12. REPRESENTATION DES NOMBRES, ERREURS

12.1. Généralités sur les erreurs et précisions

Cette partie s'applique à tout système de numération, au décimal comme au binaire.

12.1.1. Définitions

12.1.1.1. Erreur absolue

C'est la différence entre la valeur calculée ou mesurée et la valeur exacte. Elle est positive ou négative. Elle s'exprime le plus souvent avec une unité.

- ***Sur une variable ou résultat d'un calcul***

On ne connaît que l'erreur maximale.

Si l'erreur est toujours de même signe, on parle d'**erreur max** (par excès) ou (par défaut).

$x = 5,67$ avec $dx_{max} = + 0,01$ signifie que x est compris entre 5,67 et 5,68.

Il vaut mieux s'arranger pour que l'**erreur** soit « **Centrée** » et dire alors:

$x = 5,675$ avec $dx_{max} = \pm 0,005$ alors x est compris entre 5,67 et 5,68

Pour une **variable** après arrondi au plus près, **l'erreur max** est donc toujours :

±½ du pas de quantification choisi

Pour un résultat d'un calcul, d'autres erreurs interviennent évidemment, voir plus loin.

- ***Sur une constante***

Elle est à priori connue :

Si k =4,373 et si on ne garde 3 chiffres, k= 4,37 avec une erreur dk de -0,003

Si k =4,375 et si on ne garde 3 chiffres, k= 4,37 avec une erreur dk de -0,005

Si k =4,370 et si on ne garde 3 chiffres, k= 4,37 avec une erreur dk de 0 !

On voit que pour un même format (ici 3 chiffres décimaux) l'erreur peut aller de 0 à la valeur max possible comme pour une variable. C'est la même chose en binaire.

Points très importants :

Si la valeur de k est par hasard, proche d'un pas de quantification binaire, l'erreur est très faible voire quasi nulle avec bien moins de bits que le nécessite la théorie. Exemple : 0,7500 se code avec une erreur nulle avec seulement 2 bits fractionnaires (,11). La théorie pour 4 chiffres fractionnaires nécessiterait au minimum 14 bits(voir plus loin).

Codage de coefficients:

On doit souvent coder des valeurs particulières de constantes (coefficients de réglage, d'étalonnage) susceptibles en fait d'être modifiées pour s'adapter par exemple à tel ou tel environnement physique. Il faut que le programme marche dans tous les cas, et garantisse la précision finale souhaitée. On doit donc absolument raisonner **pour ces constantes** sur **l'erreur maximale possible** dans le format choisi, comme pour des variables (Sinon une référence change, les constantes doivent être modifiées même très légèrement et le programme n'est alors plus utilisable, ou sinon ne fournit plus la même précision. !

Vrai constantes :

Ce n'est que pour de vraies constantes qui ne changeront jamais (par exemple coefficient 1/3 ou 1/2 ou 3/4 lors de calculs), que l'on a intérêt à prendre l'erreur minimale possible dans le format choisi (qui peut alors être nulle !)

12.1.1.2. Erreur relative

Si on divise par la grandeur elle même, on parle d'erreur relative **dx/x**, sans dimension, que l'on peut exprimer en **%**. Pas toujours significative car tend vers l'infini si x tend vers 0.

12.1.1.3. Précision et incertitude

Lorsqu' elle est Centrée, l'erreur maximale porte le nom de précision ou d'incertitude absolue ou relative. On note : Δx et $\frac{\Delta x}{x}$ les incertitudes absolues et relatives sur x.

12.1.2. Arrondi ou troncature, technique

On est forcé très souvent de limiter le nombre de chiffres d'une variable ou d'un résultat :

Soit parce que la précision est totalement inutile ou n'a aucun sens (chiffres non significatifs) : par exemple une tension de 12,564979808 volts, une dimension de 2,54678 mètres pour la longueur d'une table en bois …. !

Soit pour les calculs intermédiaires (exemple : lors des produits, le nombre de chiffres augmente sans arrêt).

a)**troncature** La troncature brutale d'un certain nombre de chiffres de faible poids fournit une valeur par défaut ou par excès (le signe de dx dépend du signe du nombre, et du type de code utilisé)

b)**arrondi au plus près** l'erreur est centrée

Ex en décimal: 2,35427 tronqué: 2,3542 $dx = +7.10^{-5}$

arrondi au plus prés: 2,3543 $dx = -3.10^{-5}$

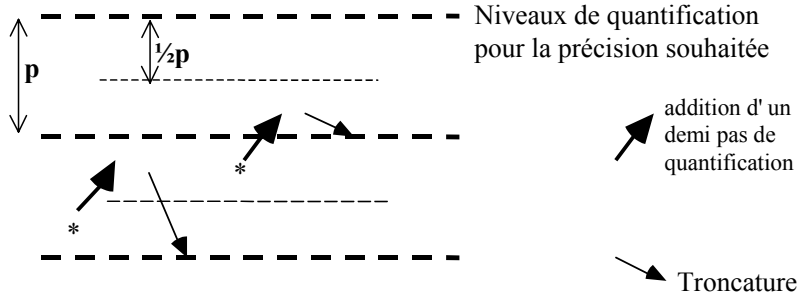
Avantages : L'erreur maximale est moindre

Suppression des effets cumulatifs d'erreurs, très important pour certains calculs répétitifs.

On doit donc souvent effectuer cette opération.

Méthode toute simple (pour tout système de numération et code) :

Pour arrondir **x** à un chiffre de **poids faible donné**, donc pour obtenir un résultat avec un pas de quantification **p**, on ajoute à la **valeur absolue ½ du pas de quantification** conservé, donc on fait **+½ p**) avant de tronquer. L'erreur maximale est alors **dx = ± ½ p**



Exemples :

Pour arrondir 2,367 à l'entier le plus proche, on fait 2 ,367 + 0.5 = 2,867 → 2

Pour arrondir 2,667 à l'entier le plus proche, on fait 2 ,667 + 0.5 = 3,167 → 3

Pour arrondir -2,667 à deux chiffres après la virgule, on fait : $-(2{,}667 + \frac{0.01}{2}) = -(2{,}667 + 0{,}005) = -2{,}672 \rightarrow -2{,}67$

Avantage en binaire et pour le code complément à deux que l'on reverra, il n'est pas nécessaire de passer par la valeur absolue, quelque soit le signe on pourra faire +1/2 pas (donc +1 sur le poids juste inférieur) et tronquer. Nous reverrons ceci bientôt.

12.1.3. Erreur sur le résultat d'une opération ou d'un calcul.

Le plus simple et le plus rapide est un **calcul d'erreur**, en cherchant les **erreurs maximales** d'un calcul, **sans s'occuper de la fréquence** de ces erreurs, **ni de compensations statistiques** éventuelles.

Une autre approche existe, (en traitement du signal par exemple), elle s'appui sur la statistique, et consiste à calculer les bruits d'échantillonnage, les bruits d'arrondis,..., nous ne l'étudierons pas ici.

Il faut tenir compte :

-**Des erreurs des paramètres d'entrée** (ou incertitudes), sur les **variables** mais aussi sur les **constantes introduites** (que l'on traite comme des variables si ce sont des coefficients de réglage et non de vraies constantes mathématiques). Le **calcul** différentiel permet de trouver facilement l'erreur finale sur le résultat, et la précision finale dans le pire des cas (passage aux incertitudes).

-**Des erreurs dues à la limitation du nombre de bits après certaines opérations** (multiplication par exemple). Il faut toujours effectuer si possible les calculs intermédiaires avec davantage de bits.

-**Des erreurs de méthode de calcul** (développements plus ou moins linéaires...)

Toutes ces erreurs s'ajoutent en principe, et le calcul reste simple ou du moins pas trop compliqué, si on considère ce qui est plus simple, toutes les erreurs centrées. (Des compensations peuvent parfois s'effectuer avec des arrondis par défauts ou par excès mais l'étude est alors bien plus délicate ! ! !)

12.1.4. Importance de la précision des opérandes de départ

12.1.4.1. Cas « standards »

Le calcul d'erreur par les méthodes classiques (calcul différentiel et petites variations) fournit évidemment le résultat. Dans de nombreux cas, il faut des opérandes un peu plus précis que la précision finale souhaitée, mais sans plus.

12.1.4.2. Cas plus délicats

Il peut être bon de mettre en évidence quelques cas surprenant, où la précision des opérandes de départ doit être étudiée avec le plus grand soin si on veut éviter des résultats aberrants.

Donnons un exemple de calcul ou le résultat peut être entaché d'une forte erreur, même sans aucun dépassement, si on ne prend pas la précaution de travailler dès le départ avec un grand nombre de bits significatifs.

On cherche par exemple les racines de l'équation :

$$f(X) = 0{,}1.X^2 + 300.X + 2 = 0$$

- **calcul avec 10 chiffres décimaux significatifs.**

  $\sqrt{\Delta} = \sqrt{(90000 - 0.8)} = 299.9986667$

  $x1 = (-300 + \sqrt{\Delta})/0.2 = -0.0066665$ $\quad f(x1) = 5{,}4.10^{-5}$,correct

  $x2 = (-300 + \sqrt{\Delta})/0.2 = -2999{,}9933$

- **calcul avec 6 chiffres décimaux significatifs**

  au plus près $x1 = (-300 + 299{,}999)/.2 = -0.005$ **erreur de 24%**

  pour x2, pas de problème:

  $x2 = (-300 - 299{,}999)/0{,}2 = - 2999{,}99$

- **calcul avec 5 chiffres décimaux significatifs**

  au plus près : $x1 = (-300 + 300)/0{,}2 = 0$ **erreur de 100%**

en tronqué x1 = (-300 + 299,99)/0,2 = -0,05 erreur **600%** !
x2 = (-300 -299,99)/0,2 = -2999,95 erreur 10^{-4}

➢ **Conclusion !**

Une **forte erreur** peut survenir lors de **LA SOUSTRACTION DE NOMBRE DE VALEUR TRES VOISINE** (l'erreur d'arrondi se reporte alors sur les forts poids).

➢ **Remèdes :**

a) Trouver une autre formule pour calculer x1, on sait que (produit des racines) x1.x2 = 2/0,1 = 20

Le calcul avec 5 chiffres décimaux significatifs fournirait :

x1 = 20/(-299,999) = -0,0066665 en tronqué, ce qui est acceptable, précision 10^{-4} .

b) Garder un grand nombre de bits significatifs (pour avoir 10 chiffres décimaux), et ne limiter qu' après la soustraction.

12.1.4.3. Arrangement des expressions pour limiter les erreurs.

Nous allons prendre l'exemple du calcul de A - B qui peut, comme nous l'avons vu introduire de fortes erreurs.

➢ **A et B sont deux opérandes quelconques.**

Aucun remède si ce n'est de travailler avec plus de précision, ou de trouver une autre formule fournissant directement A-B. (ou une expression contenant ce A - B) .

➢ **A et B dérivent de la même fonction** $A = f(x + e)$ et $B = f(x)$

a) On écrit la différence sur une autre forme:

si f(x) = 1/x, $f(x + e) - f(x) = \frac{1}{x+e} - \frac{1}{x} = \frac{-1}{x(x+e)}$

b) Utilisation des dérivées :

$f(x + e) - f(x) = e\, f'(x)$ valable si e très petit

ou Il existe τ tel que $f(y) - f(x) = (y-x) \cdot f'(\tau)$ avec $x < \tau < y$. Il n'est pas évident de trouver τ minimisant les erreurs. On peut prendre pour τ le milieu de xy .

Exemple : $\sin(x + e) - \sin(x) = e.\cos\tau$, avec $\tau = x + e/2$

12.1.5. Exemple de calcul complet (précision des opérandes de départ et effets d'un arrondi)

Aux erreurs des opérandes de départ se superposent les erreurs d'arrondis et de méthode de calcul, **le calcul différentiel** permet d'estimer les erreurs max possibles:

Exemple soit y = R. cos(T)

R et T sont des variables, erreurs dR et dT .

Le développement de COS fournit une erreur dC (erreur de méthode et erreurs de calculs).

Soit dP l'erreur d'arrondi du produit.

dy = R . d(COS(t)) + COS(T).dR + dP

dy = -R.(SIN(T).dT + dC) + COS(T).dR + dP

$$\frac{dy}{y} = -tg\ T.dT + \frac{dR}{R} + \frac{dP}{R.COS\ T} - \frac{dC}{COS\ T}$$

erreur relative produite par les erreurs sur les paramètres d' entrée + erreur d'arrondi sur le produit + erreur de méthode et de calcul sur COS

On peut alors passer aux incertitudes, ce qui fournira l'erreur max.

Il faudra alors tracer la courbe de l'erreur en fonction des variables, ce qui permet de trouver parfois des zones où l'erreur devient trop importante (voir infinie !) .

On utilisera un certain calcul dans certaines zones d'opérandes, pour d'autres, il faudra chercher une autre formule, ou un autre arrangement .

Après un calcul de ce type, on peut éventuellement étudier des compensations d'erreurs (en choisissant volontairement des arrondis ou des troncatures), plus délicat !

12.2. Code Binaire Virgule Fixe

Rappel : **MSB** = Most Significant Bit = Bit le plus à gauche
LSB = Least Significant Bit = Bit le plus à droite

12.2.1. Entier non signés

- n bits **non signé** $X = a_{n-1}.2^{n-1} + - - + a_1.2^1 + a_0$

dynamique n bits: non signé de **0 à $2^n - 1$**

12.2.2. Entiers signés mode complément à 2

1) Explication (en décimal) de l'intérêt de ce code, le calcul algébrique :

Pour effectuer par exemple les additions 6 + (-3) et 6+ (3), nous sommes obligés de tester, même par la pensée, le signe du second opérande et effectuer dans le premier cas une soustraction et dans le second une addition. Le processeur doit donc effectuer un test et lancer l'une ou l'autre des opérations, donc une perte de temps et la nécessité de deux opérateurs câblés distinct .

On cherche donc à trouver un code spécifique pour représenter le second opérande –3, et on ajoutera donc soit le code de –3 soit celui de 3.

Il faut se fixer un nombre N de bits : Par exemple sur 4 bits :

0110 - 0011 = 0110 +(10000 - 0011) - 10000 = 0110 + (1111 - 0011) + 1 - 10000

Complément VRAI Complément RESTREINT
Complément à 2^N Complément à 2^{N-1} , appelé
Complément à 2 "Complément à 1"

Pour calculer:	0110 (6 décimal)	On calcule:	**0110**
-	0011 (3 décimal)	+	**1100**
	?	+	**1**
		1	**0011**

Donc 3 en ignorant la retenue.

Note : On pourrait montrer ceci bien plus rapidement par : on calcule x-y au moyen de : $x + (2^n – y)$ qui est égal à $2^n + x-y$ soit x-y en ignorant la retenue donc le terme 2^n.

Le **complément restreint** s'obtient juste par **inversion des bits** .

0011 donne 1100

Le **complément vrai s'obtient ensuite en rajoutant 1.**

le nombre 1100 + 1 = 1101 peut donc représenter le nombre - 0011 = -3

Un nombre négatif se représente donc par le complément VRAI du nombre positif correspondant.

Résumé : Soit le nombre **X**, Le **code de –X est $2^N – X$**

2) Allure du code, dynamique

Exemple sur 3 bits	
0 11	+3
0 10	+2
0 01	+1
0 00	0
1 11	-1
1 10	-2
1 01	-3
1 00	-4

On peut coder sur 3 bits :
$2^3 = 8$ nombres de -4 à +3

Très important :
Les bits à droite du bit de signe ne sont pas la valeur absolue !!

Dynamique : 2^N nombres de -2^{N-1} à $2^{N-1} - 1$
Sur 4 bits, on aurait 16 nombres de -8 à + 7
Sur 8 bits, on aurait 256 nombres de - 128 à + 127.

On notera le dissymétrie (une valeur possible de plus en négatif)
Le nombre -2 s'écrit Sur 3 bits 110
Sur 4 bits 1110
Sur 8 bits 11111110

Pour coder un même nombre avec plus de bits, on effectue une "**EXTENSION DE SIGNE "**: On rajoute des 0 en tête pour un nombre positif, des 1 pour un nombre négatif.

- **Le code est pondéré** :

Sur 4 bits

1	0	0	0	=	- 8
|	|	|	|		
-8	4	2	1		

\|
bit de signe.

Si x <0 code_x = 2^n -x
-x = code_x - 2^n
décalage de -2^n par rapport à code_x, entier non signé

Le bit à gauche est le signe et à pour poids -2^{N-1}
Donc : $X = -a_{n-1}.2^{n-1} + \dots + a_1.2^1 + a_0$

On peut utiliser directement cette pondération pour convertir à la main :
Exemple 10011000 = -128 + 16 + 8 = - 104

- En **écriture hexadécimale**, on peut écrite ce nombre : 98 hexa . Attention, on ne peut pas le convertir directement de l'hexa en décimal par les puissances de 16, il faut passer par le binaire. Seul le poids fort à un poids négatif . Ce n'est pas de la numération hexadécimale mais simplement une représentation rapide du code en lecture 4 bits par 4 bits !

Une calculette en mode hexadécimal travaille le plus souvent en entier signé mode complément à 2. Un nombre est négatif si le poids fort est à 1, donc si le chiffre hexa **le plus à gauche** est supérieur ou égal à 8 de 8 à F, le plus à gauche voulant dire le niéme chiffre pour une calculette n chiffres. Donc sur une calculette 10 chiffres, FE donnera 254 tandis que FFFFFFFFFE donnera -2.

3) Avantage du code "complément à 2"

- Rapidité des calculs, le code étant pondéré, il permet le calcul algébrique pour les additions et soustractions, par contre pour les multiplications et divisions, l'algorithme de calcul est différent !

- La suite binaire signée (mode complément vrai) est facile à obtenir à partir de la suite des entiers non signés (code binaire naturel): simple inversion du bit de signe.

Exemple sur 4 bits:

1111	(15)			0111	(+7)
1110				0110	
1101		Par INVERSION	0101		
1100		du bit de signe :		0100	
1011				0011	
1010				0010	
1001				0001	
1000	(8)		----------0000----------(0)------		
0111				1111	
0110				1110	
0101				1101	
0100				1100	
0011				1011	
0010				1010	
0001				1001	(- 7)
0000	(0)			1000	(- 8)

Code binaire Naturel Code binaire signé, mode complément vrai (ou à 2
Donc pratique pour les CAN et CNA.

- ***Pour l'arrondi au plus près***

Du fait de l'allure du code qui va du plus négatif au plus positif à chaque fois en ajoutant 1, l'arrondi au plus près se fait **en additionnant 1/2LSB** avant de tronquer **directement** sur le nombre (passage à la valeur absolue inutile)

12.2.3. Code signe et valeur absolue

Le signe est mis à part :

- Par exemple en tête du nombre, la valeur absolue est alors sur N-1 bits :
  exemple - 8 **0** 000 1000 Le premier bit n'a pas de poids, c'est juste le signe.
- Ou on réserve un octet complet pour le signe (00 pour +, 11 pour -, ou les codes ASCII de ces 2 caractères)
  exemple - 8 **00000000** 0000 1000 Un octet de signe.

Utilisation : intermédiaire obligatoire pour le monde décimal, les valeurs décimales introduites sur un clavier ou affichées sont dans ce mode: signe en tête, puis valeur absolue !

12.2.4. Nombre non entier (N = m+i bits)

La virgule se déplace, elle est en fait seulement « dans la tête du programmeur !», avec produit ou division par deux à chaque cran.

Exemple en code complément à 2 :

$$X = \pm\ a_{m-1}.2^{m-1} + \ldots + a_1.2^1 + a_0 + a_{-1}.2^{-1} + a_{-2}.2^{-2} + \ldots a_{-i}.2^{-i}$$

m bits de partie entière **,** i bits de partie fractionnaire

Pas de quantification : entier pas **p =1** non entier pas $\mathbf{p = 2^{-i}}$

Exemple : 01001100 = 76 0100,1100 = 4,75 (76/16)
0,1001100 = 0,59375 (76/128)

12.2.5. Erreur ou précision

- après troncature (à i bits fractionnaires)

erreur absolue $0 < \Delta x < p = 2^{-i}$

erreur relative $\Delta x / x = \varepsilon r = \varepsilon a/x$ tend vers l'infini pour x = 0.

- arrondi au plus près erreur centrée

incertitude absolue : $\Delta x \leq p/2 = 2^{-i-1}$ donc

précision de ± 1/2 LSB

l'incertitude relative tend vers l' infini pour x tendant vers 0.

12.2.6. Notation pratique, formats classiques

Qi(n) format sur **n bits** dont **i de partie fractionnaire.**

On les considère ici **pour des nombres signés**, **code complément à 2**

	Dynamique	***Précision (au plus près)***
Entiers **Q0(8)** --------	-128 à 127	± 1/2
Entiers **Q0(16)** ----------	-32768 à 32767	± 1/2
Entiers **Q0(32)**	Environ $-2.14.10^9$ à $+2.14.10^9$	± 1/2
Q7(8) - , -------	-1 à presque 1	$\pm 2^{-8} = \pm 4.10^{-3}$
Q15(16) - , ----------	-1 à presque 1	$\pm 2^{-16} = \pm 1.5.10^{-5}$
Q8(16) -------- , --------	-128 à presque 128	$\pm 2^{-9} = \pm 2.10^{-3}$
Q16(32) ----------, ----------	**-32768** à presque **32768**	$\pm 2^{-17} = \mathbf{\pm 7.6\ 10^{-6}}$

On peut aisément écrire (ou on les possède déjà) en assembleur ou en C des sous programmes ou des fonctions permettant leur affichage par envoi sur une ligne série, ou directement sur des afficheurs cristaux liquides, leur saisie sur un clavier …

De telles fonctions seront étudiées ultérieurement.

Le format **Q16(32)** est d'après sa dynamique et sa précision est **assez polyvalent**.

12.2.7. Choix du nombre de bits, formats classiques

12.2.7.1. Partie entière : Dynamique

On veut coder tous les entiers de valeur absolue de **0 à Nmax**

Il faut $2^n \geq N\max$ (**valeur absolue** sur **n bits**).

On en tire $n \geq \dfrac{\log N_{\max}}{\log 2}$

12.2.7.2. Partie fractionnaire: Précision

Il faut $2^{-n'} \leq 10^{-d'}$ (**partie fractionnaire** sur **n'** bits pour assurer **d' chiffres décimaux fractionnaires**). On en tire $n' \geq d'.\dfrac{log10}{\log 2} = \dfrac{d'}{\log 2}$

12.2.7.3. Exemple : nombre min de bits nécessaires

On veut coder en code complément à deux de –3000 à +3000 avec 4 chiffres décimaux après la virgule.

Partie entière : valeur absolue $n \geq \frac{Log3000}{Log2} = 11{,}6$ donc 12 bits minimum.

Plus le bit de Signe (et poids -2^{n+1})

Partie fractionnaire : $n' \geq \frac{4}{\log 2} = 13{,}3$ donc 14 bits.

Au total, au minimum **13 bits** de partie **entière** et **14** de partie **fractionnaire**.
Conclusion le format classique **Q16(32)** convient

12.3. Code Binaire Virgule Flottante X = M.2^E

Il existe plusieurs variantes correspondant à plusieurs normes. Nous ne donnons ici qu'un exemple montrant le principe du codage.

M valeur absolue de la mantisse, sur **n** bits, **non signé**, et par exemple : $0{,}5 \leq |M| < 1$

E exposant sur **m** bits, par exemple en signé mode complément vrai,

Le mot complet est de n + m + 1 bits (bit de signe compris)

S	Exposant m bits signé	|mantisse| non signée n bits

- **Pas de quantification** sur la |mantisse| $p = 2^{-n}$ et sur le nombre $2^E.2^{-n}$

Précision absolue : arrondi au plus près $dx = \pm p/2 = \pm 2^E.2^{-n-1}$

Précision relative: arrondi au plus près $dx/x = dx / (M.2^E) = \pm 2^{-n-1}/M$

Cas le plus défavorable M = 0.5 D'ou $0 < |dx/x| < 2^{-n}$

Précision relative constante : $\pm 2^{-n}$ (exprimable aussi en %)

Choix du nombre de bits

On veut coder par $m.2^e$ un nombre décimal $M.10^E$.

Dynamique: il faut que $2^e \geq |M_{max}| . 10^{|E_{max}|}$ on en tire **e**.

Précision: les mantisses doivent avoir des précision comparables, d'ou la valeur de m.

Format classique : le float en C

4 octets : n=23, m=7

Dynamique $0{,}5.2^{-64}$ à presque 1.2^{63} soit environ 10^{-19} à $9{,}2.10^{18}$

Précision relative $\pm 2^{-24}$ ou **$5{,}9.10^{-8}$** arrondi au plus près, *constante,* et donc 7 chiffres environ significatifs décimaux.

12.4. Code DCB virgule fixe, Code Hexadécimal et Code ASCII

1) **Code Décimal Codé Binaire** (Chaque chiffre décimal codé en binaire).

Indispensable pour les Entrées_sortie dans le système décimal plus convivial pour l'être humain ! (petits claviers numériques, afficheurs)

Les chiffres (de 0 à 9) sont codés (en machine) en binaire. Le signe est à part.

On peut avoir un tableau d'octets pour la partie entière, et un autre pour la partie fractionnaire (ou utiliser un code de la virgule).

- **Codage 1 chiffre = 1 octet**

Exemple +124,96 peut donner :

1 octet de signe	$00 ($FF si il était négatif, ou autres codes !)	
3 octets de partie entière	$01 $02 $04	ou ASCII : $31 $32 $33
2 octets de partie fractionnaire	$09 $06	ou ASCII : $39 $36

Obtention du code ASCII :

Pour travailler avec les caractères de '0' à '9' et nom les chiffres de 0 à 9 (pour clavier ASCII, lignes de transmission, série ..).

On passe très facilement du chiffre de 0 à 9 au caractère correspondant en ajoutant **$30** au code binaire, et dans l'autre sens en remettant à zéro le quartet haut.

- **Codage compact 2 chiffre = 1 octets**

On pourrait avoir : 1 quartet de signe (0 ou F ou autre code)
2 octets partie entière $01 $24
1 octet de partie fractionnaire : $96

Peu utile dans des applications microprocesseurs en fait, sauf pour stocker un grand nombre de valeurs décimales. S'utilise en Cobol dans le code compact nommé COMP-3.

Pratique par contre dans les systèmes logiques câblés ou programmés (compteurs DCB). (On peut câbler directement dessus des afficheurs)

2) **Code Hexadécimal** :

Groupement 4 bits par 4 bits de n'importe quel octet, et conversion pour afficahge en caractères ASCII de 0 à F. Pratique parfois pour afficher du binaire quelconqe.

Obtention du code ASCII :

Pour toutes les valeurs de 0 à F on ajoute **$30** (ca marche comme en DCB de 0 à 9).

Pour les valeurs de A à F, on ajoute **encore $7**

Exemple : pour 010 = A on arrive à A + $30 + $7 = $41 et c'estr correct puisque les codes ASCII des lettres majuscule de A à Z commece à $41.

12.5. Choix des modes de représentation

	Avantages	*Inconvénient*
VIRGULE FIXE	calculs + rapides conversions simples (binaires, DCB ..)	Débordements délicats à gérer, dynamique faible. Format pouvant changer à chaque étape.
VIRGULE FLOTTANTE	Grande dynamique et précision Dépassements rares Précision relative constante sur un opérande **(pas toujours sur le résultat d' un calcul)**	Calculs plus complexes et plus lents (sauf si processeur flottant) Conversions plus délicates

Existence de **bibliothèques flottantes en C** pour de nombreux microprocesseurs !

Sur un microcontrôleur standard (sans unité arithmétique flottante), très pratique si on n'a pas de contraintes ni de vitesses, ni de taille de code.

	Avantages	*Inconvénient*
Binaire	Proche de la machine Calculs rapides	Nécessitent conversions Binaire↔DCB
DCB	Chiffres décimaux directement visibles. Nécessaire pour les entrées/sortie vers un utilisateur humain !	Calculs lents

13. GENERALITES SUR L'ARITHMETIQUE BINAIRE VIRGULE FIXE

Dans ce chapitre, tous les **nombres signés** seront en **mode complément à 2** (complément vrai), les nombres sur 8, 16 bits ou plus.

13.1. Introduction

Format général **Qi(n):** ---------- , ----------

La ***dynamique*** est fonction du nombre de bits de la partie entière, soit n-i

La ***précision*** (pour des nombres fractionnaires), dépend du nombre de bits de la partie fractionnaire, soit i.

D'autre part après chaque addition et surtout chaque multiplication, le nombre de bits théorique augmente sans cesse, une *limitation à un format maximum est obligatoire*.

Deux dangers:

- **Le débordement**: le pire ! . Il provoque l'erreur maximale !.

exemple sur 8 bits signés: 0111 1111 + 1 = 1000 0000

127+1 = -128 !!

A éviter absolument !!!

Si pour quelques opérandes seulement on risque un débordement, il faut programmer un écrétage à la valeur max (positive ou négative).

-La perte progressive de précision: la dégradation est progressive avec éventuellement passage à zéro d'un opérande ("underflow"):

exemple sur 8 bits, multiplications successives par 1/4:

0010 0011 *1/4 = 0,000 1000

0,000 1000 *1/4 = 0,000 0010

0,000 0010 *1/4 = 0,000 0000 = 0 !!!

On ne peut pas toujours la supprimer, mais il faut la limiter par un recadrage éventuel du résultat (un peu comme en virgule flottante finalement ou cela est automatique !!).

Dynamique théorique: (Dynamique = étendue des nombres représentables)

Pour un calcul ou une opération donnée, c'est la dynamique du résultat, pour toutes valeurs d'opérandes théoriquement possibles:

Exemple en entier non signé, 8 bits:

les opérandes X et Y vont de 0 à 255, donc S = X+Y va de 0 à 510

P = X.Y va de 0 à 65025

Il faut donc 9 bits pour coder S, et 16 pour coder P.

Dynamique pratique:

Les nombres sur 8 bits précédents peuvent être des valeurs toujours entre 0 et 99.

Alors S = X+Y va de 0 à 198 et P = X.Y va de 0 à 9801

Il suffit donc pratiquement de 8 bits pour S et de 14 pour P.

Il est donc inutile de prévoir à ce niveau un bit de plus pour S, et pour P sur 16 bits, on a 2 bits de **réserve de dynamique**, ce qui peut être pratique pour les calculs ultérieurs!.

Conclusions :

→Pour chaque problème, pour diminuer les erreurs, en virgule fixe, il faut:
-Sauf dans quelques cas particuliers, interdire tout dépassement accidentel !
-évidemment tronquer ou arrondir le moins souvent possible.
-éviter les pertes de précision trop importantes!!

→Le passage à 1 bit de plus ou un octet de plus n'est pas du tout évident après calcul, il vaut mieux **travailler tout de suite sur des opérandes étendus sur plus de bits que nécessaires initialement** (exemple : calculs sur 16 bits pour des opérandes sur 8 bits).

→Attention !

En **langage C seul** :

Pas d'**accès aux bits du code condition** du processeur (Retenue C et Overflow V).

La **taille d'un résultat** est toujours **celle de ses opérandes** (exemple produit 8bits par 8 bits donne toujours 8 bits), donc **cette méthode est obligatoire**.

Donc si l'on veut détecter des ***débordements*** par des indicateurs C et V, et agir en conséquence, il faudra programmer les ***fonctions C en assembleur***.

→Si on doit réduire la taille d'un résultat après une opération, deux solutions :

Solution 1: **Utiliser juste le nombre de bits de poids forts nécessaires**, on gardera peut-être des bits de poids faibles non significatifs (si on part d'opérandes de 8 bits, un poids 2^{-15} par exemple n'est pas significatif) . Ceci n'est pas gênant si une dynamique plus élevée n'est pas nécessaire dans des calculs ultérieurs .

Solution 2: **Garder des poids forts en réserve pour des calculs futurs, en prenant un compromis**: si l'incertitude sur un opérande est telle que la précision du résultat ne peut pas être meilleure que 2^{-7} , on peut supprimer les poids 2^{-12}, 2^{-13} , 2^{-14} ,2^{-15} par exemple.

→Sauf cas très particulier, si on redoute un débordement pour quelques valeurs des opérandes, et que celui ci n'est pas trop important ni trop fréquent, on peut envisager deux solutions:

Détection du débordement et message à l'utilisateur.

Ecrêtage du résultat à sa valeur maximale (avec message éventuel). Sauf cas très particulier, un écrêtage perturbe infiniment moins qu'un débordement incontrôlé, qui entraîne lui immédiatement une erreur sur les poids forts du résultat !!!

→**Pour simplifier,** on raisonnera sur **les formats standards:**

-sur des **entiers** Q0(n)
-sur des **nombres de module < 1** Qn-1(n)
-sur des nombres en Q16(32)

→Des **études similaires** sont très aisées à effectuer pour **d'autres formats.**

13.2. Les indicateurs C, N, Z et V en assembleur

Ils ne sont rappelons le pas accessible en C seul !

Remarque : Utiliser surtout les tests classiques de branchement conditionnels tels que : BGE,BGT, BHI, BLS, BPL, BMI, BEQ …en signé (testant N, Z et V) ou en non signé (testant C et Z) car ces instructions testent souvent une fonction logique de plusieurs indicateurs.

13.2.1. Opérations élémentaires

13.2.1.1. addition et soustraction non signé

Cas général : Il faudrait un bit de poids fort supplémentaire.
Exemple : Q0(8) + Q0(8) = Q0(9)

Le bit **C** ou **CARRY** du code condition d'un processeur est le bit de **dépassement.**
Exemples :

	*0*11111111	255		*0*00000010	2
	+ *0*00000001	+1		- *0*00000011	-3
1	00000000	0 au lieu de 256 !	1	11111111	255 au lieu de -1
C	N		**C**		

N n'est que le poids fort du résultat. Donc **sans autre signification** ici.
Z est significatif (résultat 0) seulement si on ne déborde pas !

