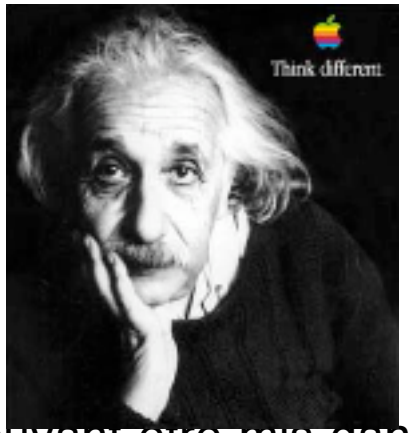
2 CODAGE DES INFORMATIONS



DU BIT A L'OCTET

L'électronique fournit des circuits électriques pouvant être mis dans un état ouvert ou fermé, allumé ou éteint, +5v ou 0v; on peut ainsi représenter deux états: faux ou vrai, 0 ou 1, et on dit qu'**on code l'information sur un bit**.

De même que l'oeil peut percevoir des informations complexes dans des séries "binaires" de points noirs et blancs d'une image (par exemple le groupe de points A peut être lu comme étant associé à la lettre "A" de l'alphabet, ou comme 3 segments formant un dessin vu en perspective), de même on peut imaginer regrouper des suites de bits en paquets pour représenter beaucoup plus que de simples états binaires.

De l'écran (N&B), l'ordinateur ne connait que 800 lignes de 600 points noirs ou blancs mis bout à bout. C'est illisible pour lui, mais pas pour nous qui les voyons **regroupées** les unes au dessous des autres pour former une image bidimensionnelle. Nous allons voir ce qu'il est possible de faire avec des regroupements de bits de plus en plus complexes.

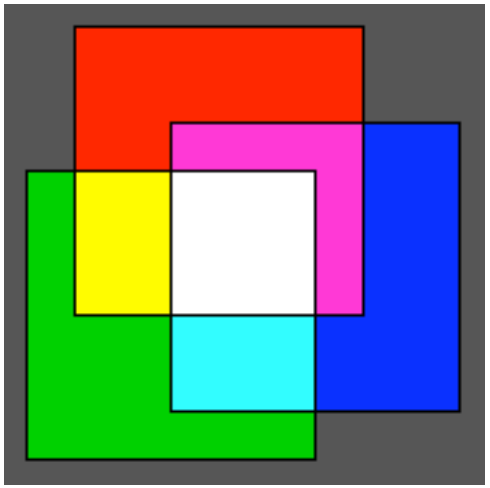
En regroupant **2** bits, on obtient 4 combinaisons notées de 0 à 3:

```
0      0  0          -> noir
1      0  1          -> gris foncé
2      1  0          -> gris clair
3      1  1          -> blanc
```

exemple: le premier peut être associé à la couleur noire, le dernier à la couleur blanche, les deux intermédiaires au gris clair et au gris foncé.

En regroupant **3** bits, on obtient 8 combinaisons notées de 0 à 7:

```
0      0  0  0      -> noir
1      0  0  1      -> bleu
2      0  1  0      -> vert
3      0  1  1      -> cyan
4      1  0  0      -> rouge
5      1  0  1      -> magenta
6      1  1  0      -> jaune
7      1  1  1      -> blanc
```



exemple: notez à droite une correspondance couleur, permettant de coder les 3 couleurs primaires, les couleurs complémentaires et le noir et blanc. Total 8.

En regroupant **4** bits, on obtient 16 combinaisons notées en utilisant les symboles de 0 à 9, puis au delà les 6 premières lettres de l'alphabet:

0	0	0	0	0	(codage hexadécimal)
1	0	0	0	1	
2	0	0	1	0	
3	0	0	1	1	
4	0	1	0	0	
5	0	1	0	1	
6	0	1	1	0	
7	0	1	1	1	
8	1	0	0	0	
9	1	0	0	1	
A	1	0	1	0	
B	1	0	1	1	
C	1	1	0	0	
D	1	1	0	1	
E	1	1	1	0	
F	1	1	1	1	

La lettre F représente ainsi le nombre 15.

