C PARTY : Un exposé applicatif du langage C pour les microcontrôleurs

Sommaire

Chapitre 5 : APPLICATIONS

1) LEDS

- Une led qui clignote
- Une led qui clignote à une fréquence précise (timer + interruption)
- Deux leds qui clignotent à des vitesses différentes (timer+ interruption)
- Une led + un bouton poussoir → Variation de luminosité (PWM)
- 8 leds + un bouton poussoir → variations lumineuses, effet scanner, etc.
- Une led bicolore
- Un stroboscope à leds
- Un vu-mètre à leds (voltmètre/ampèremètre)
- Matrice de leds ou comment optimiser les ports du µC
- Un dé électronique
- Mise en œuvre d'une led de forte puissance

2) Afficheur 7 segments

- Compteur/Décompteur 1 digit
- Compteur/Décompteur 2 digits (technique du multiplexage)
- Compteur/Décompteur 4 digits (technique du multiplexage)
- Voltmètre/Ampèremètre 4 digits (multiplexage et ADC)
- Horloge temps réel → DS1337+2 boutons+4 digits (I2C, interruptions, multiplexage)
- Convertisseur numérique (binaire/décimal/hexa) sur 8 digits

3) Afficheur LCD alphanumériques

- Câblage, initialisation, mise en œuvre
- Affichage de caractères spéciaux (utilisation CGRAM)
- Voltmètre/Ampèremètre
- Fréquencemètre, Période-mètre
- Capacimètre, inductancemètre, milliOhmmètre
- Mesure de distance avec des ultra-sons
- Mesure de distance avec des leds infrarouges
- Une calculatrice

4) Buzzer et haut-parleurs

- Interfaçages
- Générer des fréquences
- Un mini piano
- Une sirène Américaine

5) Port série (USART)

- RS232 Câblages
- RS485 Câblages
- Piloter son µC par PC simplement.

6) Bootloader

- Bootloader pour PIC18F8722

Chapitre 1 : PRESENTATION GENERALE

1 C pourquoi ? La genèse de C PARTY

Si vous tapez sous Google les mots clés suivants : C+microcontrôleur vous obtenez, à la date où sont écrits ces lignes, plus de 954.000 résultats, donc tout porterait à croire qu'apprendre le C orienté microcontrôleurs est à la portée de tous et que tout cela est d'une banalité totale.
Il n'en est rien.
Il suffit pour s'en convaincre de piocher au hasard quelques résultats pour en tirer quelques constats et émettre quelques critiques :

- Cours très complets sur le C mais aussi très théoriques, très généralement illustrés d'exemple orientés vers la console PC.
- Cours survolant (trop) les bases fondamentales de numération, les vocabulaires ou les méthodes de raisonnements, d'organisation, de choix.
- Applications vers les microcontrôleurs n'expliquant que rarement les méthodes de réflexions, les calculs ou encore les mécanismes d'utilisation ou de configurations des ressources matériels.
- Trop souvent des codes extrêmement mal commentés ou, nous verrons que ce point est très important en programmation.
- Trop souvent des codes issus de compilateurs non C ANSI et donc non portable d'un µC à l'autre, alors que c'est un des arguments fort de ce langage à condition de faire ce qu'il faut.
- Des documents issus de l'enseignement, biens faits mais qui répondent à des exigences de programmes scolaires avec des choix trop souvent orientés commercialement (marque de compilateur, marque de microcontrôleurs, etc) et ne répondant pas toujours à mon sens à une nécessaire universalité ou diversité…
- Ou encore ne traitant que d'aspects trop limitatifs pour avoir une vision globale.
- Je passe sur les sites ou blogs de bricoleurs du dimanche qui disséminent un tas d'énormités, d'inexactitudes qui étrangement reçoivent plus de crédits et pénètrent plus durablement, plus insidieusement les esprits.

Aujourd'hui la très grande majorité des sites dédiés aux microcontrôleurs est essentiellement axée sur les PIC, laissant croire aux novices qu'il n'existerait que cette marque, que cette solution.
Pire encore, que cette référence est incontournable, la meilleure preuve étant sa prolifération.
C'est évidemment faux.
Juste que commercialement ils ont mieux fait leur boulot sans doute et aussi que Microchip est arrivé au bon moment et avec de bons arguments, à l'époque tout du moins…

Comme dans ce domaine il n'est nulle éternité, ni supériorité indéniable et incontournable, il est donc souhaitable de ne pas rester prisonnier et d'avoir les idées aussi larges que possible en ce domaine.

Les novices sont souvent très effrayés par l'assembleur, même si de très bon cours existent et notamment celui de Bigonoff qui fait référence en la matière sur le net.
L'assembleur possède de nombreux avantages et aussi quelques restrictions notamment d'être lié à une marque de microcontrôleur voir à une famille dans une même marque et intransportable en terme de code.
Egalement il ne permet pas une maintenance aisée dans des applications lourdes.

L'époque a changé, autrefois les émulateurs coûtaient très chers et l'assembleur était souvent le moyen bon marché de faire du développement dans les petites structures, les microcontrôleurs avaient des ressources très limités ce qui imposait naturellement l'emploi de l'assembleur.
Et puis il y avait surtout quelque chose d'assez nouveau dans le milieu de l'électronique qui historiquement était depuis toujours dominé par la compétence analogique : l'émergence de la vague numérique avec l'apparition de gens à compétences informatique dont l'assembleur est la langue maternelle.

Bref, aujourd'hui pour des raisons d'efficacités et de rendement, par le fait que les microcontrôleurs d'aujourd'hui possèdent d'énormes ressources mémoires, l'assembleur perd du terrain au profit des langages dit « évolués » tels que C, Pascal, Basic.
Nous verrons dans cet exposé que la meilleure efficacité se trouve souvent dans l'usage conjoint de l'assembleur et du C, le C étant le langage le plus proche de l'assembleur, langue native du microcontrôleur.
Le C comme tous langages évolués masquent en partie l'intimité du matériel ce qui la plupart du temps est plutôt un avantage quand on doit écrire une application complexe mais qui peut nécessiter l'usage de l'assembleur pour par exemple optimiser l'usage de certaines ressources ou écrire des *drivers* performants et rapides.

Par contre quand il s'agit de faire des calculs complexes le C s'avère bien plus efficient, vous l'aurez compris il n'est pas question d'abandonner l'assembleur mais d'exploiter à bon escient les avantages respectifs de l'un et de l'autre.