13.2.1.2. Additions et soustractions signé

Cas général : Il faudrait aussi un bit de poids fort supplémentaire.
Exemple : Q0(8) + Q0(8) = Q0(9)

V (overflow) du code condition d'un processeur, **indique** le dépassement en signé.
Exemples :

	01111111	127		10000000	-128
	00000001	+1		11111111	-1
0	10000000	-128 au lieu de +128	1	01111111	127 au lieu -129
C			**C**		

C ne signifie rien !
N indique résultat négatif si pas de dépassement seulement (V = 0) !
Z est significatif également seulement si V = 0

13.2.1.3. Décalages arithmétiques

- **Gauche : Multiplication par 2**
  1) non signé simple décalage gauche (avec introduction de 0).
  2) signé idem
  Les instructions **ASL** (décalage arithmétique) et **LSL** (décalage logique) ont le même effet.
- **Droite : Division par 2**
  1) non signé décalage droite avec entrée de zéro sur le MSB
  ex 10000000 donne 01000000 (128 donne 64). Décalage logique **LSR**

  2) signé décalage droite, avec recopie MSB sur le MSB.
  ex 10000000 donne 11000000 (-128 donne -64). Décalage arithmétique **ASR**

13.2.2. Opérandes de taille multiple de la taille de base

Résonnons sur une taille de base égale à l'octet.

- **Additions, soustractions, décalages , en signé ou non signé**
  Faire l'opération binaire avec tout d'abord le passage simple de la retenue C d'un octet à l'autre (ou d'une taille de base à la suivante).
  On ne doit tester N, C ou V que la **dernier octet.**

Z ne signifie **rien**, il faudrait tester en fait le OU des différents Z à chaque étape du calcul.

Les instructions de test sont donc à utiliser avec réflexion ! exemple BHI qui teste C+Z ne marche pas !

- **Multiplication et autres**

Attention : l'algorithme, et donc l'instruction de multiplication n'est pas le même en signé mode complément à deux.

En HC11 MUL ne travaille qu'en non signé !

Il vaut donc souvent mieux travailler sur les **valeurs absolues**, en binaire naturel (usage de C seulement pour le report des retenues) et ne remettre le signe qu'en fin d'opération !

Exemple pour un microprocesseur 8 bits, sans un langage évolué C qui permettrait de traiter des produits jusqu'à 32 bits par 32 bits, pour calculer le produit de 16 bits par 16 bits donne 32 bits, on décomposerait comme suit.

X = 256.Xh + Xl

Y = 256.Yh + Yl

X.Y = 65536.Xh.Xl + 256(Xh.Yl + Xl.Yh) + Xl.Yl

13.3. Point important : calcul d'une somme, le résultat final pratique ne débordant pas.

Exemple, calcul de $S = \sum_{k=0}^{N-1} x_k$. ou $S = \sum_{k=0}^{N-1} x_k y_k$ ou $S = \sum_{k=0}^{N-1} x_k^2$

Si les opérandes Xk sont sur Qi(j), et que la dynamique pratique est telle que la somme finale tient aussi en Qi(j), grâce au modulo 2^n du code binaire signé, les résultats intermédiaires peuvent déborder sans problème !

Cette propriété est très utile dans certains cas.

De même si la somme finale demande par exemple en pratique 1 bit en plus, on peut calculer la somme avec seulement ce bit en plus, et laisser déborder dans les résultats intermédiaires. Le résultat final sera correct !

Même propriété donc pour calculer des sommes de produits, des sommes de carrés …

→ Il ne faut surtout pas dans ce cas programmer un écrêtage à la valeur max, qui fausserait alors le calcul !

14. LE LANGAGE EVOLUE (C) ET L'ARITHMETIQUE

→ Si le processeur contient une « unité arithmétique » en virgule flottante, il n'y a pratiquement pas de problème de vitesse ni de taille de code, on travaille donc avec des entiers ou des nombres en virgule flottante selon les besoins.

→ Si il ne travaille que sur des entiers (cas le plus fréquent sur les microprocesseurs et microcontroleurs), on doit :

Travailler sur des **entier**s, ou du moins **en virgule fixe** en rusant, en faisant croire au C que ce sont des entiers, la virgule n'est en fait que dans la tête du programmeur … !

Si pas de contraintes de vitesse ni de taille de code, on peut tout de même heureusement se servir de **bibliothèques virgules flottantes** sans aucun problème.

14.1. Calculs en virgule fixe

- ***Très important :*** le C fournit le **résultat** d'une opération quelconque dans la **même taille** (8, 16 ou 32 bits) **que** celle des **opérandes de départ**. (par exemple un produit 16bits par 16 bits donne 16 bits toujours !)
- En **C seul** (sans fonction C en assembleur), on devra donc très souvent **forcer une opération** (par un cast**) à s'effectuer dans un format plus grand**.

14.1.1. Introduction de valeurs en Qi(n) dans du C

Si on travaille sur des opérandes 16 bits, même pour des nombres fractionnaires, on déclarera les opérandes par **int**. Le C ne connaissant en virgule fixe que des entiers, on lui fournit le même code en raisonnant entier, en décalant la virgule complètement à droite :

Explication pour une valeur en Q15(16) :

Valeur en Q15(16) - , - - - - - - - - - - - - - - -

Le même code sans la virgule : - - - - - - - - - - - - - - - -

La virgule est seulement « dans la tête du programmeur ». Le C travaillera sur ce dernier nombre, ici 32768 fois plus grand. Le produit est effectué évidemment en flottant une fois pour toute par le compilateur, puis le cast en int.

Exemples :

Coefficient	***Déclaration C***
k= + 0,456 en **Q15(16)** - , - - - - - …. -	int k = 0.456***32768** ;
k = -1,456 en **Q14(16)** - - , - - - - … .-	int k = - 1.456***16384** ;
x = -345,78 en **Q16(32)**	long x = -345.78***65536**;

14.1.2. Additions et soustractions

Pour gérer les débordements, on doit raisonner selon la **dynamique pratique**, en sachant ou se trouve en réalité la virgule en permanence, et travailler sur une **taille suffisante**.

Attention : Pas de message d'erreurs en cas de débordement !!!!.
Pas d'accès aux bits C et V !

Pour détecter un débordement éventuel, il faut travailler au départ sur une taille plus grande, ou sinon programmer évidemment en assembleur la fonction C !

14.1.2.1. Différents cas possibles

On déclare par exemple int x,y,**s** ; en **Q15(16) -,------- --------**
long **ls** ;

Dans le **cas général**, il faut un bit de plus, il faut donc **passer à 32 bits avant le calcul** :

- ***Obtention de la somme sur 32 bits***

On calcule **ls = (long)x + (long)y**

On obtient alors **ls en Q15(32)** **-------- -------- -,------- --------**

On peut si besoin sur ce long tester si la dynamique dépasse ou non la dynamique16 bits. C'est la seule façon (sans utiliser l'assembleur) pour tester si on déborde ou non.

- ***Pour se ramener à 16 bits dans le cas général :***

Un bit de plus est suffisant, on décale à droite, on perd un bit de précision.

On fait **s = ((long)x + (long)y) >>1**;

Résultat : **s en Q14(16)** **--,------ --------**

(plus précis que s = (a>>1) + (b>>1) ;)

- ***Cas particulier, si on est sûr d'après la dynamique pratique de ne pas déborder,***

On peut faire le calcul sur 16 bits évidemment : **s = x + y ;**

C'est le cas aussi d'une somme de N termes, le résultat final ne débordant pas.

14.1.2.2. Exemple de fonction C d' addition avec test de débordement

Cette fonction C a déjà été écrite en assembleur dans un précèdent chapitre, et le temps d'exécution était alors évidemment nettement inférieur. Nous l'écrivons ici tout en C.

Prototype : **char somme(int *tab, unsigned int taille, int *resultat) ;**

Cahier des charges : Opérandes et résultats 16 bits signés
Détection d'un éventuel dépassement en cours de calcul.
Retour: **1** si OK, -1 si dépassement

```
charsomme(int *tab, unsigned int taille, int *resultat)
{
long S = 0 ;
char flag = 1 ;       /* initialisé à 1 à chaque départ de calcul */
char k ;
for(k = 0 ; k < taille ; k++)
        {   S = S + (long)tab[k] ;
                    if(  (S < -32768) || (S > 32767) )   flag = -1 ;
    }
*resultat = S ;
return(flag)
}
```

Remarque : ce drapeau mémorise un débordement éventuel en cours du calcul. Le résultat final n'est pas faux, nous l'avons déjà vu, si la dynamique pratique de celui ci est assuré….(ici 16 bits signé). On peut par contre dans ce cas programmer ainsi un Warning.

14.1.3. Multiplications

14.1.3.1. Produits d'entiers signés X et Y

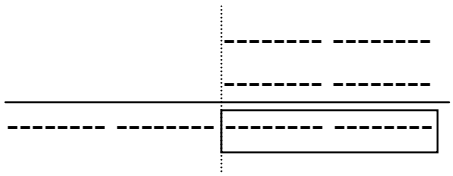
Mathématiquement : **N bits par N bits donnent 2N** bits

Mais Attention: En C, le produit **P = X * Y** avec X, Y entiers signés **8, 16 ou 32 bits**, donne un P entier signé sur **même taille 8, 16 ou 32 bits,** *sans message d'erreur* en cas de dépassement. *Une étude de la dynamique pratique est indispensable!*.

Exemple d'opérandes sur 16 bits : int x,x,p ;
Valeur sur 32 bits : long lp ;

- ***Si la dynamique pratique le permet :***

p= x*y ;
résultat 16 bits, partie basse suffisante !

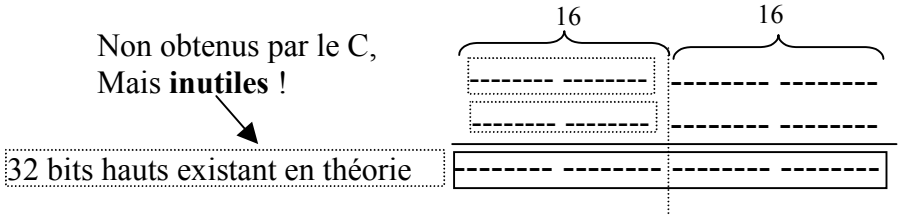


- ***Passage à 32 bits:***

Si **lp** est un entier long sur 32 bits, l'écriture lp = x*y ("cast" implicite) donne en fait lp = (long)(x*y) il y a donc juste passage sur 32 bits d'un résultat limité à 16 bits, et donc cela de sert à rien, dépassement dans de nombreux cas !

Il donc forcer un produit 32bits par 32bits résultat 32 bits et écrire :

lp = (long)x*(long)y;



Le temps de calcul est évidemment bien plus long!

14.1.3.2. Produits de nombres signées de module < 1

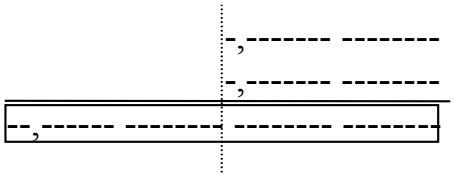
On doit déclarer les opérandes comme des entiers. Soit par exemple:

int x,y; (16 bits signés) en fait x, y fractionnaires, **Q15(16)** !
long lp P en 32 bits signés
int p P ramené à 16 bits.

- ***Produit exact sur 32 bits:***

P en Q30(32)

lp = (long)x * (long)y;



Et non lp = (long)(x*y); !! qui ne prendrait que la partie basse et l'étendrait sur 32 bits!!. L'écriture en "cast" implicite lp = x*y, donne le même résultat, erroné !

Dynamique pratique –1 à +1
Dynamique théorique possible –2 à presque +2 donc un bit de réserve de dynamique

Ce bit est pratique pour calculer :

Une somme de deux produits S = ax + by

Une somme de produits $S = \sum x_k y_k$ si en pratique le résultat final tient dans la dynamique.

Attention cependant : s'interdire tout de même tout à –1, car dépassement, pour la seule valeur +2 !

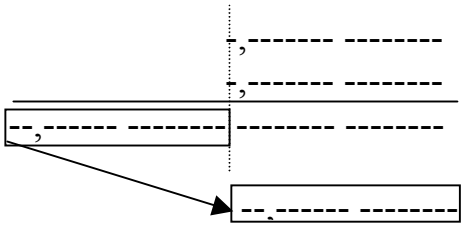
➢ ***Réduction finale à 16 bits :***

On part d'opérandes avec 15 bits fractionnaires, il est donc inutile de conserver les 30 bits de précision obtenus !

On pourra ainsi sans aucun inconvénient fournir un produit seulement sur 16 bis !

Partie haute :

p = ((long)x * (long)y) >> 16 ;



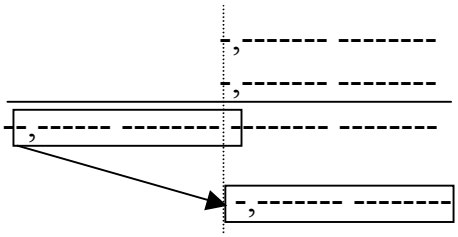
On obtient :

P en Q14(16) ou P/2 en Q15(16)

Donc **1 bit** de « **réserve de dynamique** » pour un calcul futur.

Produit dans le même format Q15(16) :

p = ((long)x * (long)y) >> 15 ;



P en Q15(16)

Ou si l'on veut faire l'arrondi au plus près :

p = ((long)x * (long)y + 0x8000) >> 15;

14.1.3.3. Produit d'un entier par un nombre de module < 1

On déclare: int x,y; (16 bits signés) en fait **x entier**, **y** fractionnaire **Q15(16)**.

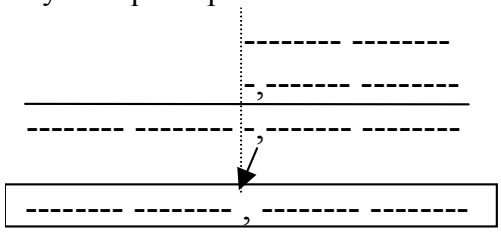
long lp; (32 bits signés)

int p ; (16 bits)

➢ ***Exemple assez général*** :

Obtention d'un produit en Q16(32), bien pour dynamique et précision.

p = ((long)x * (long)y) << 1 ;



P en Q16(32)

14.1.3.4. Produits de nombres de format quelconques

Faire toujours un petit dessin pour voir ou se trouve la virgule après le produit, ce que l'on veut obtenir et trouver le nombre de décalages à effectuer, c'est tout simple !

14.1.4. Exemple de calcul de y = ax+b

Etudions un cas pratique, correspondant nous le verrons plus loin à la mesure d'une grandeur physique (calcul de celle ci à partir d'un code binaire).

x entier sur 16 bits, mais de dynamique pratique pas forcement maximale (exemple code 12 bits en provenance d'un CAN, étendu à 16 bits).

a et b des coefficients (pouvant éventuellement un peu varier, mais restant dans une dynamique Qi(16).

Exemple : y = -2,46.x + 60

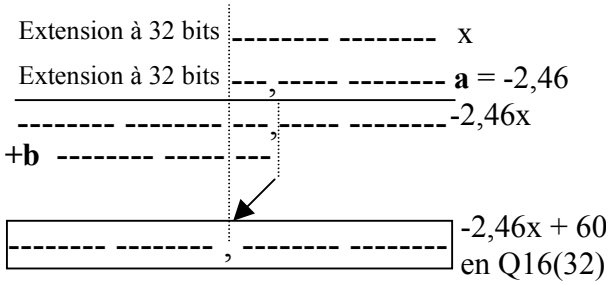
Avec x entier signé Q0(16), mais dynamique pratique 12 bits

-2,46 en Q13(16) (Dynamique possible –4 à presque +3)

60 entier sur 16 bits

Résultat y en Q16(32) pour bonne précision, dynamique pratique souvent amplement suffisante.

Soit : Int x ;
Int a = -2.46*8192 ;
Int b = -60 ;
Long y ;
y =((long)x * (long)a +
((long)b <<13)) <<3 ;
(b<<13 faux car on resterait en 16 bits !)
y en Q16(32)



Remarque : cette écriture en C peut générer donc un produit effectivement de 32 bits par 32 bits, ce qui n'est pas du tout utile, un produit 16 bits par 16 bits → 32 bits est suffisant ! D'ou optimisation possible en programmant en assembleur si besoin est.

Performances : environ **171 Tck** (et ***139 octets*** de code) sur HC11

108 Tck (et ***87 octets*** de code) seulement sur HC12, car ce processeur possède une instruction assembleur de produit 16bits*16bits → 32 bits. L'HC11 a besoin d'un sous programme pour la même opération.

On pourra comparer ces performances à l'utilisation des virgules flottantes (voir plus loin). On peut gagner beaucoup en virgule fixe (et en programmant en assembleur si besoin est).

14.1.5. Division

14.1.5.1. Généralités

- Pour des valeurs déclarées int ou long, le résultat est dans le même type (int ou long).
- X/Y donne donc 0 si X < Y !
- Cas général : $\frac{Qi(n)}{Qi(n)} \longrightarrow Q_0(n) \quad entier$

Il faudra ruser pour obtenir des bits fractionnaires.

Par contre X/Y peut fournir un message d'erreur pour Y = 0.

14.1.5.2. Astuce pour avoir des bits fractionnaires

1) Soit à effectuer le calcul de X/Y avec opérandes 16 bits

X et Y signés sur 16 bits, entiers **Q0(16)** ou de **même format Qi(16)**

Résultat en **Q16(32)** **-------- -------- , -------- --------**

➢ Rappel de la théorie:

On veut calculer $\frac{X}{Y} = Q + \frac{R}{Y}$ Le C ne donne que Q entier

On calcule donc plutôt : $\frac{2^{16}X}{Y} = 2^{16}(Q + \frac{R}{Y})$ On raisonne sur **un numérateur 2^{16} fois plus grand**, on obtient un quotient entier que l'on peut alors **interpréter en Q16(32)** pour connaître le vrai résultat (la division par 2^{16} se fait par la pensée, le C ne voit que des entiers).

➢ Exemple : int X = 4 ; int Y = 3 ; long Q1632 ;

Q1632 = ((long)X << 16)/ (long)Y ;

On a ainsi calculé 4/3 et résultat en Q16(32), pour obtenir 1,3333

Performances : sur compilateur IAR Systems, HC11 2MHz, optimisé en vitesse :

540 Tck = 270 µs = 0,27ms

2) Calcul général, opérandes de 32 bits

X en **Qi(32)** divisé par Y en **Qi(32)** donne **Q entier sur 32 bits**.

Pour avoir des bits fractionnaires, pas facile, il faudrait avec la technique précédente une division 64 bits par 32 bits !

Utiliser plutôt les virgules flottantes !

3) Cas fréquent : calcul précis d'une moyenne de nombres entiers sur 16 bits

Soit : la **somme S** sur **32 bits,** le **nombre de valeurs N** sur **16 bits**.

On désire la **moyenne** en **Q16(32).**

S'agissant d'une moyenne de nombres sur 16 bits, le résultat de la division tient forcément sur 16 bits seulement.

On doit effectuer le calcul en deux temps, et **sur les valeurs absolues** !

```
Unsigned long PE,PF;          /* partie entière et partie fractionnaire */
Unsigned long R ;
Char signe = 0 ;
long S=0 ;                    /* pour contenir la somme */
unsigned int N ;              /* le nombre de valeurs */
long MOY_1632 ;               /* pour la moyenne en Q1632 */
/*  ici on suppose que l'on calcule la somme des valeurs en long */
/* on calcule maintenant la moyenne :*/
if(S<0 ){ S = -S ; signe = 1 ;}      /* prise de la valeur absolue */
PE = S/N ;                    /* partie entière du quotient */
R = S%N ;                     /* reste R < N */
PF = (R <<16)/N ;   /*  entier 2^16 R / N  ou R/N en Q0(16)  ,-------- -------- */
MOY_1632 = (PE << 16) + PF ;  /* valeur absolue de la moyenne en Q1632 */
if(signe = = 1) MOY_1632  = - MOY_1632 ;      /* remise du signe */
```

→ On pourrait faire une fonction de prototype **long division(long N, long D) ;**

14.2. Calculs en virgule flottante

Permet de manipuler des nombres algébriques, de valeur absolue pouvant être très élevée, avec toujours le même nombre de bits significatifs, sans pratiquement aucun problème de débordement .

Pour des microcontroleurs classiques, sans opérateurs flottant câblés :

- Possible dans une boucle si pas de problème de temps d'exécution.
- Sinon à n'utiliser que lors des initialisations de certaines variables de calcul après des introductions au clavier ou pour calculer des valeurs à afficher.
- Attention à la taille du code généré, plus importante que pour des calculs en virgule fixe.

Un **langage évolué tel que le C** possède les **bibliothèques de calcul en virgule flottante**, et s'occupe évidemment des opérations qu'il faudrait faire si on travaillait en assembleur (travail sur les mantisses et les exposants, recadrage pour normalisation …).

Tous les calculs de **fonctions mathématiques**: sinus, cosinus, racine carré …. sont possibles en C. Les routines lancées sont bien évidemment assez longues d'exécution sur les petits microcontrôleurs ne travaillant à la base qu'en fixe. Cette lenteur est relative ! (quelques millisecondes par exemple) et n'est nullement gênante si on n'a pas de contraintes de temps.

Un problème subsiste toutefois, celui des conversions en virgule fixe dans les deux sens…

14.2.1. Exemple de calcul en virgule flottante

14.2.1.1. Calcul d'un division, conversion du résultat en Q16(32) pour affichage aisé.

Soit le même exemple que précédemment en virgule fixe : calcul en virgule flottante de **4/3** et résultat en Q16(32), pour obtenir 1,3333

Performances : elles sont données sur compilateur IAR Systems, HC11 2MHz, optimisé en vitesse.

- ***Opérandes déjà en flottant :***

```
float  x = 4 ;    /* on peut écrire 4.0 aussi */
float  y = 3 ;
float    fQ ;  /* quotient flottant */
long Q1632 ;   /* quotient converti en Q1632 */
fQ = fx/fy ;                          3000 Tck   donc   1,5ms ! !
Q1632 = 65536.0*fQ1;                  +1500 Tck   donc   +0,75ms ! !
```

(voir plus loin l'explication de la conversion flottant → Q16(32))

Total 4500 Tck = 2,25 ms

- ***Opérandes initiaux en virgule fixe***

On les transforme en flottant pour le calcul :

```
int  X = 4 ;
int  Y = 3 ;
float  fQ ;
long     Q1632 ;
fQ1 = (float)X/(float)Y ;      5800 Tck   donc   2,9ms ! !
```

On voit que si les opérandes sont déjà en flottant la division est évidemment plus rapide.

De même : `Q1632 = 65536.0*fQ1;` **+1500 Tck** donc **+0,75ms** ! !

Total 7300 Tck = 3,65 ms

→ Ces temps sont à comparer au calcul de 4/3 tout en virgule fixe, résultat en Q16(32) vu précédemment, il fallait plus de 10 fois moins de temps : 540Tck soit de 0,27ms.

14.2.1.2. Calcul de y = ax + b

Soit le même calcul que précédemment en virgule fixe :

Y = -2.46x + 60

Résultat en Q16(32) pour affichage.

Soit a et b les deux coefficients.

Pratiquement pas de problème de dynamique sur y en flottant, juste évidemment au niveau de la conversion finale en Q16(32).

```
Float a = -2.46 ;
Float b = 60 ;
Float fy ;
Long    y1632 ;
```

Calcul évident !

Performances sur HC11 ou HC12 1289 Tck + 560 Tck soit **1849 Tck**.

Et ***taille code 1120 octets***

(Comparez avec le calcul précédent en virgule fixe …. !)

```
fy = a*x + b ;
y1632 = 65536.0*fy ;
```

(voir ci-dessous le principe de cette conversion)

14.2.2. Conversions Virgule fixe ↔ Virgule flottante

➢ **Flottant vers fixe, exemple conversion en Q16(32) :**

En Q16(32), on raisonne sur un nombre entier qui est 65536 fois plus grand. Il faut donc multiplier en flottant par 65536. Attention ce n'est pas 16 décalages, qui en flottant serait stupide, et donnerait n'importe quoi

Donc, pour convertir en Q16(32) il faut simplement faire :

Soit par exemple: **float x = -3,4627 ;**

Soit : **long x_1632 ;**

Alors: **x_1632 = 65536.0*x ;**

(Le cast en long est implicite, et il n'est pas utile de l'écrire)

➢ **Fixe vers flottant**

Soit une valeur par exemple en Q15(16) **-,------- --------**

On veut la convertir en flottant.

Soit par exemple : **int val_1516 = 0,467*32768 ;**

Soit : **float x** ;

Alors : **x = (float)val_1516/32768 ; (ou val_1632/32768.0)**

Attention ! l'écriture x = (float)(val_1516/32768) est fausse et donnerait 0, il faut forcer la division à s'effectuer en flottant et non en entier !

14.3. Conversions Binaire (Virgule fixe) ↔ DCB Notions de « Driver »

Nous étudions ici les algorithmes de conversion qu'on utilise obligatoirement dès que l'on effectue des entrées sorties de **valeurs décimales** (gestion de clavier numérique, d'afficheurs ..). Nous reverrons ces fonctions (sous d'autres formes éventuellement) dans le chapitre sur la gestion de clavier numérique et d'affichage sur panneau cristaux liquides.

On travaille toujours sur les **valeur absolue** et **on traite le signe à part**.

14.3.1. Virgule fixe DCB → BINAIRE Virgule fixe

Soit le nombre DCB : **dcba,ij** = 1000d + 100c +10b+ a+ 0,1i + 0,01j

➢ **Partie entière :**

il faut calculer 1000d + 100c +10d+a

a) On suppose que l'on dispose des chiffres a, b, c …

Un algorithme est pratique: on multiplie par 10 le résultat précédent (en commençant par zéro) et en ajoute le nouveau chiffre : **dcba** s'obtient par

[{ (0*10+**d**)*10+**c** }*10 + **b**]*10 + **a**

Cet algorithme est aussi pratique, en effet on peut si besoin est effectuer au fur et à mesure la conversion sans attendre d'avoir tout le nombre (cas de nombres introduits au clavier).

b) Exemple de fonction C

Conversion de la partie entière :

Paramètres d'entrée :

Tab_entier	Le tableau qui contient les chiffres DCB de la partie entière
n	Taille du tableau

Retour : L'entier binaire obtenu après conversion

Dynamique : Résultat max possible, un long non signé, donc on a de la marge !

```
unsigned long conversion_entiere(char n, char *tab_entiers)
{
char k;
unsigned long x=0;      /* calcul en 32 bits pour détecter si dépassement 16 bits */
for(k=0 ; k<n ; k++) x=10*x+tab_entiers[k];
return(x);
}
```

Programme main() de test :

```
Main()
{
char tab[3] = {2,7,9} ;
unsigned long PE ;
PE = conversion_entiere(3, tab) ;
}
```

➢ partie fractionnaire :

a)Méthode

-On peut prendre chaque chiffre et les multiplier par 0,1 0,01 etc.. ., délicat et des erreurs se cumulent.

-Il est plus simple de traiter la partie décimale comme une partie entière, on obtient x entier. Si le nombre de chiffres de la partie fractionnaire est Nf, on divise ensuite par 10^{Nf}, ce 10^{Nf} est calculé (produits successifs par 10) au fur et à mesure que l'on convertit l'entier pour avoir x. On effectue non pas la division $x/10^{Nf}$ qui donnerait évidemment 0, mais $((long)x<<16)/10^{Nf}$ c'est l'astuce déjà vue qui permet d'avoir les bits utiles désirés. Un calcul en flottant bien que faisable prendrait trop de place bêtement en mémoire, ici on reste ainsi tout en virgule fixe !

b) Exemple de fonction en C:

La division avec le numérateur décalé à gauche de 16 donne le quotient 16 bits fractionnaire dans un long, donc sous la forme 0000000000000000, - - - - - - - - - - - - - - - -

Le débordement est simple à gérer. Avec 16 bits fractionnaires, le pas de quantification est de $2^{-16} = 1/65536 = 1{,}5.10^{-5}$ donc 4 chiffres décimaux après la virgule suffisent. Il suffit donc de ne pas prendre en compte des chiffres supplémentaires.

Conversion de la partie fractionnaire :

Paramètres d'entrée :

Tab_decimaux	le tableau qui contient les chiffres DCB de la partie fractionnaire
m	taille du tableau

```
Retour :      La partie fractionnaire en binaire format virgule en tête   , - - - - - …..
Dynamique : max  4 chiffres décimaux (m ≤ 3)
unsigned long conversion_decimale(char m, char *tab_decimaux)
{
char k;
unsigned long x=0;
unsigned int div=1;
for(k=0 ; k<m ; k++) x=10*x+tab_decimaux[k];
x = x <<16;                              /*   x=65536.x   */
for(k=0 ; k<m ; k++)div=div*10;    /* création du terme en 10 puissance  */
return ( x/div );
}
```

Programme main() de test, on doit obtenir 455*65536 = 28818_d = 7092_{hexa}

```
Main()      {        char tab[3] = {4,5,5} ; unsigned long PF;
   PF = conversion_decimale(3, tab) ;
        }
```

➢ Mélange des deux parties pour obtenir du Q16(32) :

Une fois récupérées la partie entière PE et la partie fractionnaire PF sur 16 bits en non signé, ou comme plus haut sur 32 bits mais calées à droite, il suffit de faire :

```
long  q1632 ;   /*  signé 32 bits */
Q1632 =  PE << 16 +  PF ;
```

On traite le signe ensuite (on effectue ou non le complément à 2).

14.3.2. Virgule fixe Binaire → DCB et Affichage, notion de Driver et de Couches de programmation

➢ Partie entière

On effectue des divisions successives par la base 10, le **reste** fournit chaque **chiffre**, on trouve les unités en premier. C'est la méthode la plus aisée à programmer.

➢ partie fractionnaire

a) On peut multiplier successivement par 10 et garder à chaque fois les unités (la partie entière), on obtient ainsi chaque chiffre en commençant par les 10ème .

b) On peut obtenir aussi directement en binaire le nombre de 100ème ou de 1000ème ou de 10000ème ... en multipliant en une seule fois par 100 ,1000 ou 10000 . Le programme précédent de conversion entier binaire → DCB fournira alors ce résultat en DCB.

➢ Passage à des caractères ASCII

Il suffit d'ajouter **$30** à chaque octet chiffre pour passer à l'octet caractère ASCII (de 0 à 9)

➢ Exemple de conversion Binaire → DCB et d'affichage décimal.

Il faut distinguer **deux niveaux** de programmation:

Couche de Bas niveau: envoyer 1 octet (1 code ASCII) sur un afficheur ou un périphérique quelconque de visualisation (par exemple ligne série).

Couche immédiatement supérieure: envoyer un nombre décimal (sous forme de chaîne de caractères). Cette couche appellera à plusieurs reprises la couche de bas niveau précédente.

Couches encore supérieures …: on peut aussi imaginer des fonctions qui se servent des fonctions de la couche précédente.

Intérêt:

On voit que **changer d'afficheur** revient alors juste à **changer le bas niveau**. Ce bas niveau peut se nommer "**Driver**".

Cette façon de procéder permet de créer des couches de programmation qui ne dépendent pas des périphériques de sortie, et donc des fonctions bien plus polyvalentes.

L'étude d'un panneau cristaux liquides étant faite dans un chapitre ultérieur, nous supposons ici que nous possédons déjà le Driver suivant, propre à ce périphérique:

Prototype du Driver:	Cahier des charges:
void putch(char c);	Envoi d'un caractère ASCII vers l'afficheur, à la position courante du curseur. On suppose que le curseur se décale automatiquement.

Ce sera la couche de bas niveau.

On peut alors étudier ici la fonction générale **void outinteger(int)** d'affichage d'entier signé. Couche immédiatement supérieure.

On étudiera ensuite une fonction générale **void affich_q1632(long)** pour le format standard **Q16(32) signé**. Ce sera une couche encore au dessus, cette fonction pouvant appeler la précédente.

16 bits fractionnaires donnent raisonnablement 4 chiffres décimaux fractionnaires.

/ Fonction de conversion et d'affichage d'entier signé 16 bits, utilise putch(char) */**

On remarquera l'utilisation d'un drapeau pour éviter d'afficher les zéros en tête.

```
void outinteger(int val)
 {
 char flag_zero = 0;
 char k;   char ascii[5];            /* 5 chiffres à afficher sont suffisant */
 if(val ==0)putch('0');              /* si Zéro, afficher 0 */
      else
       {
       if(val < 0)   { val = - val;  ascii[0] = '-'; putch(ascii[0]);   }     /* signe affiché si < 0 seulement */
       for(k=4;k>=0;k--)    /* Conversion Binaire DCB (1 octet par chiffre) et ASCII */
                                    /* Chiffre de poids faible en premier */
       { ascii[k]= (val%10) + 0x30;
                        /*  Reste des divisions successives par 10, et passage ASCII */
       val = val/10;
       }
       for(k=0;k<=4;k++)    /* sortie du tableau, suppression des zéros en tête, outchaine pas possible ! */
                {                   /* Chiffre de poids fort en premier */
                        if(ascii[k]!= 0x30) flag_zero = 1;
                        if(flag_zero != 0)putch(ascii[k]);
                }
       }
 }
```

/** Fonction de conversion et d'affichage de nombres signés de format Q16(32) */**

/* utilise les fonctions précédentes putch et outinteger */

```
Void affich_q1632(long q1632)
{
unsigned int entiere, frac;
unsigned long val32;
int k;
if(q1632 < 0) {     q1632= -q1632;     /* valeur absolue */
                    putch('-');        /* q1632 n'est pas modifié à l'extérieur de
                    la fonction, car passage par valeur … */
                }
/* ***********    Traitement de la partie entière :  *********** */
entiere=q1632 >> 16;
outinteger(entiere);putch(',');      /* fonction précédente supprimant les zéros en tête */
/* ********     Traitement de la partie fractionnaire :   ********* */
frac = 0xffff & q1632;               /* partie fractionnaire dans frac */
for(k=0;k<=3;k++)                    /* Boucle de 4 tours pour trouver 4 chiffres */
 {
val32 = 10*(unsigned long)frac;  /* val32 variable de travail, on multiplie 4 fois par 10*/
entiere = val32>>16;    /*chaque chiffre (dixième, centièmes ..) se trouve dans la partie entière */
putch(entiere + 0x30);     /* on affiche en ASCII chaque chiffre */
frac = (unsigned int)(0xffff & val32);    /* on enlève à chaque fois la partie fractionnaire */
 }
}
```

Remarque : Pour la partie fractionnaire, on aurait pu multiplier frac une seule fois par 1000 (pour obtenir le nombre de millième, et afficher alors par outinteger. Mais cette fonction enlève les zéros en tête, donc une valeur telle que ,0351 fournirait ,351 à la place ! d'ou faire une autre fonction outinteger ou calculer comme ici chiffre par chiffre.