Comment compte-t'on en hexadécimal ? Comme en décimal:

SYSTEME DECIMAL

00	10	..	90
01	11	..	91
02	12	..	92
03	13	..	93
04	14	..	
05	15	..	
06	16	..	
07	17	..	
08	18	..	98
09	19	..	99
			total 100

SYSTEME HEXADECIMAL

00	10	20	..	E0	F0
01	11				
02	12				
03	13				
04	14	24	..	E4	
05	15				
06	16				
07	17				
08	18				
09	19	29	..	E9	
	0A	1A	2A	..	EA
0B	1B	2B	..	EB	FB
0C	0C				
0D	0D				
0E	1E		..	EE	FE
0F	1F	2F	..	EF	FF
					total 256

En regroupant sur **8** bits, on obtient 2^8 = 256 états:

```
0    0000 0000        (00)
1    0000 0001        (01)
2    0000 0010        (02)
3    0000 0011        (03)
4    0000 0100        (04)
     ..................
126  0111 1110        (7E) ( 7 et E )
127  0111 1111        (7F) ( 7 et F )
128  1000 0000        (80)
     ..................
253  1111 1101        (FD)
254  1111 1110        (FE)
255  1111 1111        (FF)
```

qu'on peut représenter de façon plus compacte en utilisant le codage hexadécimal indiqué à droite entre parenthèses.

Noter qu'il suffit de connaitre les 16 premiers hexadécimaux pour calculer un octet.

Par exemple le nombre binaire **10000010**
est lu comme **1000 et 0010,**
le premier associé à **8**, le second à **2**,
donc le tout correspond en hexadécimal à **82**.

Inversement le nombre hexadécimal FA
est lu comme F et A,
le premier associé à 15, le second à 10 (...,8,9,A,B,...)
ce qui donne en binaire **1111 et 1010**
soit **11111010.**

On verra l'utilité dans un moment. Patience !

Le groupe de 8 bits, encore appelé **octet**, est ainsi l'unité fondamentale pour coder tous les types d'information en informatique, le bit n'étant utilisé que dans des cas très particuliers.

On est conduit à définir des multiples de cette unité:

1 ko	**$=2^{10}$**	**= 1024 octets**
1 Mo	**$=2^{20}$**	**= 1024 ko**
1 Go	**$=2^{30}$**	**= 1024 Mo**

Exemple:

Un format classique des logiciels bitmap définit la **page A4 de dessin noir et blanc** sur 720 lignes de 576 points (pourquoi 576, parce que 576 = 72 * 8 = 72 octets).

Chaque point peut être noir ou blanc et "pèse" 1 bit.
Le "poids" de la page est donc 576 * 720 bits = 414720 bits soit 51840 octets soit **50,625 ko**.

REPRESENTATION DES DONNEES

CARACTERES ALPHANUMERIQUES

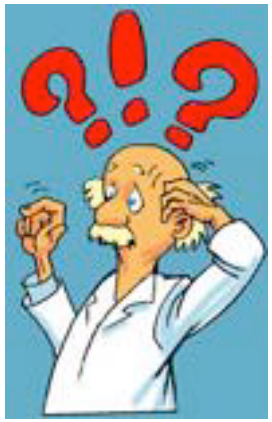
NOMBRES ENTIERS

NOMBRES FRACTIONNAIRES

NOMBRES REELS

LES INSTRUCTIONS

LES ADRESSES EN MEMOIRE



CARACTERES ALPHANUMERIQUES

Un octet permet de coder tous les caractères alphanumériques ainsi que divers symboles et les opérateurs comme:

