Représentations numériques

Revenir aux bases

Représentations en *bases*

- Bases ou radix r :
  - utilisation des symboles de 0 à r-1
  - Coefficients des puissances de r
- Base 10 :

$$142 = 1 \times 10^2 + 4 \times 10^1 + 2 \times 10^0$$

- Base 10, nombre décimal

$$142.34 = 1 \times 10^2 + 4 \times 10^1 + 2 \times 10^0 + 3 \times 10^{-1} + 4 \times 10^{-2}$$

Généralisation

La formule générale pour la base 10 est bien entendu :

$$x = \sum_{i=-\infty}^{i=\infty} c_i.10^i, \text{avec } c_i \in \{0 \dots 9\}$$

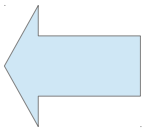
et se généralise pour toute base r :

$$x = \sum_{i=-\infty}^{i=\infty} c_i.r^i, \text{avec } c_i \in \{0 \dots (r-1)\}$$

Savoir compter et convertir

Compter en binaire

Decimal count	Binary count				
	16s	8s	4s	2s	1s
0					0
1					1
2				1	0
3				1	1
4			1	0	0
5			1	0	1
6			1	1	0
7			1	1	1
8		1	0	0	0
9		1	0	0	1
10		1	0	1	0
11		1	0	1	1
12		1	1	0	0
13		1	1	0	1
14		1	1	1	0
15		1	1	1	1
16	1	0	0	0	0
17	1	0	0	0	1
18	1	0	0	1	0
19	1	0	0	1	1
	2^4	2^3	2^2	2^1	2^0
	Powers of 2				



MSB LSB

A quoi ça sert ?

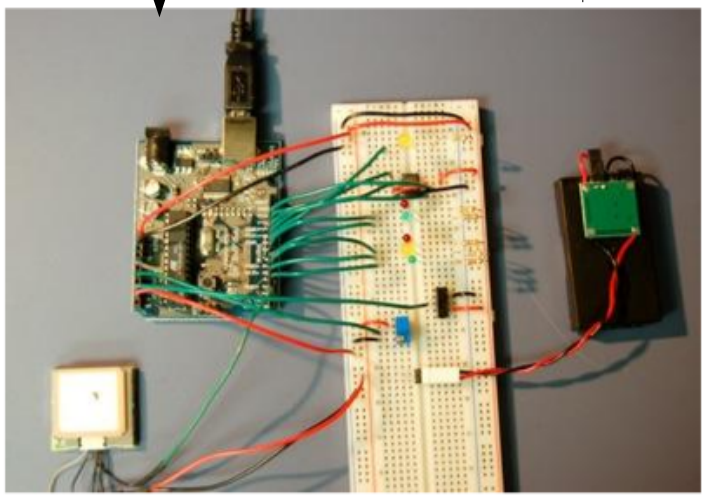
des variables de contrôle.....ou de statut

```
public class TcpClientSample
{
    public static void Main()
    {
        byte[] data = new byte[1024]; string input, stringData;
        TcpClient server;
        try{
            server = new TcpClient(" . . . . ", port);
        }catch (SocketException){
            Console.WriteLine("Unable to connect to server");
            return;
        }
        NetworkStream ns = server.GetStream();
        int recv = ns.Read(data, 0, data.Length);
        stringData = Encoding.
            ASCII.GetString(data, 0, recv);
        Console.WriteLine(stringData);
        while(true){
            input = Console.ReadLine();
            if (input == "exit") break;
```

"ctrl" "status"

bit (byte)

7 6 5 4 3 2 1 7 6 5 4 3 2 1



Applicable partout !

Conversions entre base

par tables

Decimal (Base 10)	*Binary (Base 2)*	*Octal (Base 8)*	*Hexadecimal (Base 16)*
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Conversions entre base

par tables

le nombre $101101_2 = 10\_1101_2 = 0010\_1101_2 = 2D_{16} = \mathbf{0x2D}$

Dans le cas de nombre fractionnaires, le sens de lecture s'inverse pour la partie fractionnaire. Ainsi :

le nombre $1011.01_2 = 1011.0100_2 = B.4_{16}$

On procède de même pour la traduction octale.

