scientific applications which often use very large and very small numbers. To represent a wide range of numbers using relatively few digits, is well known in the decimal system, the scientific representation or exponential notation. For example, $0,00000025 = 2,5*10^{-7}$. in general, for any numbering system, a real number can be expressed as:

$$N = M * B^E \text{ or } N = (M;B;E)$$

where:

M: mantissa

B: base

E: exponent

The internal representation that makes this format on computers is known as floating point.

For instance in decimal (B=10) $259,75_{(10} = 0,25975*10^3$ or (0,25975;10;3) or, in binary code (B=2)

$$259,75_{(10} \rightarrow 100000011,11_{(2} \rightarrow 0,10000001111 * 2^9{}_{(2} \rightarrow 0,10000001111 * 2^{1001}{}_{(2} \rightarrow (0,10000001111;1001)$$

The representable numbers range for a given value of B, is fixed by the number of bits of the exponent E, while the accuracy is determined by the number of bits of M.

Normalization

The same real value can be represented in infinite ways by exponential notation. For example, 2,5 can be represented as $0,25*10^1$, $0,025*10^2$,$250*10^-$

^{2},... To avoid confusion, we should choose one of these formats as the standard for floating point representation of a real number. The form chosen is called *Normalized form* and it is one that maintains the highest accuracy in the representation of numbers. This is achieved when the binary point is located immediately to the left of the first significant digit, so that no space is wasted representing no significant digits.

For instance:

- 2,5 represented in normalized form is 0,25*10^{1}
- (0,000011101 ; 2 ; 0111) → (0,11101 ; 0011) [4] $\xrightarrow{\text{Exponent excess-k}}$ (0,11101 ; 1011)

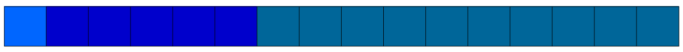
In general, to represent negative exponents the *excess-k* method is used. In the other hand, for represent negative *mantissas* signed magnitude method

- (100,11110 ; 2 ; 0010) → (0,10011110 ; 2 ; 0101) $\xrightarrow{\text{Exponent excess-k}}$ (0,10011110 ; 1101)
- (101,001 ; 2 ; 0100) → (0,1010010 ; 2 ; 0111) $\xrightarrow{\text{Exponent excess-k}}$ (0,10011110 ; 1111)

IEEE754

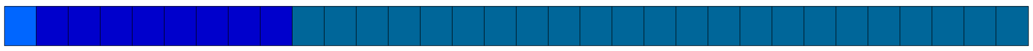
The most popular format for representing floating points in binary was developed by the *Institute of Electrical and Electronics Engineers* (IEEE) and it is called the IEEE754. This format can represent special cases such as infinite values and undefined results, *NaN* or *Not a Number* results. It proposes 3 formats:

Half precision. It uses 16 bits



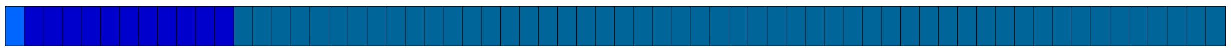
It uses 16 bits: One bit for the sign, 5 for the exponent, 10 for the mantissa

Simple precision.



It uses 32 bits: One bit for the sign, 8 for the exponent, 23 for the mantissa

Double precision.



It uses 64 bits: One bit for the sign, 11 for the exponent, 55 for the mantissa

🟦 Sign
■ Exponent
▨ Mantissa

All three formats use normalized mantissa, so the first bit on the left hand on the mantissa will be a 1 (the first significant digit have

4 We remove the base value because we assume that the system base is 2

to be a 1). Because of this, three formats do not encode this 1 in the mantissa, although it is taken into account when operating with the number. In other words, in these formats the MSB is on the left of the decimal comma and they only save the bits on right side.

