En raison de l'architecture matérielle des machines numériques (processeurs, mémoires, …) toutes les données numériques doivent être construites à partir d'un alphabet binaire constitué d'un 0 (associé à l'état logique bas) et d'un 1 (associé à l'état logique haut).

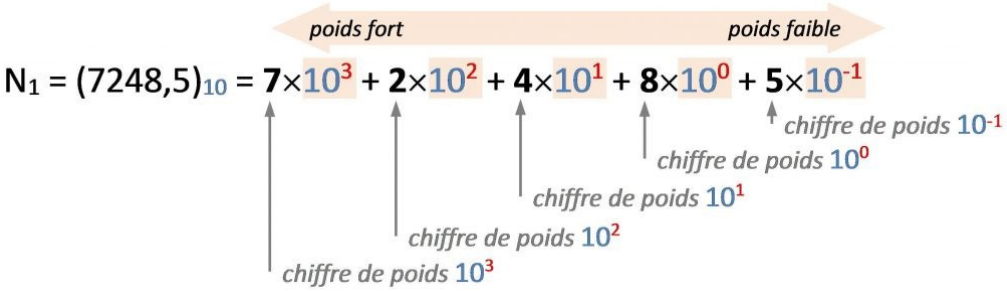
Il est donc nécessaire d'associer plusieurs chiffres binaires (0 et 1) pour représenter des données. Le support d'une information numérique est donc représentée par des mots binaires est numérique.
Un mot de format 8 bits est un octet. Les informations sont souvent représentées en binaire sous la forme de mots dont le format est multiple d'un octet (8bits, 16bits, 32bits …).

Dans notre langue, nous associons des lettres pour construire des mots auxquels nous attribuons un sens. De même, une signification est donnée aux mots binaires par une opération appelée codage binaire de l'information. Le codage permet de donner un sens aux mots binaires. Le codage permet de traduire une information directement compréhensible par des humains (nombre décimal, image, texte, son, …) dans un langage binaire compréhensible par l'unité de traitement des machines.

I. Les systèmes de numération

Un système de numération est un ensemble de règles permettant de représenter les nombres. Dans les systèmes numériques, on utilise principalement les systèmes suivants :
• Le système de numération décimale utilise les dix chiffres 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
• Le système de numération binaire utilise exclusivement les deux chiffres 0 et 1. (que l'on appelle alors bit – binary digit)
• Le système de numération hexadécimale utilise les seize chiffres 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

Par convention, l'écriture d'un nombre N s'effectue par la juxtaposition de chiffres possédant chacun un poids égal à une puissance entière de la base de numération. Les chiffres s'écrivent de la gauche vers la droite par valeur décroissante de leur poids. Le poids d'un chiffre dépend donc de son rang et du système de numération adopté :
En base 10 – écriture décimale :



En bases 2 et 16 – écritures binaire et hexadécimale :

$$N_2 = (1\,0\,0\,1\,1\,0)_2 = 1\times2^5 + 0\times2^4 + 0\times2^3 + 1\times2^2 + 1\times2^1 + 0\times2^0 = (38)_{10}$$

$$N_3 = (E7B)_{16} = 14\times16^2 + 7\times16^1 + 11\times16^0 = (3707)_{10}$$

II. Représentation des nombres entiers naturels

Les nombres entiers naturels sont représentés par le code binaire naturel qui utilise l'expression naturelle du nombre en base 2 (système de numération binaire) :

Codage binaire naturel des 10 chiffres de la base 10 :

	2^3	2^2	2^1	2^0
0	0	0	0	0
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

On dit que ce code est un code pondéré : le ième bit (en partant de la droite) « pèse » 2^{i-1}.