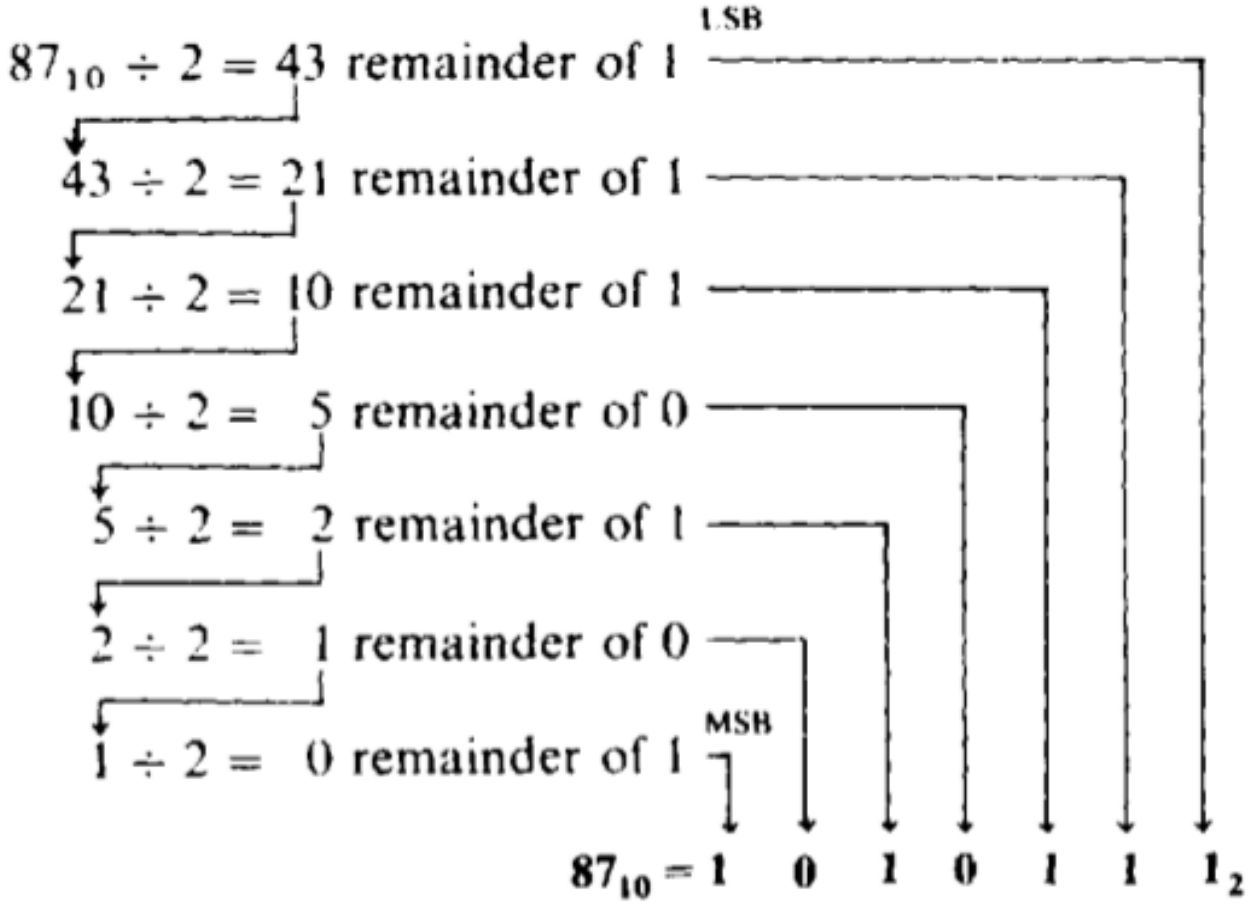
$(1011000110101 1.1111)_2 = (101\ 110\ 001\ 101\ 011\ .\ 111\ 100)_2 = (26153.74)_8$