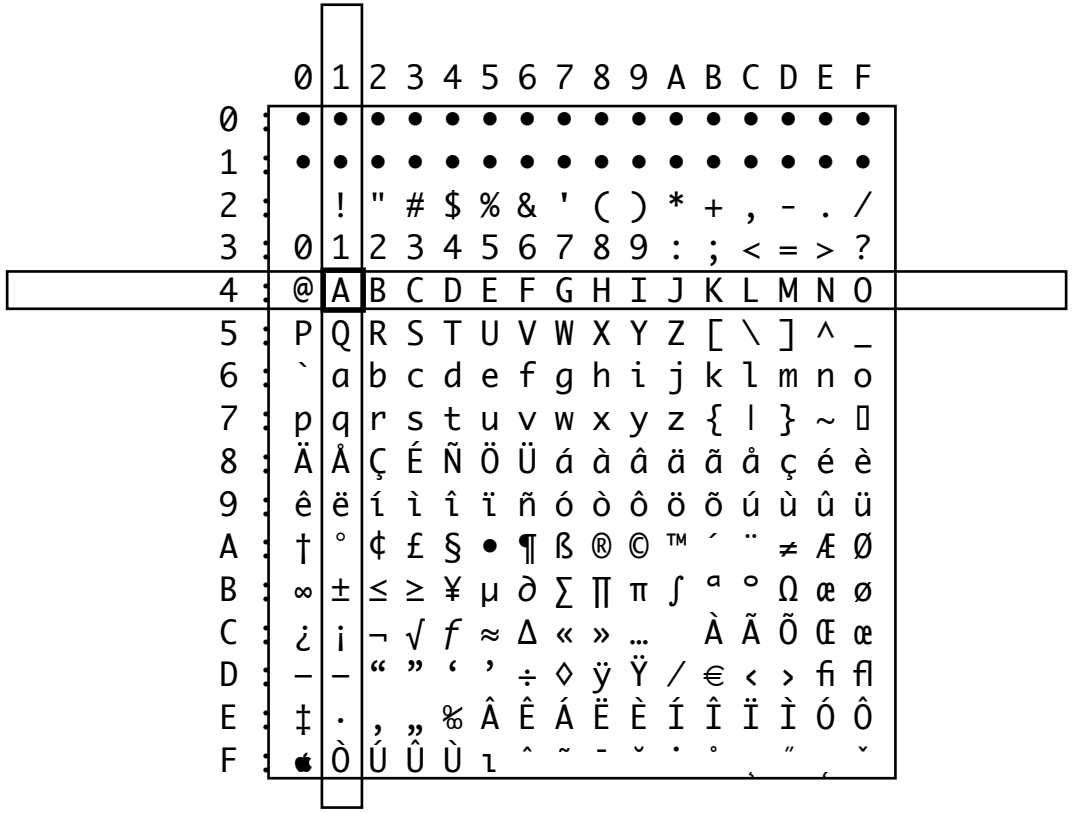
`+, -, *, /, return, tab, etc...`

Le standard ASCII associe aux 128 premières combinaisons de huit 0 ou 1 les caractères alphabétiques et numériques courants ainsi que quelques instructions et opérateurs du type +,-,*,/.

Par exemple, le caractère "**A**" est associé à la **65**ème combinaison, le caractère "**1**" est associé à la **49**ème combinaison, le **retour chariot** est associé à la **13**ème combinaison, etc...

Note 1: on passe des majuscules aux minuscules en ajoutant 32 (ou 20 hex) : “A”=65, “a”=97.

Note 2: La taille d’une page A4 de 50 lignes de 80 caractères se calcule ainsi: 50*80=4000 octets ≠ 3,9 ko.



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
1	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	□
8	Ä	Å	Ç	É	Ñ	Ö	Ü	á	à	â	ä	ã	å	ç	é	è
9	ê	ë	í	ì	î	ï	ñ	ó	ò	ô	ö	õ	ú	ù	û	ü
A	†	°	¢	£	§	•	¶	ß	®	©	™	´	¨	≠	Æ	Ø
B	∞	±	≤	≥	¥	µ	∂	∑	∏	π	∫	ª	º	Ω	æ	ø
C	¿	¡	¬	√	ƒ	≈	Δ	«	»	…		À	Ã	Õ	Œ	œ
D	–	—	“	”	‘	’	÷	◊	ÿ	Ÿ	⁄	€	‹	›	fi	fl
E	‡	·	‚	„	‰	Â	Ê	Á	Ë	È	Í	Î	Ï	Ì	Ó	Ô
F		Ò	Ú	Û	Ù	ı	ˆ	˜	¯	˘	˙	˚	¸	˝	˛	ˇ

TABLEAU DES 256 CARACTERES
les 128 premiers sont les caractères ASCII
les 128 suivants sont différents suivant les plateformes (UNIX, WINDOWS, MACINTOSH,...)

Le codage ASCII s'impose à tous les constructeurs d'ordinateurs. Mais cela ne concerne que les 128 premiers caractères codables sur un octet, dont sont exclus les caractères accentués ou propres à une langue comme le français, le suédois ou l'espagnol.

Comme il n'y a pas de standard de fait en la matière, chacun a choisi son propre code pour les 128 derniers caractères, et en conséquence, un texte écrit sur une machine WINTEL doit être filtré dans une matrice de transcodage pour être lu correctement sur un MAC ou une station UNIX.

Et de toute façon, les langues orientales, avec leurs milliers de caractères sont exclues du jeu. Aujourd'hui, avec Internet, un nouveau codage se profile sur la base de deux octets (soit plus de 30 000 caractères possibles). Il s'agit du système UNICODE qui devrait remplacer les acrobaties imposées aujourd'hui pour les échanges, dans le genre de ce qu'on utilise dans le langage HTML où par exemple: "é" s'écrit : é .

Pas vraiment joli !

NOMBRES ENTIERS

Un **octet** peut représenter 256 états et donc en particulier 2^8 nombres entiers de :

0 à 255 (00 à FF).

et les entiers relatifs de: **-128 à 127**.

En regroupant **deux octets**, on peut représenter 2^{16} nombres entiers de:

0 à soit 65535 (0000 à FFFF)

et les entiers relatifs de : **-32768 à 32767**.