Pour interpréter un nombre en binaire naturel, il faut connaître les puissances de 2

Puissance de 2 →	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Valeur en décimal →													

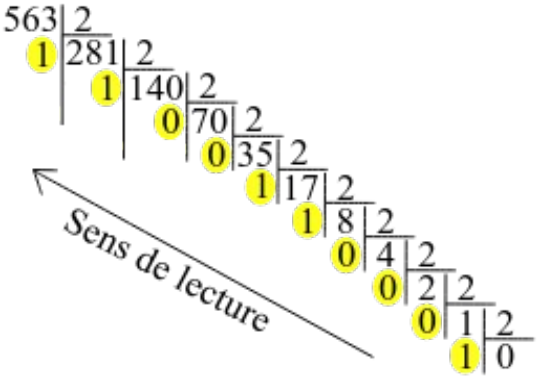
Méthodes de conversion

Pour un nombre entier naturel (positif), on peut présenter son écriture binaire dans un tableau faisant apparaître les puissances successives de 2 (la plus grande puissance de 2 doit être juste inférieure au nombre à coder). Cette méthode marche pour les conversions binaire-décimal et décimal-binaire.

	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	64	32	16	8	4	2	1
$(87)_{10}$	1	0	1	0	1	1	1

Sinon, voici une méthode « manuelle » simple, pour les conversion décimal-binaire :

- On prend le nombre en base 10 (forme normale).
- On le divise par 2 et on note le reste de la division (soit 1 soit 0)
- On refait la même chose avec le quotient précédent, et on met de nouveau le reste de côté.
- On réitère la division, jusqu'à ce que le quotient soit 0.
- Le nombre en binaire apparaît alors : il suffit de prendre tous les restes de bas en haut.



$$563_{10} = 1000110011_2$$

Opérations arithmétiques

Les opérations sur les nombres binaires s'effectuent de la même façon que sur les nombres décimaux toute fois il ne faut pas oublier que les seules symboles utilisés sont le ''1'' et le ''0''. Les opérations fondamentales sont les suivantes :

L'addition
0 + 0 = 0
0 + 1 = 1
1 + 1 = 0

La soustraction
0 – 0 = 0
0 – 1 = 1 et on retient 1
1 – 1 = 1

III. Le code hexadécimal

Le code hexadécimal permet une représentation simplifiée des nombres binaires, à l'usage des humains ! L'écriture et la lecture d'une succession de 0 et de 1 est fastidieuse pour l'homme et source d'erreurs.

10011011110010010101101

Pour raccourcir les mots binaires, un groupe de 4 bits consécutifs est représenté par un chiffre hexadécimal.

0001 0011 1110 0100 1010 1101
1 3 E 4 A D

4 bits permettent de coder 16 éléments ($1111_2 = 15_{10}$)
Pour coder 16 chiffres, on utilise ceux de la base 10 (0 à 9) auxquels on ajoute les 6 lettres de A à F : 0 1 2 3 4 5 6 7 8 9 A B C D E F

Binaire (base 2)	Décimal (base 10)	Hexadécimal (base 16)
0	0	0
1	1	1
10	2	2
11	3	3
100	4	4
101	5	5
110	6	6
111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Le saviez-vous ? Un groupe de quatre bits s'appelle un nibble !

Pour interpréter un nombre hexadécimal, il faut connaître les puissances de 16 :

Puissance de 16 →	16^5	16^4	16^3	16^2	16^1	16^0
Valeur en décimal →						

Méthodes de conversion

- Pour **convertir un nombre décimal en hexadécimal**, la méthode est similaire au binaire, sauf que cette fois on divise par 16.
- Pour c**onvertir un nombre hexadécimal en décimal**, le principe est le même que pour la conversation "binaire en décimal" sauf qu'au lieu d'utiliser des 2^n on utilise des 16^n. On ajoutera que dans le cas où l'on rencontre un A, on le remplace par un 10 ; un C, on le remplace par un 12 ; un D, on le remplace par un 13 ; un E, on le remplace par un 14 ; un F, on le remplace par un 15.
- Pour **convertir un nombre binaire en hexadécimal**, on groupe les bits par 4 en partant de la droite et on convertit chaque groupe en hexadécimal.
- Pour **convertir un nombre hexadécimal en binaire**, on convertit chaque nombre et lettre en groupe de 4 bits binaire puis on les écrit les uns à la suite de l 'autre.