4	7	.	F	E
↓	↓		↓	↓
0100	0111	.	1111	1110

$47.FE_{16} \simeq 1000111.1111111_2$

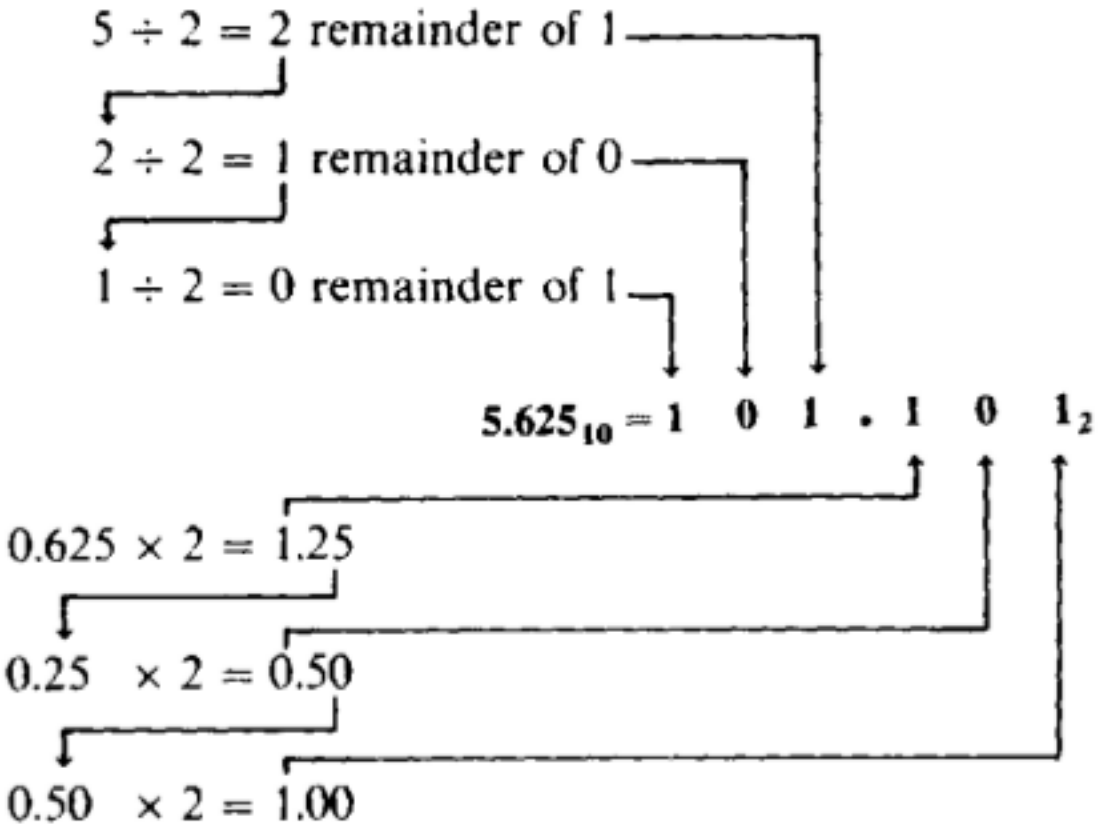
Conversions entre base

par calcul



Conversions entre base

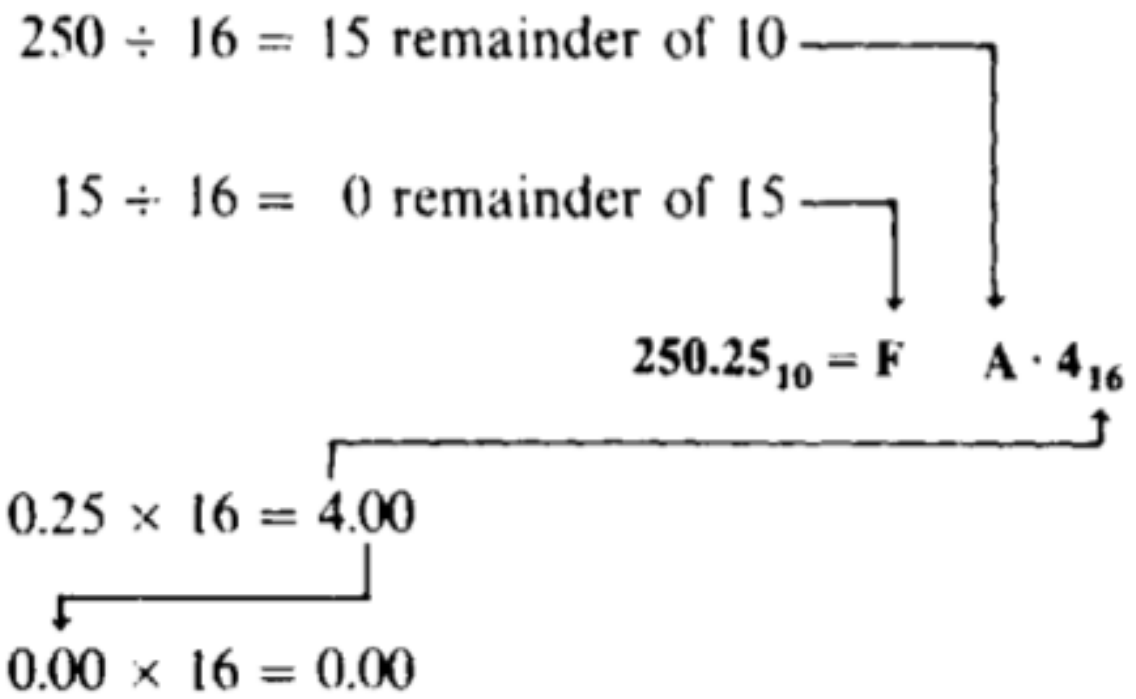
par calcul



Conversions entre base

par calcul

Même principe pour la base hexadécimale



Autres codes binaires

BCD : binaire codé décimal

Chiffre	Quartet	Chiffre	Quartet
0	0000	5	0101
1	0001	6	0110
2	0010	7	0111
3	0011	8	1000
4	0100	9	1001

Chaque chiffre de 0 à 9 est ici représenté par 4 bits.
Ceci simplifie la conversion : elle revient à étudier la juxtaposition des bits

Ex : 123 = 0001 0010 0011

Attention ! C'est simple, mais ça ne marche qu'en codage BCD, peu utilisé...

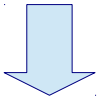
Autres codes binaires

Code de Gray ou code réfléchi

décimal	binaire classique	Gray
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100

$0 \rightarrow 0000$

$1 \rightarrow 0001$



$0 \rightarrow 0000$

$1 \rightarrow 0001$

$- - - - - -$

$2 \rightarrow 0011$

$3 \rightarrow 0010$

Il existe un algorithme pour passer d'un nombre x à l'autre $x+1$:

- on calcule le nombre de 1 dans x. On inverse le dernier bit de x quand ce nombre de 1 est pair.
- si le nombre de 1 est impair, on inverse le bit à gauche du 1 qui est le plus à droite.