En regroupant quatre octets, on peut représenter 2^{32} nombres entiers de:

0 à 4 294 967 295 (0000 0000 à FFFF FFFF)

et les entiers relatifs de:

-2 147 483 648 à 2 147 483 647.

Et la couleur ? Sur un moniteur la couleur est obtenue par composition de trois composantes: ROUGE, VERT, BLEU (RVB ou RGB in english). Pour obtenir une qualité photo-réaliste avec des dégradés sans bandes, il faut dépasser 200 nuances pour chaque composante. Avec 1 octet (soit 8 bits) par couleur, on atteint 256 nuances et on peut composer:

$256 \times 256 \times 256 = 2^8 \times 2^8 \times 2^8 = 2^{24} \neq 16$ millions couleurs.

Le processeur préfèrant travailler sur des combinaisons paires d'octets, on code en fait sur 4 octets soit 32 bits; le 4ème octet permet de représenter 256 états de ce qu'on veut, et on en profite pour décrire le caractère plus ou moins transparent de la couleur (nom de code alpha channel).

codage:	A	R	G	B
Le rouge opaque s'écrit:	FF	FF	00	00
Le vert opaque s'écrit:	FF	00	FF	00
Le gris transparent 50% s'écrit:	80	80	80	80

Ce codage est utilisé sur Internet (langage HTML).

NOMBRES FRACTIONNAIRES

En regroupant 2 octets, on peut convenir que l'octet de gauche représente la partie entière et l'octet de droite la partie décimale; il existe plusieurs méthodes.

Par exemple, dans la représentation dite **FIXED**, le nombre hexadécimal "**01 80**" représente :

01 correspond à 1

et 80 correspond à 128 que l'on divise par 255

soit 128/255 = 0.5

soit au total: **1,5**.

rappel: 80 se lit: 8 et 0 soit 1000 et 0000, soit 10000000
soit la 128ème combinaison d'un octet.

NOMBRES REELS

Etudions les nombres réels suivants:

1280	128	12,8	1,28	0,128	0,0128

1280 peut s'écrire: $+0.128\ 10^{+4}$
soit sous la forme des 2 entiers relatifs: +128 et +4

128 peut s'écrire: $+0.128\ 10^{+3}$ +128 et +3

12,8 peut s'écrire: $+0.128\ 10^{+2}$ +128 et +2

1,28 peut s'écrire: $+0.128\ 10^{+1}$ +128 et +1

0,128 peut s'écrire: $+0.128\ 10^{+0}$ +128 et +0

0,0128 peut s'écrire: $+0.128\ 10^{-1}$ +128 et -1

etc...

Cette représentation normalisée des nombres réels (virgule flottante) permet de ramener leur représentation à celui d'un couple de deux nombres relatifs, l'exposant et la mantisse.

En codant sur 4 octets, on peut représenter les nombres réels sur l'intervalle :

$$[\ -1.5*10^{45}\ ,\ 3.4*10^{38}\ ]$$

avec une précision de 7 à 8 décimales.

En codant sur 8 octets, l'exposant atteint environ 300 et la précision 16 décimales.

On code également sur 10 et même 12 octets.

Remarque: c'est toujours le même principe: on ramene le problème complexe à deux problèmes plus simples.

LES INSTRUCTIONS

Il faut bien agir sur les données: aligner des caractères, additionner des nombres, tracer des lignes de pixels, mélanger des couleurs.... Il faut donc définir des opérations, des instructions.

Avec **un octet** on peut coder **256** instructions; on trouve ce choix dans certaines calculatrices programmables (ma vieille HP67) où la place mémoire est comptée. Dans ce cas les instructions seront directement entrées au clavier ou mémorisées dans une partie de la mémoire de la machine pour être exécutées ensuite séquentiellement.

Dans les microordinateurs on utilise **deux octets** et l'on peut coder en principe **65 536** instructions.

On compte tout d'abord le bon millier d'instructions de base (code cablé ou microprogrammé) du microprocesseur .

A titre d'illustration la valeur hexadécimale : 2C4F
correspond sur le microprocesseur MOTOROLA 68k à une instruction dont la mnémonique en langage assembleur est :

```
MOVE.L  A7,A6.
```

Cette instruction transfère simplement 4 octets (appelé aussi mot long, d'où le .L) d'une mémoire interne au microprocesseur (registre A7) à une autre (registre A6).

Concernant les combinaisons inutilisées (65000 - 1000), chaque constructeur peut les utiliser à sa guise pour appeler un ou deux autres milliers de routines (cas de la ToolBox du Macintosh par exemple). Un processeur MOTOROLA aura un comportement différent suivant qu'il sera installé sur un MAC, un NEXT ou une station graphique.