2 C pour qui ?

- C'est bien sur pour tous et plus largement pour tous ceux qui se promettent depuis des années de s'y mettre, tout en remettant aux calendes grecques la date effective….
- L'étudiant de terminale y trouvera un (très constructif) complément aux cours qu'il reçoit.
- L'électronicien à la retraite mais passionné par son métier et qui n'a jamais eu l'occasion de se mettre à la page de cette technologie.
- L'électronicien qui cherche à ajouter une nouvelle corde à son arc, ou plutôt à son CV.
- Le non électronicien qui a pour loisirs l'électronique.
- Le scientifique qui n'a pas une formation spécifique à l'électronique mais qui a des besoins qui nécessitent dans son corps de métier des compétences en microcontrôleurs, par exemple dans le cadre de matériels pour compléter ou réaliser une expérience, un outil spécifique.
- L'informaticien qui viendra chercher ici davantage d'informations pratiques sur le coté matériels des microcontrôleurs.
- Plus globalement tous les gens curieux d'apprendre quelque chose d'actuel et de particulièrement généralisé dans les technologies modernes des équipements qui nous entourent.
- Pourquoi pas aussi des enseignants cherchant des exemples d'applicatifs pour leurs cours, cet exposé étant libre de droit et aucunement restrictif en terme d'usage.
- Ceux qui pensent que l'on ne peut pas compter jusqu'à 10 avec deux doigts seulement (lol).
- Enfin ceux qui connaissent un autre langage comme basic et qui ne réalisent pas encore qu'ils vont se faire contaminer grave, qu'ils n'en ressortiront pas indemnes.
- Les adeptes de l'assembleur… non je déconne, ceux-là sont quasiment en religion avec l'assembleur donc peu de chance de les voir débouler ici.

- Et j'ai le droit de rêver un peu … si cela pouvait convaincre le monde de l'enseignement de bien vouloir bouter énergiquement hors de nos écoles tous ces pseudos logiciels comme flowcode par exemple, dits intelligents, qui n'enseignent rien, n'amènent à rien, ne servent à rien sauf à engourdir des esprits qui ne demandent qu'à progresser et comprendre le sens des choses.

Pour conclure sur le sujet de l'intelligence :

«L'intelligence c'est ce qui se passe quand rien n'empêche l'intelligence de fonctionner » Louis Pauwels.

Et je partage à 100% cette idée d'où les fondements de cet exposé.

3 C facile ?

Question légitime que les plus anxieux se posent déjà j'en suis sûr.
Ben « oui et non » comme disent les Normands.
Oui, si on a un minimum de bon sens et de logique.
Non, car seul le temps apporte l'expérience qui servira de véritable ciment à ces nouvelles connaissances pas forcément naturelles au premier abord.

Mais rassurez-vous, le principe que j'adopte est le même que j'emploie lors des nombreuses formations que j'ai été amené à conduire dans le cadre de mes activités professionnelles depuis 25 ans.
J'essaye de me mettre toujours à la place d'un débutant absolu, j'estime qu'il n'y a jamais de mauvaises questions quand on apprend, bien souvent la « mauvaise » question vient d'un manque de clarté de la part de celui qui enseigne ou forme.
« Ce que l'on conçoit bien s'énonce clairement » telle est ma devise.
Bon, même si ce n'est pas toujours évident d'être en forme ou inspiré chaque jour...

Je dirais que pour pouvoir suivre cet exposé il faut les dispositions suivantes :

- L'envie
- La curiosité
- La pugnacité

Accessoirement si vous avez un niveau de 3ème vous êtes aptes à tout comprendre.
En tout cas si vous savez compter jusqu'à 10 vous êtes bien équipé pour suivre jusqu'au bout.

La différence avec un exposé devant de vrais gens est que d'ici je n'ai aucun recul ni aucun retour en temps réel de la manière dont les gens perçoivent l'exposé.
C'est pour cela que le document sera en permanence remis en forme, enrichi et complété.
La version est toujours visible en haut de chaque page, donc la mise à jour consistera à tout simplement écraser une version antérieure.
N'imprimez pas !
Vous allez gâcher du papier inutilement, en tout cas dans un premier temps, les versions vont évoluer très rapidement donc attendez au moins le début des applications pour imprimer ce qui précède.

4 C pratique ?

Ca c'est un des objectifs de ce projet.
J'ai prévu de faire des fiches très complètes et individuelles sur chaque sujet.
Par exemple sur les décalages et le masquage en un coup d'œil vous savez à quoi ça correspond, j'ai favorisé le coté visuel, manière bande dessinée, même un non francophone pourra comprendre donc à priori même un rappeur...
L'intérêt également est que le plan de l'exposé n'est pas restrictif, ce qui nous permettra selon l'évolution des choses à enrichir progressivement les exemples d'applications.
Celui qui voudra réaliser un exemple, aura un schéma complet, des explications complètes, un code 100% fonctionnel, il sera à même de pouvoir le modifier.
Mieux qu'un livre, là il suffira d'imprimer uniquement la section concernée.
Même pas peur d'Hadopi, vous pouvez télécharger autant que vous voulez, vous ne devrez rien à personne.

5 C long ?

Ben ça ne dépendra que de vous, du temps que vous y consacrerez.

Quelqu'un de normalement constitué devrait en moins de un mois avoir acquis les principes de bases lui permettant de reconnaitre tous les éléments de base du langage, être capable de faire clignoter une led voir plusieurs.

Je suis persuadé que la grande majorité sera bien au-delà dans un mois ou deux, en tout cas que le virus du C vous aura contaminé au point que vous ne voudrez plus autre chose pour programmer vos précieux microcontrôleurs.
Il n'y aura plus que votre imagination pour limiter vos projets les plus fous, bientôt vous parlerez couramment le C à votre µC adoré.

Pour atteindre notre objectif et arriver dans le vif du sujet équipé des outils indispensables nous allons passer par une petite phase de révision des principes élémentaires de la numération afin de bien comprendre comment et de quoi est composé un nombre, comment le manipuler afin de le traduire dans le langage natif de la machine et réciproquement pour exploiter ses résultats.

Chaque partie sera illustrée d'exemples d'applications commentés et conclue par un résumé des notions les plus indispensables pour aborder le chapitre suivant.

Chapitre 2 : NUMERATION ET CONVERSIONS

1. Décimal, Binaire, bit, hexa, BCD, octets, etc. C par ici…

Depuis que l'homme possède deux mains, et donc 10 doigts à moins accident, il compte en base 10 que l'on appelle communément le système décimal.
Le système décimal permet de prendre en compte 10 valeurs distinctes : 0, 1, 2, 3, 4, 5, 6, 7, 8,9.

Depuis que l'ordinateur existe celui-ci ne comprend que deux états possibles : VRAI ou FAUX ou encore OUI ou NON, deux états possibles d'où l'appellation binaire relative à la base 2.
Il est donc naturel et suffisant pour lui de manipuler deux valeurs numériques distinctes: 0 et 1

En base 10 on appelle chaque élément des chiffres.
En base 2 on appelle chaque élément des chiffres binaires qui en anglais se dit : *binary digit*.
Ce terme contracté est plus couramment nommé : bit

Pour que l'homme puisse dialoguer avec sa machine il va donc être nécessaire d'effectuer des conversions numériques pour passer de la base 10 à la base 2 et vice-versa.

Voyons simplement comment un nombre entier décimal se compose en faisant un petit rappel mathématique (quelques minutes de transpiration indispensables)

Par exemple prenons au hasard le nombre entier **1024 en base 10**.