To represent the exponent the standard uses the Excess-K method con K = 2^{n-1} -1

To better understand this operation is highly recommended to watch the pill 02 *Convert real number to binary code URL*

2.1.7 Boolean algebra

Besides mathematical operations (+,-,*/), on binary numbers can apply boolean or logical operations: and, or, xor, not...

To better understand these operations is worth to name 1 as *true* and 0 as *false*

NOT:

It can be represented in various ways: NOT, ¬

NOT 0 = 1

NOT 1 = 0

AND:

It can be represented in various ways: AND, Y, ^, *

0 AND 0 = 0
1 AND 0 = 0
0 AND 1 = 0
1 AND 1 = 1

In other words, the result will be true (1) only when both digits were *true*. As can be see, the result is the same that the multiplication.

```
     10011010            1011
AND 01001100      AND 111101
    00001000          001001
```

OR:

It can be represented in various ways: OR, O, v

0 OR 0 = 0

1 OR 0 = 1

0 OR 1 = 1

1 OR 1 = 1

In thIs case, the result will be *true* as soon as one of the digits was *true*.

```
       10011010              1011
    OR 01001100          OR  111101
       11011110              111111
```

XOR:

0 XOR 0 = 0

1 XOR 0 = 1

0 XOR 1 = 1

1 XOR 1 = 0

In this case, the result will be *true* when one and only one of the digits were *true*.

```
       10011010              1011
    XOR01001100          XOR 111101
       11010110              110110
```

2.2 Octal

Beside binary, there are two other interesting numeral systems when working on issues related to information technology: the octal and hexadecimal. This is because from them are easy to convert to binary.

The octal is a numeral system with a system base equal to 8 (symbols 0,1 ,2 ,3 ,4 ,5 ,6 ,7) . Its base is a exact power of binary system base 2^3=8 or, in other words, with three binary digits (with three bits) we can represent all the octal digits.

Binary	Octal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

2.2.1 How to convert a binary number into octal

The process lies in creating groups of threes bits, starting on the right hand, and replace them for the related octal value

$1101011_{(2} \Rightarrow 1 \quad 101 \quad 011 \Rightarrow 153_{(8}$

2.2.2 How to convert a octal number into binary

The process is reversed to the previous: it is converted to binary each of the numbers of octal number

$7402_{(8} \Rightarrow 111 \quad 100 \quad 000 \quad 010_{(2} = 111100000010_{(2}$

2.3 Hexadecimal

Its system base is 16. As the number of symbols used in the system is greater than 10, 6 characters must be used, in this case from A to F. Thus, the ordered set of symbols is: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

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

2.3.1 How to convert binary numbers into hexadecimal

The process is similar to the binary-octal process, except in this case groups are in fours.

$1101011_{(2} \Rightarrow 110 \quad 1011 \Rightarrow 6B_{(16}$

2.3.2 How to convert hexadecimal numbers into binary

The process is reversed to the previous: it is converted to binary each of the numbers of octal number

$7F0A_{(16} \Rightarrow 0111 \quad 1111 \quad 0000 \quad 1010_{(2} = 111111100001010_{(2}$

2.3.3 How to convert hexadecimal numbers into octal numbers

We convert into binary and group the bits in fours or threes, whichever is the numeral system destination

$6B_{(16} \Rightarrow 110 \quad 1011 \Rightarrow 1101011_{(2} \Rightarrow 1 \quad 101 \quad 011 \Rightarrow 153_{(8}$

 The conversion between octal or hexadecimal and decimal or vice versa, can be performed following the methods to convert between binary and decimal, but multiplying by power of 8 or 16 or dividing by these numbers and to get the remainders (in hexadecimal if the remainder is greater than 9 we get its related values A..F).

However, it is usually more practical to perform directly conversion into binary and then, convert into the system requested.

3. ALPHANUMERIC REPRESENTATION

3.1 Numeric and alphanumeric data.

A data is numeric if it is possible to perform mathematical operations. In contrast, a data is alphanumeric If you can NOT perform mathematical operations on it.

numeric : *how old are you?* **45**
alphanumeric: *What is your name?* "**Roberto**"

 In order to clearly differentiate between the two types of data, it is common to use single or double quotes to indicate that data is alphanumeric.