Ces routines sont contenues en ROM ou en RAM comme par exemple la valeur hexadécimale **A8A1** correspondant sur MAC à la procédure "**FrameRect**" qui trace tout simplement un rectangle.

LES ADRESSES EN MEMOIRE

La mémoire d'un ordinateur est une suite d'emplacements pouvant contenir 1 bit. Mais en fait, on a pu constater qu'on ne s'intéresse qu'à des groupes de 8 bits successifs, formant des sortes de cases contenant la mémoire élémentaire de 1 octet; c'est à ces cases que l'on va donner une adresse.

Une donnée ou une instruction pourra donc être repérée dans la mémoire (au moins par l'intermédiaire de son octet de poids le plus fort: par exemple, l'adresse d'un entier codé sur deux octets est l'adresse de l'octet de gauche).

Allure d'une partie de la mémoire:

case 0:	0000 0000	
case 1:	0001 0000	
case 2:	0000 1111	
.....		
case 1 000 000:	0010 1000,	etc...

Mais comment représenter un adresse dans un format fixe? Par exemple, en numérotant les adresses d'une rue sur 3 chiffres décimaux on peut "adresser" 1000 batiments du numéro 000 au numéro 999. Si on sait qu'il n'y aura pas plus de 10 batiments (de 0 à 9), il suffirait de numéroter sur 1 chiffre, et on perd de la place avec les deux autres chiffres. Si par contre on sait qu'un jour il peut y avoir 1001 batiments, il faut passer à quatre chiffres.

Sur une calculatrice programmable **HP67**, on ne disposait que d'un octet (**8** bits) pour coder les adresses, soit au total 256 mémoires ou encore 1/4 de kilo octet.

Sur un **Apple II** ou les premiers **PC**, on disposait de deux octets (**16** bits), soit **64ko**. Bill Gates avait même assuré que personne ne pouvait avoir besoin de davantage de mémoire !

Sur les derniers **MAC et PC**, on dispose de quatre octets (**32** bits), soit **4 Go**, ce qui laisse de la place (pour l'instant).

Actuellement les mémoires physiques installées sont au minimum de 32 Mo, pour accueillir les logiciels complexes (Office, 3D Studio Max,...) ; une solution au coût important des mémoires physiques est d'utiliser ce que l'on appelle la mémoire virtuelle étudiée plus loin dans ce texte.

Dans tous les cas l'adressage sur 32 bits est devenu la règle,... en attendant les 64 et 128 bits.

A noter que le même problème se pose sur Internet: chaque ordinateur sur la planète doit avoir une adresse; on la code sur 4 octets soit 32 bits, on peut la voir écrite dans les configurateurs sous la forme de 4 nombres inférieurs à 256: par exemple l'adresse Internet de l'école d'architecture de Montpellier est:

194.199.210.33

Avec ce codage, il y a 4 milliards d'adresses possible. Encore un peu et ce sera saturé, et il faudra passer à 64 bits.