IV. Représentation binaire d'un entier relatif

Nous avons déjà vu comment représenter les entiers positifs, nous allons maintenant nous intéresser aux entiers relatifs.
La première idée qui pourrait nous venir à l'esprit est, sur un nombre comportant n bits, d'utiliser 1 bit pour représenter le signe et n-1 bit pour représenter la valeur absolue du nombre à représenter. Le bit de signe étant le bit dit "de poids fort" (c'est à dire le bit le plus à gauche), ce bit de poids fort serait à 0 dans le cas d'un nombre positif et à 1 dans le cas d'un nombre négatif.

un exemple : on représente l'entier 5 sur 8 bits par 00000101, -5 serait donc représenté par 10000101
Il existe un énorme inconvénient à cette méthode : l'existence de deux zéros, un zéro positif (00000000) et un zéro négatif (10000000) !
Ce problème est, pour plusieurs raisons qui ne seront pas développées ici, rédhibitoire. Nous allons donc devoir utiliser une autre méthode : le complément à deux

Le complément à deux

Avant de représenter un entier relatif, il est nécessaire de définir le nombre de bits qui seront utilisés pour cette représentation (souvent 8, 16 , 32 ou 64 bits)

Prenons tout de suite un exemple : déterminons la représentation de -12 sur 8 bits
- Commençons par représenter 12 sur 8 bits (sachant que pour représenter 12 en binaire seuls 4 bits sont nécessaire, les 4 bits les plus à gauche seront à 0) : 00001100
- Inversons tous les bits (les bits à 1 passent à 0 et vice versa) : 11110011
- Ajoutons 1 au nombre obtenu à l'étape précédente : les retenues sont notées en rouge

$$\begin{array}{r} {\scriptstyle 11} \\ 11110011 \\ +\ 00000001 \\ \hline 11110100 \end{array}$$

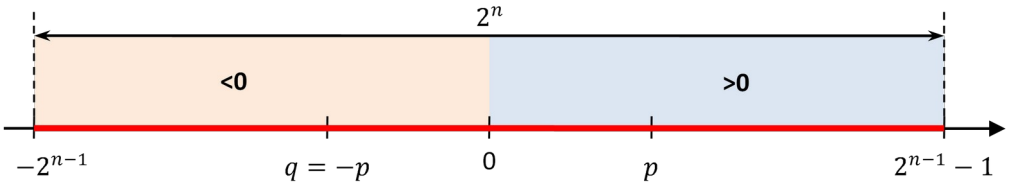
- La représentation de -12 sur 8 bits est donc : 11110100

Comment peut-on être sûr que 11110100 est bien la représentation de -12 ?
Nous pouvons affirmer sans trop de risque de nous tromper que 12 + (-12) = 0, vérifions que cela est vrai pour notre représentation sur 8 bits.

$$\begin{array}{r} {\scriptstyle 11111} \\ 00001100 \\ +\ 11110100 \\ \hline 00000000 \end{array}$$

Dans l'opération ci-contre, nous avons un 1 pour le 9e bit, mais comme notre représentation se limite à 8 bits, il nous reste bien 00000000.
Il faut noter qu'il est facile de déterminer si une représentation correspond à un entier positif ou un entier négatif : si le bit de poids fort est à 1, nous avons affaire à un entier négatif, si le bit de poids fort est à 0, nous avons affaire à un entier positif.

Le code « complément à 2 » (2 pour « base 2 »), réparti les nombres de la manière suivante :



Un octet permet donc de coder les nombres de -128 à 127.

V. Représentation de la partie décimale d'un nombre

Nous avons vu comment sont représentés les entiers relatifs au sein d'un ordinateur. Nous allons maintenant voir comment sont représentés les nombres réels, appelés ici nombres flottants.

Méthodes de conversion

Partons tout de suite sur un exemple : comment représenter 5,1875 en binaire ?
Il nous faut déjà représenter 5, ça, pas de problème : 101
Comment représenter le ",1875" ?