On peut exprimer littéralement que 1024 c'est **une fois 1 millier plus 0 centaine plus deux dizaines plus quatre unités.**
Ce qui mathématiquement se traduit par la forme canonique : (1x1000) + (0x100) + (2x10) + (4x1)

On sait par ailleurs que 1**000**=1**0**x1**0**x1**0** ce qui peut s'écrire plus aisément : 10^{3} (**3** représente le nombre de fois que l'on multiplie 10 par lui même)

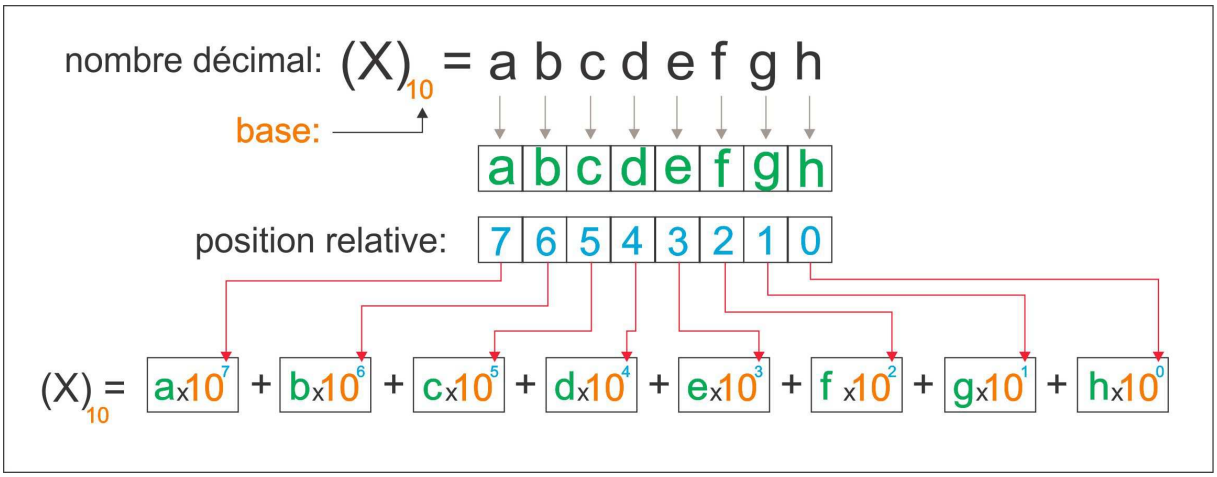
De même 1**00**=10x10=10_{2} puis 1**0** =10_{1} et enfin 1 = 10_{0}

Nous pouvons donc écrire que 1024 en base décimale se traduit mathématiquement par :

1024 = **1**x10^{3} + **0**x10^{2} + **2**x10^{1} + **4**x10^{0} 1,0,2,4 étant les quantités de chaque rang.

Cette écriture purement mathématique n'a d'intérêt que de faire apparaître clairement le format de la base ainsi que la quantité propre à chaque rang.

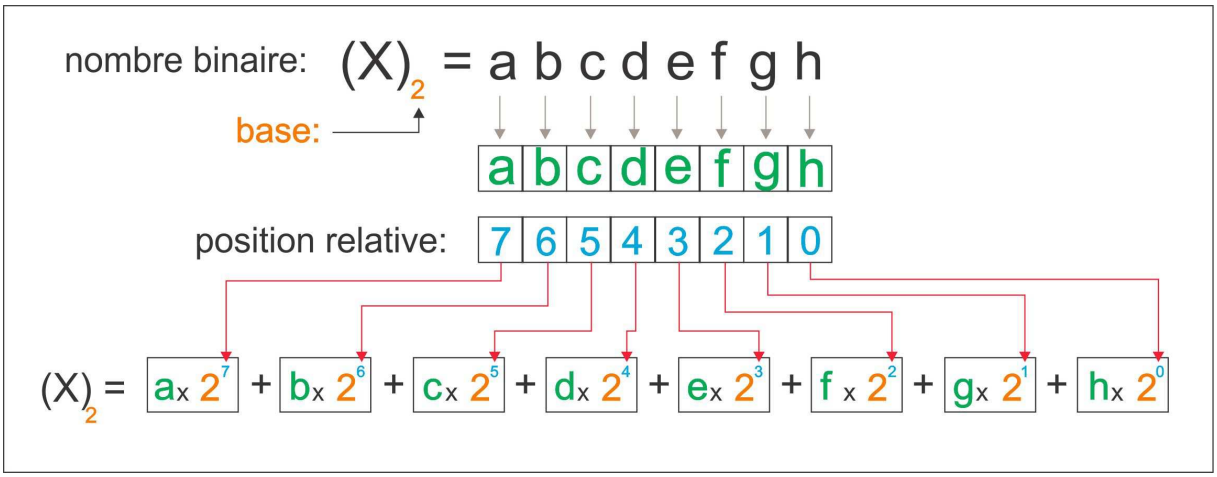
Le tableau suivant rassemble ce que nous venons de voir pour un nombre décimal composé de 8 chiffres quelconques :



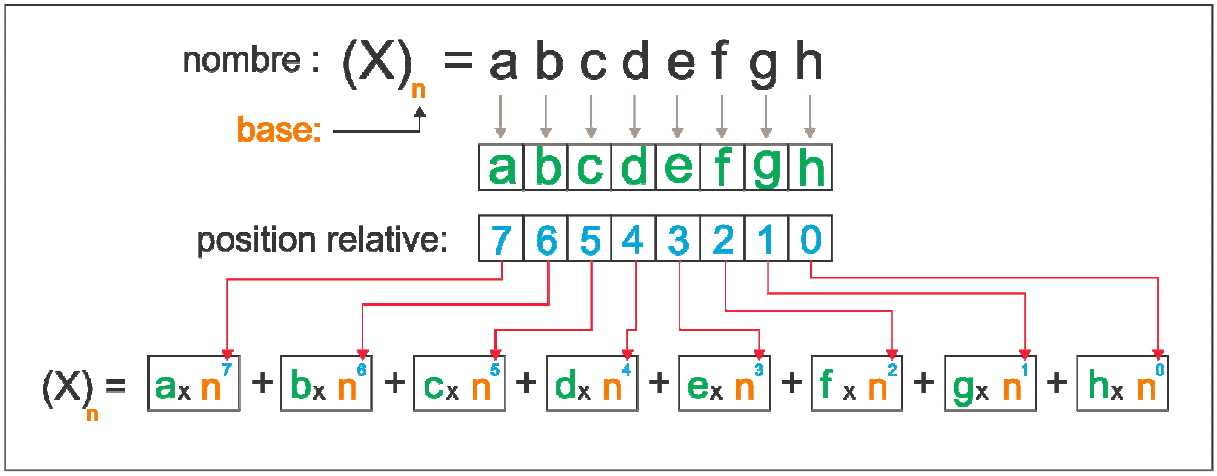
De ce qui précède nous pouvons déjà déduire un certain nombre de choses à retenir :

1. Une base B est composée de N chiffres distincts (base 10 → 10 chiffres, base 2 → 2 chiffres).
2. Le chiffre le plus grand dans cette base B vaut N-1 (base 10 →9 base 2 → 1)
3. Un nombre X peut se décomposer par une somme de produits, chaque produit étant composé de la quantité du rang multiplié par la base B à la puissance correspondant à ce rang (la puissance exprime le nombre de fois que l'on multiplie la base B par elle même).