DEFINITION D'UN PROGRAMME INFORMATIQUE

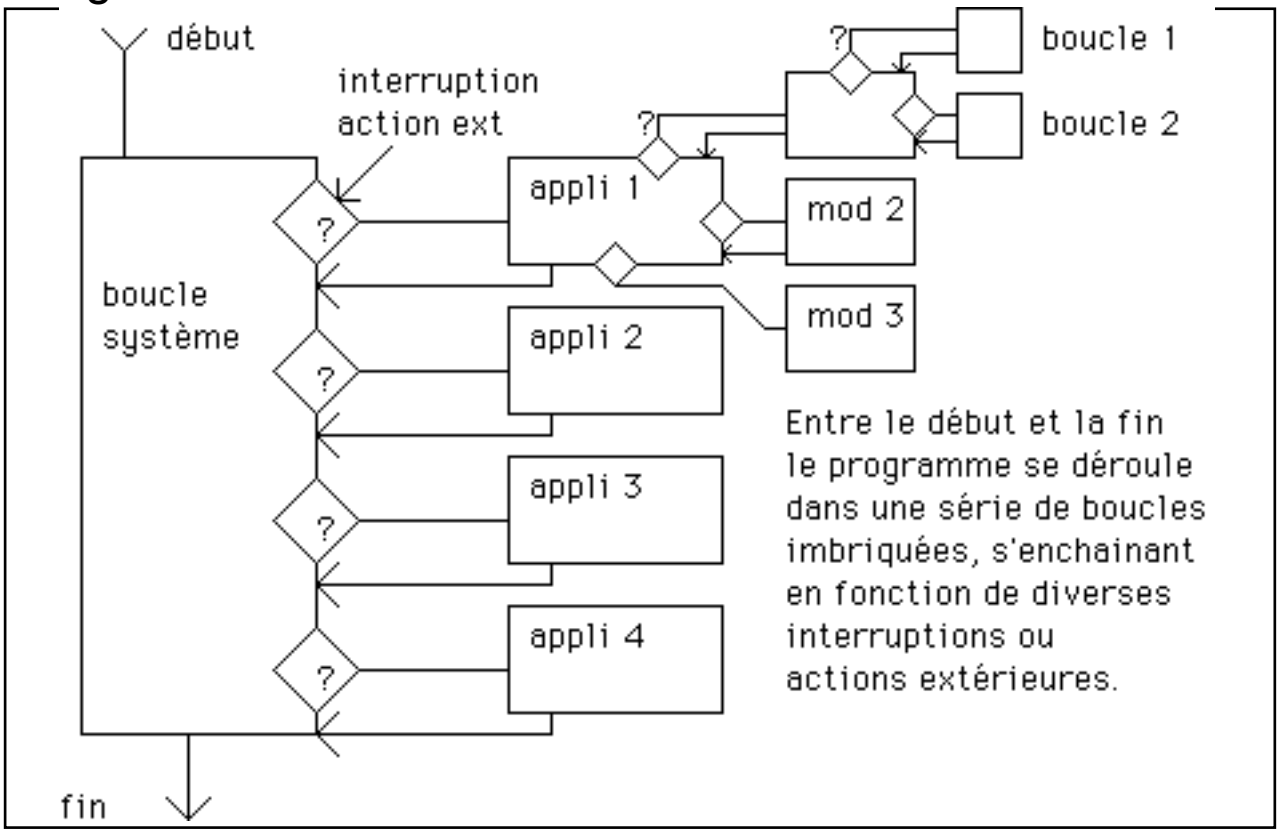
On dispose maintenant de tout ce qu'il faut pour définir ce qu'est un programme informatique.

Un programme informatique se présente sous la forme d'une suite de **données** et **d'instructions** contenues dans la mémoire. L'exécution du programme consiste à lire **séquentiellement** le contenu de la mémoire.

Chaque **instruction** agit sur les **données** qui lui sont associées ou éventuellement provoque un **saut** à une **adresse** différente de l'adresse suivante. Le **saut** peut ou non être soumis à une ou plusieurs **conditions**; de là découlent les possibilités de **bouclage** et **d'interaction** avec l'**extérieur**.

Cette définition s'applique au programme fondamental qui est lancé au démarrage de la machine (le système), au programme "interface" qui assure la gestion de base (Finder, Windows,...), et aux divers logiciels spécialisés dans diverses taches (texte, chiffres, dessins,...)

Ci-dessous une **représentation schématique de la cascade de boucles** que suit le programme contenu dans un ordinateur entre son démarrage et son arrêt:



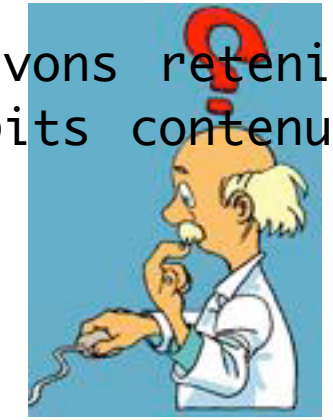
Dans les premiers temps de l'informatique, on écrivait les calculs directement en **binaire** sur des pupitres à interrupteurs, puis en **hexadécimal** à partir de claviers.

Actuellement, on peut les écrire dans un langage comme **l'assembleur** (les instructions niveau machine aperçues plus haut) ou dans un langage évolué comme le **C** et ce sont des traducteurs (interpréteurs et compilateurs) qui se chargent de recomposer dans la mémoire les suites d'octets (et donc de bits) seuls compréhensibles ou plutot manipulables par l'ordinateur, ... qui de toute façon ne comprend strictement rien à tout ce qui lui passe dedans.

Une seule forme (le bit), pour de **multiples sens** dépendant du contexte, voilà l'énorme saut qualitatif qui sépare l'informatique d'une électronique "bêtement binaire", la raison de la richesse des possibilités ouvertes par cette construction purement intellectuelle pour l'analyse et la représentation du monde réel et au delà pour la création de nouveaux mondes virtuels... (...musique, maestro!)

si vous n'êtes pas encore trop fatigués, je vous propose:

BALADE DANS LA MEMOIRE DE L'ORDINATEUR



Au point où nous en sommes, nous pouvons retenir que suivant le contexte, une suite de bits contenus dans la mémoire comme, par exemple :

.....
...00110000010111000000000000101011010000
01101110000000000000100011010000010000000...
.....

```
.....................................................
...00110000010111000000000000101011010000
01101110000000000000100011010000010000000...
.....................................................
```

pourra être considérée, suivant le contexte, comme étant des caractères, des nombres entiers ou réels, des instructions, ou un mélange plus ou moins complexe de tout cela.