Codage binaire des entiers *positifs ou « unsigned »*
résumé

Soit un nombre W de bits et soit $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$ de bits. Nous avons vu précédemment (sans la nommer) que la simple fonction :

$$B2U_w(\vec{x}) \doteq \sum_{i=0}^{w-1} x_i 2^i$$

définit la valeur non-signée ("unsigned") de $\vec{x}$, dans $\mathbb{N}$. Les valeurs limites de cette fonction sont :

$$\begin{cases} UMax_w &= \sum_{i=0}^{w-1} 2^i = 2^w - 1 \\ UMmin_w &= \sum_{i=0}^{w-1} 0 = 0 \end{cases}$$

On a donc :

$$B2U_w : \{0,1\}^w \rightarrow \{0, \ldots, 2^w - 1\}$$

Nombres signés

Complément à 2 sur w bits

De manière similaire, on définit désormais la fonction qui permet de calculer la valeur du vecteur $\vec{x}$ dans le cas d'une valeur signée, c'est-à-dire négative ou positive, **en complément à 2**.

$$B2T_w(\vec{x}) \doteq -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$$

Désormais, le bit de poids fort, le plus à gauche, contribue de manière négative à la somme totale. Précisons également que le 'T' vient de "two's complement". Calculons les valeurs min et max de cette fonction. Pour $\vec{x} = [10\ldots0]$ et $\vec{x} = [01\ldots1]$ respectivement, on trouve [6] :

$$\begin{cases} TMin_w &= -2^{w-1} \\ TMax_w &= \sum_{i=0}^{w-2} 2^i = 2^{w-1} - 1 \end{cases}$$

(! erreur dans le poly)

On a donc :

$$B2U_w : \{0,1\}^w \to \{-2^{w-1}, \ldots, 2^{w-1} - 1\}$$

On pourra également dénoter cette fonction $C_{2,w}(\vec{x})$.

Complément à 2 sur w bits

signé	bit vector	non-signé
-4	100	4
-3	101	5
-2	110	6
-1	111	7
0	000	0
1	001	1
2	010	2
3	011	3

Nombres négatifs

Le complément à 1 d'un nombre binaire consite à simplement inverser tous ses bits.

Le complément à 2 d'un nombre binaire consiste à :

- laisser inchangé le 1 le plus à droite, ainsi que tous les 0 les plus à droite.
- modifier tous les autres digits.

Il existe toute fois une autre méthode, qui se retient plus facilement (peut-être). Elle consiste en une formule simple :

$$-X = /X + 1$$

Attention !!!!
-134 nécessite 9 bits
et non 8

Addition classique (et non « OU » binaire)

Représentations obsolètes

La représentation en magnitude signée est simplement :

$$B2S_w(\vec{x}) \doteq (-1)^{x_{w-1}} \sum_{i=0}^{w-2} x_i 2^i$$

Il existe également une représentation des nombres en complément à 1, très légèrement différente du complément à 2.

$$B2T_w(\vec{x}) \doteq -x_{w-1}(2^{w-1} - 1) + \sum_{i=0}^{w-2} x_i 2^i$$

Ces représentations proposent 2 encodages pour 0 !

Dans le cas de la magnitude signée :
+0 = 0….0 et -0 =10...0

Nombres négatifs

résumé

Signed decimal	8-bit 2s complement representation		
+127	0	111	1111
+126	0	111	1110
+125	0	111	1101
+124	0	111	1100
⋮	⋮	⋮	⋮
+5	0	000	0101
+4	0	000	0100
+3	0	000	0011
+2	0	000	0010
+1	0	000	0001
+0	0	000	0000
−1	1	111	1111
−2	1	111	1110
−3	1	111	1101
−4	1	111	1100
−5	1	111	1011
⋮	⋮	⋮	⋮
−125	1	000	0011
−126	1	000	0010
−127	1	000	0001
−128	1	000	0000
	Sign	Magnitude	

same as binary numbers

-X= /X + 1

-125 ? =

125 = 111 1101

/125 = **1** 000 0010

/125+1= 1 000 0011

Noter que les nombres en complément à 2 sont négatifs s'ils présentent un MSB à 1 !

125 se représente sur 7 bits, mais pas -125 : il faut 8 bits

Exemple : -3 sur 3 bits

- $3_{10} = 011_2$
- $-3_{10} = /3_{10} + 1 = 100_2 + 1 = 101_2 = 5_{10}$????
- -3 vaut -il 5 ???
  - NON !
  - 5 ne peut pas être représenté sur 3 bits, si on y représente également les nombres négatifs

+3	0 11
+2	0 10
+1	0 01
+0	0 00
-1	1 11
-2	1 10
-3	1 01
-4	1 00

Pour un nombre donné de bits ,
il faut s'entendre sur le fait
que la représentation d'un nombre est signée ou non

Addition et soustraction

Addition : classique

	Addition
calcul	0110 + 0011
retenus	11
résultat	1001

Soustraction : x - y

1. on additionne x et le complément à 2 de y
2. si le résulat présente une retenue finale, on l'oublie!
3. s'il n'y a pas de retenue finale, on prend le complément à 2 du résultat et on palce un signe "-" devant le nombre.