It is usual to think that numeric data are numbers and alphanumeric data are only letters. But this is not correct. For instance:

What is your address? "**Avenida de las Palmeras 34**"
What is your mobile number? **"555341273"**

In the first case, **Avenida de las Palmeras 34,** is composed of letters and numbers, and in the second**, 555341273,** only by numbers, but not operables (it makes no sense to add or multiply two phone numbers).

3.2 Internal representation

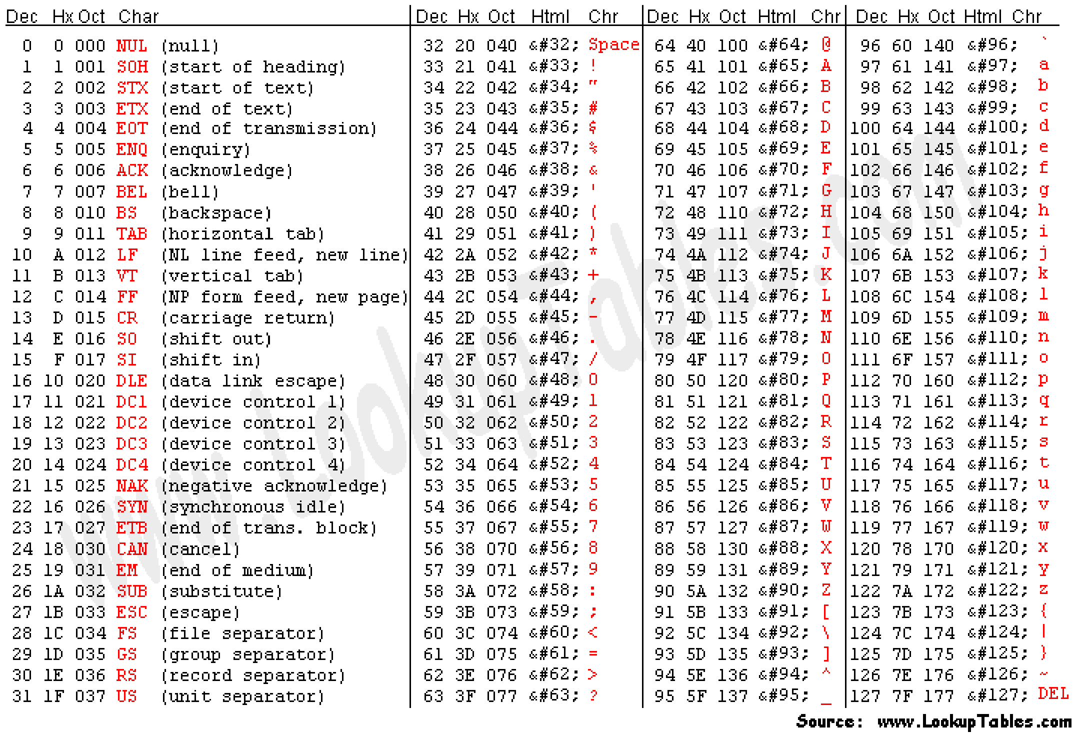
Alphanumeric characters to represent computers rely on tables, such that each of the table entries (each number) corresponds to an alphanumeric symbol.

Throughout the history of computing, there have been several tables that have always been characterized by the number of bits used to represent each character. One of the best examples in the ASCII table. The number of bits is 7, which left room for 128 characters (2^7=128)

As can be seen in the following table, each number is related with a character. For example, $73_{(10}$ is a "I", $105_{(10}$ is a "i" or $50_{(10}$ is a "2". The first entries are reserved for non-printable characters, those that are not visible, such as tabulator ($9_{(10}$) or carriage return ($15_{(10}$).