- ***Discussion sur les précisions :***

En binaire 16 bits fractionnaires donnent une erreur max de $\pm\frac{1}{2}2^{-16} = \pm 7{,}6\ 10^{-6}$

On peut donc dire que l'on pourrait raisonnablement afficher jusqu'à 5 chiffres décimaux après la virgule, le 6ème serait entachée d'une erreur max de presque 8 !

On affiche ici 4 chiffres, cela suffit dans de nombreux cas…. !

14.4. Calcul de fonctions classiques en virgule fixe

Les fonctions mathématiques classiques existent en C, mais elles travaillent sur des nombres en virgule flottante. On peut être amené à en programmer en virgule fixe pour assurer une plus grande vitesse d'exécution ou moins de place en mémoire.

Selon les fonctions, on utilise tel ou tel procédé.

14.4.1. Accès à une table, exemple: racine carré

Simple pour une fonction monotone.

14.4.1.1. Principe

Travailler en entier est peu pratique, car 16 bits fournit 8 bits. On préfère travailler sur des nombres inférieurs à 1 (et évidemment pour des nombres signés, avec le MSB à 0).

On veut donc calculer: $y = \sqrt{x}$ pour **x et y en Q15(16)**

➢ ***Technique de calcul***

Avec une table de taille raisonnable (128 valeurs) et une **approximation linéaires entre deux points**, la précision est satisfaisante.

Toutefois, la précision se dégradant pour de faibles valeurs de x, on pourrait créer une seconde gamme pour x < 1/16, et pour celle ci on calculerait $y = \frac{1}{4}\sqrt{16.x}$

- **Création de la table:**

Soit une table nommée **tabracine** contenant 128 valeurs de la fonction désirée, ici la fonction racine.

Cette table s'obtient facilement avec un tableur type Excel, en calculant pour k allant de 0 à 127, l'expression 32768*RACINE(k/127).

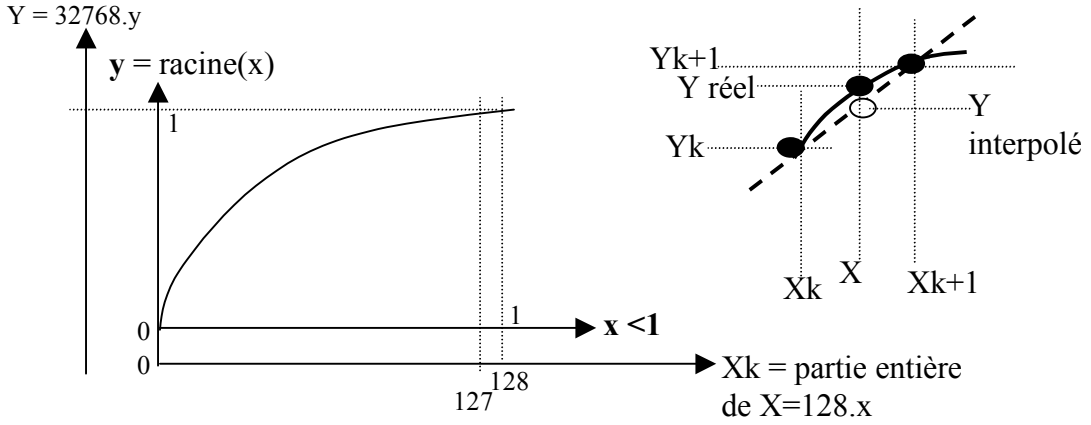
Pour ajouter ensuite une directive assembleur (fdb par exemple) à chaque ligne, il faut passer en mode texte pour concaténer, on fait alors = CONCATENER(″ fdb ″ ; case Ai), et on obtient alors aisément 128 cases avec fdb 0 fdb 2896 fdb 4096 …. On peut alors copier toute la colonne dans un fichier assembleur, ce qui donne **128 lignes de fdb**. Regrouper automatiquement plusieurs valeurs par directive fdb est moins pratique !

Pour l'interpolation linéaire, il faut une valeur de plus (valeur pour x = 1), on ne peut mettre la valeur +1 n'existant pas en Q15(16), on met 32767 qui est 1- pas, on ne commet pas de grosse erreur !.

On trouve 129 valeurs, en Q15(16), de la **racine y** d'une variable **x** telle que **0 ≤ x 1 ≤ 1,** ces valeurs seraient mises en mémoire en assembleur par des directives FDB.

- **Accès à la table: Calcul approché rapide**

A partir de **x** on calcule **X = 128.x**, la **partie entière** donne **l'indice X_k** du tableau, _tabracine(X_k) donne une valeur **Y_k** toujours par défaut. Ce calcul est suffisant dans de nombreux cas, on peut toujours augmenter la précision en augmentant la taille du tableau (1024 valeurs ou plus). Attention si on a bien $y = \sqrt{x}$ on n'as pas $Y = \sqrt{X}$!!!



- **Interpolation linéaire (en option pour plus de précision)**

A partir du point (X_k, Y_k), une interpolation linéaire s'effectue avec une pente: $\frac{Y_{k+1} - Y_k}{X_{k+1} - X_k} = Y_{k+1} - Y$ (car les Xk sont entiers). On a donc simplement:

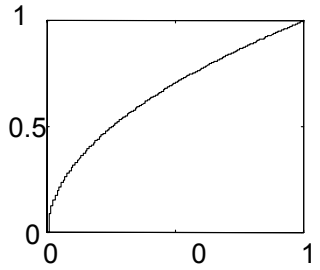
$$y = Y_k + (Y_{k+1} - Y_k)(X - X_k)$$

Il faudra ruser pour le programmer en virgule fixe, sans erreur, et sans débordement.

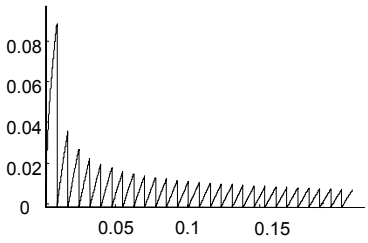
➢ **Précision obtenue**

On obtient une précision suffisante, comme le montre la simulation sous matlab suivante, tracé sans le changement de gamme.

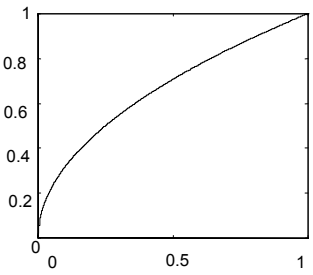
128 points, sans interpolation



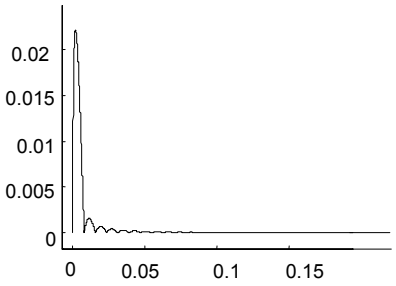
Erreur



128 points, interpolation linéaire



Erreur



14.4.1.2. Exemple de fonction en C

➢ ***Prototype :*** **int racineq15(int valeur) *;***

➢ ***Fonction écrite en C, sans interpolation***

```
int racineq15(int x)
{
return( tabracine[ x >> 8] ) ;    /*  x :  0,--- ---- | ---- ----    indice = 128x :   0--- ---- , ---- ----   */
 }
```

Et quelque part dans un **fichier assembleur,** compilateur IAR Systems

```
          RSEG   CONST
 Public tabracine
 fdb  0              128 lignes de fdb
 fdb  2896
 …….
 fdb  32767     la 128ème valeur : 1, arrondi à 1-pas  valeur max >0 en Q1516.
```

Remarque : ce tableau pourrait aussi se déclarer dans le fichier C…., simple plus simple comme ceci.

➢ ***Fonction écrite en C avec interpolation***

```
Extern  int tabracine[] ;                /* le tableau écrit dans un fichier assembleur */
int racineq15(int x)
{
int yk,ykplus1,indice ;
int x_xk,y,complement ;
indice = x >> 8 ;              /*    x :  0,--- ---- | ---- ----    indice = 128x :   0--- ---- , ---- ----   */
yk = tabracine[indice] ;
ykplus1=tabracine[indice+1] ;
x_xk = (x & 0xFF) << 7 ;   /* pour avoir x_xk en Q15(16) */
complement = ( (long)(ykplus1-yk)*(long)x_xk ) >> 15 ;    /*  produit Q1516  par Q1516  */
y = yk + complement ;
return(y) ;
}
```

```
Et évidemment le  même tableau dans un fichier assembleur, compilateur IAR Systems
        RSEG    CONST
 Public tabracine
 fdb  0,2896,4096,5017 ……32383,32511,32640 332767
```

- ***fonction sans interpolation, entièrement écrite en assembleur***

Dans un fichier assembleur, peut être le même que celui où est tabracine.

Interpolation non faite ici, car plus compliqué, mais faisable …. !

Avec l'assembleur IAR Systems, l'opérande est passé dans D=[AB], le retour se fait par D. Donc dans l'octet haut A se trouve l'indice (voir le petit dessin ci-dessus)

```
        RSEG    CODE
        Public racineq15
        (Extern tabracine)              en option, si le tableau est ailleurs
racineq15   ldx     #tabracine
            tab
            abx
            abx                         X + 2*indice  (chaque valeur est deux octets)
            ldd 0,x                     retour par D
            rts
```

- ***Fonction avec interpolation, écrite en assembleur***

Non étudiée ici…. Mais faisable !

- ***Performances :***

On peut donner un exemple sur un HC12, taille en octet et temps d'exécution dépendent évidemment aussi du compilateur.

	Fonction C SQRT flottante	**Virgule fixe Tout en C interpolation**	**Virgule fixe Tout en C** Sans interpolation	**Virgule fixe, tout assembleur, sans interpolation (gain non significatif)**
Nombre de Tck d'exécution	5200Tck	257 Tck	26 Tck	25 Tck
Taille du code (boot du C et tabracine comprise, la différence seule est significative)	1400 octets	570 octets	513 octets	510 octets

- ***Amusant :***

En testant la fonction racineq15(**sans interpolation**), et en affichant x et y en entier, on trouverait que la racine de par exemple 17102 est 23530 ! mais c'est normal, car dans la tête il faut comparer : $\sqrt{17102/32768} = \sqrt{0,5219112}$ = **0,722434** et $23530/32768$ = **0,7181 valeurs voisines.**

Avec interpolation (ou calcul en flottant), on trouverait : 23672. Donc valeur réelle : $23672/32768 = 0,722412$ valeur identique pour une précision Q15(16) ($\pm 2^{-17} = \pm 1.510^{-5}$).

On rappelle cependant que même avec une interpolation linéaire, la précision se détériore pour les petites valeurs, on a vu le remède qui consisterait à se créer des gammes : par exemple si x < 1/16 (2048 entier), on calculerait $y = \frac{1}{4}\sqrt{16.x}$

14.4.2. Méthodes d'approximation successives

♦ Exemple **algorithme de NEWTON** pour $\sqrt[n]{a}$

La suite $X_{k+1} = \frac{1}{n} X_k . (n - 1 + \frac{a}{x_k^n})$ converge vers $\sqrt[n]{a}$

On atteint 10^{-7} au bout de 5 à 6 itérations, si on part d'une valeur estimée à 20 ou 30% près . Pour n = 2, nous avons la $\sqrt{}$ déjà étudiée. n peut être < 0 : n = -1 fournit 1/x .

♦ **Dichotomie**

Pour des **fonctions monotones**, **et dont l'inverse se calcule aisément**.

Il suffit d'essayer à chaque fois la valeur centrale au domaine de valeurs possibles de départ (En virgule fixe on essaye de proche en proche les poids en commençant par le plus fort). On teste si la valeur essayée est plus élevée ou moins élevée que celle recherchée ce qui réduit par deux le domaine d'investigation, et on corrige en conséquence (remise à zéro au non du bit essayé). Pour un codage sur N bits il suffit de N essais.

14.4.3. Développements limités, polynômes de Tchebycheff

Pour Log, sin, cos ….

La théorie mathématique dépasse le cadre de ce cours.

Mais pour calculer un développement tel que :

$$y = a_0 + a_1 x \ + a_2 x^2 + a_3 x^3 + ... + a_{n-1} x^{n-1}$$

On utilise la technique itérative analogue à celle déjà vue pour la conversion DCB, binaire,

$$y = (....((((0.x + a_{n-1}).x + a_{n-2}).x + a_{n-3}).x + ... + a_1).x + a_0$$

Formule très aisée a programmer par une boucle $y = y.x + a_k$ avec y partant de 0.

En virgule flottante, aucun problème évidemment, en virgule fixe, on raisonne avec la dynamique pratique tout au long du calcul.

15. GENERATION D'INTERVALLES DE TEMPS, TIMER

15.1. Par logiciel

Application : Génération d'un **simple retard** d'une durée déterminée
Inconvénient :
Possible si le processeur n'a rien d'autre à faire !
Peu précis pour cadencer des opérations à effectuer, car on génère ainsi un simple retard et non une cadence.
Ralentissement si interruptions, on pourrait en fait les masquer. Inutilisable ou presque sur un système multitâche à temps partagé (durée évidemment très allongée !)

Exemple sur HC12 (Avec F_BUS quelconque)
Création d'une simple fonction en C et assembleur. Une programmation tout en C est impossible car le retard généré dépend du processeur et de la structure du code assembleur.

- ***Prototype*** : **tempo(unsigned int millisecondes) ;**
- ***Fonction écrite en assembleur :***

Sur compilateur IAR Systems, qui passe le paramètre dans D, carte à Tck = 500ns par défaut ou modifiable par label F_BUS en #define.
Technique : on parcourt un certain nombre de fois une boucle (sur X) qui dure exactement 1ms. Cette durée de 1ms est générée par une petite boucle (sur Y) elle utilise une instruction nop d'ajustement de temps. On peut remarquer la prise en compte de F_BUS poutr générer le même retard quelque soit F_BUS.

```
            #include      "ini_carte.h"   // ou est défini F_BUS
            PUBLIC  tempo
            RSEG  CODE
**********      Temporisation  en ms   HC12 pour C   ********************
*****           Avec F_BUS défini par un #define dans "ini_carte.h"
            #define     NN    250*F_BUS - 2       // calculé à la compilation
tempo:          // cycle de base 1ms réalisé avec N tours tel que:
      pshd      // 4*N + 7 + k = 1000*F_BUS (en MHz)  avec k = 1
      pshx      // N = 250*F_BUS(MHZ) - 2
      pshy
      xgdx        valeur en ms en D donc  valeur ms en X
bou6   ldy   #NN   2 cyc        | exemple: pour Fbus = 2MHz, NN = 498
bou7   dey         1 cyc        |
       bne   bou7  3 cyc        | total 498*4+6+2 =2000 cycles de Tck = 500ns
       nop         k = 2 cyc    |
       nop
       dex         1 cyc        |
       bne   bou6  3 cyc        | Fin de boucle de X.millisecondes
       puly
       pulx
       puld
       rts

    END
```

- ***Fichier entête : ini_carte.h***

```
#ifndef   F_BUS      // 2 4 8 16 20 ou 24 MHz,  Debug possible avec 2MHz ou 4 MHz
 #define   F_BUS      2      //Ici Par défaut 2 MHz, au delà de 4Mhz, télécharger puis exécuter par Reset
#endif
```

Ce fichier doit aussi être inclus dans tous fichiers C ou assembleur qui à besoin de F_BUS.

- ***Remarques :***

On a négligé les temps de passage de paramètres et de saut et retour au sous programme. Le **temps généré** est donc **très légèrement supérieur** au temps désiré.

Cette tempo peut servir évidemment aussi en assembleur seul, elle préserve en effet tous les registres, on l'utiliserait dans ce cas en faisant : LDD #ms puis JSR tempo

- ***Exemple d'utilisation en C:***

Fonction **action()** à **cadencer** toutes les 20ms environ:
(en fait un peu plus, car petite erreur dans tempo et temps d'exécution d'action() négligé !.

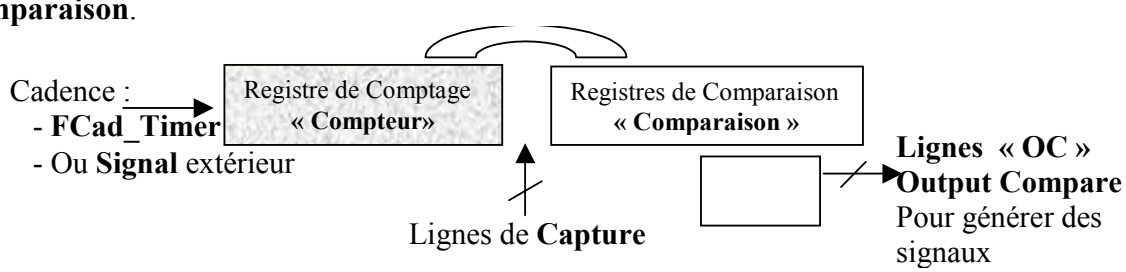
```
For( ; ;)   {   action() ;
                tempo(20) ;
            }
```

15.2. Par Timer (Exemples sur HC12)

15.2.1. Qu'est ce qu'un Timer ?

On peut évidemment mesurer une durée, ou générer un intervalle de temps donné par logiciel (boucle de comptage). Mais le microprocesseur ne fait alors rien d'autre durant ce temps et d'autre part, le calcul de la durée générée nécessite l'examen du code assembleur et de la durée d'exécution des instructions. Donc peu de souplesse. Un Timer permet de décharger le microprocesseur de cette tâche.

Un **Timer** contient **un registre de Comptage** et un ou plusieurs **registres de comparaison**.



Le compteur peut s'incrémenter à une cadence interne Fcad_Timer (Plusieurs valeurs possibles par programmation), ou sur une cadence extérieure.

De nombreux Timers ont un compteur accessible seulement en lecture.

On note plusieurs mode possible de fonctionnement :

♦ ***Le Mode Overflow :***

Le Compteur déborde. L'**événement** correspondant **Overflow** survient, signalé par un drapeau **OV** (et une interruption si validée) .

♦ ***Le Mode comparaison :***

Si Compteur = Comparaison, L'événement correspondant « **Output Compare »** survient, signalé par un drapeau **OC** (et interruption si validée).

A chaque événement comparaison, on peut aussi faire évoluer automatiquement une ligne de sortie Output Compare, afin de générer des impulsions ou des signaux périodiques.

♦ ***Le Mode capture :***

Si un front actif sur la ligne de capture survient, la valeur du Compteur est alors souvent recopiée dans le registre comparaison, ou dans un autre registre. L'événement correspondant **capture** survient, signalé par le drapeau CA (et interruption si validée).

15.2.2. Fonctionnement du Timer de l'HC12 en ' Comparaison '

Le compteur étant pour ce Timer en lecture seule, son mode Overflow ne permet pas de générer des intervalles de temps quelconques puisque l'on ne peut pas charger le compteur à une valeur initiale .

La carte 68HC12 employée utilise un oscillateur externe (XTAL) à 4 MHz, et au démarrage par défaut son horloge interne E se fixe à : **Eclck = F_BUS = Fck = 2 MHz** (sauf modification par programmation, on pourrait aller jusqu'à 24MHz !) Cette cadence sert donc de référence générale. On peut remarquer plusieurs noms pour désigner la même chose ….

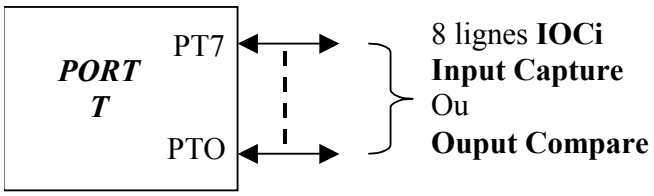
Un compteur de 16 bits, appelé « **Free-runnig counter** » s'incrémente librement, à une cadence de **Eclock/n** (rapport programmable). Sa valeur peut être lue à tout moment par l'intermédiaire du registre **TCNT** (en **lecture seulement**).

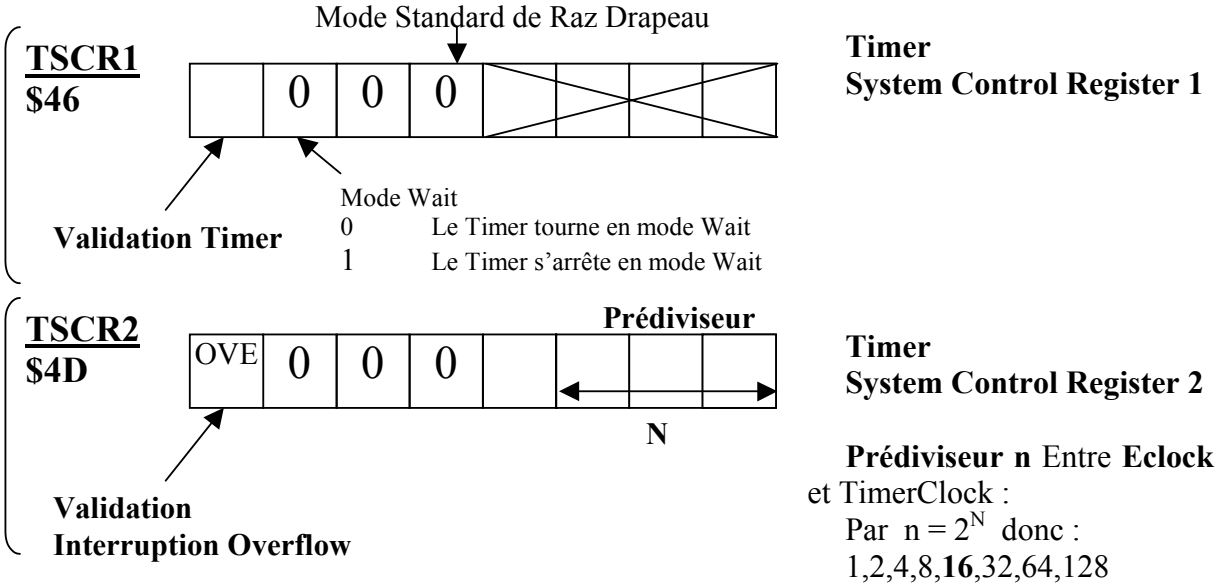
15.2.2.1. Description résumé des principaux registres :

Le timer de l'HC12 possède une soixantaine de registres, on décrit ici le minimum !

Remarque : Lors de la première initialisation des différents bits, on peut directement écrire 0 dans les bits non utilisés (soit c'est la valeur par défaut, soit écrire 0 est sans effet).

Nous ne décrivons que les fonctions de base. Il utilise le PORT T pour la gestion de **lignes IOCi** d'Input Capture (mesure de fréquences) ou Output Compare (Générateur de signaux)



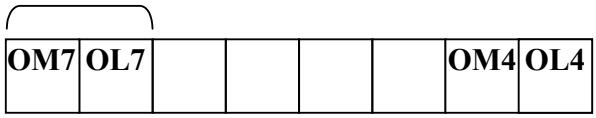


TIOS
$40

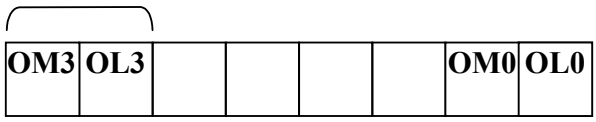
0 Travail en Input Capture
1 Travail en Output Compare
Si on se sert des lignes, on doit sélectionner leur mode de travail au moyen des registres suivant : TCTL1 et 2

Timer
Input Capture &
Ouput Compare
Selection
Obligatoire pour valider le mode Output Compare, sinon Input Capture par défaut !

TCTL1
$48

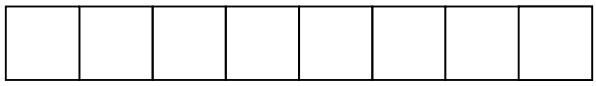


TCTL2
$49



Timer
Control Register 1 et 2
Pour lignes IOC7 --- IOC0
Si Output Compare :
00 non actives
01 IOCi Toggle
10 IOCi à 0
11 IOCi à 1

TIE
$4C



Timer Interrupt Enable

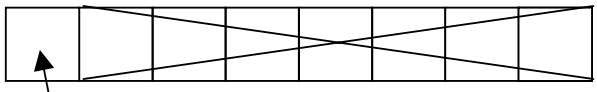
Validation Interruption lignes IOCi
En Ouput Compare ou Input Capture

TFLG1
$4E

Timer Interrupt Flag 1
Mode Standard :
Remise à zéro en écrivant 1

Drapeaux d'interruption lignes IOCi
En Ouput Compare ou Input Capture

TFLG2
$4F



Timer Interrupt Flag 2
Mode Standard :
Remise à zéro en écrivant 1

Drapeau d'interruption Overflow

TCNT
$44

Timer Counter 0 à 65535
Lecture Seule

TC0...7 soit TCi
$50+2i

Timer Compare (N°0 à N°7)

CFORC
$41

f7	f6					f1	f0

Compare Force Register

Ecrire un bit à 1 **force une action identique à celle de l'événement comparaison** sur la ligne PTi correspondante (si elle est dans ce mode).
Utilisation : Si l'action Toggle est programmée, on peut lorsque l'évènement est survenu, provoquer par programmation de nouveau un Toggle, et ainsi générer des impulsions de fréquence = 1/Ttimer (Au lieu de 1/2Ttimer), donc une **fréquence double**
(Mais on obtient des **impulsions seulement**, et non un signal carré).

15.2.2.2. Résumé du fonctionnement en « Comparaison » (Output Compare)

→ On doit tout d'abord initialiser le Timer, en validant celui ci (TSCR1), en programmant le prédiviseur (par exemple rapport k de 16, dans TSCR2), et en validant le fonctionnement Output Compare sur le canal désiré (TIOS).

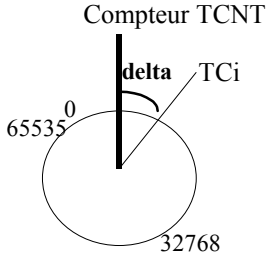
→ Lorsqu'il y a égalité entre TCNT le « Free-running counter » et un des registres TCi (i = 0 à 7), on parle d'un **événement comparaison**, et alors :
- Le drapeau associé N°i dans TFLG1 passe à 1
- Une interruption survient si validée
- On peut programmer (par TCTL1 et TCTL2) des actions sur les lignes de sorties **IOCi** (du port T). Pratique pour générer des signaux périodiques.

→ Le dernier registre CFORC permet de générer par programme si besoin est la même action, pratique pour générer un signal carré de fréquence F à partir d'une cadence F d'interruption : on programme l'action « toggle » c'est à dire changement d'état, lors de l'événement, et dans le programme d'interruption on crée de nouveau un changement d'état avec CFORC.
Ne pas oublier ensuite la Raz du drapeau concerné !

→ ***Programmation des registres TCi (F_BUS = 2MHz, et prédiviseur k = 16)***
Pour armer et réarmer le Timer pour chaque canal de comparaison OCi, on doit régulièrement écrire dans **le registres TCi** concerné de 16 bits. On doit distinguer la première fois du mode répétitif :
Avec **Fck = F_BUS = 2 MHz** on a **T_BUS = 5OOns**
Si on a programmé un rapport de division **k de 16** Le compteur s'incrémente à Fck/16, donc tous les : 16*500 = 8 µs et donc **Tcad_Timer = 8 µs**

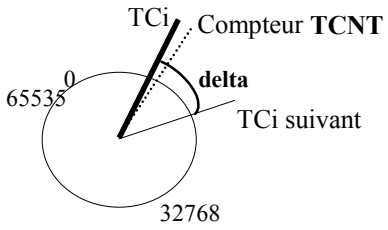
Si on initialise à un instant donné un registre **TCi** à TCNT + **delta**, un événement « Comparaison » (avec interruption si validée) doit survenir **delta*8 µs** plus tard.

Mode **une fois** ou la **première fois :**
TCi = TCNT + delta ;



Si on initialise alors de nouveau TCi à TCi + **delta** (et non TCNT car TCNT peut déjà s'incrémenter avant et durant ces instructions), un événement « Comparaison » (avec interruption si validée) se déclenchera **après exactement delta*8 µs**.

Mode **Répétitif :**
TCi = TCi + delta ;



Si on réitère indéfiniment ce procédé, on peut programmer une cadence régulière d'interruption, par conséquent une avance régulière du moteur pas à pas, à la précision du quartz du microcontrôleur près.

Remarque *: Pour d'autres processeurs (et d'autres Timer) on pourrait conserver la référence fixe et réinitialiser le compteur à chaque comparaison valide. Le 68HC12 ne le permet pas car TCNT est un registre en « lecture seule ».*

15.2.3. Travail sans interruption

Application : Génération d'un **simple retard** d'une durée déterminée, **sans utiliser les interruptions**.

Avantage : le processeur peut éventuellement effectuer quelques autres taches sur interruption (cela ne retarde pas le compteur du Timer).

Attention:

Délai augmenté légèrement si en fin de comptage le processeur est parti exécuter une interruption.

Pour cadencer des actions, préférer évidemment une programmation Timer sur interruption !

1) Exemple N°1 : Fonction de retard analogue à la précédente

Création d'une simple fonction en C analogue à la précédente :
Prototype **tempo(unsigned int millisecondes) ;**
Utilisation par exemple du registre de comparaison TC3
Interruption IT_TC3 non validée !
Temps de passage de paramètres et de saut et retour au sous programme négligés, temps de chargement de TC3 non nul... donc temps généré très légèrement supérieur au temps désiré.

Initialisations :
```
TSCR1 = 1 ;   // Validation Timer
TSCR2 = 4 ;   // Prédiviseur 16
TIOS = TIOS |  0x08 ;        // Travail en Compare sur TC3
```

```
Tempo(unsigned int ms)                  // Pour F_BUS = 2 MHz
{
TC3 = TCNT + 125*ms ;                   /* 1ms = 125*8µs  */
  TFLG1 = 0x08 ;          /*  Raz drapeau IT_TC3  initial, obligatoire !   */
  While(  (TFLG1 & 0x08) = = 0) ;   /*  attente drapeau égalité compteur et TC3 */
  }
```

Retard max : 8µs.65535 = 0,524280 s
Retards plus important : un compteur logiciel, permettrait de compter des retard élémentaires.

2) Exemple N°2 : Fonction répétitive

Dans une boucle sans fin, on peut aisément créer un *rythme précis* à la précision du quartz près avec une fonction répétitive, ou TC3 est réarmé par rapport à l'ancien TC3 et non par rapport à TCNT.

```
Tempo_repetitive(unsigned int ms)              // Pour F_BUS = 2 MHz
    {
  TC3 = TC3 + 125*ms ;                  /* 1ms = 125*8µs  */
    While(  (TFLG1 & 0x08) = = 0) ;   /*  attente drapeau égalité compteur et TC3 */
  TFLG1 = 0x08 ;            /*  Raz drapeau IT_TC3  obligatoire !   */
    }
```

Exemple :

Fonction **action()** à **cadencer** toutes les 20ms exactement (Précision du Quartz):

```
tempo(20) ;
For( ; ;)   {  action() ;
               tempo_repetitive(20) ;
               }
```

15.2.4. Travail avec interruption

Application : **cadencer des taches** à des **intervalles réguliers**.

Avantage : le processeur peut effectuer en parallèle d'autres choses dans un programme principal, et même effectuer quelques autres taches sur interruption (cela ne retarde pas le compteur du Timer), on peut donc générer plusieurs cadences en parallèles.

Peut fonctionner dans un système multitache à temps partagé, avec cette interruption prioritaire sur tout le reste.

Attention:

Délai augmenté légèrement si en fin comptage, le processeur est déjà en interruption (quelconque, ou autre Timer).

- ***Exemple avec F_BUS = 2MHz***

Soit la variable globale **cadence_ms** contenant le nombre de millisecondes désirées.

Fonction action() à cadencer exactement toutes les 20ms :

Cadence minimale :
65535*8µs = 524280
cadence_ms ≤ 524 millisecondes

```
int cadence_ms = 20 ;       /* cadence en ms */
main()
{
TIE = TIE | 0x08 ;    /*  IT_TC3 validée sans
                                modifier d'autres bits ! */
   TC3= TCNT + 125*cadence_ms ;    /*1ms=125*8µs  */
   Valid_it() ;     // Masque général des IT à 0
   While(1) ;   /*  boucle attente IT ou autre travail … */
   }

 interrupt[0xFFE8-0xFF80]   void it_tc3(void)
 {  TFLG1 = 0x08 ;     /* raz drapeau IT_tc3  */
    TC3 = TC3 + 125*cadence_ms ;      /* en répétitif */
    action() ;  /* la fonction a cadencer */
 }
```

Remarque : on a écrit TIE = TIE | 0x08 et non TIE = 0x08 ; pour ne mettre à 1 que le bit désiré (d'autres interruptions peuvent déjà être validées, cette écriture manipulant 1 bit et lui seul est possible

sur TIE car ce registre se comporte comme une mémoire en écriture et lecture, il ne faudrait surtout pas faire cela avec TFLG1 !)

➢ ***Obtention de cadences plus lentes (F_BUS = 2MHz)***

On désire générer des intervalles de temps bien plus long, donc mesurés en secondes.

On se sert du Timée et de TC3 par exemple pour générer un temps de base, et un compteur logiciel compte le nombre d'interruption avant d'actionner la tache.