- on multiplie 0,1875 par 2 : 0,1875 x 2 = 0,375. On obtient 0,375 que l'on écrira 0 + 0,375
- on multiplie 0,375 par 2 : 0,375 x 2 = 0,75. On obtient 0,75 que l'on écrira 0 + 0,75
- on multiplie 0,75 par 2 : 0,75 x 2 = 1,5. On obtient 1,5 que l'on écrira 1 + 0,5 (quand le résultat de la multiplication par 2 est supérieur à 1, on garde uniquement la partie décimale)
- on multiplie 0,5 par 2 : 0,5 x 2 = 1,0. On obtient 1,0 que l'on écrira 1 + 0,0 (la partie décimale est à 0, on arrête le processus)

On obtient une succession de "a + 0,b" (ici "0 + 0,375", "0 + 0,75", "1 + 0,5" et "1 + 0,0"). Il suffit maintenant de "prendre" tous les "a" (dans l'ordre de leur obtention) afin d'obtenir la partie décimale de notre nombre : 0011

Nous avons $(101,0011)_2$ qui est la représentation binaire de $(5,1875)_{10}$

Il est possible de retrouver une représentation décimale en base 10 à partir d'une représentation en binaire.

Partons de $(100,0101)_2$
Pas de problème pour la partie entière, nous obtenons "4".
Pour la partie décimale nous devons écrire : $0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} = 0,3125$.
Nous avons donc $(4,3125)_{10}$

Que remarquez-vous ?
Dans l'exemple ci-dessus, nous remarquons que le processus de "conversion" ne s'arrête pas, nous obtenons : "0,0001100110011...", le schéma "0011" se répète à "l'infini". Cette caractéristique est très importante, nous aurons l'occasion de revenir là-dessus plus tard.

En base dix, il est possible d'écrire les très grands nombres et les très petits nombres grâce aux "puissances de dix" (exemples "$6,02 \cdot 10^{23}$" ou "$6,67 \cdot 10^{-11}$"). Il est possible de faire exactement la même chose avec une représentation binaire, puisque nous sommes en base 2, nous utiliserons des "puissances de deux" à la place des "puissances dix" (exemple "$101,1101 \cdot 2^{10}$").

Pour passer d'une écriture sans "puissance de deux" à une écriture avec "puissance de deux", il suffit de décaler la virgule : "1101,1001 = 1,1011001.211" pour passer de "1101,1001" à "1,1011001" nous avons décalé la virgule de 3 rangs vers la gauche d'où le "2^{11}" (attention de ne pas oublier que nous travaillons en base 2 le "11" correspond bien à un décalage de 3 rangs de la virgule).

Si l'on désire décaler la virgule vers la gauche, il va être nécessaire d'utiliser des "puissances de deux négatives" "0,0110 = $1{,}10 \cdot 2^{-10}$", nous décalons la virgule de 2 rangs vers la droite, d'où le "−10"

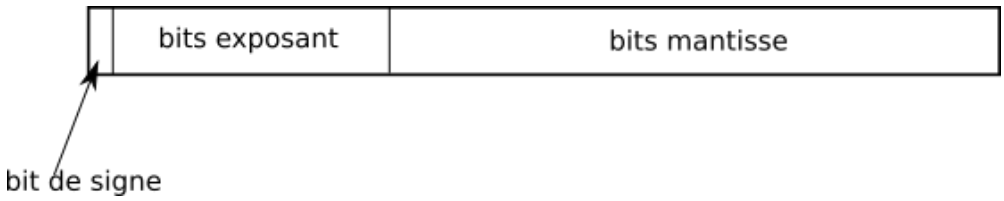
Représentation des flottants dans un ordinateur

La norme IEEE 754 est la norme la plus employée pour la représentation des nombres à virgule flottante dans le domaine informatique. La première version de cette norme date de 1985.

Nous allons étudier deux formats associés à cette norme : le format dit "simple précision" et le format dit "double précision". Le format "simple précision" utilise 32 bits pour écrire un nombre flottant alors que le format "double précision" utilise 64 bits. Dans la suite nous travaillerons principalement sur le format 32 bits.

Que cela soit en simple précision ou en double précision, la norme IEEE754 utilise :

- 1 bit de signe (1 si le nombre est négatif et 0 si le nombre est positif)
- des bits consacrés à l'exposant (8 bits pour la simple précision et 11 bits pour la double précision)
- des bits consacrés à la mantisse (23 bits pour la simple précision et 52 bits pour la double précision)



Nous pouvons vérifier que l'on a bien 1 + 8 + 23 = 32 bits pour la simple précision.