 The space is also a character: $32_{(10}$

The problem of this table is its limited space. As you can see, it has room for all the Latin spellings used in Anglo-Saxon languages, but we can not find spellings like the ñ, ç or accented vowels. Therefore, the extended ASCII table of 8 bits (256 characters) was created. This new table can incorporate all of Latin spellings plus some graphic symbols.



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

Source: www.LookupTables.com

 Today the ASCII tables are almost obsolete. The expansion of the Internet and globalization make necessary tables that incorporate not only Latin characters, but Chinese, Arabic, Korean, Russian, Hebrew...

Throughout the history of computing, there have been several t

3.3 Unicode

Unicode is a standard that can use more than one byte for each character, allowing a much wider range of representation. The most important elements of Unicode are:

- **Code point:** A "code point" represents a single character. They are written as U+nnnnnn. Unicode supports more than a million, and just under 100 thousand are currently allocated.
- **Glyph:** The glyph indicates how a code should be painted on the screen. Different glyph can exist for the same character (for example, different fonts types)
- **Codification:** There are different ways to represent a character:
  - UTF8: It has a variable length, it uses one to six bytes for each character (the most commons use only one byte)
  - UTF16: It's similar to UTF8, but a minimum of 2 bytes per character are used.
  - UTF32: It has a fixed length. 4 bytes are used for each character.

The Unicode collation algorithm (UCA) is an algorithm defined in Unicode Technical Report #10, which is a customizable method to produce binary keys from strings representing text in any writing system and language that can be represented with Unicode. These keys can then be efficiently byte-by-byte compared in order to collate or sort them according to the rules of the language, with options for ignoring case, accents, etc.

4. OTHER TYPES OF INFORMATION

The images and sound are continuous signals, with an infinite level of precision. To represent this type of signals, they must be transformed into discrete signals, that is, with limited precision. This process is called encoding, and information will be lost in it. With higher precision level, more signal will seems the original, but it will have a larger size.

4.1 Images

The usual way of representing images is to project them onto a grid of a given size and save the "color" of each of the squares on that grid. Each of these squares is called a pixel. The size of that grid is the resolution of the image.

For each one of the pixels, it is necessary to store its color. Two fundamental concepts com into play there:

- **Color Space:** Indicates which colors can be represented. There is a color space called CIELab, which contains all the colors that the human eye can see. Computing devices cannot represent all of those colors, but only a subset of them. The most common color spaces used in computing are sRGB and AdobeRGB.
- **Color model:** define how the colors of the color space will be represented, using tuples of numbers. The most common models are RGB (which indicates the levels of Red, Green (G) and Blue (B) in the image and is used for additive colors such as light), CYMK (used for subtractive colors such as ink)HSV and HSL.

To know what the color to show is, we should know what the model is and also what color space is being defined. On monitors, it is common to use RGB over the sRGB space, but for printing it is more common to use CYMK over AdobeRGB.

There are also vector formats to represent geometric images. These formats do not store pixels but a description of what needs to be drawn (for example, circle centered at 0.10 in red). There are many vector formats, such as ai, svg, etc. Vector formats can be scaled without problems, since they contain a description of the image and not the pixels that make it up.

4.2 Videos

The videos are represented by sequences of images that are slightly modified. This can be done this way since our brain, if we project them fast enough, will interpret them as a continuous signal and transform them into movement (this is not the case in other animal species).

Each image is called a frame. A video can be rendered with different number of frames per second, typically from 23.56 to 60.

Videos are usually compressed using different algorithms called Codecs, such as mpeg-2, h.s64 xvid or divx. Since the images between frames normally change little, video codecs can have very high efficiency.

The videos are stored in files with different formats. The format of the video file is called a container (for example: avi, divx or matroska) They are commonly identified by their extension. A container does not have to contain only video, they habitually contain videos and sounds. Likewise, it may contain information encoded with different codecs.

4.3 Sounds

A sound is a continuous wave of vibrations. Sampling is used to represent sound: the measurement of the intensity of the wave is taken at regular intervals. From these intensities, the original signal can be reproduced again.