exemple : $x = 1101100_2$ et $y = 1011011_2$

$$1101100 + 0100101 = 1\ 0010001$$

Le résultat est donc $x - y = 0010001_2$

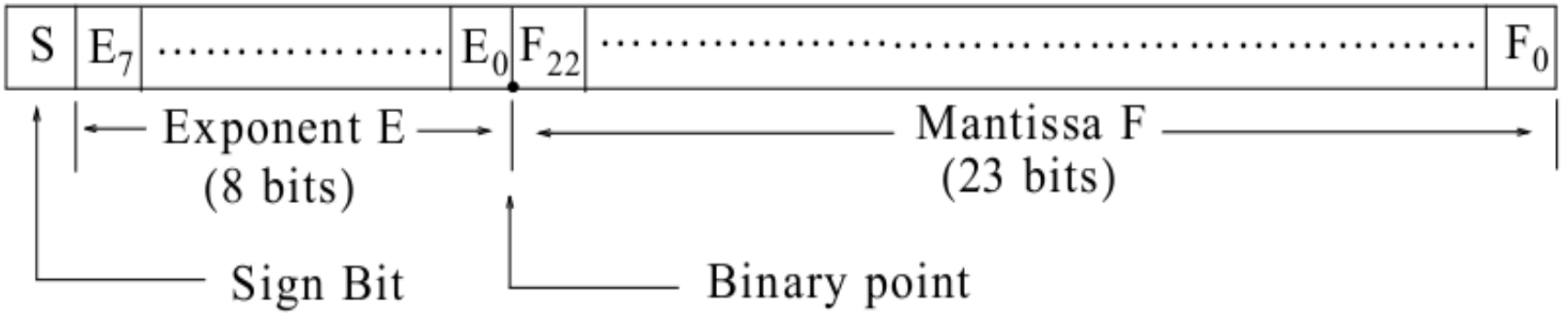
Nombre en virgule fixe

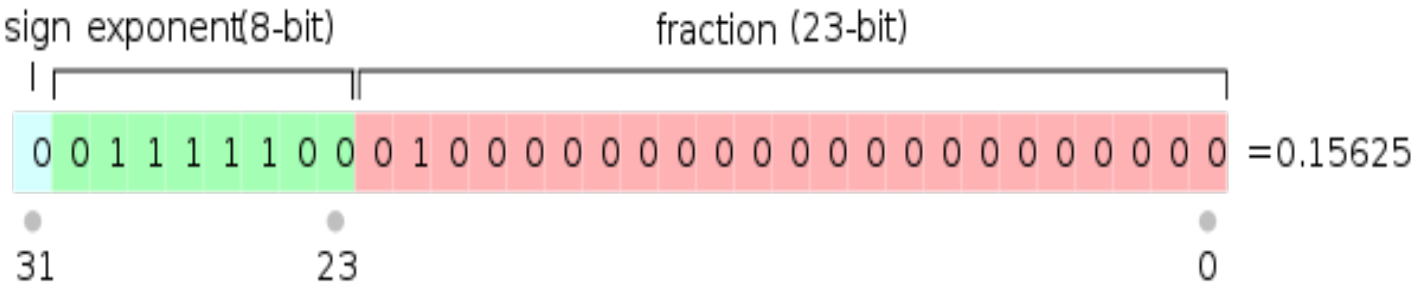
fixed point

- Déja vu précédemment...
- Le calculateur numérique devra connaître la position de la virgule, fixée une bonne fois pour toute.
- Impact sur les opérations classiques :
  - Le résultat de l'addition hérite de la position de la virgule identique à celle dans les opérandes
  - Cela est changé pour les autres opérations.

Nombres en virgule flottante

ieee754





Exo : calculer le nombre max et min représentable

IEEE 754 (SP)

- L'exposant est biaisé (en excédant) à 127. C'est-à-dire que pour coder un exposant e, on code le nombre e+127.
- Les exposants représentables vont de -126 à 127 (les valeurs 00000000 et 11111111 sont réservées pour des cas exceptionnels)
- Les nombres sont normalisés, c'est-à-dire que la mantisse est entre 1 et 2.

IEEE 754
cas exceptionnels

- Si e=11111111_2 et m=0, le nombre représenté est + ou - l'infini. Les opérations sur l'infini sont gérés correctement.
- Si e=11111111_2 et m/=0, la représentation est considérée comme n'étant pas un nombre (NaN : Not a Number).
- Si e=0 et m=0 on représente le nombre +0 ou -0 selon le signe.
- Si e=0 et m\neq0 on dit que le nombre est dénormalisé. C'est-à-dire que la mantisse est écrite sans le 1 implicite.