Soit la variable locale **nombre_it_** pour compter les interruptions.

Et la variable globale **cadence_s** pour contenir le nombre de secondes désirées.

Temps de base pratique : 0.5 seconde = 62500*8µs

Donc nombre_it = 2*cadence_s

Fonction action() à cadencer exactement toutes les T = 60s :

Cadence minimale :
65535*0.5 donc 32767
Cadence_s ≤ 32767 secondes

Donc environ 9h

```
/* on suppose masque général d'interruption à 0 */
unsigned int cadence_s = 60 ;   /* cadence en s */
main()
{
TIE = TIE | 0x08 ;   /*  IT_TC3 validée */
    TC3 = TCNT + 62500 ;
    Valid_it() ;     // Masque général des IT à 0
    While(1) ;   /*  boucle attente IT ou autre travail …
*/
    }
```

Programme d'interruption, notation IAR System.

Le compteur est statique obligatoirement sinon il serait relis à zéro à chaque interruption !

```
Interrupt[0xFFE8-0xFF80]    void it_tc3(void)
{
static unsigned int nombre_it = 0 ;
TFLG1 = 0x08 ;  /* raz drapeau */
TC3 = TC3 + 62500 ;       /* pour 0.5s */
If (nombre_it ++ > = 2*cadence_s - 1)
                    {
                    Action() ;   /* La tache à cadencer */
                     nombre_it = 0;
                     }
}
```

Remarques importantes : si on doit modifier ensuite la cadence, il faut évidemment que celle-ci soit en variable globale (comme ci-dessus), mais alors il faut bien écrire :

if (nombre_it ++ > = 2*cadence_s - 1) et non if (nombre_it ++ > = 2*cadence_s - 1)

Car dans le cas d'une cadence demandée inférieure, le compteur nombre_it++ peut avoir déjà dépassé 2*cadence_s – 1 et l'égalité ne sera possible qu'après le modulo du compteur et donc bien plus tard, surtout si le compteur est un long !

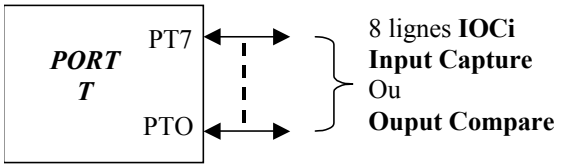
D'autre part pour repartir sur une première cadence exacte, on doit à chaque modification de celle-ci remettre le compteur nombre_it à 0, celui-ci devra donc aussi être global et non plus statique de la fonction d'interruption.

16. MESURE DE FREQUENCE ET DE PERIODE

16.1. Le Timer HC12 en 'Input Capture'

Ce mode permet de mémoriser le contenu du compteur TCNT dans un registre **TCi**
Application : Mesure de période, de durée d'impulsion.

8 entrées possibles : les lignes IOC1



16.1.1. Format des registres principaux

On retrouve les registres utilisés en mode comparaison (voir chapitre précédent sur le Timer) :

TSCR1 Validation du Timer
TSCR2 Interruption overflow du TCNT et prédiviseur.
TIOS Travail sur chaque ligne en Input Capture (**0**) ou output compare (1). Donc par défaut : Mode Capture.
TIE Validation d'interruption sur chacune des 8 lignes.
TFLG1 Drapeau correspondant à chaque ligne
TCNT Le compteur 16 bits
TC0..TC7 Qui servent aussi de registres de Capture du TCNT sur les fronts actifs des lignes TCi

Et en plus pour déterminer l'événement capture:

TCTL3
$4A

EDG 7B	EDG 7A					EDG 4B	EDG 4A

TCTL4
$4B

EDG 3B	EDG 3A					EDG 0B	EDG 0A

Timer Control Register 3 et 4
Pour lignes IOC7 --- IOC0
Pour Input Capture

00	Inactif
01	front ↑
10	front ↓
11	front ↑et ↓

16.1.2. Mode de travail principal en mode capture

Sur événement front (montant, descendant, ou les deux) sur une des lignes IOCi (lignes N°i du **PORT T**), la valeur du TCNT est stockée dans le TCi correspondant à la ligne. Le drapeau correspondant dans le registre TFLG1 passe à 1. Une interruption « Enhanced Capture Timer Channel i » est déclenchée si validée.

Application principale : mesure de largeur d'impulsion, de période.

16.2. Le Pulse Accumulateur de l'HC12

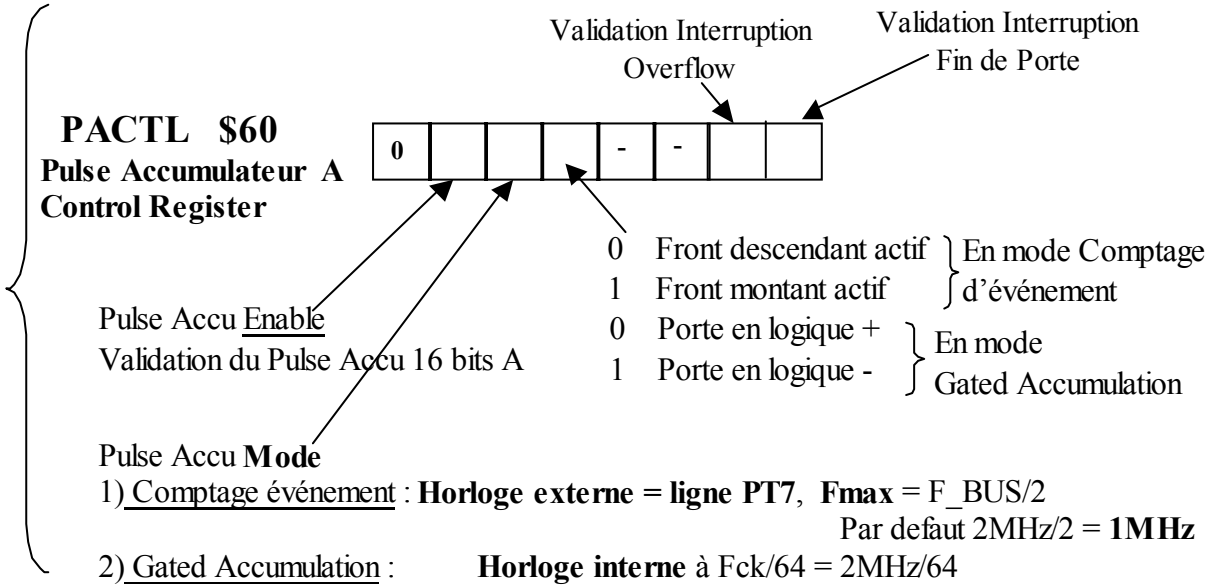
16.2.1. Principe

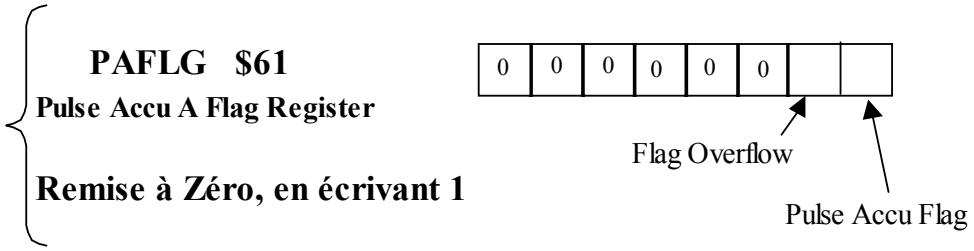
Le Pulse accumulateur, comme son nom l'indique, sert à accumuler donc à compter des impulsions. Il est tout désigné pour le **comptage d'évènements physiques** sur une ligne (fronts montant ou descendant) et pour mesurer des largeurs d'impulsion. L'HC12 contient quatre pulse accu de 8 bits (PAC3..PAC0), et de nombreux modes de travail. Ils peuvent être associés deux par deux pour une comptage sur 16 bits et former les pulse accu 16 bits A et B. Nous ne décrivons ici que le strict minimum, et seulement le Pulse accu A (formé de PAC3 PAC2).

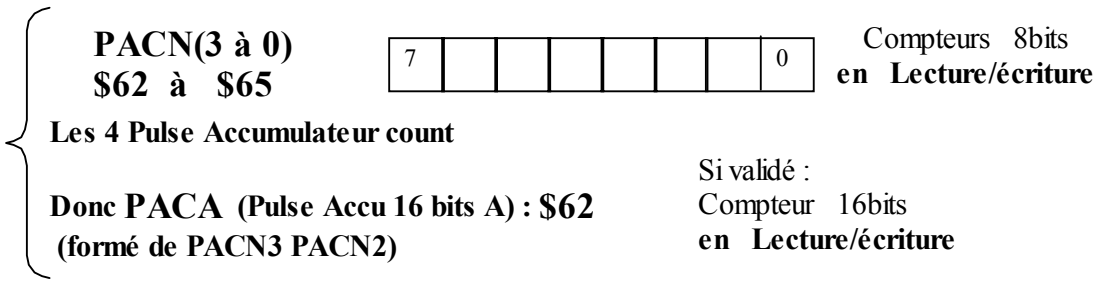
Remarque : A un pulse accu matériel de 8 bits ou de 16 bits, on peut toujours si nécessaire adjoindre un **compteur supplémentaire logiciel** de 8, 16 bits ou plus, pour réaliser des comptages sur N bits quelconques. Il suffit en effet d'incrémenter ce compteur logiciel supplémentaire sur interruption Overflow du Compteur physique.
Ceci est possible également pour le compteur TCNT du Timer.

Application principale : Mesure directe de fréquences.

16.2.2. Registres principaux du Pulse Accu 16 bits A







16.3. Mesure de fréquences et de périodes (en C)

16.3.1. Mesure directe de fréquence

On mesure durant un intervalle de temps **θ** le nombre de périodes du signal.
Donc à priori **le plus classique** , mesure durant **1seconde**.

➢ ***Calcul de F et précision :***

$F = \frac{N}{\theta}$ et $\Delta F = \frac{1}{\theta}$ ($\Delta N = 1$) **1Hz** si **θ = 1s**, 10Hz si θ = 0.1s …., 1KHz si θ = 1ms.

A priori une cadence de mesure d'environ 1 seconde est très pratique, la **durée de mesure** pourra donc être de 1 seconde exactement, ce qui fournit un résultat **à 1Hz près**.

D'autre part $\frac{\Delta F}{F} = \frac{1}{\theta . F}$ si on néglige $\Delta\theta$ ....

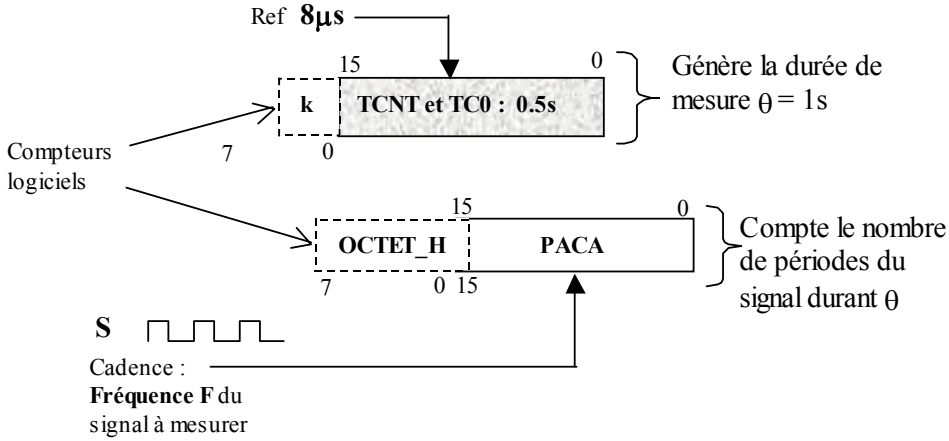
La précision relative diminue évidemment dans les basses fréquences et à moins d'augmenter la durée de mesure, il vaudra mieux passer par la période.

➢ ***Fonction C***

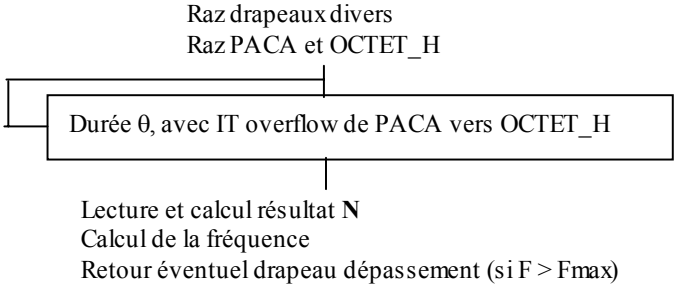
Le **signal S** est envoyé sur **PT7** (**entrée du Pulse Accu 16 bits A**). Compteur **PACA**.
On choisit **TC0** pour générer la **durée de mesure** de durée **θ**.

Pour dépasser 65536, ce qui ne ferait pour θ = 1s que 65536Hz, on ajoute un compteur logiciel OCTET_H (8 bits non signés) permettent de compter sur 24 bits, de 0 à 16777215.

On choisit par exemple le **prédiviseur classique de 4** pour le TCNT et **F_BUS** standard **de 2 MHz**. Comme TC0 seul ne peut alors pas générer un temps plus grand que 65535*8µs = 0,524s, on choisit une cadence de base de 0,5 seconde, pour générer la porte de mesure de 1s.



On suppose le pulse accu déjà initialisé dans le mode correct.



Dépassement théorique à 16777215 périodes du signal durant la durée θ=1s..

Mais **Fréquence max pratique mesurée** : environ 85% de Fck/2 soit environ **850kHz**

(au delà l'interruption overflow ralenti un peu trop)

Cahier des charges de la fonction:
En entrée : pointeur vers fréquence en Hz, en flottant.
En retour : drapeau éventuel (1 Ok, 0 débordement) de fréquence trop élevée (limite écrite dans la fonction)
Temps de calcul : 1 seconde par construction

```
char frequence_directe(float *f)                // en Hz      Prédiviseur 16, F_BUS 2MHz
{                                               // On suppose les initialisations effectuées
char k; long N;
PAFLG = 0x02;            // Raz drapeau overflow du Pulse accu A
PACA=0;OCTET_H = 0;      // Compteur 24 bits à 0
TC0 = TCNT + 62500;      // 0.5 s pour le TC0
TFLG1 = 0x01;            // raz drapeau TC0
// LA PORTE DE MESURE de 2*0,5s avec OCTET_H++ à chaque Overflow du PACA par IT
for(k = 0; k < 2; k++)   // LE PACA s'incrémente tout seul à chaque front du signal sur PT7
   { while( (TFLG1&0x01)==0);  // attente 0,5s
     TFLG1 = 0x01;                    // raz drapeau TC0
     TC0 = TC0 + 62500;       // re armement TC0
    }
N = 65536*(long)OCTET_H + (long)PACA;
*f=( float)N;  // f = N
if(N>850000) return(0); else return(1);   // Drapeau OK si F < Fmax
}
```

- ***Les initialisations préalables***

```
// Initialisations de base du Timer
TSCR1 = 0x80;     // valid timer
TSCR2 = 4;                // prediviseur de 16

// POUR COMPTER FRONTS DU SIGNAL:  Initialisation du Pulse Accu A 16 bits
PACTL = 0x52;  // 0 1 0 1 0 0 1 0   Validé, Comptage sur PT7, It overflow validé

// Initialisation du Timer TC0 pour ensuite durée mesure 1s = 2*0,5s
TIOS |= 0x01;   // TC0 en comparaison sans IT
//  Validation des interruptions générales (en assembleur cli)
valid_it();
```

- ***La fonction d'interruption overflow du Pulse Accu***

```
// INCREMENTATION DE OCTET_H: Sur Overflow du PACA
interrupt[0xFFDC - 0xFF80]   void ov_pulse_accuA_overflow(void)
{
PAFLG = 0x02;   // raz drapeau
OCTET_H ++;    // incrémentation des 8 bits supplémentaire (compteur logiciel)
OCTET_H = OCTET_H;
}
```

16.3.2. Mesure en passant par la période

On compte durant une période T du signal à mesurer le nombre de périodes θ d'une horloge de référence interne rapide.

➢ ***Calcul de F et précision***

On a : $F = \frac{1}{N\theta}$ avec θ = 8.10^{-6}, et $\Delta F = \frac{1}{\theta.N^2} = \frac{F}{N}$ (ΔN = 1)

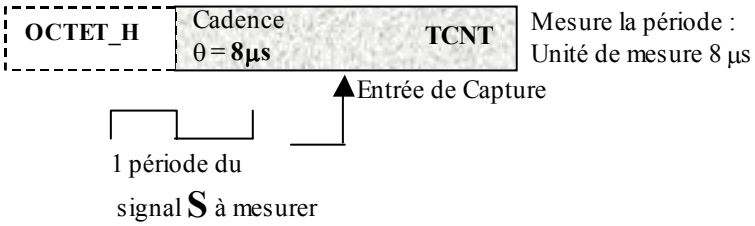
On peut calculer $\frac{\Delta F}{F} = \frac{1}{N} = F.\theta$ (Δθ toujours négligeable : Quartz du microcontroleur).

La période de référence θ doit être petite pour augmenter la précision.
Il est donc plus précis de passer par la période pour des fréquences basses.

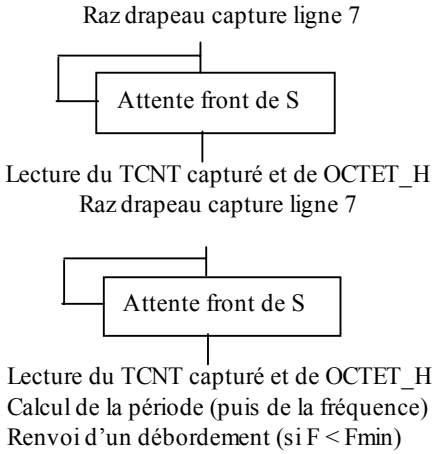
➢ ***Fonction C***

Le **signal S** est également envoyé sur **PT7,** qui est utilisée cette fois ci en entrée **Input Capture**. On choisit toujours un **prédiviseur de 16 et F_BUS de 2 MHz**. Donc l'horloge de référence fera 8μs de période.

La période maximale Tmax mesurable sera de 65536*8μs = 0,24 s, donc un peu petit. On peut donc ajouter comme précédemment un compteur logiciel OCTET_H de 8 bits pour mesurer des Tmax de 134s !



On suppose le Timer déjà initialisé dans le mode voulu



Fmin = 7,5mHz !!.

Remarque : on ne peut pas mettre à zéro le TCNT du Timer de l'HC12 (en mode standard). On devra donc tenir compte des deux lectures et du modulo du compteur.

Cahier des charges :

En entrée : pointeur vers fréquence en Hz, en flottant.
En retour : drapeau de fréquence trop basse (1 Ok, 0 débordement, F < Fmin)

Durée de mesure: par construction, entre une et deux période du signal à mesurer (dans le cas le plus défavorable, on vient juste de passer un front actif quand on commence une mesure).

```
char frequence_par_periode(float *f)        // Prediviseur 16, F_BUS = 2MHz
                                            // On suppose les initialisations effectuées
{
unsigned char octeth1,octeth2; unsigned int tcnt1,tcnt2;
unsigned long N1,N2,deltaN;
const float F_cad_timer = 125000.0;  // avec prédiviseur de 16 et F_BUS 2 MHZ
TFLG1 = 0X80;                        // Raz drapeau capture ligne 7
while((TFLG1 & 0x80)==0);       // attente premier front actif de S sur ligne 7
octeth1 = OCTET_H; tcnt1 = TCNT;   // saisie initiale des valeurs du compteur 24 bits
TFLG1 = 0x80;                        // Raz drapeau capture ligne 7
while((TFLG1 & 0x80)==0);       // attente second  front actif de S sur ligne 7
octeth2 = OCTET_H; tcnt2 = TCNT;   // saisie des nouvelles valeurs du compteur 24 bits
N1 = 65536*(long)octeth1 + (long)tcnt1; N2 = 65536*(long)octeth2 + (long)tcnt2;
if(N2>N1)           {       deltaN = N2-N1;
                    }                                       // on tient compte du modulo ici
else                {       deltaN = 65536*256 + N2-N1;
            }
*f = F_cad_timer/(float)deltaN;
if(*k > 0,0075)return(1); else return(O);
}
```

- ***Les initialisations préalables***

```
// Ini de base du Timer
TSCR1 = 0x80;  // valid timer
TSCR2 = 0x04;  // prediviseur de 16

// POUR COMPTER CLK TIMER DURANT UNE PERIODE DU SIGNAL
// Initialisation du Timer TC7 en Input capture SANS interruption
TIOS = TIOS & (0xFF - 0x80);    // inutile si au début, car Input capture par défaut, mais sécurité si modif avant
TCTL3 = 0x40;   // 01 00 00 00   Ligne 7 active montant
TSCR2 = TSCR2 | 0x80;  // Interruption Overflow du TCNT
//  Validation des interruptions générales (en assembleur cli)
valid_it();
```

- ***La fonction d'interruption overflow du TCNT du Timer***

```
// INCREMENTATION DE OCTET_H:          Sur IT Overflow du TCNT
interrupt[0xFFDE - 0xFF80]  void ov_tcnt(void)
{
TFLG2 = 0x80;   // raz drapeau;
OCTET_H++;
}
```

16.3.3. Idée de programme complet, et précision obtenue réellement

➢ ***Idée de programme complet***

En mode répétitif, on pourrait commuter automatiquement les deux méthodes selon la fréquence mesurée. Le basculement entre les deux méthodes (par un drapeau flag_methode) se ferait à 353 Hz (voir l'étude qui suit). Mais pour éviter de changer constamment de méthode à la fréquence limite, il serait judicieux de programmer une petite hystérésis.

➢ ***précision obtenues, et choix de la méthode.***

En fréquencemètre direct, soit une porte de mesure de θ = 1seconde (1000ms).

$$\frac{\Delta F}{F}(\%) = 100\frac{1}{\theta.F} = 100\frac{1}{F}$$

En périodemètre, et une référence θ à 8µs, on a : $\frac{\Delta F}{F}(\%) = 100F.\theta = 8.10^{-4}F$

Les deux courbes tracés sur le même graphique permettent de voir que pour une fréquence critique il vaut lieux changer de méthode :

$100/Fc = 8.10^{-4}.Fc$

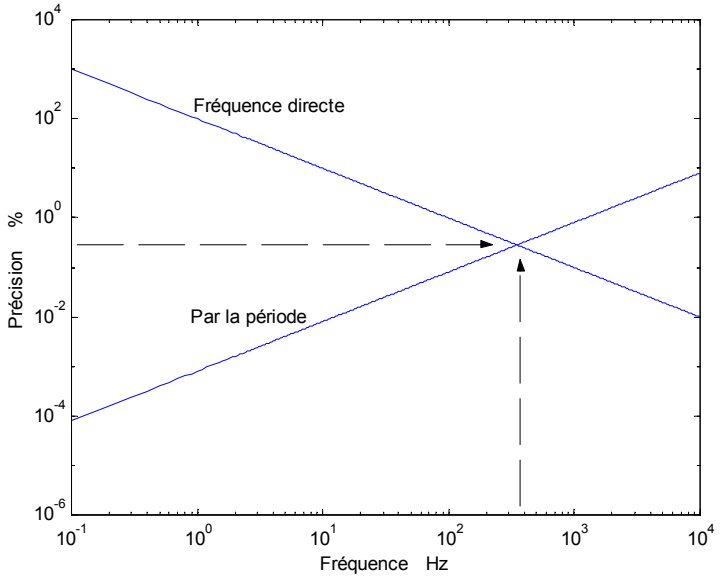
Donc Fc = 353 Hz

Pour **F < 353 Hz** par la **période**

Pour **F >= 353 Hz** fréquence **directe**

Et précision partout meilleure que :

0.28 %



17. TRAVAIL SUR DES GRANDEURS PHYSIQUES

17.1. Utilisation des Convertisseur Analogiques Numériques

De nombreux microcontrôleurs ont des périphériques de ce genre internes.

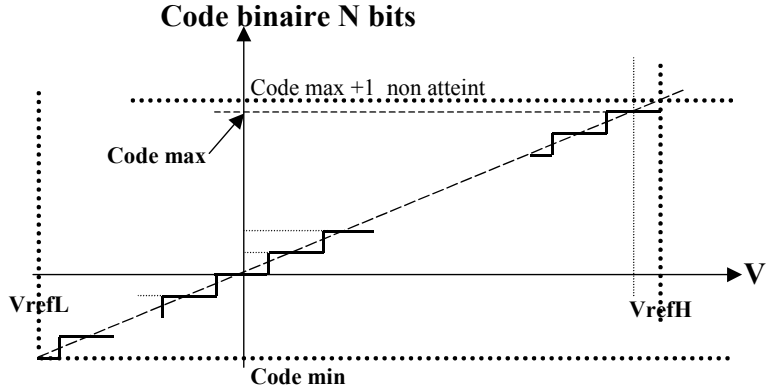
17.1.1. Rappel des caractéristiques de transfert

Généralement, après un réglage correct, aux tensions **VrefL et VrefH** (tension de références internes ou non) correspond les codes **codemin** et **codemax +1** (jamais atteint).

On supposera que ces références sont des source de tension de précision supérieure à la précision finale désirée pour ne pas avoir d'influence. Il serait facile de tenir compte par la suite de leur imprécision.

Si on recherche la précision maximale et le moins de bruit de mesure possible, VrefL pourra être la « masse Analogique » de la maquette (reliée à la masse numérique en un seul point au niveau de l'arrivée de l'alimentation. et VrefH proviendra d'un circuit de "référence de tension" et ne sera donc pas directement la tension Vcc d'alimentation du microcontrôleur.

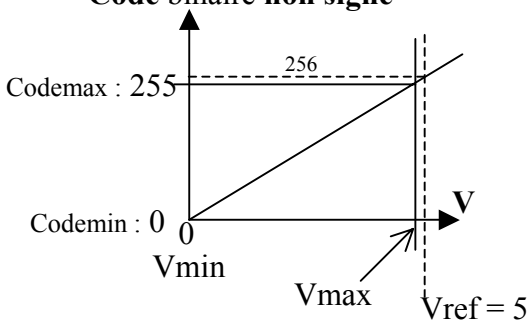
1) Caractéristique de base :



Le code binaire de N bits peut être signé ou non signé.

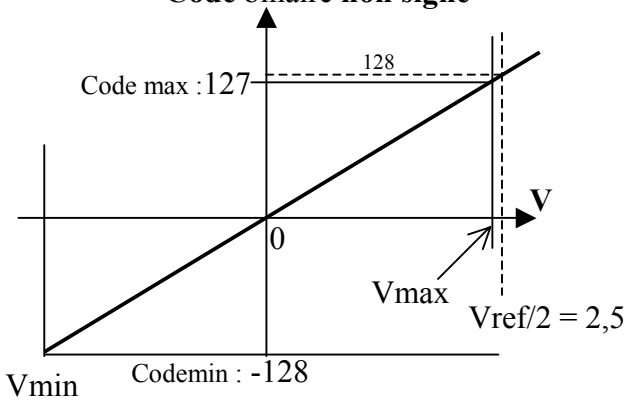
Les modes principaux sont les modes unipolaires et bipolaires, avec VrefL = 0.

2) Le mode unipolaire avec VrefL = 0 VrefH = Vref

	Entrée analogique	*Code Binaire **N bits non signé***
Général :	0 à VrefH	0 à 2^N-1
Exemple: 8 bits VrefL = 0 VrefH = Vref = 5v	Vmin = 0 Vmax ~ 5v $V = Vref * \frac{Code}{256}$	0 à 255 **Code** binaire **non signé** 

3) Le Mode Bipolaire avec VrefL= -Vref/2 VrefH = Vref/2

Un code binaire signé est évidemment plus pratique, la tension d'entrée et le code ont le toujours le même signe, et à une tension nulle correspond un code nul.

	Entrée analogique	Code Binaire **N bits signé complément à 2**
général	-Vref/2 à Vref/2	-2^{N-1} à $2^{N-1}-1$
Exemple: 8 bits Vref = 5 Vref/2 = 2,5 VrefL = -2,5v VrefH = 2,5v	Vmin = -2,5v Vmax ~ 2,5v $V=\frac{Vref}{2}*\frac{Code}{128}$	-128 à 127 **Code** binaire **non signé**  Vmin Vref/2 = -2,5

4) *Mode bipolaire pour des CAN fonctionnent toujours avec V > 0 ?*

De nombreux convertisseurs ne travaillent qu 'avec des tensions toujours positives en entrée.

On voit sur les valeurs précédentes qu'il est très facile tout de même de travailler en bipolaire par :

- **Décalage à l'entrée de Vref/2 = +2,5v** (Montage avec ampli opérationnel par exemple, réalisant Vs = Ve + 2,5v. Ainsi à ***Ve = -2,5v*** correspond à l'entrée du CAN une ***tension nulle***, et donc à ***Codemin***.
- **Choix du code binaire signé : *Codemin = -128***, Codemax = +127

5) Obtention d'un Code Signé pour des CAN toujours non signé ?

Si le CAN utilisé ne donne pas le code complément à deux, il est très simple de le réaliser, d'après la définition du complément à 2: "code de –x" = "complément à 2 de x" = "2^N-x", se reporter au chapitre arithmétique binaire et au code complément à 2.

Deux Méthodes au choix :

- Méthode N°1 : Inversion matérielle ou logicielle du bit de signe
- Méthode N°2 : Si Code > 2^{N-1} faire Code = Code – 2^N

Exemple sur 8 bits : si Code > 128 faire Code = Code - 256

17.1.2. Techniques d'utilisation

17.1.2.1. Echantillonnage et conversion demandés par logiciel

Au moyen d'une écriture dans un registre de contrôle (CAN interne), ou à une adresse déclenchant une impulsion sur un signal "Début de conversion" (CAN externe), on peut déclencher l'échantillonnage et conversion d'une voie, ou de plusieurs voies analogiques (séquence d'acquisition).

La fin de conversion est signalée par un drapeau fin de conversion (CAN interne), ou un signal (CAN externe).

Cette **Fin de conversion** :

- peut être **sondée** (par boucle d'attente).
- peut **déclencher en interruption** (interne, ou par signal externe).

Attention à la remise à zéro du drapeau ou la remise à l'état repos de la ligne externe !

17.1.2.2. Mode free running:

On peut programmer ce mode de conversion répétitives (CAN interne) ou câbler le signal fin de conversion sur l'entrée début de conversion. Le CAN échantillonne et convertit alors sans arrêt, à sa cadence propre. Une lecture permet d'accéder à la donnée.

Intérêt: Simplicité

Application : lecture d'une valeur de tension variant lentement (position de potentiomètre).

Précaution : une lecture erronée est possible si celle ci s'effectue juste au moment d'un changement d'état en sortie. On peut tester le signal ou le drapeau Fin de conversion avant de lire la donnée.

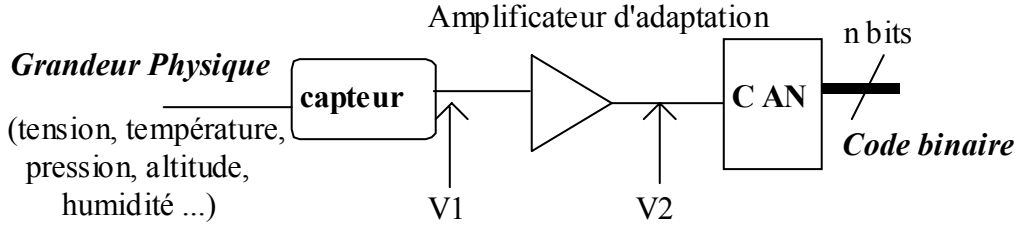
17.1.2.3. Echantillonnage de signaux à cadence régulière

Pour le traitement de signal : moyennes mobiles, filtres…, synthèse de fréquence …

Voir un peu plus loin dans ce même chapitre la petite partie traitement de signal.

17.2. Mesure d'une grandeur Physique

17.2.1. Chaîne de mesure

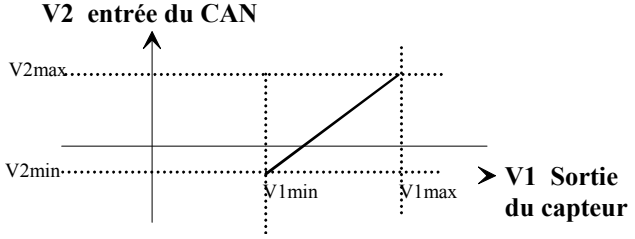


Selon le capteur utilisé, V1 peut varier de seulement quelques mv, et autour d'une tension moyenne non nulle.

Un **ampli d'adaptation** est donc indispensable pour faire travailler le convertisseur Analogique-numérique (CAN) dans la plage de tension prévue par le constructeur (par exemple 0 à 5v en « mode **monopolaire** », ou –2.5v à +2,5v en « **bipolaire** »).

Cette ampli réalise l'adaptation par :

V2 = A.V1 + B



En prenant les deux valeurs extrêmes (V1min,V2min et V1max,V2max), il est aisé de calculer la droite passant par les deux points et donc le Gain A et le Décalage B nécessaire. Pour cela, il suffit de résoudre le système d'équations

$$V2min = A.V1min + B$$
$$V2max = A.V2max + B$$

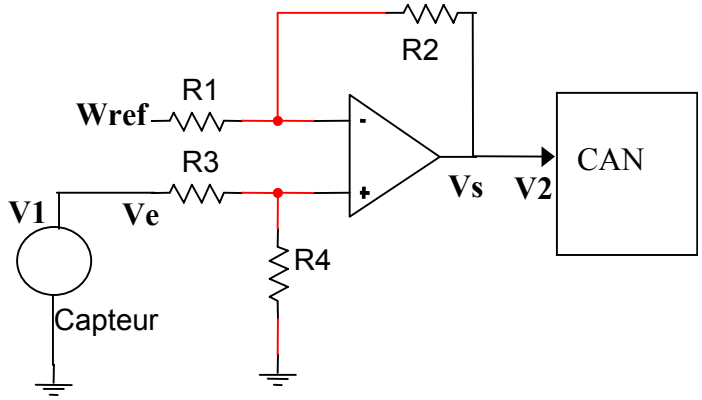
Un petit montage à amplis opérationnels (ou d'instrumentation pour plus de précision et moins de dérive) assure cette adaptation.