Pour écrire un nombre flottant en respectant la norme IEEE754, il est nécessaire de commencer par écrire le nombre sous la forme $1{,}XXXXX \cdot 2^{e}$ (avec e l'exposant), il faut obligatoirement qu'il y ait un seul chiffre à gauche de la virgule et il faut que ce chiffre soit un "1". Par exemple le nombre "1010,11001" devra être écrit "$1{,}01011001 \cdot 2^{11}$". Autre exemple, "0,00001001" devra être écrit "$1{,}001 \cdot 2^{-101}$".

La partie "XXXXXX" de "$1{,}XXXXX \cdot 2^{e}$" constitue la mantisse (dans notre exemple "1010,11001" la mantisse est "01011001"). Comme la mantisse comporte 23 bits en simple précision, il faudra compléter avec le nombre de zéro nécessaire afin d'atteindre les 23 bits (si nous avons "01011001", il faudra ajouter 23 - 8 = 15 zéros à droite, ce qui donnera en fin de compte "01011001000000000000000")

Notre première intuition serait de dire que la partie "exposant" correspond simplement au "e" de "$1{,}XXXXXX \cdot 2^{e}$" (dans notre exemple "1010,11001", nous aurions "11"). En faite, c'est un peu plus compliqué que cela. En effet, comment représenter les exposants négatifs ? Aucun bit pour le signe de l'exposant n'a été prévu dans le norme IEEE754, une autre solution a été choisie :

Pour le format simple précision, 8 bits sont consacrés à l'exposant, il est donc possible de représenter 256 valeurs, nous allons pouvoir représenter des exposants compris entre $(-126)_{10}$ et $(+127)_{10}$ (les valeurs -127 et +128 sont des valeurs réservées, nous n'aborderons pas ce sujet ici). Pour avoir des valeurs uniquement positives, il va falloir procéder à un décalage : ajouter systématiquement 127 à la valeur de l'exposant. Prenons tout de suite un exemple (dans la suite, afin de simplifier les choses nous commencerons par écrire les exposants en base 10 avant de les passer en base 2 une fois le décalage effectué) :

Repartons de "1010,11001" qui nous donne $1{,}01011001 \cdot 2^3$, effectuons le décalage en ajoutant 127 à 3 : "$1{,}01011001 \cdot 2^{130}$", soit en passant l'exposant en base 2 : "$1{,}01011001 \cdot 2^{10000010}$". Ce qui nous donne donc pour "1010,11001" une mantisse "01011001000000000000000" (en ajoutant les zéros nécessaires à droite pour avoir 23 bits) et un exposant "10000010" (même si ce n'est pas le cas ici, il peut être nécessaire d'ajouter des zéros pour arriver à 8 bits, ATTENTION, ces zéros devront être rajoutés à gauche).

Nous sommes maintenant prêts à écrire notre premier nombre au format simple précision :
Soit le nombre "-10,125" en base 10 représentons-le au format simple précision :

- Nous avons $(10)_{10} = (1010)_2$ et $(0{,}125)_{10} = (0{,}001)_2$ soit $(10{,}125)_{10} = (1010{,}001)_2$
- Décalons la virgule : $1010{,}001 = 1{,}010001 \cdot 2^3$, soit avec le décalage de l'exposant $1{,}010001 \cdot 2^{130}$, en écrivant l'exposant en base 2, nous obtenons $1{,}010001 \cdot 2^{10000010}$
- Nous avons donc : notre bit de signe = 1 (nombre négatif), nos 8 bits d'exposant = 10000010 et nos 23 bits de mantisse = 01000100000000000000000
- Soit en "collant" tous les "morceaux" : 11000001001000100000000000000000

VI. Codage des caractères

Nous savons qu'un ordinateur est uniquement capable de traiter des données binaires, comment sont donc codés les textes dans un ordinateur ? Ou plus précisément, comment sont codés les caractères dans un ordinateur ?