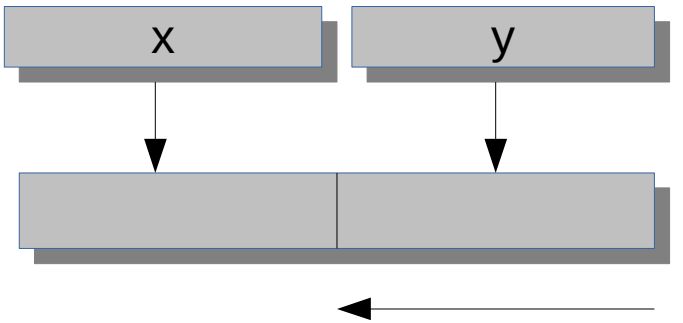
Données symboliques

- Certaines données ne sont pas numériques
  - Enumérations, collections : couleurs,alphabets,...
  - Ou compositions de données numériques
- Un encodage de ces données est nécessaire

Enumération

$$\begin{aligned} \text{bleu} &\to 0 \to 000 \\ \text{blanc} &\to 1 \to 001 \\ \text{rouge} &\to 2 \to 010 \\ \text{vert} &\to 3 \to 011 \\ \text{noir} &\to 4 \to 100 \end{aligned}$$

paire



pos = (x << 8) + y

Données symboliques

code ASCII

- Encodage des caractères usuels

b_7 →					0	0	0	0	1	1	1	1
b_6 →					0	0	1	1	0	0	1	1
b_5 →					0	1	0	1	0	1	0	1
Bits	b_4 ↓	b_3 ↓	b_2 ↓	b_1 ↓	Row ↓ / Column →	0	1	2	3	4	5	6	7
	0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
	0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
	0	0	1	0	2	STX	DC2	"	2	B	R	b	r
	0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
	0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
	0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
	0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
	0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
	1	0	0	0	8	BS	CAN	(	8	H	X	h	x
	1	0	0	1	9	HT	EM	)	9	I	Y	i	y
	1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
	1	0	1	1	11	VT	ESC	+	;	K	[	k	{
	1	1	0	0	12	FF	FC	,	<	L	\	l	|
	1	1	0	1	13	CR	GS	-	=	M	]	m	}
	1	1	1	0	14	SO	RS	.	>	N	^	n	~
	1	1	1	1	15	SI	US	/	?	O	_	o	DEL

ASCII Art

Données symboliques

Utf8 - Unicode

Définition du nombre d'octets utilisés

Représentation binaire UTF-8	Signification
0xxxxxxx	1 octet codant 1 à 7 bits
110xxxxx **10**xxxxxx	2 octets codant 8 à 11 bits
1110xxxx **10**xxxxxx **10**xxxxxx	3 octets codant 12 à 16 bits
11110xxx **10**xxxxxx **10**xxxxxx **10**xxxxxx	4 octets codant 17 à 21 bits

Ce principe pourrait être étendu jusqu'à huit octets pour un seul point de code, mais UTF-8 pose la limite à quatre[1].

Character		Binary code point	Binary UTF-8	Hexadecimal UTF-8
$	U+0024	0100100	00100100	24
¢	U+00A2	000 10100010	11000010 10100010	C2 A2
€	U+20AC	00100000 10101100	11100010 10000010 10101100	E2 82 AC
𤭢	U+24B62	00010 01001011 01100010	11110000 10100100 10101101 10100010	F0 A4 AD A2

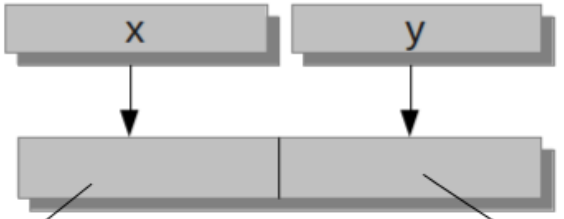
wikipedia

Manipulations binaires

récupération d'un champ binaire

pos = (x << 8) + y

Comment récupérer y ?



Il faut annuler cette composante

...tout en gardant celle-là

y= pos && 0x00FF

0000 0000 1111 1111 Masque

ex :

pos = 1011 0111 1110 1010

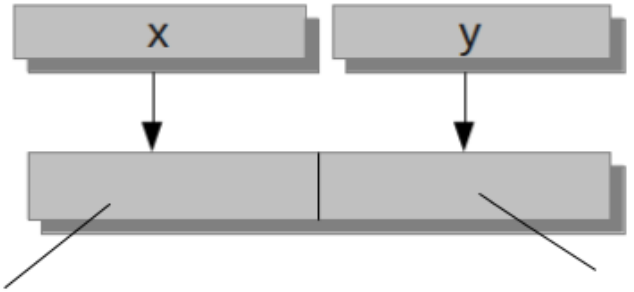
y = 0000 0000 1110 1010

Opérations dites « bit-à-bit »

Exemple (suite)