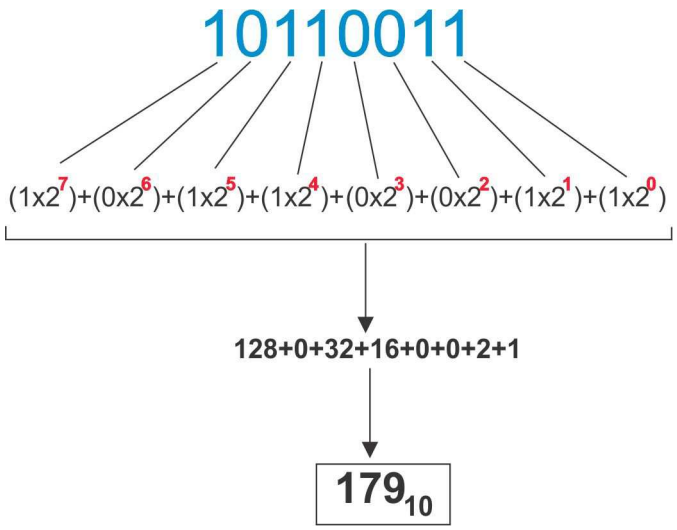
Voici le même type de représentation pour un nombre X en base binaire :



Et enfin pour un nombre X dans une base quelconque :



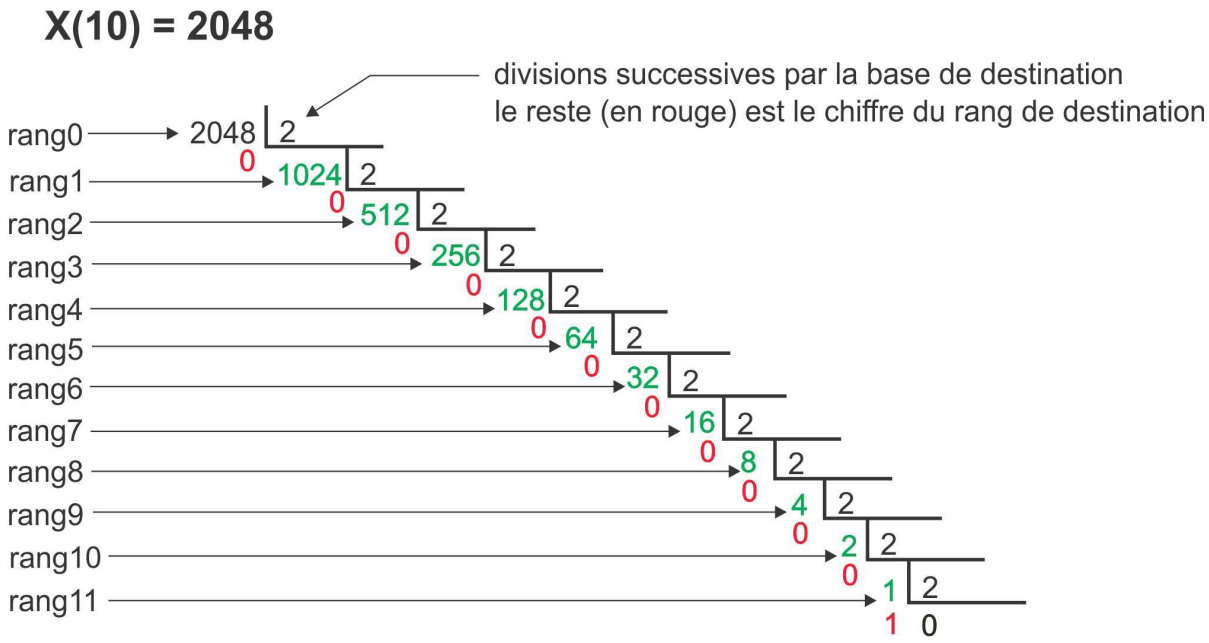
Riche de ces enseignements nous pouvons traduire un nombre binaire en son équivalent décimal.



Voyons maintenant comment convertir un nombre décimal en binaire.

La méthode est la suivante : nous allons diviser successivement ce nombre par la base de destination puis ranger chaque poids trouvé dans son rang.

Soit à convertir le nombre décimal 2048



rangs :	11	10	9	8	7	6	5	4	3	2	1	0
X(2) =	1	0	0	0	0	0	0	0	0	0	0	0

Le bit du rang le plus faible ou poids le plus faible se nomme communément le **LSB** (**L**east **S**ignificant **B**it), le bit du rang le plus fort ou poids fort se nomme **MSB** (**M**ost **S**ignificant **B**it).

On dit que l'équivalent binaire du nombre entier décimal **2048** est **100000000000.**

Nous voyons que le 1 se situe sur le rang 11, nous avons vu précédemment que pour retrouver l'équivalent décimal de ce nombre binaire il nous suffit d'élever 2 à la puissance 11 ce qui fait bien 2048.

Prenons un autre exemple : $X_{(10)}$ = 45

45 :2 = 22 → reste **1** **(rang 0 => LSB)**
22 :2 = 11 → reste **0**
11 :2 = 5 → reste **1**
5 :2 = 2 → reste **1**
2 :2 = 1 → reste **0**
1 :2 = 0 → reste **1** (dividende inférieur au diviseur => fin du calcul de division) → **MSB**

Soit $X_{(10)}$ = **45** a pour équivalent binaire $X_{(2)}$ = **101101**

Nous savons désormais traduire un nombre entier décimal en binaire et réciproquement.
Nous allons voir maintenant comment employer ces principes pour effectuer des conversions entre différentes bases telles que l'**héxadécimale (ou encore héxa pour les intimes)**.

La base héxadécimale, très utile en électronique et informatique, est la base 16, elle se compose donc de 16 éléments notés : {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}

Ce code hexadécimal à la particularité de pouvoir être codé avec 4 bits ce qui rend très commode la conversion par paquet de 4 bits d'un code binaire et le rend plus compact et plus facilement transposable en limitant les erreurs.
Ainsi il suffira de deux chiffres hexadécimaux pour représenter un **octet** (8 bits).
Un octet se dit ***byte*** en anglais à ne pas confondre avec ***bit*** qui est l'unité ou chiffre binaire.

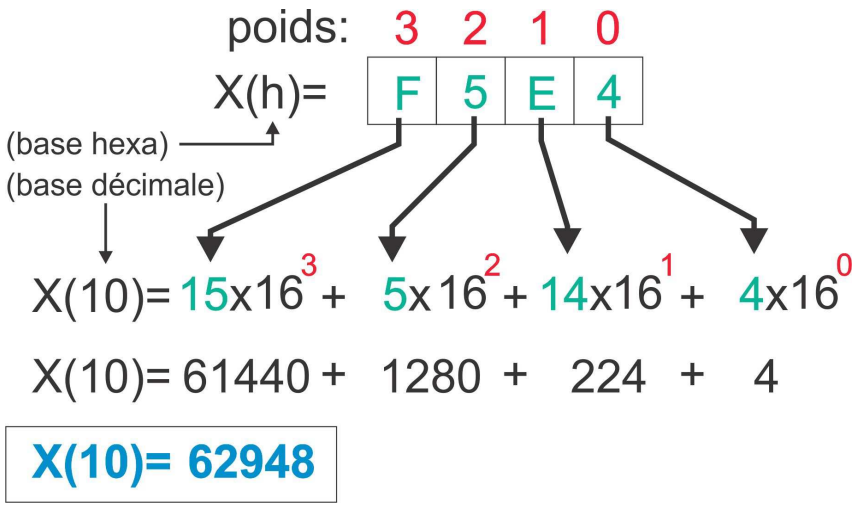
Nous pouvons dresser le tableau d'équivalences entre les trois bases :

EQUIVALENCE HEXA/DECIMAL/BINAIRE

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

Voyons comment convertir un nombre hexadécimal en nombre décimal puis comment convertir un nombre binaire en hexadécimal.