Cela peut très bien être une zone de stockage de données, ou une routine entière, ou encore une suite sans contexte précis et ne présentant aucun sens...

Analysons donc cette suite de bits quelque part dans la mémoire, en l'écrivant tout d'abord comme une suite d'octets, dont le premier octet serait par exemple à l'adresse: 0010 0000

...00110000001011100000000000010101101000001101110000000000000100011010000010000000...

binaire		hexadécimal	adresse
0011 0000	=	30	0010 0000
0010 1110	=	2E	0010 0001
0000 0000	=	00	0010 0002
0000 1010	=	0A	0010 0003
1101 0000	=	D0	0010 0004
0110 1110	=	6E	0010 0005
0000 0000	=	00	0010 0006
0000 1000	=	08	0010 0007
1101 0000	=	D0	0010 0008
0000 0100	=	40	0010 0009

soit: 302E 000A D06E 0008 D040

en regroupant les octets par deux.

On a écrit en fait trois instructions connues du microprocesseur Motorola 68k:

```
302E 000A    MOVE.W    b(A6),D0
D06E 0008    ADD.W     a(A6),D0
D040         ADD.W     D0,D0
```

qui réalisent l'expression suivante:

2 * (a + b)

Suivre séquentiellement cette suite de bits revient donc à exécuter une fonction qui calcule le **périmètre** d'un rectangle de cotés **a** et **b**.

En fait on procède dans le sens opposé: voici la même fonction périmètre composée de quatre façons différentes:

1) en écrivant les instructions en **C**,

2) en écrivant les instructions dans l'**assembleur** adapté au processeur,

3) en écrivant directement les suites d'octets équivalentes à ces instructions en représentation **hexadécimale**,

4) en écrivant directement les suites de **bits** équivalents à cette suite d'octets.

```
// 1) langage C

    int         c_perim( int a, int b )
    {
       return   2 * ( a + b );
    }
```

```
// 2) langage assembleur (embarqué dans le C)

   int    a_perim( int a, int b )
   {
      asm
      {
         MOVE.W  b(A6),D0  // charger b
         ADD.W   a(A6),D0  // ajouter a
         ADD.W   D0,D0     // (a+b)+(a+b)
      }
   }
```

remarque: les routines critiques des jeux sont écrits en assembleur, on parle presque directement au processeur.

```
// 3) "langage" hexadécimal (dans C)

int    h_perim( int a, int b )
   {
      asm
       {
          dc.w 0x302E          // dc.w
          dc.w 0x000A          // signifie
          dc.w 0xD06E          // charger les
          dc.w 0x0008          // deux octets
          dc.w 0xD040          // 0x....
       }
   }
```

remarque: certains hackers aiment ça!

```
// 4) "langage" binaire (dans C)

   int    b_perim( int a, int b )
   {
      asm
      {
         dc.w 0b 0011 0000 0010 1110
         dc.w 0b 0000 0000 0000 1010
         dc.w 0b 1101 0000 0110 1110
         dc.w 0b 0000 0000 0000 1000
         dc.w 0b 1101 0000 0100 0000
      }
   }
```