The audible spectrum of the human ear ranges from 20hz to 20,000hz, depending on the age of the individual. Therefore, and according to the Nyquist-Shannon theorem, a sampling of twice the frequency of the signal is needed, in this case 40Khz (it is fun to check this by taking photos with high exposure to screens) A sampling of 44.1 is usually used Khz in digital formats.

The number of sample bits needed to store the sound depends on the signal-to-noise ratio of the sample. Usually, 16 bits per sample is enough.

When it comes to digitally representing sounds, three types of formats are used:

- **PCM or RAW formats:** contain the sample as it was received. They occupy a large size.
- **Compressed formats:** can be lossy (mp3, ogg) or lossless (flac) Lossy formats try to remove information not perceptible to the human ear, and can be very efficient. Lossless formats allow you to reproduce the signal as it was sampled.
- **Descriptive formats:** it would be the equivalent of vector formats in images: it contains the instructions for a synthesizer to interpret a series of notes: it is similar to a score that contains the notes, the instruments and the times.

5. UNIT SYSTEM

As discussed above, the bit is the smallest unit of information. Today we do not work at bit level, but in groups of bits (in the previous section we have seen that a character is encoded using 7 or 8 bits).

Because inside the computer everything is in binary code, an easy way to handle groups is to use some value that is a power of 2, being the most basic $2^3 = 8$. A group of 8 bits is named **byte**.

Today is not usual to use the byte group, but some multiple of it: *Kilobyte* (kB), *Megabyte* (MB), *Gigabyte* (GB), *Terabyte* (TB)... . In the *International System* these multiples are powers of 10 (they are based on the decimal system), but in computing powers of 2 are used. However, the trend is to use the International System, although it should be noted that the values are similar but not the same.

We use two different kinds of words to differentiate between the system units. When we talk about kilobyte we refer to decimal system and when we talk about *kibibytes* to the binary system.

The equivalences can be seen in the following table:

Nombre SI	SI	binario	Nombre binario
Kilobyte (kB)	10^3 bytes = 1000 bytes	2^{10} bytes = 1024 bytes	Kibibyte (kiB)
Megabyte (MB)	10^6 bytes = 1000 kB	2^{20} bytes = 1024^2 bytes	Mebibyte (MiB)

Gigabyte (GB)	10^9 bytes = 1000 MB	2^{30} bytes = 1024^3 bytes	Gibibyte (GiB)
Terabyte (TB)	10^{12} bytes = 1000 GB	2^{40} bytes = 1024^4 bytes	Tebibyte (TiB)
Petabye (PB)	10^{15} bytes = 1000 TB	2^{50} bytes = 1024^5 bytes	Pebibyte (PiB)
Exabyte (EB)	10^{18} bytes = 1000 PB	2^{60} bytes = 1024^6 bytes	Exbibyte (EiB)
Zetabyte (ZB)	10^{21} bytes = 1000 EB	2^{70} bytes = 1024^7 bytes	Zebibyte (ZiB)

Although usually is used indifferently and, even today is more common *International System*, the values that are represented are different: 1 MB are 1,000,000 bytes (one million bytes), while 1 MiB are 1,048,576 bytes

It is important to differentiate between kB and kb. The first refers to kilobyte, while the second kilobit, 8 times less.

Although most of the names and abbreviations of multiples have capital letters, the kilo is defined with a lowercase.

6. ADDITIONAL MATERIAL

[1] Glossary.

[2] Videos about binary operations.

[3] Exercises.

7. BIBLIOGRAPHY

[1] Wikipedia. Signed Number Representations
https://en.wikipedia.org/wiki/Signed_number_representations

[2] Wikipedia. Signed Number Representations
https://en.wikipedia.org/wiki/Signed_number_representations

[3] Sistemas Informáticos. Isabel Mª Jimenez Cumbreras. *Garceta*. 2012