ASCII

Avant 1960 de nombreux systèmes de codage de caractères existaient, ils étaient souvent incompatibles entre eux. En 1960, l'organisation internationale de normalisation (ISO) décide de mettre un peu d'ordre dans ce bazar en créant la norme ASCII (American Standard Code for Information Interchange). À chaque caractère est associé un nombre binaire sur 8 bits (1 octet). En faite, seuls 7 bits sont utilisés pour coder un caractère, le 8e bit n'est pas utilisé pour le codage des caractères. Avec 7 bits il est possible de coder jusqu'à 128 caractères ce qui est largement suffisant pour un texte écrit en langue anglaise (pas d'accents et autres lettres particulières).

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Comme vous pouvez le constater dans le tableau ci-dessus, au "A" majuscule correspond le code binaire (1000001)2 ((65)10 ou (41)16)

Comme vous pouvez le constater, certains codes ne correspondent pas à des caractères (de 0 à $(32)_{10}$), nous n'aborderons pas ce sujet ici.

ISO-8859-1

La norme ASCII convient bien à la langue anglaise, mais pose des problèmes dans d'autres langues, par exemple le français. En effet l'ASCII ne prévoit pas d'encoder les lettres accentuées. C'est pour répondre à ce problème qu'est née la norme ISO-8859-1. Cette norme reprend les mêmes principes que l'ASCII, mais les nombres binaires associés à chaque caractère sont codés sur 8 bits, ce qui permet d'encoder jusqu'à 256 caractères. Cette norme va être principalement utilisée dans les pays européens puisqu'elle permet d'encoder les caractères utilisés dans les principales langues européennes (la norme ISO-8859-1 est aussi appelée "latin1" car elle permet d'encoder les caractères de l'alphabet dit "latin")

Problème, il existe beaucoup d'autres langues dans le monde qui n'utilisent pas l'alphabet dit "latin", par exemple le chinois ou le japonais ! D'autres normes ont donc dû voir le jour, par exemple la norme "GB2312" pour le chinois simplifié ou encore la norme "JIS_X_0208" pour le japonais.

Cette multiplication des normes a très rapidement posé problème. Imaginons un français qui parle le japonais. Son traitement de texte est configuré pour reconnaître les caractères de l'alphabet "latin" (norme ISO-8859-1). Un ami japonais lui envoie un fichier texte écrit en japonais. Le français devra modifier la configuration de son traitement afin que ce dernier puisse afficher correctement l'alphabet japonais. S'il

n'effectue pas ce changement de configuration, il verra s'afficher des caractères ésotériques.

Unicode

Pour éviter ce genre de problème, en 1991 une nouvelle norme a vu le jour : Unicode
Unicode a pour ambition de rassembler tous les caractères existant afin qu'une personne utilisant Unicode puisse, sans changer la configuration de son traitement de texte, à la fois lire des textes en français ou en japonais
Unicode est uniquement une table qui regroupe tous les caractères existant au monde, il ne s'occupe pas de la façon dont les caractères sont codés dans la machine. Unicode accepte plusieurs systèmes de codage : UTF-8, UTF-16, UTF-32. Le plus utilisé, notamment sur le Web, est UTF-8.

Pour encoder les caractères Unicode, UTF-8 utilise un nombre variable d'octets : les caractères "classiques" (les plus couramment utilisés) sont codés sur un octet, alors que des caractères "moins classiques" sont codés sur un nombre d'octets plus important (jusqu'à 4 octets).

Caractères codés	Représentation binaire UTF-8	Signification
U+0000 à U+007F	0xxxxxxx	1 octet, codant 7 bits
U+0080 à U+07FF	110xxxxx 10xxxxxx	2 octets, codant 11 bits
U+0800 à U+FFFF	1110xxxx 10xxxxxx 10xxxxxx	3 octets, codant 16 bits
U+10000 à U+FFFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	4 octets, codant 21 bits

Un des avantages d'UTF-8 c'est qu'il est totalement compatible avec la norme ASCII : Les caractères Unicode codés avec UTF-8 ont exactement le même code que les mêmes caractères en ASCII.