pos = (x << 8) + y

Comment récupérer x

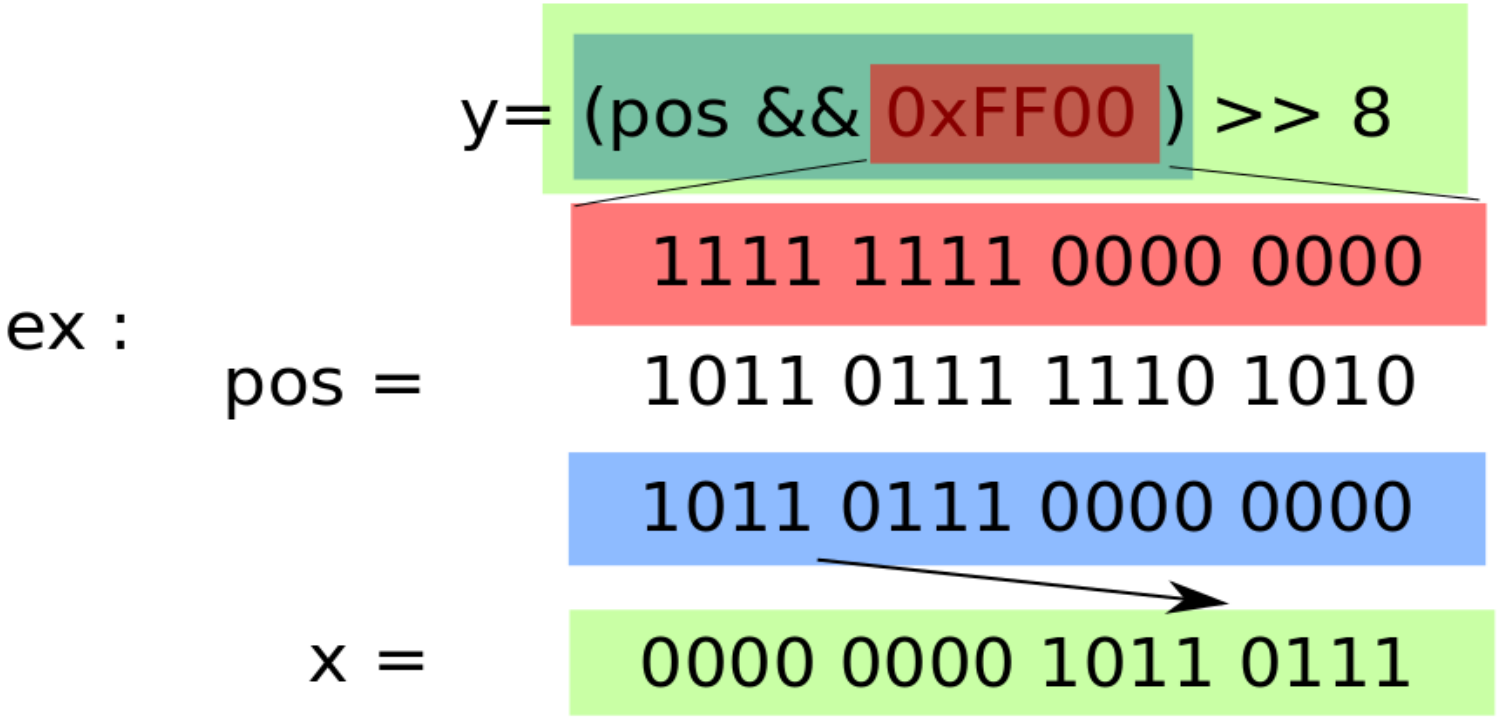


..tout en gardant celle-là

Il faut annuler cette composante

...et la décaller à droite

Exemple (suite et fin)



On peut faire plus simple.
Il suffit de décaler de 8 bits sur la gauche

(rappel) A quoi ça sert ?

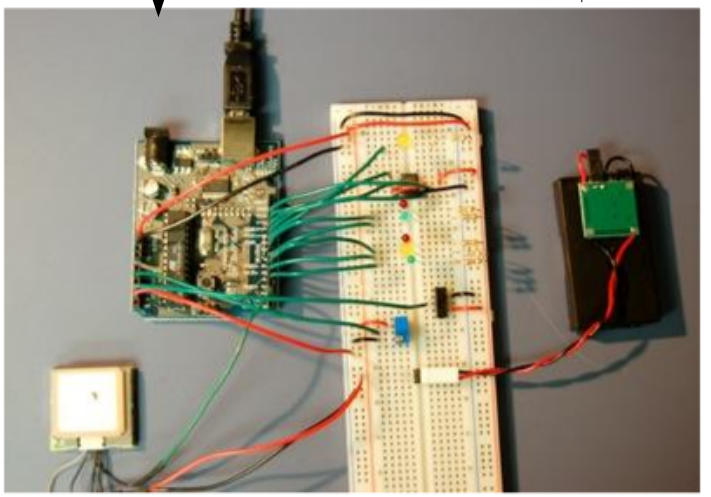
des variables de contrôle.....ou de statut

```
public class TcpClientSample
{
    public static void Main()
    {
        byte[] data = new byte[1024]; string input, stringData;
        TcpClient server;
        try{
            server = new TcpClient(" . . . . ", port);
        }catch (SocketException){
            Console.WriteLine("Unable to connect to server");
            return;
        }
        NetworkStream ns = server.GetStream();
        int recv = ns.Read(data, 0, data.Length);
        stringData = Encoding.
        ASCII.GetString(data, 0, recv);
        Console.WriteLine(stringData);
        while(true){
            input = Console.ReadLine();
            if (input == "exit") break;
```

"ctrl" "status"

bit 8 7 6 5 4 3 2 1 8 7 6 5 4 3 2 1

(byte)



Applicable partout !