Donnons ici un montage de base simple pour une précision moyenne et un gain faible. Pour des gains plus importants (50, 100 ou plus), il est préférable d'utiliser plusieurs étages et des ampli d'instrumentations pour une meilleur précision (mode commun très faible), et des précautions pour éviter les bruits parasites. Si l'on doit amplifier seulement quelques millivolts en provenance du capteur, on peut prévoir des liaisons différentielles.

Montage de base simple pour réaliser :

V2 = A.V1 + B

La précision dépend aussi de celle des résistances ! Prendre des valeurs à 1%, ou plus précis si nécessaire.



Equation aisée à retrouver à partir des deux montages de base (inverseur et non inverseur), et du théorème de superposition :

$$Vs=Ve.\frac{R4}{R4+R3}.(1+\frac{R2}{R1})-Wref.\frac{R2}{R1}$$

On identifie avec : $Vs=A.Ve.+B.$

Le décalage est réalisé par une tension de référence Wref.

Attention : si celle ci est réalisée par un pont diviseur (deux autres résistances R et R') à partir d'une autre référence de tension. Sa résistance de Thévenin équivalente ($\frac{RR'}{R+R'}$) s'ajoutera alors à R1. Elle devra donc être faible par rapport à R1, ou sinon on devra en tenir compte dans le calcul.

Une résistance variable de réglage est souvent souhaitable (R1 sous forme d'une résistance fixe en série avec une variable).

Une fois que l'**ensemble** est réglé et **correctement réglé** ! (la précision de mesure peut être détruite par un mauvais réglage initial de la chaîne), et vu de l'extérieur, on a la relation générale :

Code Binaire = f(Grandeur physique)

Et très souvent une relation linéaire : **Grandeur = K.Code + K'**

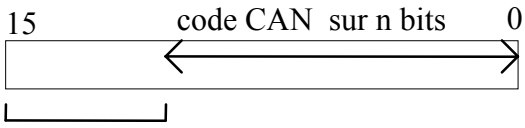
Une relation non linéaire peut évidemment se corriger en ajoutant un terme en $Code^2$ ou $Code^3$, ou par accès à une table.

On trouve souvent des cartes ou des circuits réalisant tout ou une partie de la chaîne de mesure. L'ampli d'adaptation peut être programmable pour choisir des gammes de mesure.

On suppose par la suite la chaîne correctement câblée et réglée. Attention aussi aux bruits divers, masses logiques et analogiques sont souvent nécessaires. Attention aux références de tension.

La première limitation de précision est évidemment le nombre de bits du convertisseur. Mais attention, ce nombre de bits doit être le **nombre de bits réel !** (précision et linéarité assurée à ±½ LSB) et non un nombre de bits « commercial ! ». Certains convertisseurs bon marché de par exemple 12 bits ont en effet une linéarité de ±1 LSB ou ±2 LSB ou même pire! Donc équivalant finalement à 10 bits seulement ou moins !

17.2.2. Choix du code : code entier

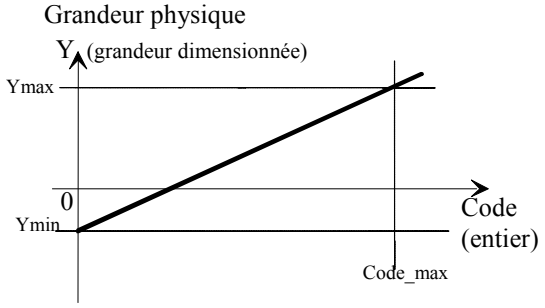
Nombre interprété en entier, cadré à droite si la taille du code est < taille donnée (8 ou 16 bits). ***Avantages :*** - Valeur binaire = entier - Simplicité ***Inconvénients :*** - Si on change le nombre de bits n du CNA, la valeur binaire change, donc le programme est à modifier	***Cadrage Droit***  Extension du signe (poids fort du code CNA) (si code signé)

17.2.3. Principe

La grandeur (**Y**) peut être **signée ou non**, *on peut choisir un code non signé* (un code signé n'est pas utile, le signe du code n'étant pas toujours celui de la grandeur physique).

On suppose une relation linéaire.

Pour coder cette grandeur physique **Y**, variant par exemple de **Ymin** à **Ymax** avec la meilleur précision possible, il faut un codage :



$$Y = K.Code + K'$$

On trouve K et K' en faisant correspondre le 0 du code avec Ymin, et le Code_max avec Ymax. Le nombre de bits nécessaire s'obtient en cherchant le nombre de pas minimum entre Ymin et Ymax.

17.2.4. Exemple

On veut coder une **distance Y variant de 1 à 5,5 m** , avec une précision absolue supérieure **à ± 5mm**.

17.2.4.1. Choix du nombre de bits du CNA

Le pas sur Y est de 10mm, il faut donc au minimum $\frac{5,5-1}{0,01}+1$ = 451 pas . Donc au minimum 9 bits. On peut conserver un code non signé, mais un code signé sur 10 bits peut servir pour des calculs futurs, avec le poids fort toujours à 0.

Pour un codage sur **10 bits** (circuits standards), le pas sera de $\frac{5,5-1}{1023}$ ≅ 4,39mm et la **précision maximale de ± 2,2mm**, donc mieux que prévu.

On a les relations: pour Ymin: 1= K.0 + K'
Ymax: 5,5= K.1023+ K'

D'ou K'=1m et **K= 4,5/1023** = 0,004398826979 m On conserve pour l'instant visiblement plus de chiffres significatifs que nécessaire !!.

Donc **Y (en mètres) = 0,004398826979.Code +1**

Autant garder l'écriture : $Y(en\ mètres) = \frac{4,5}{1023} code + 1$ et $K = 4,5/1023$ K' = 1

17.2.4.2. Précision suffisante de K (et K'). Nombre de bits pour les coder.

17.2.4.2.1. ETUDE GENERALE

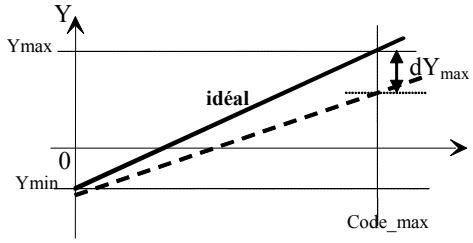
Si K et K' ne sont pas codés exactement, ils introduisent une erreur:

dY = K.dCode + Code.dK + dK'.

dCode = 0,5 (valeur arrondie à l'entier le plus proche, en effet, les bons circuits "CAN" ou convertisseurs analogique-numérique fournissent le code à 1/2 LSB près)

Cette erreur est maximale pour Code_max et vaut:

dYmax=K.O,5 + Code_max.dK + dK'



Le premier terme 0,5K = ½ LSB est la précision maximale possible théorique du convertisseur.

Les autres termes ne font que dégrader.

La dégradation est maximale pour le code max, normal, une erreur sur K étant une erreur de pente.

Pour assurer la précision maximale possible du CNA soit ±K/2 (moitié du pas), il faudrait avoir dK=dK'=0 !

Il est très aisé de faire K' entier, car K' est finalement la valeur minimale à mesurer. Par ce qui n'est pas possible pour K.

Avec une tolérance initiale dY légèrement plus grande, on va pouvoir limiter le nombre de bits nécessaires:

On choisit dK' (le plus souvent nul si K' est un entier), on trouve dKmax par:

$$dK_{\max} = \frac{dY\ \max - 0,5.K - dK'}{Code\_\max}$$

On en déduirait alors aisément le nombre de bits nécessaires pour coder K, en virgule fixe ou en virgule flottante. On ne peut pas trouver dK <0, ceci signifierait que l'on veut une précision supérieure à ce que peut donner le convertisseur !!.

On peut aussi prendre le problème à l'envers, et voir la dégradation apportée par une limitation du nombre de bits.

17.2.4.2.2. APPLICATION A L'EXEMPLE PRECEDENT

a) calcul de Y dans l'unité (ici le mètre) prévu au départ.

K'=1 donc dK'=0.

Pour avoir une précision sur Y de ±2,2mm, il faudrait mathématiquement coder K avec un nombre infini de bits!. Si on se borne à celle de départ soit dYmax = ±5mm, on en déduit:

$$dK_{\max} = \frac{0,005 - 0 - 0,5 * 0,00439453125}{1023}$$ donc ***dKmax ≈ 2,7.10^{-6}***

Il faut au moins **6 chiffres décimaux après la virgule** pour coder K.

Cette précision demandée semble bien élevée !

b) Calcul de Y dans une autre unité

On remarque dans cet exemple que les deux chiffres après la virgule sont des zéros, si on se servait de 100.K au lieu de K, on n'écrirait plus que 4 chiffres après la virgule !!.

Ceci revient à coder Y en cm , en effet: **Y (en cm) =** 4 500/1023 **. Code + 100**
K' = 100 entier à deux chiffres.

On se contente de dY=5mm = 0,5cm. Donc $dKmax \approx 2,7.10^{-4}$.

On peut donc par ce procédé limiter K à **4 chiffres après la virgule**.

Remarque importante: Il est en fait bien plus simple de calculer **Y en 128ème de mètres**, en codant 128.K au lieu de K, soit 4,5*128/1023 = Un décalage ensuite permet de redonner des mètres sans faire de division.

c) Codage de K en virgule flottante

On peut choisir d'exprimer Y en m ou en cm, cela ne change rien car

0,004398826979 = $0,4398826979.10^{-2}$ et $0,4398826979=0,4398826979.10^{0}$, le nombre de chiffres significatifs de la mantisse est le même.

On choisit la simple précision (Mantisse sur 23 bits + signe) qui permet une précision bien suffisante de **7 à 8 chiffres** décimaux sur la mantisse.

d) Codage de K en virgule fixe

Nécessaire surtout pour la vitesse de calcul ou pour réduire la taille de code sur des petits appareils à microcontrôleur.

- premier cas, **(Y en m)** pour coder K, il faudra n bits tel que $2^{-n} \le 2,7.10^{-6}$ donc $n \ge \frac{6*\log(10)-\log(2,7)}{\log(2)}$ = 18,5 soit **19 bits**. (ce qui est beaucoup, on cherche souvent à travailler avec 8 ou 16 bits maximum).

- second cas, (Y en cm), $2^{-n} \le 2,7.10^{-4}$ et $n \ge \frac{4*\log(10)-\log(2,7)}{\log(2)}$ = 11,8 soit **12 bits seulement**. Nettement mieux, on peut tout calculer en C en integer !

17.2.4.2.3. REMARQUE IMPORTANTE SUR K

Pour certaines valeurs particulières de k, on peut évidemment avoir une erreur nulle sur ce coefficient même avec très peu de bits (exemple 0,75 = 0,110). Mais ceci n'est valable que pour des cas particuliers et il faudrait réécrire tout le programme pour d'autre valeurs de k même assez voisines. Donc sauf exceptions, écrire toujours le logiciel pour l'erreur maximale possible sur k.

17.2.4.3. Exemple de calcul en C de la grandeur physique

On reprend l'acquisition précédente de la distance Y.

17.2.4.3.1. EN VIRGULE FIXE

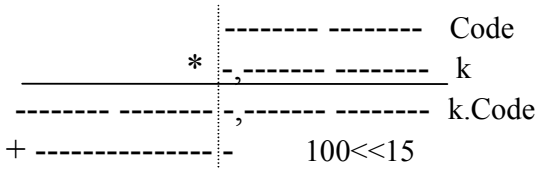
On a à calculer pour plus de précision:

Y (en m) =(100.K.Code + 100)/100

On peut travailler en signé même si tout est positif, le nombre de bits est suffisant pour k.
On désire un **résultat** dans le format toujours sympathique **Q16(32)**

```
Int k = (4.5 /1023.0)* 100.0*32768 ;  // K en Q15(16), mieux que 0,4398826979* 32768
Long Y1632;                           /*  Pour résultat en Q16(32) */
Y1632= (    ( (long)k * (long)code  + (100<<15 ) )  <<1    )  / 100;
```

Explications :



17.2.4.3.2. EN VIRGULE FLOTTANTE , CONVERSION Q16(32)

On peut calculer directement: **Y (en mètres) = (4,5/1023).Code +1**

Float K = 4.5/1023.0; // mieux que Float k = 0,004398826979
Long Y1632;
Y1632 = (long)((float)code*k + 1.0)*65536.0);
/* le produit par 65536.0 fait passer directement en en Q16(32) */

17.3. Traitement de signal, signaux de module < 1

On interprète le plus souvent le code comme nombre de **module < 1**

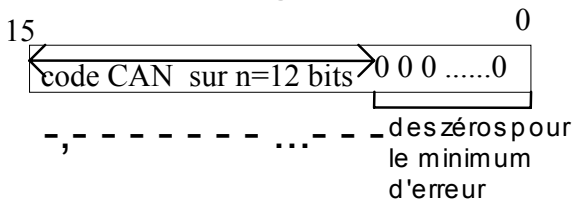
Cadrage gauche nécessaire.

Son format signé, sur 16 bits serait **Q15(16)**

Avantage :

- Programme indépendant du nombre de bits n du CNA, seule la précision de la valeur fournie change.
- Echantillon sans dimension
- Pas de débordement lors des élévations au carré (si $x < 1$, $x^2 < 1$)

Cadrage Gauche



On peut aussi l'interpréter en entier (de -32768 à un peu moins que 32767 selon le nombre de bits réels). Si on travaille en virgule flottante par la suite, ce n'est pas gênant.

17.3.1. Acquisition par CAN

A une tension V par exemple, telle que $-V_{max} < V < V_{max}$, on fait correspondre une valeur binaire x sans dimension telle que $-1 < x < 1$, de format Q15(16) par exemple.

On trouve le nombre de bits minimum du CAN tout aussi facilement à partir de la précision sur la grandeur physique de départ.

17.3.2. Restitution par CNA

Après traitement, si on désire retourner une tension par un CNA. Celui-ci sera également câblé sur la partie **gauche** du bus données. La valeur calculée devra alors toujours être au maximum comprise entre -1 et +1 (dynamique pratique).

Si la valeur calculée et envoyée au CAN dépasse même très légèrement cette dynamique de $< \pm 1$, il s'agit d'un débordement et l'erreur est maximale. Le CNA n'écrêtera évidemment pas, il faut effectuer un écrêtage éventuel par le calcul !, avant d'envoyer la valeur au CNA.

Le mieux est de se donner une petite marge de dépassement possibles avec des signaux d'entrée et de sortie par exemple d'amplitude limité à 80% de l'amplitude maximale.

17.3.3. Cas d'acquisition-restitution sur seulement 8 bits.

On raisonne soit en entier Q0(8) (de -128 à 127) ou en Q7(8) (de -1 à presque 1).

Le code à ce niveau est évidemment le même de **80** à **7F** (écriture hexadécimale)

17.3.4. Entrée-sortie de valeurs flottantes sur CNA et CAN avec cadrage gauche

♦ ***Lecture CAN*** Soit ***code*** le code entier lu sur 16 bits.
On veut passer de ce code en Q15(16) à un code flottant **x de module < 1**;

Int code; /* pour code entier lu au CAN */
Float x; /* pour récupérer en **flottant de module < 1** */

x=code/32768;

♦ ***Sortie sur CNA***

Pour dynamique complète, il faut un entier entre -32767 et +32767
On ramène un flottant **y** variant de -ymax à ymax (bornes comprises) à la dynamique totale du CNA, et on l'envoi par : **CNA = (y/ymax)*32767.0;** /* CNA adresse du CNA */
On voit bien que tous ces calculs ne dépendent pas de la taille du Convertisseur, mais seulement de la taille des codes manipulés (ici 16 bits).

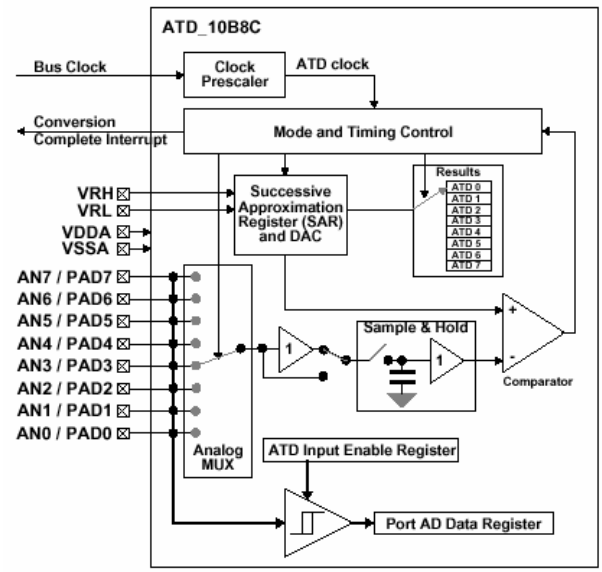
17.4. Mise en œuvre du CAN du 68HC12

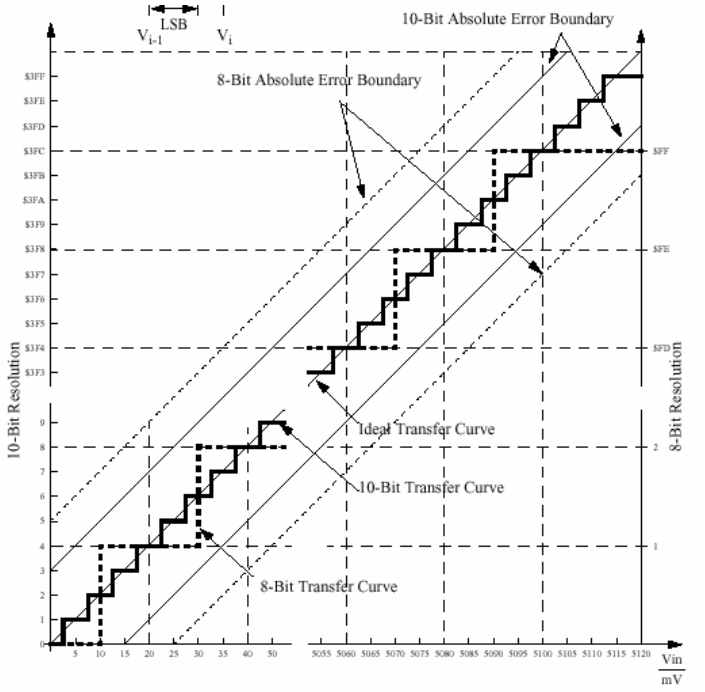
17.4.1. Fonctionnement résumé et registres de programmation

On parle aussi de **port ATD** (Analogique to Digital), car on peut confondre désormais avec les bus série : CAN !
Deux Convertisseur AN **8** ou **10** bits, de 8 voies, **ATD0 et ATD1**
Attention : l'entrée analogue n'est pas à haute impédance, résistance max de source conseillée : 1kΩ
Fck bit : FATDCKmax = 2MHz et FATDCKmin = Fckbus/2
Tension d'entrée max : 5,2 V entre **0 et 5,2V**
Temps de conversion minimum sur 10 bits : Tconv10min = 7µs
Mais si **Fckbus = 2MHz** (Fck par défaut) : FATDCKmax = Fckbus/2 = 1MHz et donc Fckbit = 1MHz et **Tconv10min = 14µs**

Schéma de principe:

Ce module possède diverses fonctions : trigger sur signal extérieur (échantillonnage), conversions multi canaux, registre tampon en sortie …. et 30 registres internes, et donc un assez grand nombre de bits actifs !
Nous n'étudions le convertisseur 8 voies ATD0, l'autre étant identique aux adresses de ses registres près.
Il fonctionne en **séquence se conversion** : une séquence = **1 à 8** conversions.





Fonction de transfert :

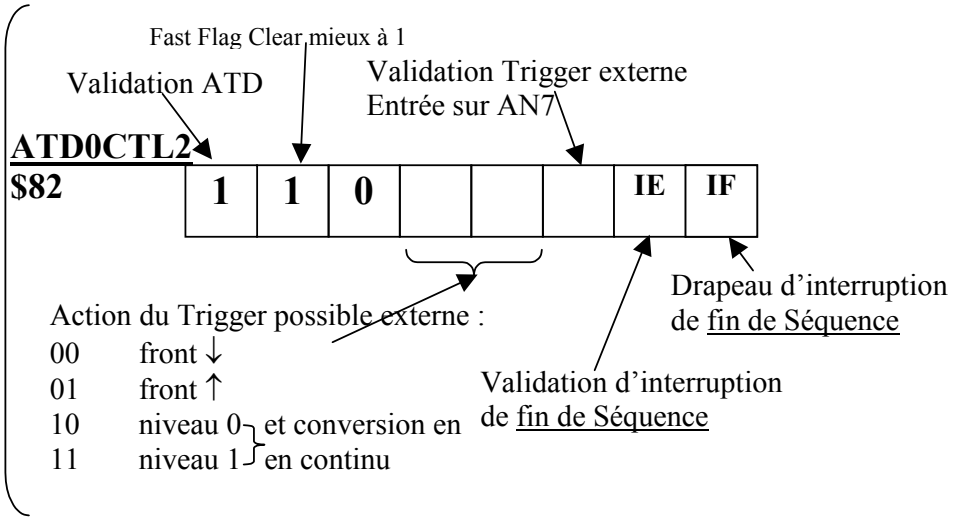
$$Code = \frac{Vin}{VrH - VrL} . 2^N$$

Vin toujours **> 0**

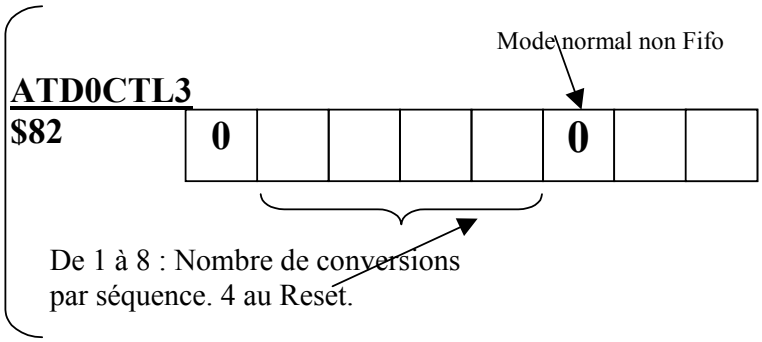
On choisira ici :

VrL = 0v
VrH = 5v

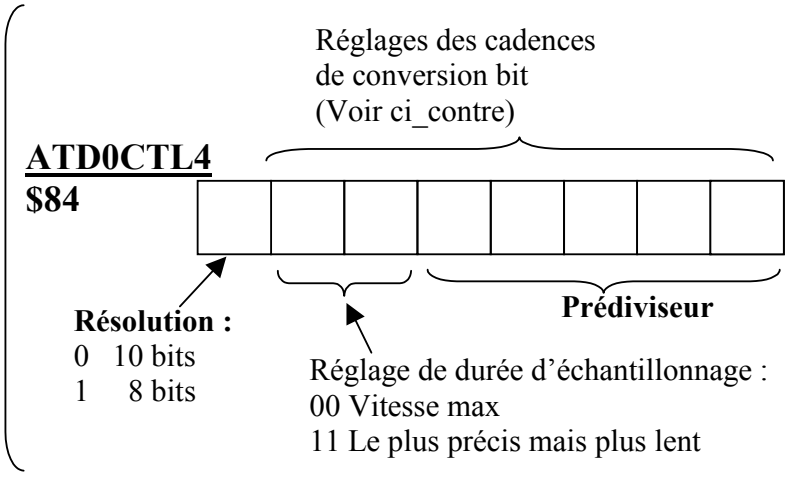
Remarque : Lors de la première initialisation des différents bits, on peut directement écrire 0 dans les bits non utilisés (soit c'est la valeur par défaut, soit écrire 0 est sans effet).



Registre Controle 2



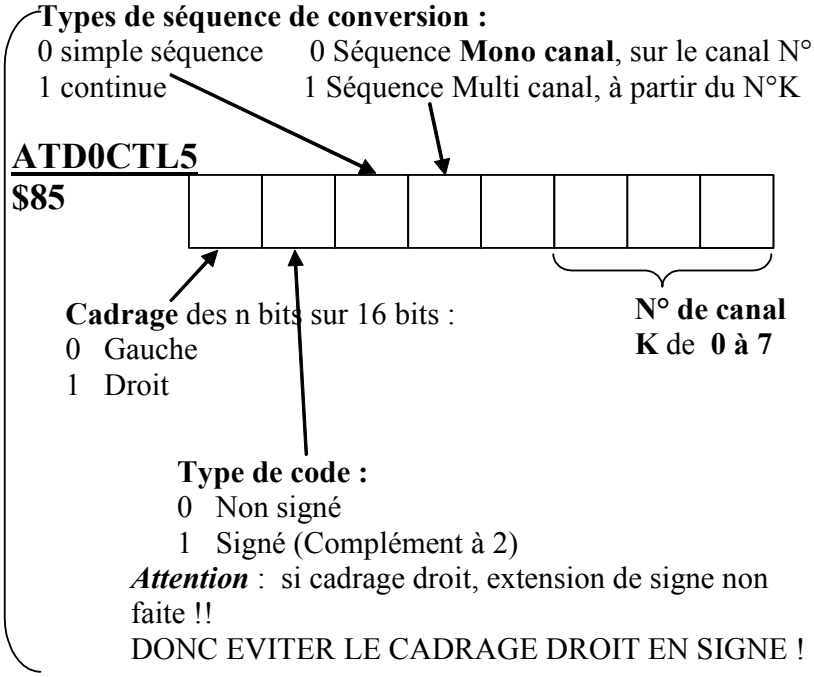
Registre Controle 3



Registre Controle 4

Prédiviseur pour Vitesse max:

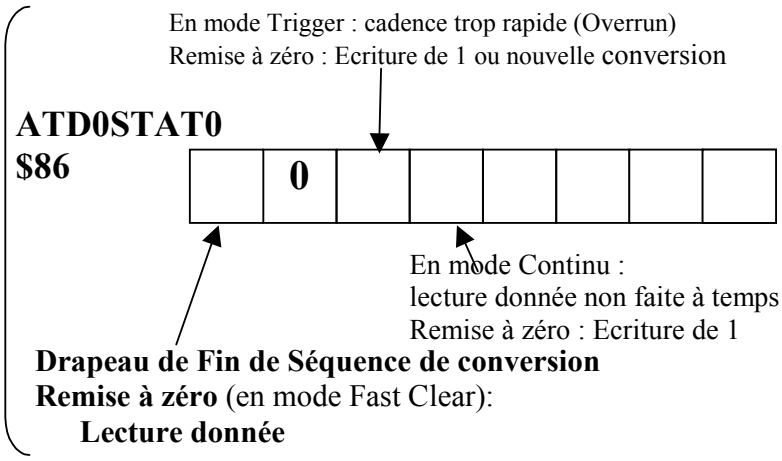
Fbus(MHz)	Prédiviseur
2	**0**
4	0
8	1
12	2
16	3
20	4
24	**5**



Registre Controle 5

→

Une écriture dans ce registre démarre une séquence de conversion



Registre Status 0

ATD0STAT1
$8…

F7	F6						F0

Registre Status 1

Drapeau individuel de fin de conversion (utile seulement si séquence de plus de 1)
->Remise à zéro (en mode Fast Clear)**:**
Lecture donnée

ATD0DRiH **ATD0DRiL**
$90 + 2i **$91 + 2i**

16 **Registres Donnée 8 bits :**
Exemple port 2:
Adresses : **$94,$95**

En mode 10 bits, On peut lire directement les 16 bits à l'adresse ATDDRiH.
En mode 8 bits :
Si **cadrage gauche**, on récupère l'octet dans **ATDDRiH**
Si cadrage droit, on récupère l'octet dans **ATDDRiL**

Différents codes obtenus :

Input Signal Vrl = 0 Volts Vrh = 5.12 Volts	Signed 8-Bit Codes	Unsigned 8-Bit Codes	Signed 10-Bit Codes	Unsigned 10-Bit Codes
5.120 Volts	7F	FF	7FC0	FFC0
5.100	7F	FF	7F00	FF00
5.080	7E	FE	7E00	FE00
2.580	01	81	0100	8100
2.560	00	80	0000	8000
2.540	FF	7F	FF00	7F00
0.020	81	01	8100	0100
0.000	80	00	8000	0000

17.4.2. Mise en œuvre simple

17.4.2.1. Résumé

On prépare tout d'abord le convertisseur en programmant ses registres contrôle : ATD0CTL2, ATD0CTL3 et ATD0CTL4.

On démarre une séquence de conversion en écrivant dans ATD0CTL5.

On attend la fin de conversion en testant le **drapeau fin de conversion** (dans ATD0STAT0).

Ce drapeau se remet tout seul à zéro en lisant une donnée (Mode Fast Clear choisi dans ATD0CTL2).

Organigramme de demande de conversion et de lecture d'une voie :
(Mode Sondage, Pulling)

17.4.2.2. Exemple : Acquisition en 8bits signés sur la voie N°0

La ligne analogique d'entrée est **AN0. Tension de 0 à 5v**
On doit demander la séquence de conversion : **canal 0**, **simple séquence**, **mono canal.**

- **Initialisation :**

```
// ini du ATD0 sans trigger 8 bits signés
ATD0CTL2 = 0xc0;  // 110 00 0 00   sans trigger, pas d'interruption fin de séquence, fast flag clear
ATD0CTL3 = 0x08;  //  0 0001 000   1 conversion par séquence, 8 bits.
ATD0CTL4 = 0x80; // 1 00 00000  8 bits, le plus rapide possible pour BUS 2MHz
//demande conversion plus loin: ATD0CTL5 = 0x40;  // 0 1 0 0 0 0 0 0  cadrage sans objet car 8 bits, signé, séquence mono canal à partir du canal 0.
```

- ***Exemple de fonction C d'acquisition :***

Elle retourne un entier de 0 à 255, on peut passer ensuite à la grandeur physique comme nous venons de le voir.

```
char lecture_can_voie0(void)
{           ATD0CTL5 = 0x40;  //  0 1 0 0 0 0 0 0  demande d'acquisition conversion
                                  // cadrage sans objet car 8 bits
                                //  signé, séquence mono canal à partir du canal 0.
            while((ATD0STAT0 & 0x80)==0);  // attente fin de conversion
     return( (char)ATD0DR0H );          // retour donnée
}
```

17.5. Petits traitements de signal sur microcontrôleur, échantillonnage d'un signal

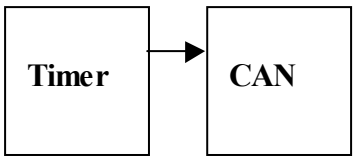
On sait que à priori pour un signal de fréquence Fmax, on doit échantillonner à une fréquence **Fech >= 2Fmax**. Des cas particuliers de sous échantillonnages sont possibles mais ce domaine ne fait pas l'objet de ce cours.

Le **CAN** travaillera évidemment en **mode Bipolaire**, et on utilisera un **code binaire signé** pour les échantillons.

17.5.1. Programmation d'une cadence d'échantillonnage

17.5.1.1. Principe

En traitement de signal, on doit échantillonner à une cadence régulière, se synchroniser sur cette cadence, et effectuer tous les calculs entre deux échantillons. Il faut donc disposer du maximum de temps de calcul.



La **cadence d'échantillonnage Fech** sera évidemment créée par **Timer**.
Sur l'interruption Timer :

- On réarme (si nécessaire) celui ci pour une durée identique.
- On remet son drapeau à 0.
- On effectue une demande d'acquisition conversion.

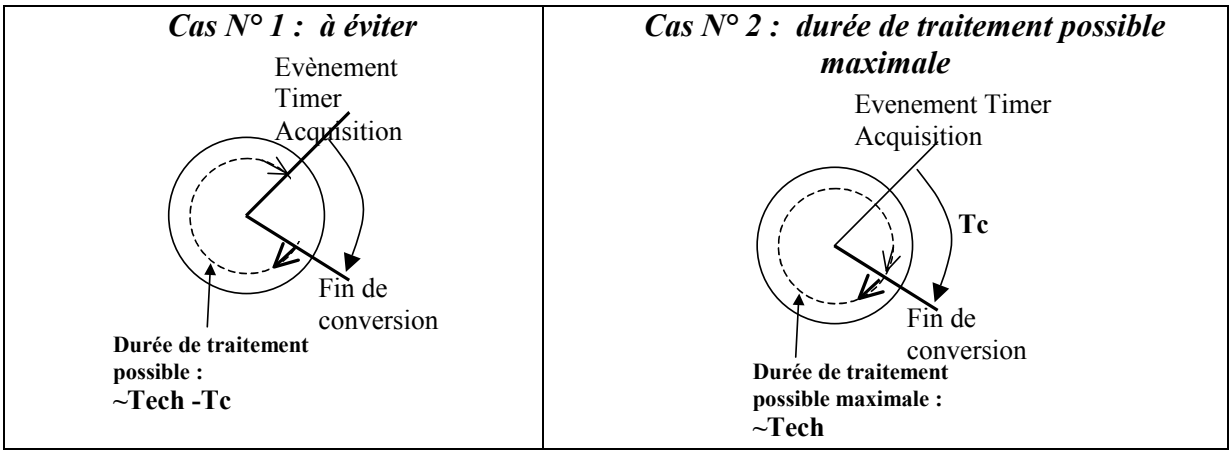
Cette demande peut se faire :

- Par des instructions placées dans l'interruption Timer.
- Ou mieux automatiquement si on programme le Timer pour générer un signal de fréquence Fech que l'on câblera sur une entrée nommé **Trigger du CAN** interne, ou sur le signal Début d'acquisition et conversion (CAN externe).

17.5.1.2. Condition de « temps réel »

Pour assurer la fréquence d'échantillonnage souhaitée, il faut que la boucle complète d'acquisition d'un échantillon (et traitement éventuel) dure moins d'une période Tech.
Sinon, la cadence réelle obtenue est inférieure.