Soit à convertir le nombre Hexadécimal F5E4 en décimal :



Décortiquons la technique utilisée pour en déduire le mécanisme :

1. On traduit les lettres de chaque rang en valeur décimales
2. Pour chaque rang on multiplie sa quantité par la base d'origine élevée au carré du rang
3. On additionne en ligne chaque terme obtenue pour chaque rang, le terme de poids fort à gauche, le terme de poids faible à droite.

Pour retrouver le nombre hexadécimal d'origine à partir de ce résultat on va effectuer des divisions entières successives par la base de destination qui est la base hexadécimale donc 16 :

62948 : 16 = 3934 reste 4 (rang 0)
3934 : 16 = 245 reste 14 (rang 1)
245 : 16 = 15 reste 5 (rang 2)
15 : 16 = 0 reste 15 (rang 3)

On va maintenant écrire en ligne les restes de chaque rang en allant de gauche à droite, à gauche se trouvant le rang de poids fort donc le rang 3 et en finissant par celui du rang 0.

Ce qui donne 15|5|14|4

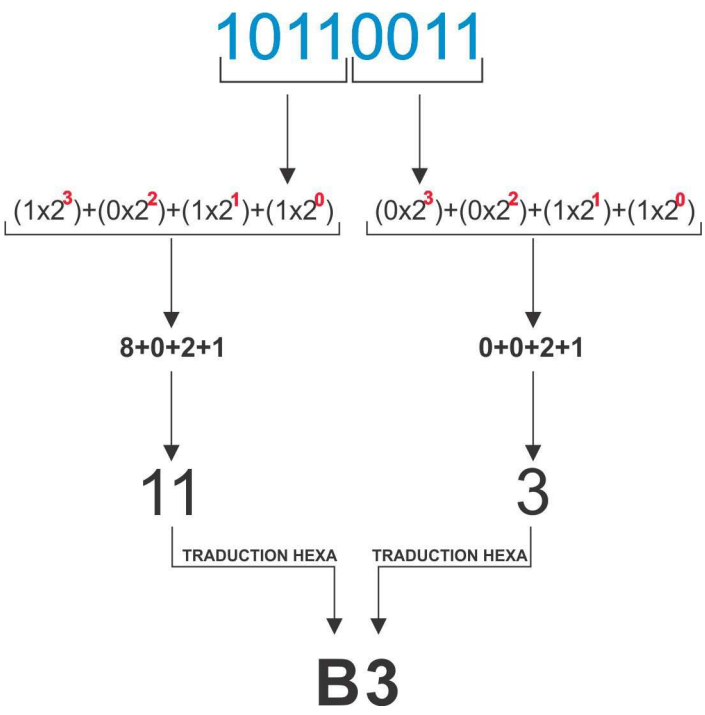
On n'oublie pas que nous avons transposé en base 16 donc il nous faut remplacer la symbolique décimale par la symbolique hexadécimale :

La symbolique de 15 en hexa est F
La symbolique de 5 en hexa est 5
La symbolique de 14 en hexa est E
La symbolique de 4 en hexa est 4

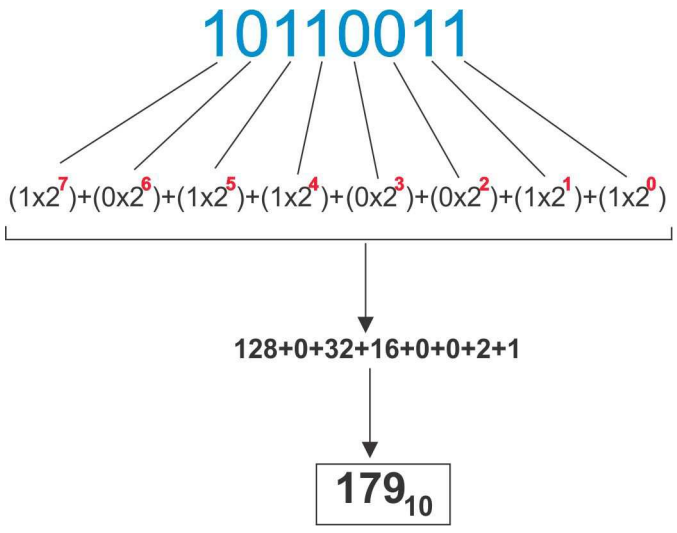
On retrouve bien le résultat d'origine : **F5E4**

Soit à convertir le nombre binaire : 10110011 en nombre hexadécimal

Nous savons qu'un nombre hexadécimal se code avec 4 bits donc nous allons effectuer **des groupements de 4 bits** et traduire chaque élément séparément.



Si nous transposons **10110011** en décimal nous faisons :



Maintenant transposons le nombre hexadécimal B3 en décimal :

$$B3_{16} = (Bx16^{1}) + (3x16^{0})$$
$$B3_{16} = (11x16) + (3x1)$$
$$\boxed{B3_{16} = 179_{10}}$$

Avec un peu d'habitude il est très facile de passer du binaire à l'hexa, ces regroupements par 4 bits faisant appel à des manipulations très abordables par simples calculs mentaux.

Le code BCD :

Nous pouvons représenter les 10 symboles du système décimal grâce à un demi-octet.

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

L'avantage est de pouvoir représenter directement une valeur binaire en valeur décimale et réciproquement.
De nombreux circuits utilisent ce mode de codage, il faut donc bien garder à l'esprit son existence.

Il ne faut juste pas mélanger deux choses :

Ecrire en binaire 1000 0100 0010 1001 revient à écrire en BCD → 8429

Ce qui ne signifie pas que 1000 0100 0010 1001 = 8429

C'est une **représentation, un codage,** pas une égalité numérique.

Le BCD est par exemple utilisé dans les horloges temps réel, les registres qui contiennent les secondes, minutes etc sont représentés avec ce système de codage.
L'avantage est encore une fois de manipuler des demi octets seulement et d'avoir une conversion directe pour chaque chiffre décimal.

Complément à un et complément à deux :

Jusqu'à présent nous n'avons parlé que de nombres entiers positifs, mais nous aurons à traiter également des nombres négatifs.
Le complément à deux permet de trouver directement le nombre négatif en connaissant un nombre positif.

Par exemple soit le nombre +55.

Son équivalent binaire sur un octet s'écrit : 00110111

Le complément à 1 consiste à inverser chaque chiffre binaire.
Le complément à 2 consiste à ajouter 1 au complément à 1 trouvé précédemment.

```
 00110111
 11001000 → complément à 1
+00000001 → on ajoute 1
_________
=11001001 → complément à 2 donne le résultat signé
```

Un nombre binaire signé utilise le 8eme bit pour distinguer le signe du nombre, 0 représente un nombre positif, 1 représente un nombre négatif.