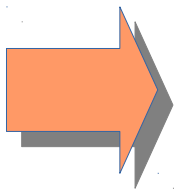
Manipulations binaires

Dans les langages **traditionnels**, les bits ne sont accessibles qu'à travers des manipulations des entiers

Listing 2.1– Quelques manipulations de bits en C

```
a = a | 0x4; /* mise a 1 du bit 2 de la variable a*/
a |=0x4; /* idem*/
b &= ~(0x4) /* mise a 0 du bit 2 */
b &= ~(1 << 2) /* idem, mais plus explicite*/
c ^= ~(1 << 5) /* inversion du bit 5*/
e >>=2 /* division de e par 4 */
```

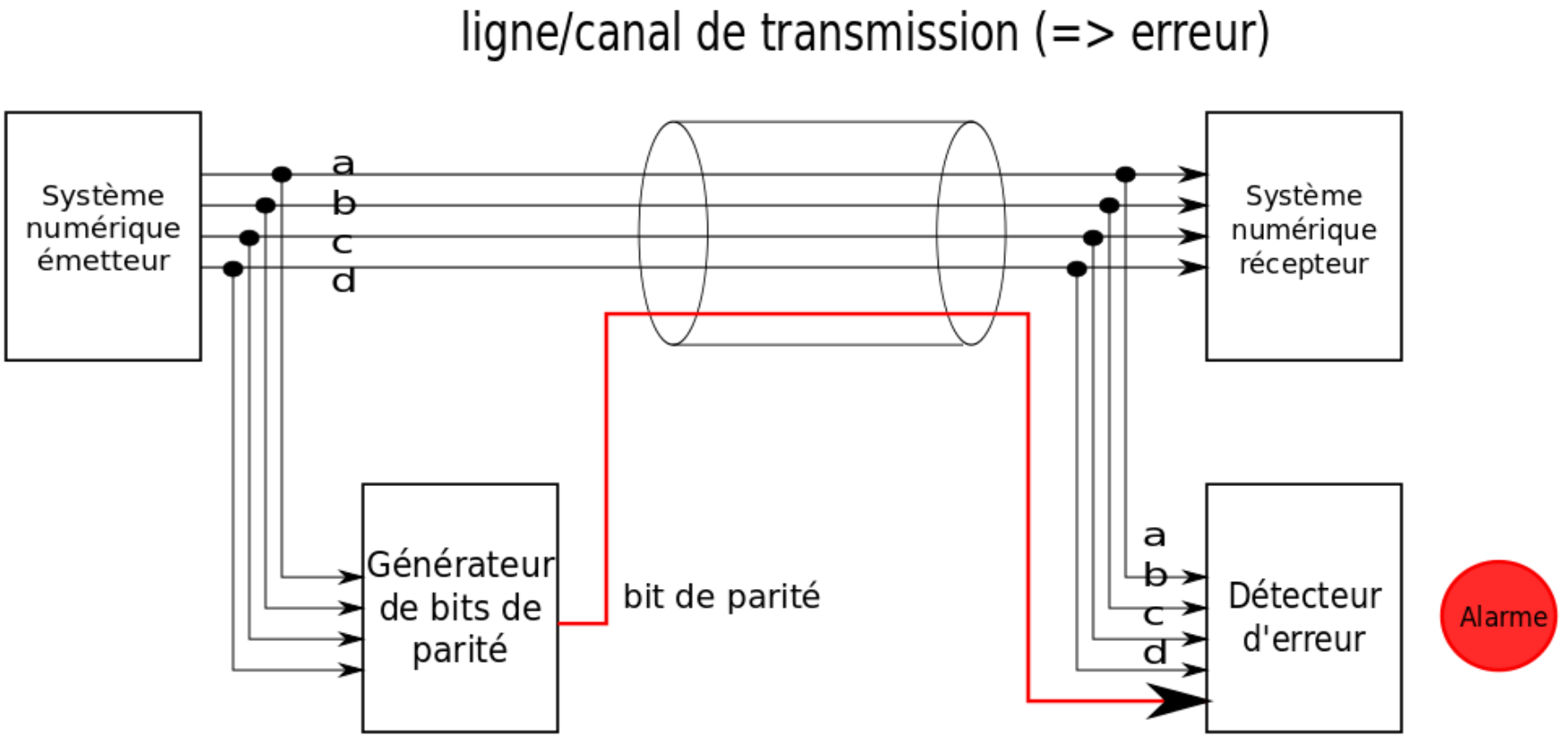
Attention : a sera un entier ! On ne verra pas explicitement ses 0 et 1 binaires !



Essentiel pour la programmation des dispositifs embarqués

Code détecteurs d'erreurs

bit de parité



Code détecteurs d'erreurs

suite

Entrées	Sortie
D C B A	P
0 0 0 0	0
0 0 0 1	1
0 0 1 0	1
0 0 1 1	0
0 1 0 0	1
0 1 0 1	0
0 1 1 0	0
0 1 1 1	1
1 0 0 0	1
1 0 0 1	0
1 0 1 0	0
1 0 1 1	1
1 1 0 0	0
1 1 0 1	1
1 1 1 1	0

mot | bit de parité

code

La sortie est fonction des entrées, et des entrées seulement.

C'est une fonction combinatoire

A propos des codes correcteurs

- Codes de Hamming
- Code de Goppa
- Code de Golay
- Code de Reed-Solomon
- …
- Turbo codes
  - Permet de s'approcher de la limite de Shannon