Un petit dessin montre qu'il ne faut pas se synchroniser sur le Timer (drapeau ou interruption), mais sur la fin de conversion, sous peine d'avoir moins de temps de traitement entre deux échantillons. Avec le cas N° 2, même si la conversion dure un certain temps, le temps de traitement pour le temps réel reste pratiquement égal à Tech = 1/Fech.



Le programme complet pourra alors se **synchroniser sur la fin de conversion** (Par **boucle d'attente** du drapeau, **ou par interruption**).

17.5.1.3. Exemple simple en HC12: Échantillonnage d'un signal

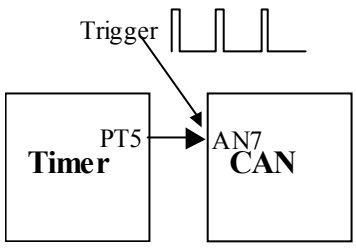
Se reporter au chapitre sur le Timer.

On désire Fech voisin de 20 kHz

On utilise le Timer et l'évènement Output Compare 5 pour générer un signal de fréquence **Fech** sur son PT5. PT5 est câblé sur l'entrée AN7 programmée en Trigger.

On choisit ici **F_BUS de l'HC12 = 24MHz**, fréquence qui permettra par la suite un traitement 12 fois plus rapide qu'avec la F_BUS de base de 2 MHz. (voir dans la partie matériel comment modifier Fech qui est par défaut de 2 MHz).

L'évènement Timer fera passer PT5 à 1. On remettra PT5 à 0 dans l'interruption Timer.



➢ ***Fonction d'initialisation, canal 1, 10 bits signés, avec F_BUS déjà à 24 MHz***

On remarquera le cadrage gauche de la donnée 10 bits sur un mot de 16 bits, plus pratique en traitement de signal (pour un raisonnement éventuel en Q15(16), nombres de module < 1).

```
void ini_can_ech(void)
{
ATD0CTL2 = 0xcc;  //  110 01 1 00   trigger montant, pas d'interruption fin de séquence, fast flag clear
ATD0CTL3 = 0x08;        //  0 0001 000   1 conversion par séquence
ATD0CTL4 = 0x05;        //  0 00 00101   10 bits, le plus rapide possible Pour BUS 24MHz
ATD0CTL5 = 0x40;  //  0 1 0 0 0 0 0 1  cadrage gauche signé canal 1  mono canal
// ini du timer et TC5 à Fech   pour F_BUS = 24 MHz
TSCR1 = 0x80;             // validation Timer
TSCR2 = 1;                // Prédiviseur de 2
```

```
TIOS = 0x20;                // TC5  en Output Compare
TCTL1 = 0x04;               // Mode toggle de PT5 cablé sur Trigger CAN (AN7)
TIE = 0x20;                 // interruption Timer TC5
delta_fech = 635;  // correct pour Fbus = 24,      Fech = 18897,63779
TC5 = TCNT + delta_fech;
}
```

- ***Fonction d'interruption du Timer et TC5***

```
interrupt[0xFFE4-0xFF80] void  Tc5_pour_Fech(void)
{
TFLG1 = 0x20;
TC5 = TC5 + delta_fech;          // On réarme pour Tech
}
```

- ***Fonction C: int attente_échantillon(void)***

Exemple de fonction C en assembleur, incorporable à un traitement de signal tout en assembleur pour plus de rapidité. On pourrait écrire aussi simplement la fonction en C !

```
    .public attente_echantillon
attente_echantillon:
wait        ldaa ATD0STAT0    attente fin conversion
            bpl wait
            ldd ATD0DR0       lecture donnée 16 bits et raz drapeau (mode fast)
            movb #1,CFORC     retombée du trigger, évènement OC1 forcé
            rts
```

- ***Exemple de programme principal se synchronisant sut Fech pour traitement***

Traitement en Virgule fixe

```
Int x ;    // Q15(16)    module < 1
While(1)
  {
  x = attente_echantillon() ;  // Q15(16)
…… //  Traitement (par filtrage numérique)
}
```

Traitement en flottant (si temps … !)
Et échantillon ramené entre –1.. +1

```
Float x ;       // module < 1
   While(1)
     {
     x = attente_echantillon()/32768.0 ;   // float
…… //  Traitement (par filtrage numérique)
        }
```

En flottant, si on désire, ce qui peut être plus pratique pour des études de dynamique par exemple, travailler sur des échantillons réellement entre –1 et +1, on doit diviser par 32768. L'acquisition renvoi en effet un nombre en Q15(16) (cadrage gauche) donc en réalité des entiers 32768 fois plus grands.

17.6. Exemple de petits traitements de signal : valeur efficace d'un signal, sur HC12

On décide ici de travailler seulement sur des **échantillons de 8 bits**, et avec la **F_BUS** par défaut de **2MHz**. On choisit par contre pour le Timer un prédiviseur de 2 seulement.

Si on se contente d'une résolution de 8 bits, travailler sur des échantillons de 8 bits permet aussi quelques raisonnements simple en entiers sans soucis de débordements, en effet 8 bits par 8 bits donnent 16 bits et donc un format encore manipulable facilement et rapidement en C, et sur l'HC12 qui est un microcontrôleur presque 16 bits.

On désire une **Fréquence d'échantillonnage Fech de 10 kHz.**

17.6.1. Fonctions de base :

- ***Fonction d'initialisation, canal 1, 8 bits signés, avec F_BUS = 2MHz et prédiviseur de 2***

```
// Fcad Timer = 1MHz.
// Cadence Fech définie par : Fech = 1000000/delta_fech
// Exemple :          Si delta_fech = 1000, Fech = 10 kHz
void ini_can_ech(void)   // BUS 2 MHZ à Fech = ? 8 bits Signés SAns interruption Fin sequence
{
ATD0CTL2 = 0xcc;  //  110 01 1 00   trigger montant, pas d'interruption fin de séquence, mode fast flag clear
ATD0CTL3 = 0x08;  //   0 0001 000   1 conversion par séquence
ATD0CTL4 = 0x80; //  1 00 00000    8 bits, le plus rapide possible pour  BUS 2MHz
ATD0CTL5 = 0x41;  //  0 1 0 0 0 0 0 1   signé  canal 1  mono canal
// ini du timer5 à Fech = ?
TSCR1 = 0x80;     // validation Timer
TSCR2 = 1;                // Prédiviseur de 2
TIOS = TIOS | 0x20;       // Timer 5 en Output Compare  - - - - 0 1 - -
TCTL1 = TCTL1 | 0x04;  // toggle de PT5 cablé sur Trigger CAN (AN7)
TIE = TIE | 0x20;
delta_fech = 100;                 //  Fech = 10000 Hz
TC0 = TCNT + delta_fech;
}

interrupt[0xFFE4-0xFF80] void timer5_pour_Fech(void)
{
TFLG1 = 0x20;  // Raz drapeau IT TC5
TC5 = TC5 + delta_fech;
}
```

- ***Fonction d'attente et de récupération d'un échantillon 8 bits***

```
char attente_echantillon(void)
{
while( (ATD0STAT0 & 0x80)==0);
CFORC = 0x20;             // Pour Freq de PT5 = Freq cadence Timer TC5, sinon 1/2 !
return (ATD0DR0H);
}
```

17.6.2. Puissance d'un signal, par block

$$p=\frac{1}{N}\sum_0^{N-1} x_n^2 \quad \text{et} \quad Veff=\sqrt{\frac{1}{N}\sum_0^{N-1} x_n^2}$$

On fait l'acquisition de N valeurs du signal. On calcule ensuite la puissance moyenne et la valeur efficace. Plus la fréquence du fondamental est basse, plus N devra être élevé. Mais attention ici, pour un signal x(t) à la fréquence F, la puissance instantanée $x(t)^2$ sera à la fréquence 2F, en tenir compte pour le choix de Fech.

On peut raisonner en entier : si les échantillons sont sur 8 bits, la puissance est sur 16, la valeur efficace redevient sur 8 bits.

On peut remarquer la génération pratique sur PT2 d'un signal au niveau haut durant le temps de calcul (lecture de l'échantillon non comprise) pour vérifier si celui ci permet l'échantillonnage à la cadence prévue. En réalité la durée de boucle sera un peu supérieure car on doit tenir compte aussi de la fonction attente_echantillon() et de son temps d'exécution la donné étant juste présente.

```
Unsigned int N = 500 ;
void main(void)
{
unsigned long k; long p; float veff;
long sn2;int xn;
ini_can_ech();              // fonction précédente

PTT          &= 0xFF-0x04;        //à 0 :   PT2 pour test duree calcul
DDRT |= 0x04;               //---- --1--
valid_it();
// Début de la boucle d'acquisition de N valeurs, avec calcul de la somme des carrés
while(1)
 {
  sn2 = 0;
  for(k =0 ; k < N; k++ )      // Moyenne sur  N valeurs
     {
      PTT &= 0xFF-0x04;    // PT2 à 0  fin de calcul
      xn = attente_echantillon();
      PTT |= 0x04;         // PT2 à 1 début de calcul
      sn2 = sn2 + xn*xn;
   }
// Lecture de la puissance, et  calcul  Veff ensuite
   p = sn2 ;
   veff = sqrt(  (float)sn2/N );
   // On peut afficher ici Veff après chaque mesure
 }
}
```

→ On vérifierait que l'acquisition s'effectue bien à 10kHz sans problème. **(F_BUS = 2MHz).**

17.6.3. Puissance d'un signal par filtre moyenneur simple

On réalise ici un vrai filtre passe bas sur les échantillons élevés au carré, et on peut disposer ainsi à tout moment (plus loin dans la boucle, ou sur interruption) de la puissance, et donc par la suite si nécessaire de la valeur efficace du signal.

Remarque : Eviter de prendre la racine carrée dans la boucle, ce qui augmenterait vraiment trop le temps de calcul. En effet, on peut travailler tout le temps sur la puissance, et passer à la valeur efficace que lors d'un affichage.

17.6.3.1. Principe du moyenneur récursif

On montrerait en traitement du signal que l'on réalise un très bon filtre passe bas, et donc un filtre moyenneur par l'expression récursive suivante :

Yn = (1+a)*Xn -a*Yn_1; avec a = -0.99 ou a = -0.999

Plus a est voisin de –1 plus le passe bas est bon, mais évidemment, en contre partie, plus la constante de temps d'établissement est grande. Les valeurs indiquées sont des valeurs très classiques. Se reporter à des cours de traitement du signal pour plus d'information.

17.6.3.2. calculs en C, virgule flottante

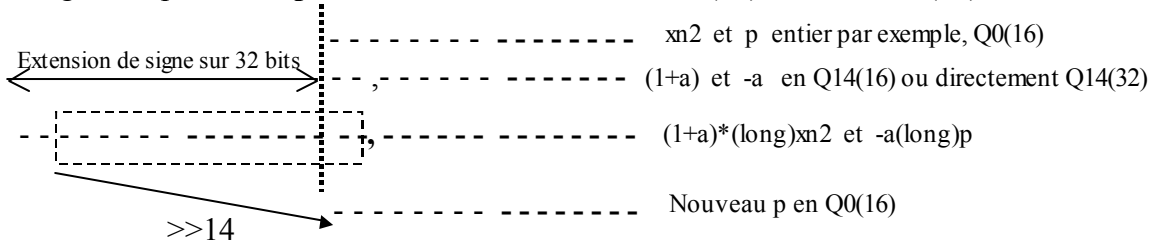
```
int xn; float a = -0.99 ; float xn2;
float p ;  // puissance moyenne du signal
…..
// Boucle de travail
while(1)
  {
PTT &= 0xFF-0x04;            //  PT2 niveau Haut durant traitement
xn = attente_echantillon();
PTT |= 0x04;
x2 = xn*xn;                                  // Carré
p = (1+a)*x2 -a*p;          // Moyenneur sur x2 en flottant ,  puissance du signal en flottant
}
```

→ On mesurerait ici **(sur PT2) un temps de calcul de 1,1ms ! (F_BUS = 2MHz).**
Donc : Fechmax théorique : 909 Hz seulement !
En pratique : environ 900Hz, presque identique.

17.6.3.3. Calculs en C, virgule fixe

Principe :

On peut raisonner avec les échantillons xn en entiers 8 bits Q0(8) aussi bien qu'en Q7(8).
On filtre les carrés, donc des xn2 sur 16 bits (entiers Q0(16) ou bien Q14(16)).
Soit 1+a et a en Q14(16), dynamique –2 à +2, ou directement Q14(32).
Le signal de puissance **p** filtré est sur 16 bits entiers Q0(16) ou bien Q14(16).



Programme :

```
const long  unplusa =       (1-0.99)*16384;     // pour  a = -0.99
const long  moinsa =        0.99*16384;
int xn,xn2 ;
Int p ;   // puissance moyenne en temps reél
……
while(1)
  {
PTT &= 0xFF-0x04;             //  Sur PT2 niveau Haut durant traitement
xn = attente_echantillon();
PTT |= 0x04;
xn2 = xn*xn;    //  OK  16 bits par 16 bits donnent 16 bits, suffisent car xn 8 bits
p  = ( unplusa*(long)xn2 + moinsa*(long)p ) /16384 ;
   }
```

On pourrait remarquer sur le code généré, que ce microcontrôleur effectue une division par 16384 bien plus rapidement qu'un décalage à droite de 14. On évitera donc l'écriture >>14 (qui nécessiterait en HC12 une boucle de 14 fois un décalage droit sur 32 bits !).

→ On mesurerait ici **(sur PT2) un temps de calcul de 220 µs (F_BUS = 2MHz).**
Donc : Fechmax théorique : 4,545 Khz
En pratique : 4,5 KHz.

17.6.3.4. Avec fausse fonction en assembleur, et instruction spécifique HC12

On doit pouvoir faire mieux en programmant le moyenneur en assembleur !

Comme le filtre ne comprend pas de boucle de calcul, pour optimiser au maximum on peut choisir une fausse fonction, **sans paramètres**, de prototype **void moyenneur(void)**; travaillant sur les **variables globales de 16 bits: entrée xx**, **sortie yy** de même format.

On gagnera ainsi sur l'envoi et le retour d'opérandes. Mais attention, la fonction réalisée sera alors figée et devra toujours travailler sur ces variables globales, de même nom.

Instructions assembleur spécifiques utilisées :

EMACS m

Produit signé de l'opérande 16 bits (pointé par X) par l'opérande 16 bits pointé par Y.
Résultat 32 bits en m, m+1,m+2,m+3.
(Sans auto incrémentation des pointeurs, dommage !).

EDIVS

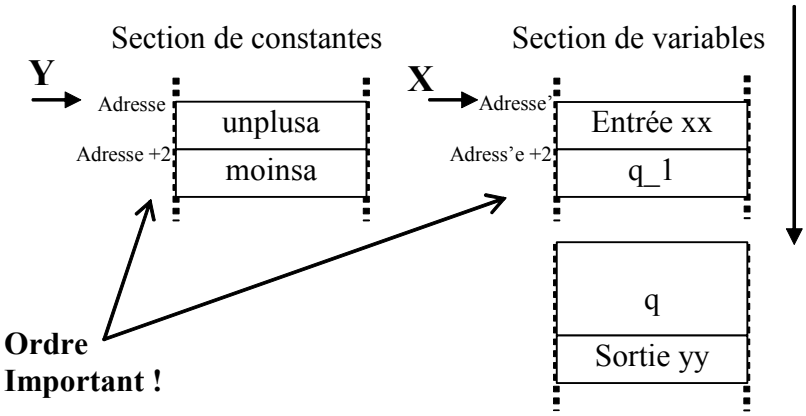
Division signé de l'opérande 32 bits en Y D par 16 bits en X.
Résultat 16 bits en Y.

Disposition des variables :

Pour profiter de l'instruction EMACS, on devra disposer en mémoire les opérandes comme suit :

Variables globales du filtre :
Entrée 16 bits **xx**
Sortie 16 bits **yy**

Variables statiques de travail :
q 32 bits (en fait la sortie yy sur 32 bits)
q_1 16 bits (la sortie yy à l'instant précédent)



Constantes 16 bits :
Unplusa et **moinsa**

Important : Certaines de ces constantes et variables devant être ordonnées, on doit forcément les déclarer en assembleur. Nous pouvons donc finalement les déclarer toutes en assembleur.

Sous programme de la fonction void moyenneur(void) écrite en assembleur :

```
            PUBLIC  moyenneur
            PUBLIC  xx,yy       // xx  Qi(16) quelconque   yy 16bits même Qi(16)
        RSEG CONST
unplusa     fdb   164    // 1 +(-0.99) en Q14(16)  Ordre important pour unplusa et moinsa
moinsas             fdb  16220   // -(-0.99) en Q14(16)    Et sur 16 bits ici forcément.

            RSEG UDATA1
xx          rmb 2     entrée 16 bits.  Ordre important pour  xx  et q_1
q_1                 rmb 2     16 bits

q                   rmb 4     32 bits variable de travail statique du filtre
yy          rmb 2     sortie 16 bits
        RSEG  CODE
moyenneur:
  pshx
  pshy
*---------------------- filtre récursif premier ordre  y = (1+a)x -ay    a en q14(16) -2 à +2
  ldx #xx           X sur xx
  ldy #unplusa              Y sur unplusa
  movw #0,q
  movw #0,q+2
  emacs q
  leax 2,x          Incrémentation de X de 2, X pointe q_1
  leay 2,y          Incrémentation de Y de 2, Y pointe moinsa
  emacs q
* >> 14             Décalage de 14 à droite par division (sinon boucle de 14 décalages !)
  ldy q
  ldd q+2
  ldx #16384
  edivs
  sty q+2           Résultat nouveau (16bits) en q+2,q+3  en Qi(16)
```

```
  movw q+2,q_1    glissement : nouveau résultat → ancien

* resultat dans yy
  movw q+2,yy      // en Qi(16)
  puly
  pulx
  rts
  END
```

Fonction C d'utilisation :

```
  extern int xx,yy;
  extern void moyenneur(void);  // Entrée xx  Sortie yy
void main(void)
{
int xn;
…….
while(1)
  {
PTT &= 0xFF-0x04;                 // Sur PT2 niveau Haut durant traitement
xn = attente_echantillon();
PTT |= 0x04;                      //****** Temps de traitement 56 micro secondes (F_BUS 2MHz)
xx = xn*xn;        //  Fechmax theorique 17800 Khz mais pratique environ 11Khz car 15 à 20 micros perdues dans IT timer 5
moyenneur();   // yy = (1+a)*xx -a*yy;  // ICI EN VIRGULE FIXE ET ASSEMBLEUR
}
// La puissance moyenne est en permanence dans yy
```

→ On mesurerait ici **(sur PT2) un temps de calcul de 56 µs (F_BUS = 2MHz).**
Donc : Fechmax théorique : 17,8 KHz
En pratique, seulement 11kHz

On peut se demander ici pourquoi la fréquence maximale pratique est bien inférieure à la théorie. On atteint ici les limites de l'HC12 à 2 MHz, donc les temps négligés ont d'avantage d'importance que précédemment.

Deux raisons :

-On a négligé le temps d'acquisition de l'échantillon xn dès l'interruption fin de conversion : test positif du drapeau, lecture xn er retour de sous programme.

-On a négligé le temps d'exécution de l'interruption TC5.

18. PORT PWM PULSE WIDE MODULATION

18.1. Principe : un pseudo CNA ?

Une sortie PWM permet de commander des éléments analogiques sans utiliser de vrais Convertisseur Analogique Numérique (D'ailleurs, l'HC12 n'en possède pas).

➢ ***Signal PWM***

Soit $\mathbf{F_{pwm} = 1/T}$

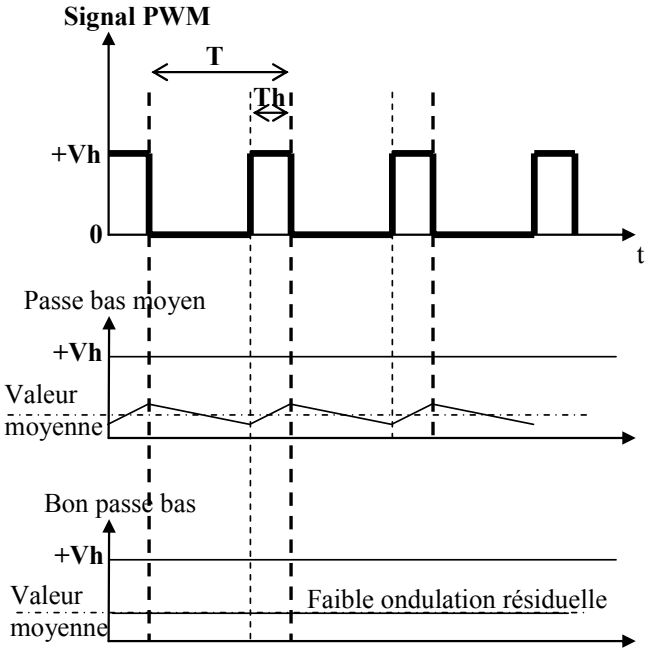
Th durée au niveau haut.

La valeur moyenne de ce signal est proportionnelle à son rapport cyclique, et varie ici en pratique de presque 0 à presque Vh (souvent 5 volt), en gardant un petit résiduel).

$$PWM_{moy} = Vh.\frac{Th}{T}$$

si $0 < Th < T$: $0 < PWM_{moy} < Vh$

Le passe bas est d'autant plus aisé à réaliser (nécessitant une constante de temps plus petite) que F_{pwm} est élevé (50kHz, 300 kHz ou plus).



Le port PWM est donc en fait un modulateur PWM.

On initialise le port en fixant tout d'abord une fréquence $\mathbf{F_{pwm}}$.

D'un code **N** (entre 0 et N_{max}) dépend la durée **Th.**

On peut alors modifier à tout instant Th, et donc $PWM_{moy} = Vh.\frac{N}{N_{max}}$ on réalise donc une **sorte de conversion analogique** numérique si on s'intéresse à cette valeur moyenne.

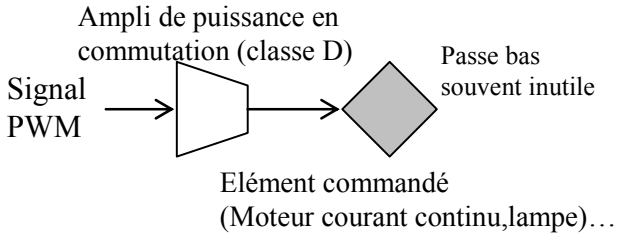
Le passe bas pourra être électronique (simple RC ou filtre amélioré), mais être aussi souvent directement l'élément physique commandé, ou parfois un mélange des deux.

18.2. Applications

18.2.1. A travers un ampli de puissance classe D

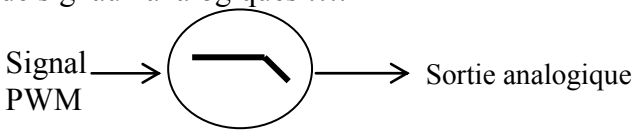
Commande d'éclairage, de moteurs. L'amplificateur travaille **directement en commutation** (**classe D**), et donc est de rendement élevé (souvent 90%).

Pour de petite puissance, le passe bas est en fait souvent l'élément physique actionné (inertie du moteur, rémanence de l'éclairage). Un passe bas supplémentaire (après l'ampli de puissance) peut toutefois être ajouté pour diminuer les interférences de commutation.



18.2.2. Un pseudo CNA ?

A travers un passe bas, on obtient un pseudo convertisseur numérique analogique. Exemple : génération de signaux analogiques ….

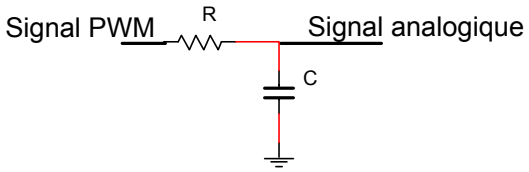


1) Génération d'un signal analogique de fréquence quasi nulle ou de très basse fréquence.

Un simple intégrateur RC peut suffire.
Pourcentage d'ondulation crête-crête (si pas trop élevée) :

$$\frac{\Delta Vmoy}{Vmoy} = \frac{T - Th}{RC}$$

Mais se souvenir du compromis : une faible ondulation nécessite une constante de temps élevée, et donc aussi un régime transitoire long.



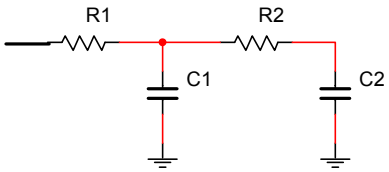
2) Génération d'un signal analogique de Fmax donnée.

Le spectre du signal PWM contient la bande de 0 à Fmax, mais aussi des fréquences autour de F = 1/T et bien au-delà.

Un filtrage aisé et peu onéreux s'obtient facilement lorsque F >> 10 ou 20 Fmax.

Et alors un simple second ordre réalisé à partir d'un double RC peut suffire.

Une astuce évitant l'étude d'un vrai second ordre consiste à prendre **R2 = 10R1**, on peut alors dire que le second RC charge peu le premier, et le montage équivaut pratiquement à deux premier ordre en cascade.
Attention tout de même à l'impédance d'entrée du circuit qui suit …..



18.3. Le port PWM de l'HC12

L'HC12 possède **8 canaux** possible PWM. Avec sortie sur les **8 bits** du **port P**.

On peut travailler en **8 canaux 8 bits** ou **4 canaux 16 bits** (ce dernier mode non étudié ici).

Il contient une trentaine de registres, on décrit ici le minimum !

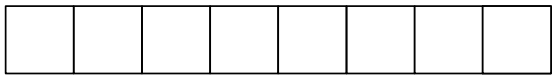
Remarque : Lors de la première initialisation des différents bits, on peut directement écrire 0 dans les bits non utilisés (soit c'est la valeur par défaut, soit écrire 0 est sans effet).

PWME
$A0

Validation de chaque canal PWM

PulseWideModulation Enable

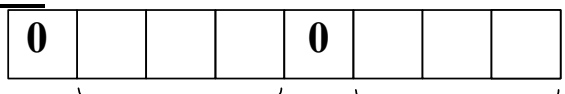
PWMPOL
$A1



Polarité de chaque canal PWM :
0 signal à 0 en début de cycle
1 signal à 1 en début de cycle

PulseWideModulation Polarity

PWMPRCLK
$A3



N_A Prédiviseur A Pour canaux : 0 1 4 5

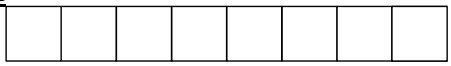
N_B Prédiviseur B Pour canaux : 2 3 6 7

PulseWideModulation Prescale Clock

Clock interne = $\frac{F\_BUS}{2^N}$

N = N_A ou N_B
(prédiviseur de 1 à 128)

PWMCNTi
$AC+i

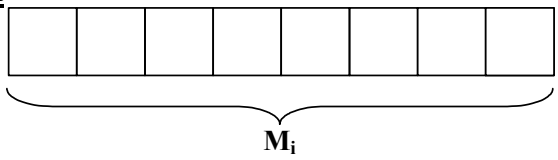


Lecture à tout instant
Une écriture quelconque le met à 0 !

PulseWideModulation Counter_i

Utile pour re démarrer à zéro

PWMPERi
$B4+i



PulseWideModulation Period$_i$

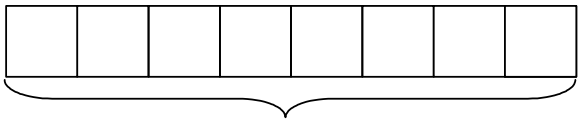
Second diviseur, et alors:

$$F_{Pwm\_i} = \frac{F\_BUS}{M_i.2^N}$$

N = Na ou NB déterminait un prédiviseur identique pour chaque groupement de 4 canaux.

$\mathbf{M_i}$ est un diviseur supplémentaire spécifique pour chaque canal i.

PWMDTi
$BC+i



Première partie de cycle

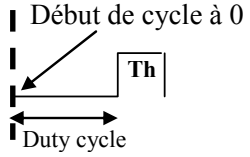
C'est **l'entrée véritable** du modulateur PWM réalisé.
Et avec toujours : **$PWMDT_i < PWMPER_i$!**

PulseWideModulation Duty$_i$ cycle

Réglage du rapport cyclique:

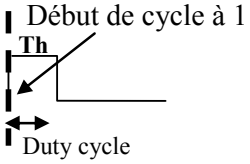
Cas polarity 0 : **0 donne** $Th/T = 1$

$$\text{Rapport cyclique} = 100.\frac{PWMPERi - PWMDTYi}{PWMPERi}$$



Cas polarity 1 : **0 donne** $Th/T = 0$

$$\text{Rapport cyclique} = 100.\frac{PWMDTYi}{PWMPERi}$$



18.4. Exemple d'application : pseudo CNA signé 8 bits

Pseudo CNA signé, de bande passante jusqu'à environ 5000Hz.
Signal de sortie de 0 à Vh, et donc centrée sur Vh/2 (+5 volt).
Un décalage de –Vh/2 permettrait un centrage sur zéro (ou un simple circuit CR si on ne cherche pas à passer le continu).

Pour **F_BUS à 24 MHZ** (voir PLL).
Au-delà, le passe bas tout simple du second ordre n'est plus assez raide.

1) Initialisation du canal 0

```
void ini_pwm0(void)          // Pour F_bus = 24 MHz   sinon 12 fois moins vite !
{
PWMPOL =PWMPOL | 1;          // high au départ pour cycle Duty : polarité 1
PWMPRCLK = 0;                // Prescaler 1/1  donc 24/1 = 24 MHz
PWMPER0 = 150;               // PWfrequency = 160 kHz
PWMDTY0 = 75;        // valeur initiale : ici ensuite excursion de environ +-64 autour (donc de 11 à 139)
PWMCNT0 = 0;                 // on démarre le cycle.
PWME = PWME + 1;             // Validation du canal 0
}
```

Comme PWMDTY0 doit rester évidemment inférieure à PWMPER0 = 150.
On voit qu'une excursion **de +-64** permet d'avoir 11 <= PWMDTY0 <= 139
Il reste une petit marge de chaque coté. Mais on est sûr ainsi de ne pas avoir avoir PWMDTY0 en débordement sur PWMPER0.

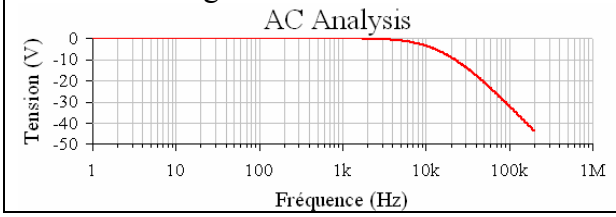
2) Sortie d'une valeur X signé de 8 bits sur le PWM0

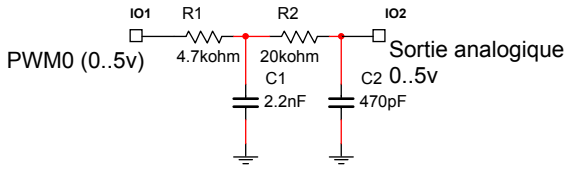
Pour X = 0, on obtient la valeur centrale 75, et donc Vs = V/2 = 2,5 volts.
On écrit simplement :

PWM0 = 75 + X/4 ; ou PWM0 = 75 + X>>4 ;

3) Filtre RC

Module du gain en dB :





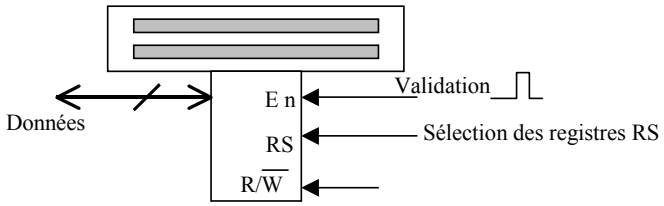
19. AFFICHAGE SUR PANNEAU CRISTAUX LIQUIDES

On trouve de nombreux afficheurs alphanumériques à cristaux liquides dans le commerce pouvant se câbler directement sur un bus d'un microcontrôleur dans une zone d'adresse libre, ou sur un port d'entrée sortie.

19.1. Exemple de composant

Afficheur à panneau cristaux liquides **2 lignes** associé au contrôleur **Hitachi HD44780U.**

Lignes de 40 caractères, dont 16 visibles.



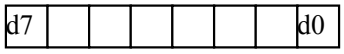
➢ ***Registres du contrôleur***

On peut remarquer deux registres aux mêmes adresses, mais en lecture on accède à BAC et en écriture à IREG automatiquement.

Adresse base + 0 **IREG**

Instruction Register (écriture seule)

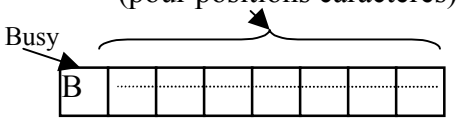
IREG



Adresse base + 0 **BAC**

Busy/Adress counter register

BAC Adress counter (pour positions caractères)



Le contrôleur contient une mémoire de 80 caractères, adressé par un compteur d'adresse.

Registre en lecture seule.