Ainsi **1**1001001 en écriture signée représente **-55**

Avec 8 bits en écriture signée nous voyons facilement que seuls 7 bits pourront permettre d'écrire la valeur absolue d'un nombre.

Donc les valeurs min et max représentables sont **-128** et **+127** soit **256** valeurs (ne pas oublier de compter le 0).

En écriture non signée les valeurs positives représentables sont 256 valeurs (de 0 à 255 en décimal ou encore de 0 à FF en hexa).

De la même manière nous pouvons voir que pour deux octets soit 16 bits nous avons les résultats suivants :

Ecriture signée : le plus petit **1000 0000 0000 0000** et **0111 1111 1111 1111** pour le plus grand.

0111 1111 1111 1111 → +32767
1000 0000 0000 0000 → -32768

Ce qui s'écrit dans les différentes bases que nous connaissons :

Pour un octet : 1000 0000 → $(-128)_{10}$ → $(80)_{hexa}$
0111 1111 → $(+127)_{10}$ → $(7F)_{hexa}$

Pour un mot de 16 bits soit deux octets : min → 1000 0000 0000 0000 → $(-32768)_{10}$ → (**8000**)$_h$
max →0111 1111 1111 1111 → $(+32767)_{10}$→ (**7FFF**)$_h$

Identification et reconnaissance des nombres dans un programme :

Faisons une petite parenthèse pour voir comment les nombres dans différentes bases seront identifiés dans un programme, même si nous y reviendrons par la suite de l'exposé.
Nous avons pour le moment noté ceux-ci avec une écriture mathématique comme par exemple $(55)_{10}$ en base décimale ou encore $(7FFF)_{16}$ en hexadécimal ou $(10001100)_2$ en binaire.

Pour le décimal c'est simple on oublie de noter les parenthèses ainsi que la base, normal puisque c'est par défaut la base que nous utilisons couramment chaque jour.

Pour l'hexadécimal on rencontre en informatique les suffixes suivants : **$**, **h** ou **0x**

Par exemple : **$**7FFF ou 7FFF**h** ou encore **0x**7FFF

Pour le binaire ont rencontre deux écritures : **0b** ou **% (retenez surtout le premier le second est utilisé aussi pour le reste de la division euclidienne en C…)**

Quand on se dit que même pour les nombres les informaticiens n'ont pas réussi à se mettre d'accord on en déduit que ça va pas être simple tous les jours…

Il va également falloir vous habituer à compter mentalement pour traduire un nombre binaire en décimal, en tout cas sur 8 bits dans un premier temps, c'est vraiment une formalité vous verrez avec un peu d'entrainement.

La méthode est toute simple il vous suffit d'additionner de **droite à gauche** les bits à 1 selon leur poids binaire : 1, 2, 4, 8, 16, 32, 64, 128 ou bien de prendre votre calculatrice pour les moins courageux.
Pour l'hexa nous avons déjà vu qu'il suffit de faire des groupements par 4 bits et de procéder de même sans oublier de traduire ce qui est au dessus de 9 par la lettre qui va bien dans cette base.

2. La logique combinatoire : NON, ET, OU, NON ET, NON OU, XOR…

Voici un volet important et incontournable dans l'apprentissage d'un langage informatique, car aussi bien qu'en électronique il va nous falloir effectuer des opérations logiques en faisant appel à l'algèbre de Boole (merci donc à Mr George Boole qui en a établi les principes mathématiques).

Je ne vais pas ici faire un cours approfondi sur la logique combinatoire, ni sur les méthodes de résolutions d'équations avec les tableaux de Karnaugh et autres amusements mais tenter de donner les grandes lignes et résultats qui nous seront utiles en programmation.
Nous y verrons par contre les astuces à retenir au fur et à mesure lorsque une application concrète orientée programmation se présentera, l'application étant notre objectif avant tout.

La logique combinatoire qui nous intéresse repose sur la base 2 donc fait appel à deux états possibles étant les 2 éléments de cette base comme nous l'avons déjà vu : 0 et 1.
Sommairement cela consiste à présenter une ou des valeurs variables en entrée d'un opérateur logique et d'en obtenir un résultat prédéterminé en sortie.

Pour cela il nous faut établir des règles de base, nous allons donc postuler que 0 représente un NON et 1 un OUI, ou encore que 0 est FAUX et 1 est VRAI (logique positive).
L'analogie en électricité serait de dire que 0 est éteint et 1 allumé.

Pour le moment afin de simplifier le propos nous allons traiter 2 variables d'entrée que nous appellerons A et B, la variable de sortie ou résultat s'appellera Z.

- **Opérateur logique NON (ou fonction complément)**

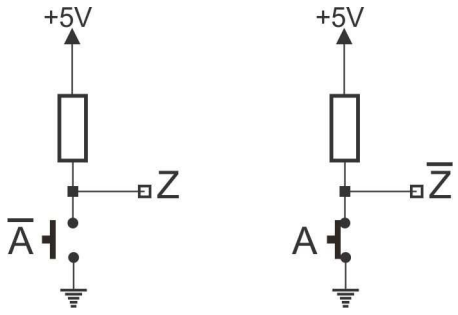
Si **A=0** on peut dire que **NON** A=1 nous avons affaire à une seule entrée à laquelle nous lui affectons un opérateur de négation cela implique que le résultat en sortie est l'inverse de celui présenté à l'entrée.
L'écriture générale sera donc : **$A=\bar{Z}$ (La barre symbolisant l'état inverse de Z ou NON Z)**

Le tableau suivant résume tous les états possibles de Z selon les valeurs prises par A :

A	Z
0	1
1	0

$Z=\bar{A}$

Exemple concret : conversion d'un état mécanique en état électrique inverse



Lorsque le bouton poussoir est à l'état repos (A=0 => $\bar{A}$) => Z est à 1 (+5V)
Si on appui sur le bouton poussoir (A=1) => Z est à 0 => $\bar{Z}$ (0V)

Nous avons converti une information d'état mécanique (le bouton est enfoncé ou non) en un état électrique (Z est à 0V ou à +5V).

De manière générale une variable barrée indique un état 0 et non barrée un état 1, il est en effet plus fréquent d'utiliser la logique dite positive, mais ce n'est qu'une affaire de **convention**, le tout étant de s'y tenir et de ne pas en changer en cours de route, sans quoi il n'y a plus de cohérence… logique.

→ Une analogie électrique de convention : le sens de parcours du courant ou le sens des fléchages des tensions dans une maille.

Exemples d'applications du NON : le complément à 1 d'un octet, le clignotement d'une led, etc

Nous avions vu plus tôt que le complément à 1 consistait à inverser chaque bit d'un octet par exemple.

Nous verrons plus en détail en programmation C qu'il existe deux moyens d'exprimer une négation logique selon si l'on souhaite inverser une variable booléenne ou chaque bit d'un mot binaire.