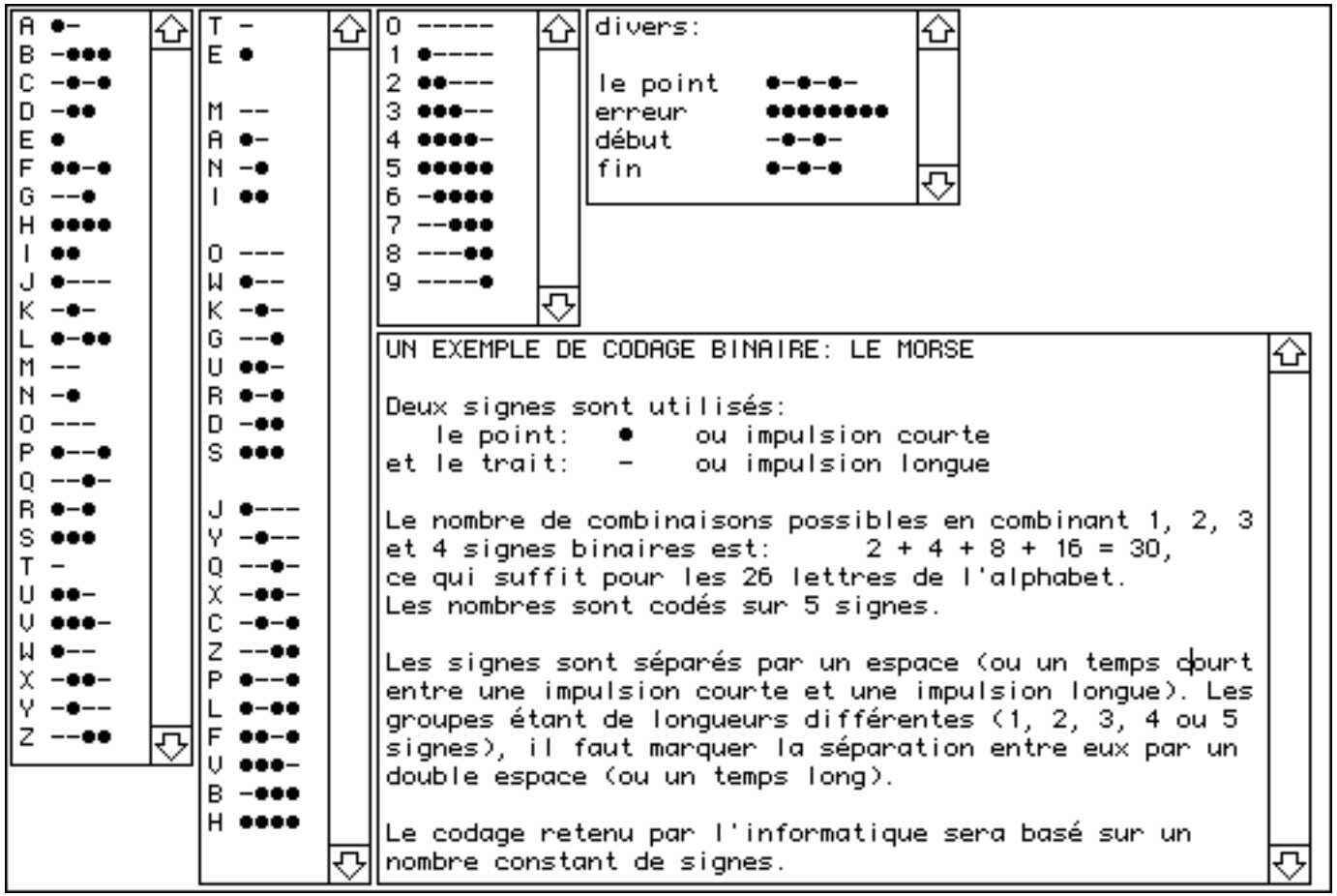
remarque: les premiers informaticiens entraient ainsi les données.

Une fois compilées, les trois premières écritures aboutissent toutes à la dernière, seule comprise par l'ordinateur.

...00110000001011100000000000001010110100000
01101110000000000000100011010000010000000...

et voila comment on peut faire tracer un rectangle à un ordinateur qui ne connait que le langage binaire !

... cool, non ?


A •-
B -•••
C -•-•
D -••
E •
F ••-•
G --•
H ••••
I ••
J •---
K -•-
L •-••
M --
N -•
O ---
P •--•
Q --•-
R •-•
S •••
T -
U ••-
V •••-
W •--
X -••-
Y -•--
Z --••
T -
E •
M --
A •-
N -•
I ••
O ---
W •--
K -•-
G --•
U ••-
R •-•
D -••
S •••
J •---
Y -•--
Q --•-
X -••-
C -•-•
Z --••
P •--•
L •-••
F ••-•
V •••-
B -•••
H ••••
0 -----
1 •----
2 ••---
3 •••--
4 ••••-
5 •••••
6 -••••
7 --•••
8 ---••
9 ----•
divers:
le point •-•-•-
erreur ••••••••
début -•-•-
fin •-•-•
UN EXEMPLE DE CODAGE BINAIRE: LE MORSE
Deux signes sont utilisés:
le point: • ou impulsion courte
et le trait: - ou impulsion longue
Le nombre de combinaisons possibles en combinant 1, 2, 3
et 4 signes binaires est: 2 + 4 + 8 + 16 = 30,
ce qui suffit pour les 26 lettres de l'alphabet.
Les nombres sont codés sur 5 signes.
Les signes sont séparés par un espace (ou un temps court
entre une impulsion courte et une impulsion longue). Les
groupes étant de longueurs différentes (1, 2, 3, 4 ou 5
signes), il faut marquer la séparation entre eux par un
double espace (ou un temps long).
Le codage retenu par l'informatique sera basé sur un
nombre constant de signes.