Adresses de **0 à 3F** : ligne du haut N° 0

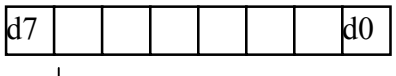
Adresses de **40 à 7F** : ligne du bas N° 1

Adresse base + 1 **DREG**

Data register

Pour l'envoi des caractères (codes ASCII)

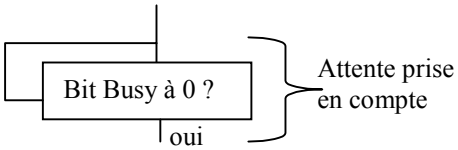
DREG



Ecriture code commande ou donnée

Organigramme d'envoi d'une commande :

(Les données peuvent s'envoyer sans tester forcément Busy)



➢ **Tableau de commandes (codes à envoyer dans IREG)**

Sur la partie gauche de l'octet envoyé, on remarque les codes fixes qui permettent l'accès à telle ou telle commande :

0 0 0 0 0 0 0 1 commande Clear Display

0 0 0 0 0 0 1 commande Return Home

1 - - - - - - - Accès en écriture au compteur interne de position des caractères

B Accès en lecture à ce compteur, et au bit Busy B

Tableau d'instructions :

Instruction	Code RS	R/$\overline{W}$	DB_7	DB_6	DB_5	DB_4	DB_3	DB_2	DB_1	DB_0	Description
Clear display	0	0	0	0	0	0	0	0	0	1	Clears entire display and sets DD RAM address 0 in address counter.
Return home	0	0	0	0	0	0	0	0	1	—	Sets DD RAM address 0 in address counter. Also returns display from being shifted to original position. DD RAM contents remain unchanged.
Entry mode set	0	0	0	0	0	0	0	1	I/D	S	Sets cursor move direction and specifies display shift. These operations are performed during data write and read.
Display on/off control	0	0	0	0	0	0	1	D	C	B	Sets entire display (D) on/off, cursor on/off (C), and blinking of cursor position character (B).
Cursor or display shift	0	0	0	0	0	1	S/C	R/L	—	—	Moves cursor and shifts display without changing DD RAM contents.
Function set	0	0	0	0	1	DL	N	F	—	—	Sets interface data length (DL), number of display lines (N), and character font (F).
Set CG RAM address	0	0	0	1	A_{CG}	A_{CG}	A_{CG}	A_{CG}	A_{CG}	A_{CG}	Sets CG RAM address. CG RAM data is sent and received after this setting.
Set DD RAM address	0	0	1	A_{DD}	A_{DD}	A_{DD}	A_{DD}	A_{DD}	A_{DD}	A_{DD}	Sets DD RAM address. DD RAM data is sent and received after this setting.
Read busy flag & address	0	1	BF	AC	AC	AC	AC	AC	AC	AC	Reads busy flag (BF) indicating internal operation is being performed and reads address counter contents.

Pin Functions

Signal	No. of Lines	I/O	Device Interfaced with	Function
RS	1	I	MPU	Selects registers. 0: Instruction register (for write) Busy flag: address counter (for read) 1: Data register (for write and read)
R/$\overline{W}$	1	I	MPU	Selects read or write. 0: Write 1: Read
E	1	I	MPU	Starts data read/write
DB_4 to DB_7	4	I/O	MPU	Four high order bidirectional tristate data bus pins. Used for data transfer and receive between the MPU and the HD44780U. DB_7 can be used as a busy flag.
DB_0 to DB_3	4	I/O	MPU	Four low order bidirectional tristate data bus pins. Used for data transfer and receive between the MPU and the HD44780U. These pins are not used during 4-bit operation.

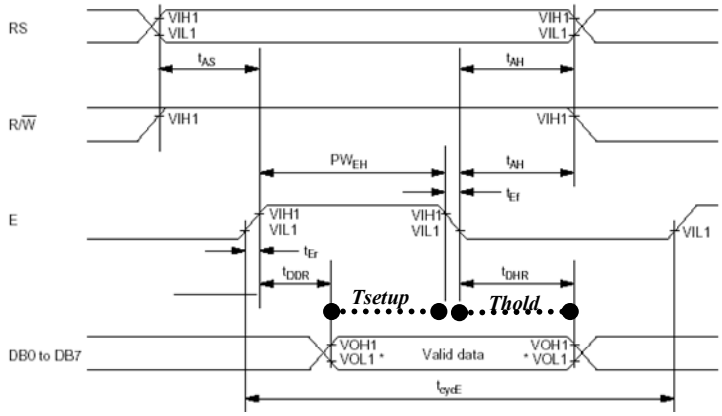
19.2. Câblage sur un microcontrôleur, Driver

1) Technique N° 1 : directement sur un bus d'un microcontrôleur en mode étendu.

Avantage :

Les deux registres 8 bits sont vus par le processeur comme des cases mémoire, accessibles par simples instructions assembleur de lecture/écriture.

Les 3 lignes de contrôle du circuit d'affichage (En, Rs, et $R\overline{W}$) sont gérés automatiquement lors des cycles d'accès mémoire en lecture/écriture. Il faut juste veiller au bon timing sur les chronogrammes constructeurs. De nombreux circuits sont prévus pour un tel câblage. Pour notre circuit :



On peut noter les temps importants principaux:

En lecture, les temps tAS, tDDR, et tDHR

En écriture, tAS, *Tsetup*, et *Thold*

Ainsi que d'autres comme tcycE …

Inconvénient :

Le microcontrôleur doit être en mode étendu, il faut des circuits de décodage d'adresses.

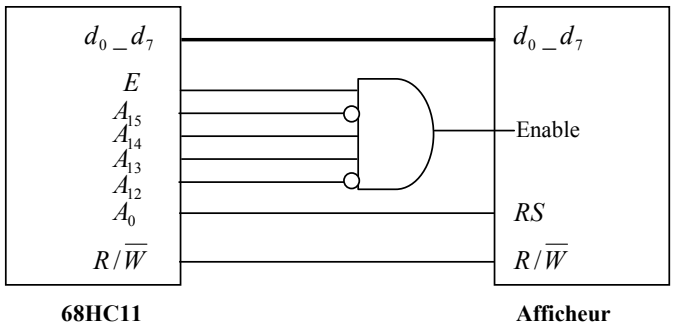
Exemple :

Ce câblage sur un bus HC11 correspond au bloc d'adresses suivantes :

$6000 à $6FFF

Adresses les plus simples à utiliser : $6000 (pour A0 = 0)

$6001 (pour A0 = 1)

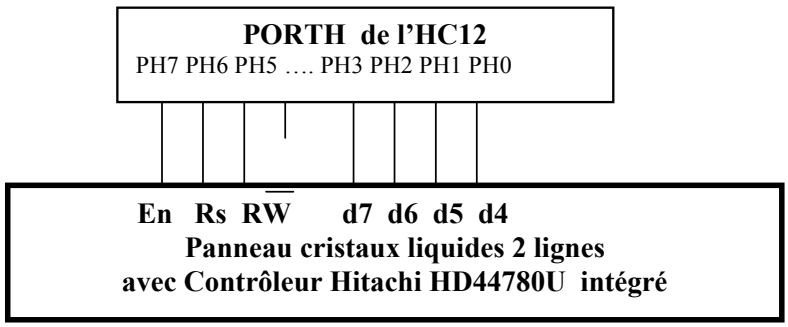


2) Technique N° 2 : Sur un port parallèle 8 bits. NOTRE CAS

Avantage :

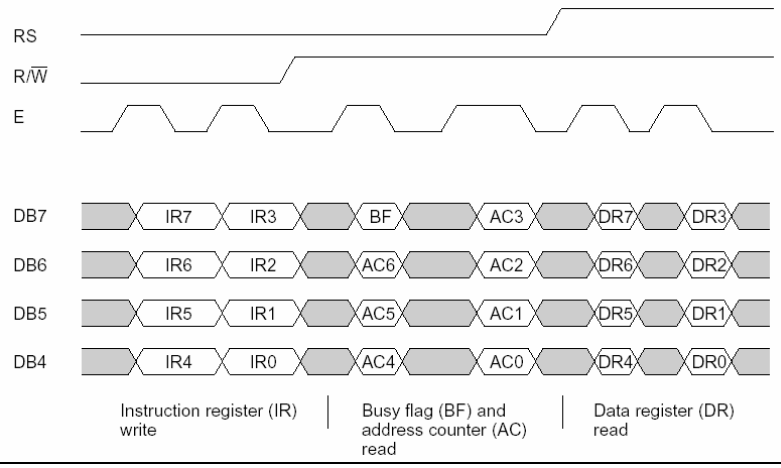
Les microcontrôleurs actuels possédant un grand nombre de ports d'entrée sortie (parallèles et série) et bien plus de mémoire RAM et FLASH, on les utilise désormais de plus en plus en mode microcontrôleur seul, ou « Mono-chip », sans décodage d'adresses extérieur. On réserve donc un port pour l'afficheur.

Exemple :



Inconvénient :

L'afficheur réclamant 3 lignes de contrôle, travailler sur un port de seulement 8 bits nécessite un mode d'accès 4 bits par 4 bits. Ce que prévoit l'afficheur.

Chronogramme Mode 4 bits : On voit que les tous quartets sont transférés sur la partie haute des données : DB7 à DB4, et que deux impulsons E sont nécessaire par octet.	

La gestion des 3 lignes n'est donc plus automatique, il faut les gérer par logiciel.

Les accès au panneau nécessiteront donc forcément un certains nombre de cycle microcontrôleur (envoi de quartets, instructions faisant évoluer les signaux de contrôle). Chaque accès en lecture ou écriture demandera donc plusieurs dizaines de cycles microcontrôleur, donc fort ralentissement mais pas gênant dans le cadre d'un simple afficheur à cristaux liquides !

3) Différences au niveau logiciel, usage d'un Driver

Rappelons ici l'intérêt d 'utiliser un 'Driver'

Il faut distinguer **plusieurs niveaux** de programmation:

Couches de Bas niveau:

Ecriture et lecture dans les registres d'un composant: Couche élémentaire

Envoyer 1 octet (1 code ASCII) sur un afficheur ou un périphérique quelconque de visualisation (par exemple ligne série).

Couche immédiatement supérieure: envoyer un nombre décimal (sous forme de chaîne de caractères). Elle appellera à plusieurs reprises la couche de bas niveau précédente.

Couches encore supérieures …: on peut aussi imaginer des fonctions qui se servent des fonctions de la couche précédente.

Intérêt:

On voit que **changer de périphérique d'affichage** (ou de câblage), revient alors juste à **changer le bas niveau**. Ce bas niveau peut se nommer "**Driver**".

Cette façon de procéder permet de créer des couches de programmation qui ne dépendent pas des périphériques de sortie, et donc des fonctions bien plus polyvalentes.

Etudions ici le cas de l'écriture d'un caractère dans le registre de donnée DREG. Il s'agit bien du plus bas niveau, la fonction putch d'écriture d'un caractère y fait appel, ainsi que d'autres fonctions plus spécifiques à cet afficheur.

Cas N 1 du câblage sur le bus

Soit DREG le label défini par :

#define DREG *(char*)0x4000 par exemple

```
// Ecriture du registre de donnée
void W_DREG(char c)
{
DREG = c;
}
```

Et c'est tout !

Cas N°2 du câblage sur port H de l'HC12

```
#define EN (char)0x80
#define RS (char)0x40
#define RWn (char)0x20

void W_DREG(char c)    // Ecriture du registre de donnée
{
unsigned cc;char i;
i = test_iset();
inhib_it();
PTH = 0;
DDRH = 0xEF;    // Port en sortie tout à zéro
cc = (unsigned char)c >> 4;
W_4(cc + RS);              //  envoi de 0 1 0 0 D7 D6 D5 D4
cc = (unsigned char)c & 0x0F;
W_4(cc+RS);                //  envoi de 0 1 0 0 D3 D2 D1 D0
if(i ==0)valid_it();
}

void W_4(char c)    //Ecriture de 8 bits en deux fois 4 bits
{
Write(c);
Write(c + EN);
Write(c);
}
void Write(char c)
{  PTH = c;  }
```

Fonction assembleur pour cas N°2 :

```
*  test de I, et retour B=0 si I=0, B!=0 si I = 1
test_iset :
        tpa   code condition dans A
        tab
        andb #%00010000
        rts
```

Important : Pour le cas N°1, fonction W_DREG semble peu utile en apparence ! on aurait pu écrire dans les fonctions principales à chaque fois : DREG = c ;

Mais l'avantage est de pouvoir, si besoin est, changer facilement le type de panneau ou son câblage, sans modifier les fonctions principales, en effet, pour un câblage sur le bus on prend la fonction void W_DREG(charc) de gauche, pour un câblage sur le port H comme précédemment, on prend celle de droite.

On remarquera aussi que, en mode 4 bits **l'envoi des 2 quartets** haut et bas est une **opération non divisible**, il faut donc pour éviter tout aléas **inhiber** au début toutes les **interruptions** par inhib_it() ; (appelant en assembleur sei), pour les **revalider** ensuite si le bit I était à 0 (fonction assembleur test_iset) par valid_it() ; (appelant en assembleur cli).

19.3. Fonctions C utilitaires développées pour le composant précédent

Plusieurs modes sont possibles, et une séquence d'initialisation logicielle est à effectuer avant tout affichage. Le mode 4 bits si il est utilisé doit être initialisé dès le début.

19.3.1. Prototypes

Ces fonctions C peuvent être aussi programmées en assembleur non pas par soucis de rapidité, mais de taille du code. Optimiser peut parfois être utile pour des fonctions devant occuper un minimum de place en mémoire, destinées à des cartes ou des microcontrôleurs n'ayant pas forcément des mémoires de taille importante. Il est évident que ces fonctions sont bien plus aisées à écrire tout en C ! Nous fournissons ces dernières.

➢ ***Fonction d'initialisation de l'afficheur (nécessaire)***

Void config_aff(char * tableau) ;

Le mode 4 bits est tout d'abord initialisé dans cette fonction. Puis, les codes placés dans un tableau se terminant par le caractère NULL (code hexa 00) sont envoyés dans l'ordre à l'afficheur : 28 0D 01 06 00 avec curseur

Exemple de configuration classique (4 codes). La partie soulignée correspond à la commande.

0x28 001**0 10**00 instruction Function Set avec :
DL = 1 (bus 4 bits) **N**=1 (2 lignes)
F=0(caractères 5x8 points)

0x0D 0000 1**101** instruction Display on/off control :
D=1 (visu validée) **C**=0 (curseur off)
B=1 **curseur clignotant**

0x01 0000 0001 instruction clear display :
Rempli l'afficheur de n espaces (code ASCII 0x20).
Raz du compteur d'adresse (position caractère à 0)

0x06 0000 01**10** instruction Entry Mode Set
I/D = 1 (mode **incrémentation automatique** du compteur, pour écriture automatique de gauche à droite)
S=0 (pas de commande de recadrage de la ligne à gauche ou à droite)

0x00 Caractère NULL fin de tableau à écrire.

➢ **Fonctions diverses, travaillant avec un curseur courant**

Void change_ligne(void); Passe d'une ligne à l'autre sur l'afficheur, sans effacer.
Void efface_ligne(void); Efface la ligne courante

Void outinteger(int valeur); Affichage valeur binaire entière signée (avec suppression des zéros en tête)
Void putch(char caractère) ; Affichage de tout caractère.
Void virgule(void); écrit virgule, équivaut à putch(',') ;
Void espace(void); écrit espace, équivaut à outch(' ') ;
Void outhex8(char octet) ; affichage héxadécimal d'un octet
void outhex16(int valeur) ; affichage en hexadécimale d'un entier 16 bits.
void outchaine(char chaine) ; écriture de chaîne de caractère
void affich_q1632(long q1632, char * texte) ; Ecriture de valeurs signées en Q1632 avec texte associé.
void tempo(unsigned integer millisecondes) ; temporisation en millisecondes, écrite obligatoirement en assembleur pour le respect du Timing, sert pour le clavier.

19.3.2. Ecriture de ces fonctions en C

➢ **ROUTINES « C » SPECIFIQUES DE CET AFFICHEUR**

```
// ****  FONCTIONS d'AFFICHAGE  sur PANNEAU CRISTAUCX LIQUIDES tout en C  *********
//                  Panneau en mode 4 bits, 2 lignes, sur port H                */
//            Les CHAR doivent être  DES CHAR SIGNES     Janvier 2005  G. Pallot
//       pour F_BUS 2MHZ  (Sinon petit délai nécessaire dans les deux drivers write et read)
 #include <string.h>
 #include "registres_hc12.h"
// Prototypes des DRIVERS ELEMENTAIRE DU PANNEAU CRISATUX LIQUIDES
 char R_BAC(void);    void W_IREG(char c);   void W_DREG(char c);
 void attente_libre(void);  void ini_mode_4bits(void); void tempo(unsigned int ms);
```

```
void config_aff(char *tab)       // chaîne classique à envoyer 28 0C 01 06 00  sans curseur
{                        //                        28 0D 01 06 00  avec cur seur
PTH = 0;   // port H au repos pret à emettre
DDRH = 0xEF;
tempo(200);     //   repos après mise sous tension
ini_mode_4bits();                // Initialisation du mode 4 bits
while(*tab!=0)
   { W_IREG(*tab++);                                    // Driver d'écriture dans IREG
     attente_libre();  /* attente prise en compte */
   }  }
/* ************************************************************** */
void change_ligne(void)
{
if( (R_BAC() & 0x40)==0)W_IREG(0xC0) ;  /*  si première ligne, envoi debut seconde  1 100 0000 */
else  W_IREG(0x80) ;  /* sinon  ligne, envoi debut première  1 000 0000 */
attente_libre();  /* attente prise en compte */
}
/* ************************************************************** */
void efface_ligne(void)
{
 char ini_curseur; char k;
 ini_curseur = R_BAC() & 0x40;  /*  - 000 0000  ou - 100 0000  */
 W_IREG(ini_curseur | 0x80 );          /*  1 000 0000  ou 1 100 0000 */
 attente_libre();  /* attente prise en compte */
 for( k = 0; k <= 15;k++) {        W_DREG(0x20);     /* 16 espaces */
                                    // attente_libre() non utile
                                  }
  W_IREG(ini_curseur | 0x80);   /* curseur de nouveau au début de ligne courante */
  attente_libre();       /* attente prise en compte */
}
/* ************************************************************** */
void curseur(signed char position)      // de 0 à 15 sur ligne courante
{
char ini_curseur;
if(position < 0)           position = 0;
if (position > 15)         position = 15;
ini_curseur = R_BAC() & 0x40;  /*  1 000 0000  ou 1 100 0000  */
W_IREG(ini_curseur + position + 0x80);
attente_libre();       /* attente prise en compte */
}
/* ************************************************************** */
```

➢ ROUTINE DE BASE PUTCH

```
void putch(char c)    // ENVOI CARACTERE  (Code ASCII)
{  W_DREG(c);         //  attente prise en compte inutile sur DREG
}
/* ************************************************************** */
```

➢ ROUTINES « C » POLYVALENTES, utilisant PUTCH

```
void outchaine(char *ch)
{
while(*ch !=0) putch (*ch++);    // Appel à la fonction de bas niveau putch, mieux que W_DREG
        // attente prise en compte pas utile pour data
}
 /* ************************************************************** */
void espace(void)   {  putch(0x20); }
void virgule(void)  {  putch(','); }
/* ************************************************************** */
```

```
void outinteger(int val)
 {
 char flag_zero=0;
 char k;   char ascii[5];     /* tableau de 5 chiffres   car  de -32768 à 32767 */
 if(val ==0)putch('0');
 else
 {
  if(val < 0){ val = - val;  putch('-'); }     /* signe affiché si < 0 seulement */
  for(k=4;k>=0;k--)           /* Conversion Binaire DCB (1 octet par chiffre) et ASCII */
    { ascii[k]= (val%10) + 0x30;
      val = val/10;
    }
   for(k=0;k<=4;k++)     /* sortie du tableau, chiffre fort en premier, avec suppression des zéros en tête,
outchaine pas possible ! */
    {
      if(ascii[k]!= 0x30) flag_zero = 1;
      if(flag_zero != 0)putch(ascii[k]);
    }
}
}
/* ************************************************************** */

void outhex8(unsigned char c)
 {
 unsigned char cc;
 cc = (c >> 4)+0x30 ;  // prise du quartet haut 0011 ----
 if(cc >0x39)cc = cc+0x07;   // pour ABCDEF
 putch(cc);
 cc = (c & 0x0F) + 0x30;
 if(cc >0x39)cc = cc+0x07;   // pour ABCDEF
 putch(cc);
 }

 void outhex16(int val)       // on aurait pu aussi utiliser 2 fois la précédente fonction ….
 {
 char ascii[5];   unsigned int n;  char k;
 n = (unsigned int)val;      /* traitement en non signé */
 ascii[4]=0;        /* fin de chaine   */
  for(k=3;k>=0;k--)           /* Conversion Binaire DCB (1 octet par chiffre) et ASCII */
    { ascii[k]= (n%16)+ 0x30;        /* prise simple 4 par 4 */
      if(ascii[k] > 0x39)ascii[k]=ascii[k]+0x07;
      n = n/16;
    }
 outchaine(ascii);
 }
/* ************************************************************** */
void affich_q1632(long q1632,char *unite)
  {
  unsigned int entiere;unsigned int frac;unsigned long int val32;
  int k;
  efface_ligne();
  if (q1632 < 0)   {   q1632 = -q1632;  putch('-');   }
        /* remarque : q1632 non modifiée à l'extérieur de la fonction, car passage par valeur  */
  entiere=q1632 >>16;                 /* ici la partie entière est donc toujours > 0 */
  outinteger(entiere);virgule();   /* affichage de la partie entière suivie de la virgule */

  frac = 0xffff & q1632;   /* partie fractionnaire   dans frac */
  for(k=0;k<=2;k++)        /* on multiplie 3 fois par 10 pour avoir 3 chiffres après la virgule */
     {
```

```
   val32 =(unsigned long)10*(unsigned long)frac;       /* val32, variable de travail de 32 bits */
   entiere = val32 >> 16 ;       /* chaque chiffre (dixième, centième .., se trouve dans la partie entière */
   putch(entiere+0x30);          /* on affiche le caractère ASCII de chaque valeur de 0 à 9 */
   frac = (unsigned int) (0xffff & val32);        /* en enlève à chaque  fois la partie entière */
     }
outchaine(unite) ;
  }
```

➢ LA COUCHE ELEMENTTAIRE et DIVERS

```
/*****************************************************************************************
/************* LES DRIVER  pour le C, AVEC panneau  MODE 4 BITS SUR PORTH *********
/***   Format  8 bits  EN RS RWn -  Db7 Db6 Db5 Db4      RS = 0: command   RS = 1:  Data   **/
#define EN (char)0x80
#define RS (char)0x40
#define RWn (char)0x20
void Write( char);  char Read(void);  void W_4(char c);
extern void tempo(unsigned int ms);
char R_BAC(void)              // Lecture Registre Bac : registre d'état et d'adresse
{
char c,i;
i = test_iset();
inhib_it() ;
PTH=0; DDRH = 0xE0;           // bits utiles du Port LSB en entrée
Write(RWn);                   // envoi de 001-(0000)
Write(RWn+EN);                // envoi de 101-(0000)
c = Read()<<4;                // Lecture D7 D6 D5 D4 0 0 0 0
Write(RWn);                   // envoi de 001-(0000)
Write(RWn+EN);                // envoi de 101-(0000)
c = c + (Read()& 0x0F);       // Lecture D7 D6 D5 D4 D3 D2 D1 D0
Write(RWn);                   // envoi de 001-(0000)
if(i ==0)valid_it() ;
return(c);
}
void W_IREG(char c)                   // Ecriture du registre d'instruction
{
unsigned cc; char i;
i = test_iset();
inhib_it() ;
PTH = 0;
DDRH = 0xEF;          // Port en sortie tout à zéro
cc = (unsigned char)c >> 4;
W_4(cc);                  //  envoi de 0 0 0 0 D7 D6 D5 D4
cc = (unsigned char)c & 0x0F;
W_4(cc);                  //  envoi de 0 0 0 0 D3 D2 D1 D0
if(i ==0)valid_it();
}

void W_DREG(char c)                   // Ecriture du registre de donnée
{
unsigned cc; char i;
i = test_iset();
inhib_it() ;
PTH = 0;
DDRH = 0xEF;                  // Port en sortie tout à zéro
cc = (unsigned char)c >> 4;
W_4(cc + RS);                 //  envoi de 0 1 0 0 D7 D6 D5 D4
cc = (unsigned char)c & 0x0F;
W_4(cc+RS);                   //  envoi de 0 1 0 0 D3 D2 D1 D0
if(i ==0)valid_it() ;
}
```

```
void W_4(char c)                //Ecriture de 8 bits en deux fois 4 bits
{
Write(c);
Write(c + EN);
Write(c);
}

void ini_mode_4bits(void)       //Initialisation du mode 4 bits
{      // Pour comprendre, il faudrait voir la doc complète de l'afficheur, sur le NET.
W_4(3);       // Nécessaire si le Reset Interne de mise sous tension de l'afficheur n'existait pas
tempo(5);    // > 4.1 ms
W_4(3);
tempo(1);     // > 100micro
W_4(3);
tempo(1);
W_4(2);
tempo(1);
}
void attente_libre(void)
{
while( (R_BAC() & 0x80) !=0);   // tant que busy = 1
}
void Write(char c)                                  char Read(void)
{                                                   {
PTH = c;                                            // delay(ddelay);          // Si F_BUS > 2MHz
// delay(ddelay);          // Si F_BUS > 2MHz       return(PTH&0x0F); // les 4 LSB seulement
}                                                   }


/**************    Utile pour F_BUS > 2MHz, par exemple le maximum  24 MHz  *****/
unsigned int ddelay = 30;   // OK pour 24 MHZ  fonction appelée juste dans write et Read
void delay(unsigned int dd)     // Délai non chiffré ici mais correct
{
static unsigned int k;
for(k=0;k<dd;k++);
}
// Et Appeler ce delay dans les deux fonctions concernées write et Read (mis en commentaire actuellement)
```

➢ **Routines assembleur** pour le C:

Tempo : Fonction assembleur de retard en millisecodes, étudiée précédemment.

test_iset : **Test de I, et retour B=0 si I=0, B!=0 si I = 1**

```
*       psha    si assembleur seul
        tpa     code condition dans A
        tab
        andb #%00010000
*       pula    si assembleur seul
        rts
```

(Remarque, test_iset ici ne sauve pas forcément A, car le compilateur IAR l'autorise. Si on appelait cette fonction en assembleur seul, il faudrait mettre les deux instructions psha et pula !).

19.3.3. Exemple d'écriture en assembleur pour moins de place en mémoire (et pour compilateur IAR)

Les fonctions seront aussi plus rapide d'exécution, mais l'intérêt est ici pour un afficheur une place en mémoire moindre si besoin est. Et ce sont des exemples de fonction C écrites en assembleur. Nous ne donnons ici que quelques fonctions. Bus 2MHZ.

```
PUBLIC change_ligne,efface_ligne,curseur,config_aff,outchaine,outinteger,outhex16,espace,virgule,putch
extern tempo
```

```
* IREG  RS=0  instruction register,   DREG  RS=1 data register,   BAC   RS=0   busy flag/adress counter
          RSEG  RCODE
* Envoi les mots de configuration de l'afficheur, en mode 4 bits :  En pratique 28 0D 01 06 00 avec curseur
   config_aff:
        jsr        ini_mode_4_bits    initialisation mode 4 bits, avec mini tempo initiale incluse
        pshd       en entrée D contient l'adresse du premier code de configuration
        pshx
        xgdx       adresse du premier code de configuration dans X
        ldab   0,x
   p1   jsr   W_IREG
        jsr attente_libre
        inx
        ldab   0,x
        cmpb   #$00       Code de fin de tableau de configuration (00)
        bne    p1
        pulx
        puld
        rts

   * Conversion d'un nombre binaire de 16 bits signé et affichage decimal
   * SANS LES ZEROS EN TETE       opérande passé par D
   outinteger:
        pshd
        addd #0           pour tester nombre négatif ? sur 16 bits
        bpl  positif
        coma              ici nombre negatif
        comb
        addd #1           prise du complement à 2
        jsr moins
   positif:  pshx
        pshy
        cpd    #0         si opérande nul, on doit afficher 0 sans le traiter
        beq    nul
        ldy    #5         on tourne cinq fois (D max = 65535)
   encore  ldx    #10      CONVERSION
        idiv              division de D par X,  Q en X  reste en D  suffit en B
        addb   #$30       conversion ASCII
        pshb              on conserve les chiffres ASCII dans la pile
        xgdx              Q revient en D
        dey
        bne    encore
        ldy    #5         AFFICHAGE des codes ASCII
        ldx    #0         X sert de drapeau de leadings zeroes
   encore1 pulb
        cmpb   #$30       comparaison a zero
        beq    no         tant qu'on a des zeros en tete, X reste
        ldx    #1         drapeau X=1  ok on affiche
   no     cpx    #0
        beq    null       on affiche pas tant que X = 0
        jsr    putch      envoi d'un chiffre
   null   dey
        bne    encore1
   pt     puly
        pulx
        puld
        rts
   nul    ldab   #$30       caractere 0
        jsr    putch
        bra    pt
```

```
   * Sous programme d'affichage chaine de caractères,  $00 est le code de fin de chaîne
   outchaine:
        pshd
        pshx
        xgdx     x pointe alors debut chaine
   encore5 ldab    0,x
        beq     fini
        jsr     putch
        inx
        bra     encore5
   fini   pulx
        puld
        rts
```

```
   attente_libre:  // Attente Busy Flag à 0
        pshb
   occ   jsr        R_BAC
        bmi         occ
        pulb
   rts

**Envoi caractère de l'accu B codé ASCII vers l'afficheur **
   putch:
         pshb
   cc    jsr    W_DREG   Dans Driver
   *     attente_libre non utile dans l'envoi dans DREG
         pulb
          rts
```

```
**********************************************************************************************
**                          LE DRIVER  AVEC LCD MODE 4 BITS SUR PTH                          ****
** Plus simple ici : pas de Test_iset :  SUPPOSENT donc que les interruptions sont toujours à priori validées. On pourrait l'ajouter aisément !
***          Format  8 bits  EN RS RWn -  Db7 Db6 Db5 Db4      RS = 0: command   RS = 1:  Data
   PTH             equ $260
   DDRH            equ $262
   RWn             equ $20
   RS    equ $40
   EN   equ $80
   ENN             equ $7F
   R_BAC:      // Lecture en deux fois 4 bits.  Retour opérande par B
     sei           // inhib it
     Movb          #0,PTH
     movb          #$E0,DDRH   Mise du port: 4 bits LSB en Entrée
     psha
     ldab          #RWn
     jsr           w_octet    envoi de 0 0 1 (0  0 0 0 0)      EN = 0  RS  = 0  RWn = 1
     ldab          #RWn+EN
     jsr           w_octet             EN à 1:  1 0 1 (0  0 0 0 0)
     jsr           r_octet    lecture de ---- D7 D6 D5 D4 dans B   soit  Bac Ac6 Ac5 AC3
     tba
     ldab          #RWn
     jsr           w_octet             EN à 0:  0 0 1 (0  0 0 0 0)       EN = 0  RS  = 0  RWn = 1
     lsla
     lsla
     lsla
     lsla       D7 D6 D5 D4 0 0 0 0    récupéré dans A
     ldab          #RWn+EN
     jsr           w_octet             EN à 1:  1 0 1 (0  0 0 0 0)
     jsr r_octet    lecture de ---- D4 D3 D2 D1 dans B
     andb          #$0F     0 0 0 0 D4 D3 D2 D1
     aba          D7 D6 D5 D4 D3 D2 D1 D0 dans A
     ldab          #RWn
     jsr           w_octet             EN à 0:   0 0 1 (0  0 0 0 0)
     tab
     pula
     cli           // valid it
     rts
   W_IREG:   // envoi de 8 bits de commande (D7 D6 D5 D4 D3 D2 D1 D0 en B) en deux fois 4, RWn = 0
     sei           // inhib it
     Movb          #0,PTH
     movb          #$EF,DDRH      Mise du port: 7 bits en sortie
```

```
    pshb
    pshb
    lsrb
    lsrb
    lsrb
    lsrb
    jsr         W_4             envoi de 0 0 0 0 D7 D6 D5 D4   avec RS = 0
    pulb                        D7 D6 D5 D4 D3 D2 D1 D0
    andb        #$0F            0  0  0  0  D3 D2 D1 D0
    jsr         W_4             envoi de 0 0 0 0 D3 D2 D1 D0   avec RS = 0
    pulb
    cli         // valid it
    rts
ini_mode_4_bits: //  pour Bus 4 bits
    pshx
    pshd
    movb        #0,PTH
    movb        #$EF,DDRH       Mise du port: 7 bits en sortie.  E à zéro
    ldd         #50             c'est le tempo du C avec paramètre par D !
    jsr tempo   50 ms           attente afficheur bien sous tension
    ldab        #3              0 0 0 0 0 0 1 1   10 lignes nécessaire si plantage complet de l'afficheur
    jsr         W_4             Celui ci n'effectue son Reset interne Qu'à la mise sous tension
    ldd         #5              plus de 4.1 ms
    jsr         tempo
    ldab        #3              0 0 0 0 0 0 1 1
    jsr         W_4
    ldd         #1              plus de 100micro
    jsr         tempo
    ldab        #3              0 0 0 0 0 0 1 1
    jsr         W_4
    ldd  #1     plus de 100micro
    jsr tempo
    ldab        #2              0 0 0 0 0 0 1 0
    jsr         W_4
    ldd #1      plus de 100micro
    jsr tempo
    puld
    pulx
    rts
W_4:        // Envoi des 4 bits MSB de B avec juste impulsion EN
    pshb
    jsr         w_octet     0 RS 0 0 d  c  b  a
    orab        #EN
    jsr         w_octet     1 RS 0 0 d  c  b  a
    andb        #ENN
    jsr         w_octet     0 RS 0 0 d  c  b  a
    pulb
    rts

w_octet:        // envoi octet en B
    stab        PTH
    rts
r_octet:
    ldab        PTH
    andb        #$0F
    rts
W_DREG:     // envoi complet de 8 bits de DATA (D7 D6 D5 D4 D3 D2 D1 D0 en B) en deux fois 4
    sei         // inhib it
    Movb        #0,PTH
    Movb        #$EF,DDRH   Mise port: 7 bits en sortie
    pshb
    pshb
    lsrb
    lsrb
    lsrb
    lsrb        0 0 0 0 D7 D6 D5 D4
    orab        #RS
    jsr W_4     envoi de 0 1 0 0 D7 D6 D5 D4    RS = 1
    pulb        D7 D6 D5 D4 D3 D2 D1 D0
    andb        #$0F
    orab        #RS
    jsr         W_4   envoi  0 1 0 0 D3 D2 D1 D0, RS = 1
    pulb
    cli         // valid it
    rts
```

20. GESTION DE CLAVIER 16 TOUCHES

Il peut s'agir de touches isolées, ou de regroupement de touches en matrice, avec gestion matérielle ou logicielle … Les claviers type ordinateur débordent du cadre des microcontroleurs, et s'utilisent en fait simplement sur un des lignes séries par exemple RS2322 ou USB.