Pour commencer à vous y habituer je vous les présente car ils vous seront très utiles pour la suite :

l'opérateur qui **inverse bit à bit** un nombre binaire → **~**

l'opérateur de **négation logique** → **! (ne retourne que 1 ou 0)**

Si **etat_poussoir = 0** cela implique que **! etat_poussoir = 1**

Si **PORTA = 0b01101110** alors **~PORTA= 0b10010001**

Voilà il suffit juste de le savoir et de ne pas se tromper entre les deux symboles et leur implication respective, bien identifier si l'on traite d'une **négation logique** ou d'une **complémentation logique.**

Nous verrons qu'en C la variable de type bit n'existe pas concrètement, dans les faits le C ne sait traiter que des octets, on accède aux bits élémentaires par des artifices tels que les champs de bits que nous verrons quand nous serons entrés de plein pied dans le langage **C** proprement dit.

- ## Opérateur logique ET (AND)

L'opérateur ET logique consiste à donner un résultat Z=1 ou VRAI si les deux variables A et B sont à 1, si au moins une des deux entrées est à 0 le résultat sera 0 ou FAUX.

Le tableau suivant résume les différents états possibles de Z selon A et B :

A	B	Z
0	0	0
0	1	0
1	0	0
1	1	1

$$Z = A.B$$

Pour ce souvenir de cette table il est pratique de savoir que le point représente la multiplication, donc on voit bien que Z est le résultat du **produit logique** des deux variables.

En C afin de distinguer l'opérateur arithmétique produit et l'opérateur logique ET on utilise le symbole suivant pour ce dernier : **&**

- **Opérateur logique OU (OR)**

L'opérateur OU logique consiste à donner un résultat Z=1 ou VRAI si au moins une des deux variables prend la valeur 1.
Ce qui se traduit par le tableau suivant :

A	B	Z
0	0	0
0	1	1
1	0	1
1	1	1

Z=A+B

Là encore pour s'en souvenir on se rappelle que le signe + de l'addition symbolise l'opérateur OU et Z est le résultat de l'**addition logique** des deux variables.
Nous pouvons remarquer qu'il existe un élément neutre pour la fonction ET et la fonction OU :

A+**0**=A donc 0 est élément neutre pour la fonction OU.
A.**1**= A donc 1 est élément neutre pour la fonction ET

Il existe également un **principe de dualité** entre ces deux fonctions logiques, on peut en effet remplacer **+** par **.** et 1 par 0 et réciproquement.

Exemple : A+1 = 1 est équivalent à A.0 = 0

Nous verrons qu'en C afin de ne pas confondre avec l'opérateur arithmétique d'addition nous adopterons un autre symbole pour représenter l'opérateur OU logique : |

- **Opérateur logique NON ET (NAND)**

Cet opérateur est ce qu'on appelle en logique un **opérateur logique complet,** car toutes les fonctions logiques sont réalisables avec une combinaison utilisant ce seul opérateur.

On s'en doute un peu le **NON ET** réalise une fonction ET complémentée (appelée aussi fonction de Sheffer), dont le tableau est facile à déduire ayant vu au préalable l'opérateur ET :

A	B	Z
0	0	1
0	1	1
1	0	1
1	1	0

$Z = \overline{A.B}$

- **Opérateur logique NON OU (NOR)**

Cet opérateur est également un opérateur logique complet, il réalise une fonction OU complémentée (appelée aussi fonction de Pierce) dont le tableau suivant en illustre les résultats :

A	B	Z
0	0	1
0	1	0
1	0	0
1	1	0

$$Z = \overline{A+B}$$

Pour les opérateurs que nous venons de voir jusqu'à présent nous avons à chaque fois représenté un tableau à deux variables mais bien entendu tout cela reste vrai quelque soit le nombre des variables d'entrées.

Tout ça pour introduire deux nouveaux opérateurs qui eux n'admettent que **deux variables** seulement :

- **Opérateur logique OU EXCLUSIF (XOR)**

Cet opérateur retourne un résultat VRAI si une et une seule variable est égale à 1.
Ce qui nous donne le tableau suivant :

A	B	Z
0	0	0
0	1	1
1	0	1
1	1	0

$$Z = A \oplus B$$

Vous noterez le symbole qui représente cet opérateur, nous verrons qu'en C nous adopterons un autre symbole plus commode à taper dans un code : ^

- **Opérateur logique ET EXCLUSIF (ou opérateur identité)**

Cette fonction est VRAI si et seulement si les deux variables ont la même valeur logique.

D'où le tableau correspondant :

A	B	Z
0	0	1
0	1	0
1	0	0
1	1	1

$$Z = A \odot B$$

$$Z = \overline{A \oplus B}$$

Vous aurez sans doute remarqué que cet opérateur est le complément du OU EXCLUSIF vu précédemment.

3. Décalage à droite, décalage à gauche, masquage

- **Décalage à droite : X >> Y**

X >> Y signifie que l'on va décaler les bits de X de Y bits vers la droite donc dans le même sens que les 2 flèches.
Les conséquences d'un décalage à droite sont que les bits de poids faibles sont perdus et que l'on introduit des 0 par la gauche donc par les bits de poids fort.

Exemple : 10110001 on veut faire un décalage de 3 bits vers la droite

10110001 >> 3

Premier décalage : **01011000** => un premier 0 est introduit à gauche et le bit de poids faible (MSB) est perdu à droite, il ne fait plus partie de l'octet dans notre exemple.

Deuxième décalage : **00101100** => un deuxième 0 est introduit à gauche et le deuxième bit de poids faible initial est perdu à droite.

Troisième décalage : **00010110** => **résultat final**

Si on prend la valeur initiale de notre nombre avant décalage il valait **177** en décimal**.**
Après décalage il vaut 22.

Si nous divisons 177 par $\mathbf{2}^{\mathbf{3}}$ nous obtenons 22 en valeur entière**.**

Donc décaler un nombre binaire de Y vers la droite revient à diviser ce nombre par $\mathbf{2}^{\mathbf{Y}}$

- **Décalage à gauche : X << Y**

X << Y signifie que l'on va décaler les bits de X de Y bits vers la droite donc dans le même sens que les 2 flèches.
Les conséquences d'un décalage à gauche sont que les bits de poids forts sont perdus et que l'on introduit des 0 par la droite donc par les bits de poids faibles.

Exemple : 00010001 << 3 => 10001000 est le résultat final

On voit que le nombre initial valait **17** en décimal et vaut **136** après décalage.

Or **136 = 17 + 2** ^{3} donc **décaler à gauche de Y revient à multiplier le nombre X par 2** ^{Y}

A quoi ça vient bien pouvoir nous servir en C tous ces décalages ?

Quand on programme un microcontrôleur il est fréquent de devoir reconstituer une donnée qui se trouve partagée dans deux registres différents.
Par exemple quand on fait une conversion analogique/digitale sur 10 bits ou 12 bits le résultat se trouve dans le registre ADRESL et ADRESH, pour pouvoir récupérer la valeur dans sa globalité on va utiliser les décalages.
ADRESL est un octet qui contient les 8 bits bas du résultat et ADRESH contient les bits hauts restant, il faut donc transvaser tout ça dans une variable plus grande (2 octets) pour reconstituer la valeur totale que l'on pourra manipuler plus facilement.
Un autre exemple peut être de faire défiler l'une après l'autre une led sur un PORT dans ce cas il est simple d'écrire PORTx << y en incrémentant automatiquement y dans une boucle comme nous le verrons plus loin.
Les décalages permettent également de faire de la conversion binaire/BCD et réciproquement.

Il est également très utile de pouvoir extraire l'état d'un bit dans un registre ou de pouvoir le modifier sans perturber les autres, ou encore de modifier d'un coup la moitié d'un octet ou plusieurs octets en même temps, c'est le rôle des **masques** que nous allons voir tout de suite.
Les masques vont permettre de comprendre tout l'intérêt de ce que vous avez appris plus haut avec les opérateurs logiques.

Ne pas perdre de vue qu'en maitrisant parfaitement les premiers outils que nous venons de voir la suite de l'exposé ne devrait pas vous poser de problèmes insurmontables, car le C comme tous les langages de programmation utilisent des concepts liés étroitement à la logique et aux manipulations de nombres pour pouvoir modifier ou extraire les informations utiles contenues dans le cœur du microcontrôleur, faire des tests, des calculs, afficher, etc.

- **Masques**

Cette technique porte bien son nom, le but étant de cacher la partie que l'on ne veut pas exposer lorsqu'on souhaite affecter une partie d'une donnée.
Un masque sert à modifier un ou des bits d'un nombre binaire ou à les tester (les lire).
Par exemple si on souhaite activer un timer ou si on souhaite savoir si un évènement a été détecté dans un registre tel que par exemple un caractère reçu dans le buffer de réception d'un UART.

Tout de suite des exemples concrets de mise en œuvre pour chaque type de masquage :

On veut mettre à 1 les bits 3 et 5 du nombre binaire suivant : 01000111 → **01101111**

Pour masquer tous les bits que l'on ne veut pas modifier on va associer un opérateur avec un nombre qui n'affectera que les bits que l'on veut modifier, cet opérateur ici est le **OU** et on se rappelle pour l'avoir déjà vu plus haut que **le 0 est un élément neutre** pour cet opérateur logique.

```
  01000111   (opérande 1)
+ 00101000   (opérande 2 = masque)
```

01**1**0**1**111 = résultat

En C pour réaliser cela nous écrirons :

```
resultat = 0b01000111 | 0b00101000 ;  /* résultat contient 0b01101111 */
```

On veut mettre à 0 les bits 3 et 5 du nombre binaire suivant : 01101111 → **01000111**

Cette fois nous allons utiliser l'**opérateur logique ET** associé à son **élément neutre 1** pour ne pas affecter les bits que nous ne voulons pas modifier.

```
  01101111   (opérande 1)
& 11010111   (opérande 2 = masque)
```

01**0**0**0**111 = résultat

En C nous écrirons :

```
resultat = 0b01101111 & 0b11010111  /* resultat contient 0b01101111 */
```

On veut connaître l'état du bit 3 du registre suivant : b7b6b5b4b3b2b1b0 → **00001000**

Pour connaître l'état d'un bit dans un registre nous allons appliquer un masque qui va nous retourner à coup sur la valeur du bit à la position que nous désirons.
Ce genre de manip est utilisée pour détecter, surveiller un changement d'état, comme par exemple lorsque une conversion A/D est terminée ou lorsqu'on surveille le ***busy flag*** pour un afficheur LCD.

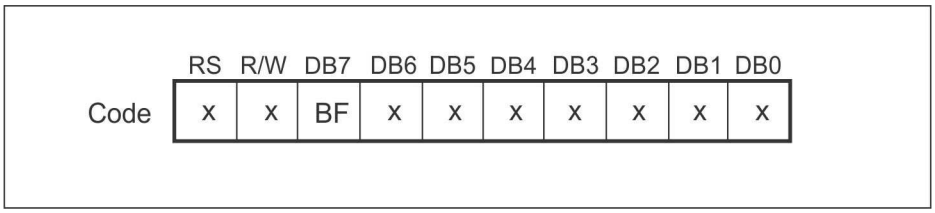
L'opérateur logique que nous allons utiliser ici est le XOR ou OU EXCLUSIF.
Nous savons que pour le **XOR le 0 est l'élément neutre et le 1 l'élément absorbant**, en d'autres termes **si nous affectons un 0 à notre masque** et que la valeur retournée est un 1 cela voudra dire que la valeur cherchée est forcément 1 et réciproquement **si la valeur retournée est 0** c'est que **la valeur cherchée est 0** puisque **le XOR retourne 1 si et seulement si une seule des deux variables est positionnée à 1**.

Voyons tout ça à travers un exemple concret d'application :

Lorsqu'une commande est envoyée à un afficheur LCD, le LCD est positionné en mode écriture puis on envoie sur son bus la commande et la donnée, comme par exemple écrire le caractère « C » sur la première ligne, première colonne.
Pendant le temps que le contrôleur de l'afficheur est occupé à effectuer cette tâche il indique qu'il est occupé et ne peut recevoir d'autres données en positionnant le bit7 du bus de data à 1, d'où le nom donné à ce bit « busy flag » ou « drapeau occupé ».
Une fois le travail accompli le contrôleur de l'afficheur remet à 0 ce bit pour signifier qu'il a terminé son travail et que l'on peut lui renvoyer d'autres ordres, c'est là précisément que nous allons nous mettre en lecture du bus de données pour attendre que ce bit retombe à 0, c'est là que l'emploi du masque va nous être utile.
Il est très fréquent de lire sur les forums que la plupart se contentent, pire, conseillent à d'autres, de mettre une simple temporisation pour laisser le temps à l'afficheur de faire son travail.
C'est jouer avec le feu car rien ne peut mieux garantir que le travail est réellement fait que par celui qui justement le fait, comme dans la vraie vie en somme… chaque afficheur n'ayant pas tout à fait les mêmes caractéristiques ni les mêmes contrôleurs et inutile de faire perdre du temps au reste de l'application à attendre pour rien.

Vous allez voir combien cette possibilité sous estimée du test du busy flag est d'une simplicité d'emploi déconcertante et c'est simple ignorance que de s'en passer.
Cela illustre bien que la méconnaissance des choses s'impose souvent avec plus de force, comme quoi *force n'est pas loi* comme dis le dicton et donc bien lire les documents, là est un des secrets de la réussite dans ce métier.

Voici comment se présente les différentes broches d'accès au controleur d'un afficheur LCD alphanumérique :



On y trouve RS : Register Select ou sélection de registre
R/W : Read/Write ou Lecture/Ecriture (1 → lecture 0 → écriture)
DB7 ~ DB0: Data Bus ou bus de données sur 8 bits

Il y a également une broche E non représentée (Enable ou activation) que nous considérons par défaut à 1.

Nous n'allons pas ici entrer dans le détail des choses, nous reverrons cela de manière plus approfondie dans les applications proprement dites, mais sachez que ce type d'afficheur peut être géré par 4 bits seulement en l'occurrence les 4 bits de poids forts, l'exemple sera traité pour le moment en mode 8 bits.
L'intérêt tout relatif est de gagner 4 fils, par contre on ralenti les transferts puisqu'il faut deux temps pour envoyer les données au lieu d'un temps en 8 bits.

→ Envoyer une séquence permettant d'afficher le caractère « C » sur la première ligne, 3eme colonne.

(…. A suivre)