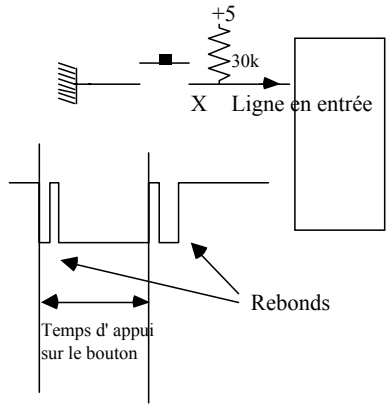
20.1. Simples touches isolées

On veut détecter, par sondage ou par interruption, la présence d'un niveau ou d'un front descendant, par exemple sur une ligne X.

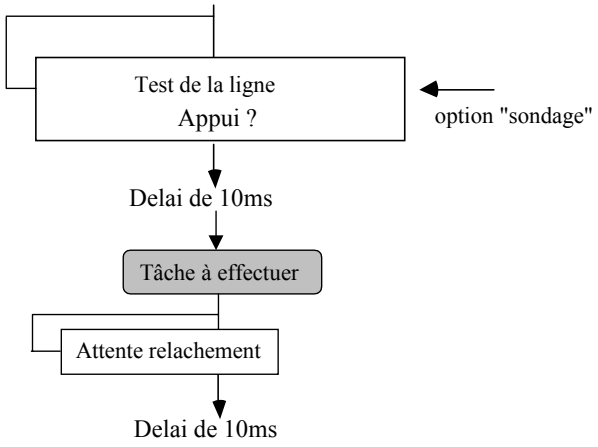
Les **rebonds** peuvent avoir lieu à l'**appui** et au **relâchement** de la touche, la largeur minimale d'une impulsion de rebond est de 0,1ms environ , la durée maximale de la salve est de **5 ms** à **10ms** (attention, la durée augmente avec l'âge du contacteur) .

Si la tâche à traiter dure moins de 10ms, les rebonds aussi bien a l'appui qu'au relâchement peuvent provoquer des interruptions et des prises en compte multiples !

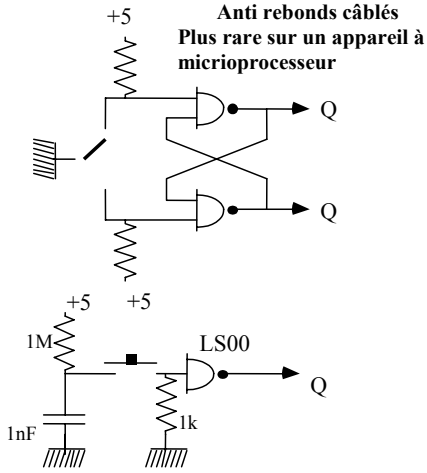
Un **antirebond** est donc très souvent indispensable.



Anti rebond programmé



Anti rebonds câblés
Plus rare sur un appareil à micrioprocesseur



20.2. claviers

Un clavier est un ensemble de boutons poussoirs disposés, pour le câblage, en matrice. (La répartition physique peut être quelconque).

On peut utiliser, soit une structure câblée (avec contrôleur de clavier intégré ou non au clavier), soit une gestion logicielle .

Le microprocesseur utilisé peut être celui du système complet. On travaille par "sondage", (le microprocesseur alternant différentes tâches, peut revenir périodiquement pour scruter le clavier), ou par "interruption".

Les claviers actuels d'ordinateurs (de 64 touches) utilisent quant à eux une structure programmée avec un micro contrôleur propre.

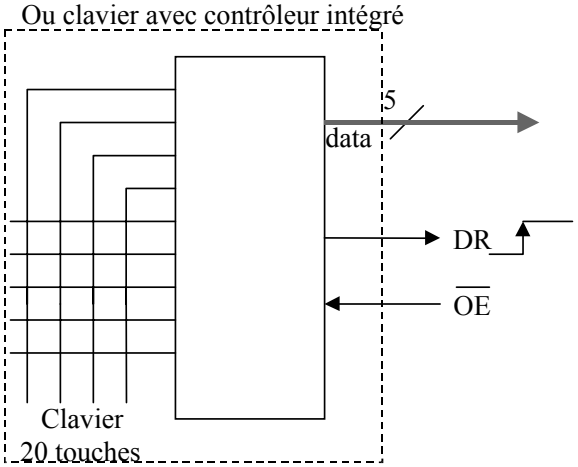
20.2.1. Clavier associé à un circuit codeur de clavier (externe ou incorporé)

Exemple : un clavier matriciel de 20 touches. On s'affranchit de la gestion logicielle de ce clavier par un circuit logique, assurant la lecture du clavier par balayage.

Vu de l'extérieur, il fournit un signal (front montant) sur DR (Data ready) dès qu'une touche à été frappée et que la donnée est donc accessible.

La mise à basse impédance de la donnée sur 5 bits s'effectue par l'entrée $\overline{OE}$

Il existe de nombreux claviers qui possèdent déjà leur contrôleur intégré.



Le branchement sur un bus ou sur un port parallèle est évident. On peut utiliser la ligne DR en « interruption », ou en « sondage ».

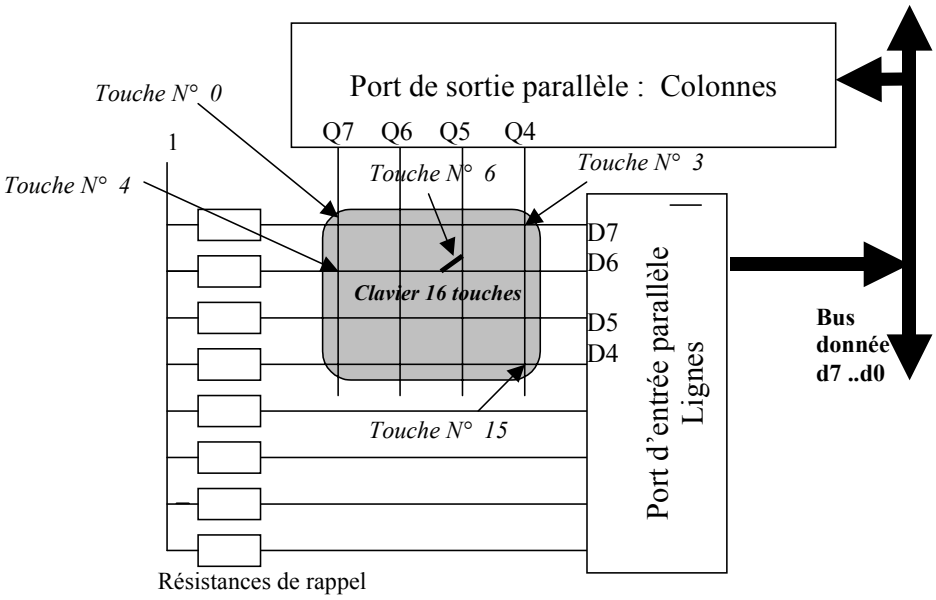
20.2.2. Clavier matriciel géré par logiciel (balayage)

➢ ***Principe***

Les touches sont réparties électriquement en lignes et colonnes.

Exemple, un clavier 16 touches, 4 lignes, 4 colonnes. Les colonnes sont câblées sur un port de sortie parallèle, les lignes sur un port d'entrée parallèle. On pourrait étendre facilement à un clavier 32 touches (8 lignes, 4 colonnes), ou même d'avantage.

Si les ports sont des ports d'entrée sortie programmables, on peut programmer une fois pour toute leur direction



Les niveaux logiques des lignes sont souvent en logique négative (consommation de courant moindre si on est en technologie TTTLS).

On scrute donc une colonne en envoyant un code contenant un seul 0 parmi 4 bits.

Une **séquence de lecture** clavier consiste à **balayer successivement chaque colonne,** en envoyant les codes **colonnes** successifs :

CC = **0** 1 1 1 - - - - 1 **0** 1 1 - - - - 1 1 **0** 1 - - - - 1 1 1 **0** - - - -

Si aucune touche n'est enfoncée, les 4 bits utiles lus sont à 1, on lit un code ligne :

CL = 1 1 1 1 - - - -

Une touche pressée un code avec les 4 bits non égaux à 1, par exemple :

CL = 1 **0** 1 1 - - - -

On connaît alors à cet instant un code ligne et un code colonne qui renseignent sur la position de cette touche : pour celle dessinée sur la figure, on aurait : CC = 1 1 **0** 1 - - - - et CL = 1 **0** 1 1 - - - - , donc la touche colonne 2, ligne 1 (numérotation de 0 à 3).

A partir des codes CC et CL, on calcule facilement (position du zéro) les numéros correspondants de ligne et de colonne, **NC** et **NL** (Si le câblage est effectué correctement, ordre et cadrage des bits, la fonction de passage est la même pour traiter CC et CL).

Si on numérote les touches de 0 à 15 ligne par ligne (de gauche à droite), on passe alors simplement à un numéro de touches : **NT = 4.NL + NC** (pour la touche appuyée du dessin : NT = 4.1 + 2 = 6).

Ce nombre pris comme indice permet alors d'accéder à un **tableau de codes de touches** :

Code_touche = Tab_codes[NT]

- ***Les trois techniques de travail***

Attente de touche (sans rien faire d'autre)

On peut réaliser une fonction lisant le clavier et attendant que l'on frappe sur une touche). On programmera pour cela une boucle attendant un bit à zéro dans le code lu.

Lecture au vol

Lors d'une procédure répétitive, on peut venir de temps en temps lire le clavier pour voir si on appui ou non sur une touche, et agir en conséquence.

Interruption sur certaines touches.

Pratique si on programmer des touches pour reprendre la main, arrêter un programme, etc ..

- ***Choix des codes de touches, hiérarchie de touches ...***

Un clavier standard utilisable en informatique, pour du traitement de texte, de la gestion de fichier, de la programmation ..., travaille avec un code de touche international: ASCII par exemple, et à priori toutes les touches sont valides. On peut par contre en inhiber par programme par la suite.

Soit le cas d'un appareil quelconque (appareil de mesure, automate,..) piloté par microprocesseur. L'utilisateur dispose d'un certain nombre de touches pour piloter l'appareil : Touches de **commandes**, touches **numériques**, touches **d'unités** (mV, Hz, etc …). On conçoit qu'il ne faut pas appuyer sur n'importe quelle touche à tout instant !. Si on ne prévoit pas toutes les fausses manipulation possible, il y aura forcément plantage du système, ou introduction de valeurs totalement erronées. !

Deux exemples très simples :

Lors d'une introduction de valeurs numériques, taper -0,-78 ou –8,67A ou 67,78,9 … sont des suites de touches stupides, il faudra donc interdire certaines touches à des moments précis.

Taper des nombres alors que l'on doit taper au préalable une touche de commande peut aussi faire planter le programme si rien est prévu !

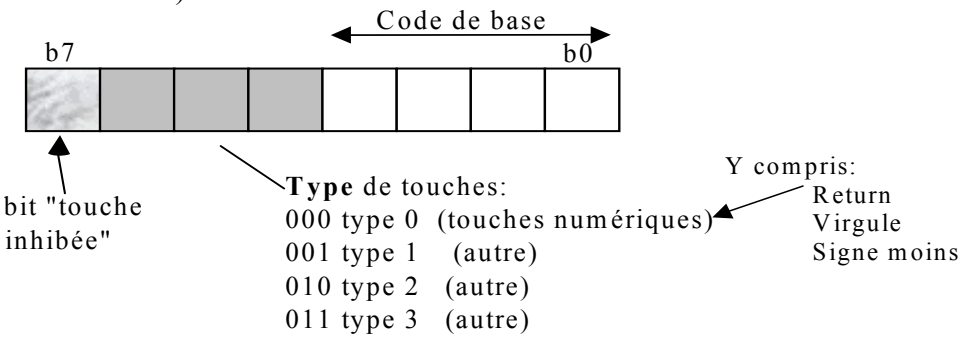
Conclusion :
Un organigramme de **gestion des fausses frappes** (ou des frappes « idiotes » ou sans objet) est toujours nécessaire.
Un **code de touche astucieux** facilitera sa programmation.

L'organigramme de gestion des frappes idiotes est nécessaire et souvent spécifique de chaque application. Nous verrons plus loin des fonctions d'acquisition de valeurs décimales, ou toutes les fausses frappes sont protégées.
Etudions ici l'utilisation de codes de touches pouvant faciliter la programmation future.

- ***Exemple de codes de touches***

Il faut :
Avoir quelques touches travaillant en interruption
Pouvoir **inhiber** des touches de manière individuelle
Pouvoir **autoriser** ou non des touches d'un type donné (touches "numériques" .. "touches d'unités" ...)

Un codage tel que ci-dessous possède ces propriétés, ces codes de touches devront êtres placés en RAM, pour pouvoir ainsi modifier certains bits de contrôle. Les codes par défaut doivent être évidemment en ROM et recopiés au démarrage du programme (ce sont des « variables initialisées »).



Pour attendre une touche d'un type donné, le microprocesseur, devra par exemple utiliser un paramètre **Type** (passé à la fonction de lecture clavier)

$00 pour une touche **numérique**. Le plus simple, pour que valeur du chiffre = code.
$10 pour attendre une touche de type 1 : par exemple **unités (mV, A ..)**
$20 type 2
$30 type 3 : commandes divereses par exemple.

Avant de reconnaître la touche par son code spécifique sur 5 bits, le logiciel associé de lecture de clavier devra, dans l'ordre:

- **Tester le poids fort**, si ce bit est à 1, pas d'action (sauf pour une touche en interruption).
- Effectuer un **masque** avec **%01110000 et comparer** au paramètre **Type.**

Si il n'y a pas égalité. Pas d'action, retour a la lecture du clavier.
Si il y a égalité, le type de touche attendu est correct, la fonction lecture clavier peut retourner le code de la touche appuyée.

On peut avoir besoin de touches sur interruption. Cela est possible, nous le verrons dans un câblage ultérieur.

On pourrait imaginer aussi un bit de touche prioritaire supplémentaire, (par exemple le bit le plus à gauche des 3 bits du type). C'est le programme ayant lancé la lecture du clavier qui devra alors voir si la touche est prioritaire ou non, et agir en conséquence. Cette notion de touche prioritaire en fait n'est pas toujours utile, des touches en interruptions peuvent suffire.

Ces codes sont évidemment dans des variables initialisées (et non des constantes !)

20.3. Exemple de câblage d'un tel clavier sur un microcontrôleur

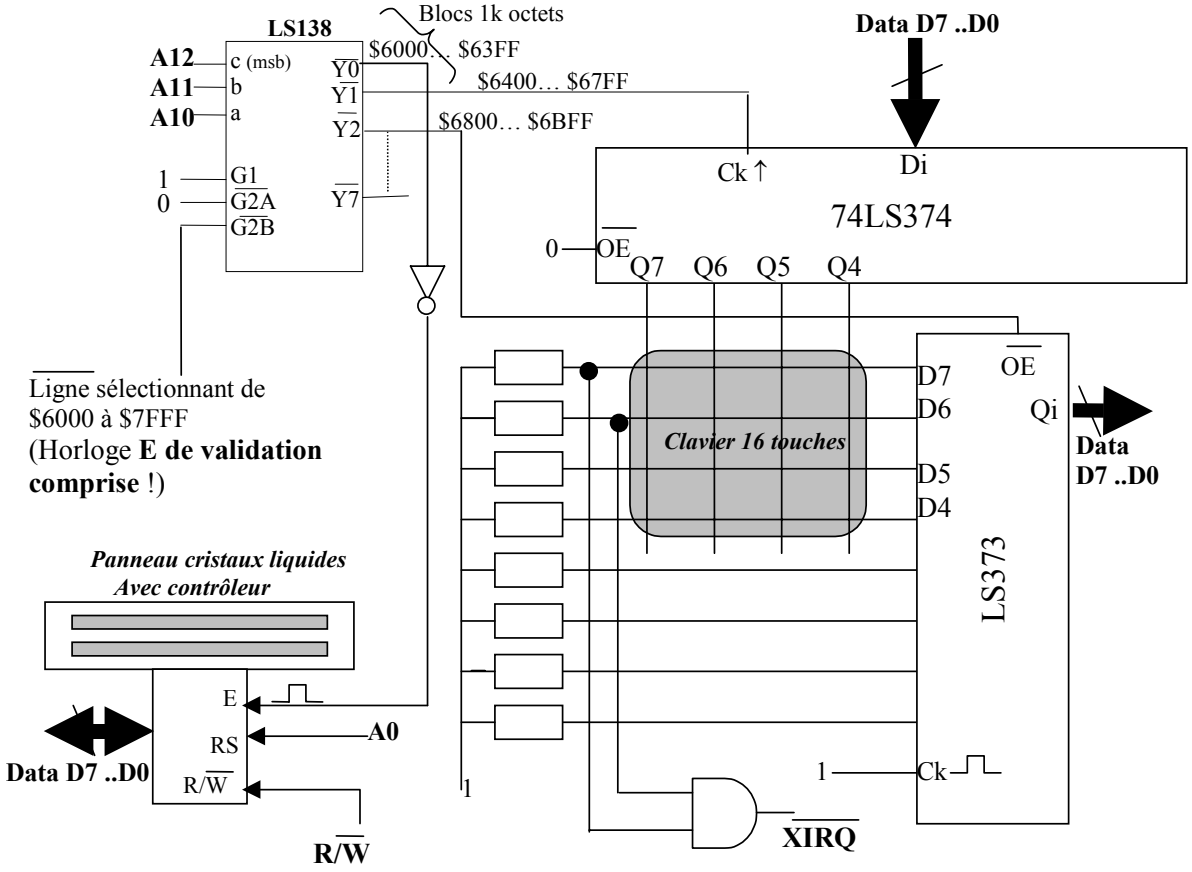
20.3.1. Technique N°1 : câblage direct sur le bus en mode étendu, avec circuits IO parallèles supplémentaires.

Exemple d'un montage complet sur un bus de HC11, avec même le panneau d'affichage précédent (en profitant d'un unique décodage par décodeur 3/8).

Si on dispose de l'espace mémoire libre $6000..$7FFF, le câblage est le plus simple possible, toutes adresses images permises dans les blocs de 1k octets (Décodeur 3→ 8)

Les ports d'entres sortie pour le clavier sont ici simplement des registres classiques 8 bits, avec écriture par un Ck front montant, et lecture par un signal $\overline{OE}$ de mise à basse impédance des sorties Qi.

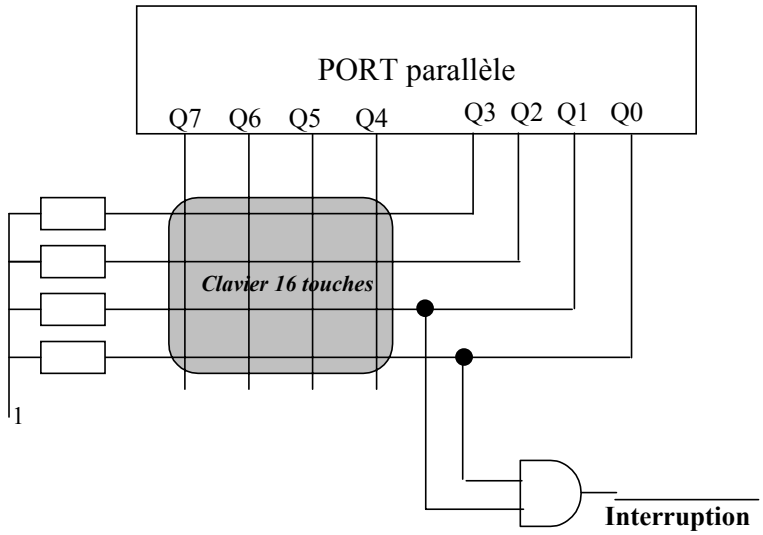
On peut aussi travailler sur interruption sur certaines touches, voir plus loin.



On accède directement aux deux ports colonnes et lignes par leur adresses.

Adresses les plus simples :	Afficheur	$6000
	Port Colonnes	$6400
	Port lignes	$6800

20.3.2. Technique N°2 : câblage sur un port parallèle du microcontrôleur

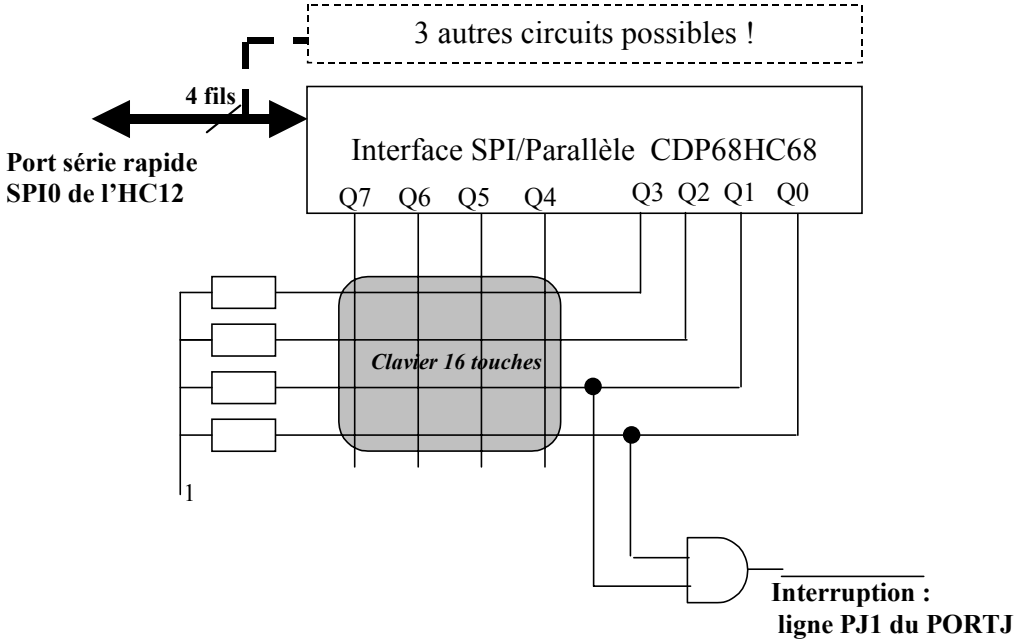


Si on veut manipuler comme précédemment les codes lignes, il suffit en lecture de les décaler à gauche de 4, en mettant à 1 le quartet bas.

Avantage : pas de circuits supplémentaires. Le microcontrôleur peut rester en « single chip »

Inconvénient : Il faut encore un port parallèle, et encore une liaison à 8 fils !

20.3.3. Technique N°3 : câblage sur un port série rapide SPI de l'HC12 avec circuit interface SPI/Parallèle supplémentaire. NOTRE CAS



Avantages :

Le microcontrôleur peut rester en « single chip ». On n'utilise pas de port parallèle supplémentaire.

On pourrait câbler au total 4 interfaces SPI/parallèles de ce type sur le même port SPI. Et l'HC12 possède 4 ports SPI !

Il existe des claviers (et aussi d'autres composants) prévus directement pour être câblés sur un tel bus.

Peu de fils de liaison entre le processeur et ces périphériques.

Donc très utilisé !

Inconvénient ?

Ralentissement des transferts, bien que le bus série soit rapide (1Mbits par secondes ou plus). Mais aucune gêne pour gérer un périphérique tel qu'un clavier ! sauf pour peut être des extra terrestres tapant à 1 kilo touches/seconde …..

Nécessité évidemment de driver, et d'une initialisation du port série SPI.

Circuits supplémentaires d'interface, mais au niveau du périphérique seulement, ne risquant donc pas de perturber le microcontrôleur qui reste en Single Chip.

20.4. Logiciel de gestion du clavier

Ce logiciel peut tourner sur le montage précédent sur bus SPI0.
Clavier matriciel 16 touches
Gestion par balayage colonne par colonne

20.4.1. Prototype et Cahier des charges des deux fonctions

Char lecture clavier(char type) ;

Lecture clavier avec attente de frappe de touche. Avec possibilité :

- d'attendre la frappe d'un type de touche donnée (par la variable type qui doit correspondre au masque à faire)
- de ne pas réagir sur une touche inhibée individuellement.

Il renvoi le code de la touche.

Char acquisition_q1632(long *q1632) ;

Acquisition de valeurs signés en **Q16(32)** (de –32767,999 à 32767,999) avec :

Echo des touches sur la ligne courante de l'afficheur cristal liquide précédent.

Renvoi un drapeau de dépassement. Ancienne valeur conservée si dépassement.

Protection des fausses frappes.

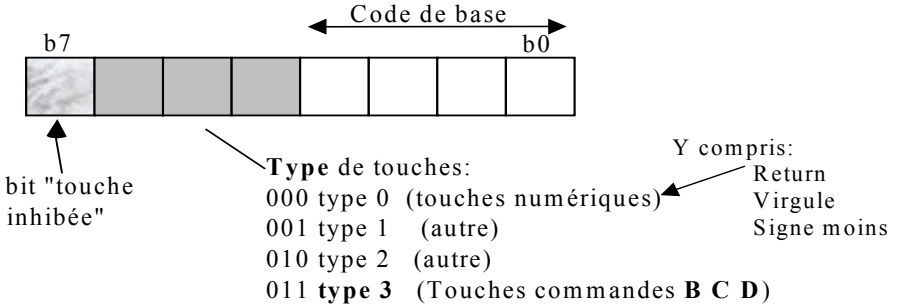
20.4.2. Choix des codes

- ***Codes colonnes* successifs** pour l'exploration : **0** 1 1 1 1 **0** 1 1 1 1 **0** 1 1 1 1 **0**
- ***Choix des touches***

1	2	3	Ret
4	5	6	**B**
7	8	9	**C**
-	0	.	**D**

B,C,D touches de fonction, **C et D** possibles en interruption sur **PJ1 du PORTJ.**

- ***Codage des touches*** (gestion aisée des types de touches, des touches inhibées …) :

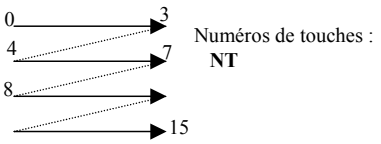


- ***Codes hexadécimaux correspondant***

01	02	03	0b
04	05	06	3B
07	08	09	3C
0f	00	**0e**	**3D**

Ici B, C, D touches de type 3

Numéros des touches de 0 à 15 (ligne par ligne)

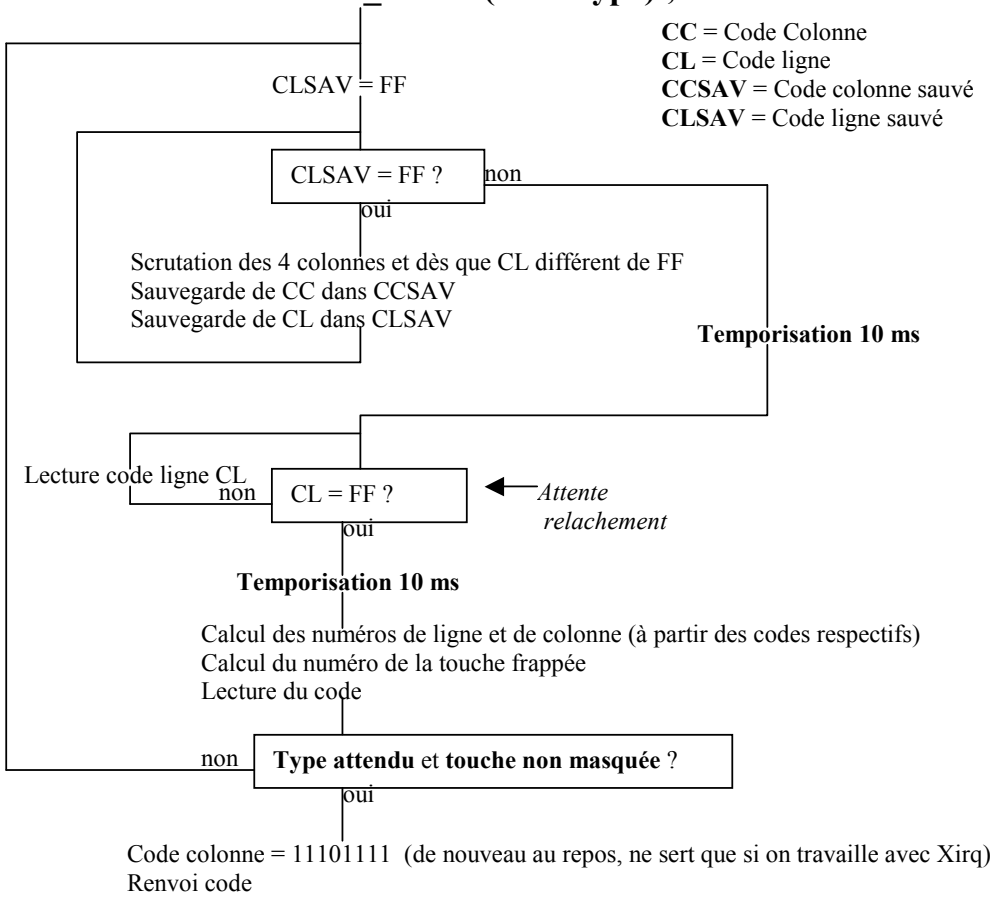


20.4.3. Organigrammes

1) Lecture clavier :

On notera les deux temporisations d'environ 10ms comme anti rebond à l'appui et au relâchement. La prise en compte définitive est faite au relâchement.

Char lecture_clavier(char type) ;



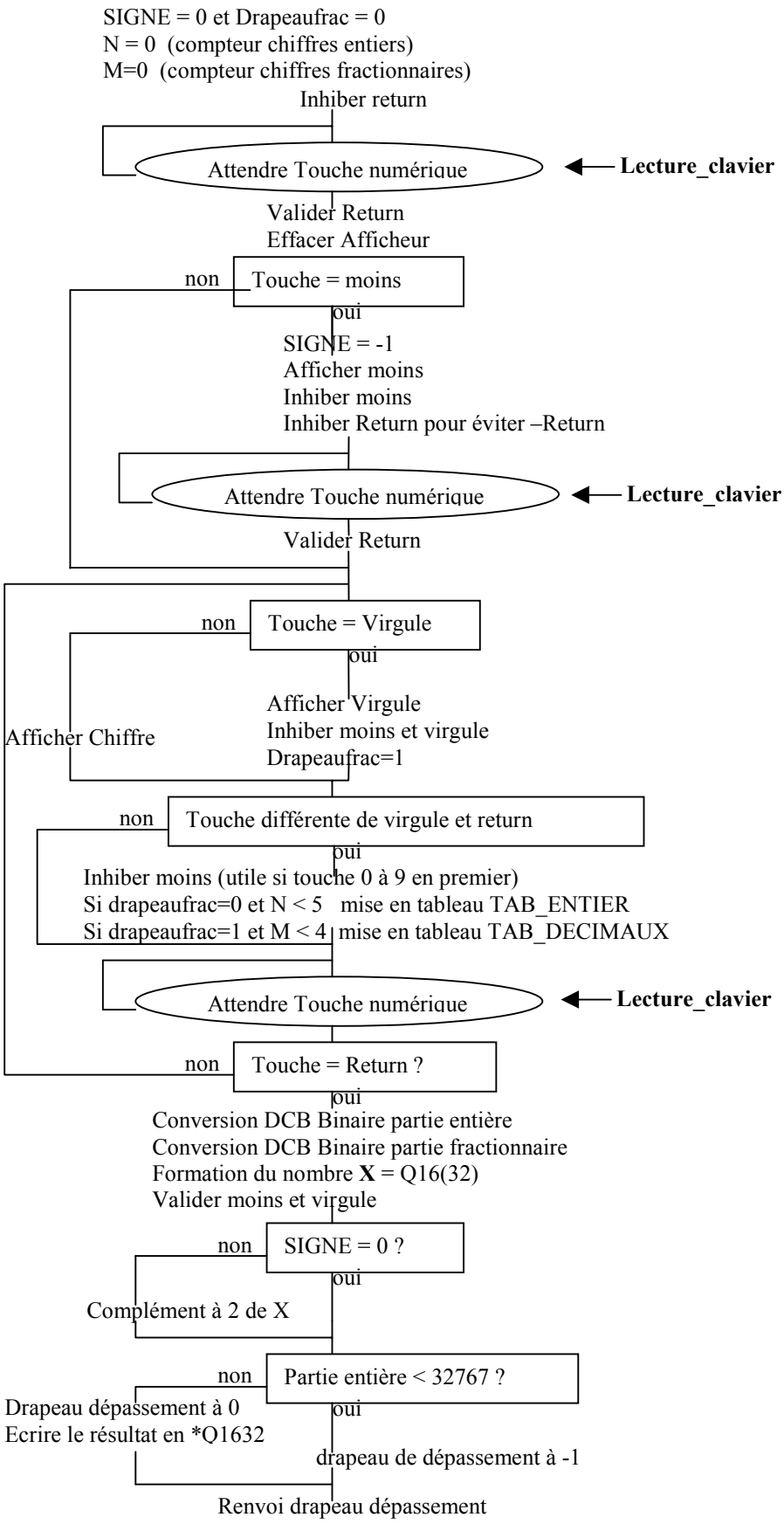
2) Acquisition de valeurs numériques en q1632

Admirer la gestion de tous les cas interdits possibles ! Sinon le reste est finalement très simple. On effectue l'acquisition de la partie entière dans le tableau TAB_ENTIER, et de la partie décimale dans TAB_DECIMAUX.Les conversions se font après.

Même si on frappe plus de chiffres, on note la limitation volontaire à 5 chiffres pour la partie entière, et à 6 pour la partie fractionnaire. Un drapeau de débordement est renvoyé.

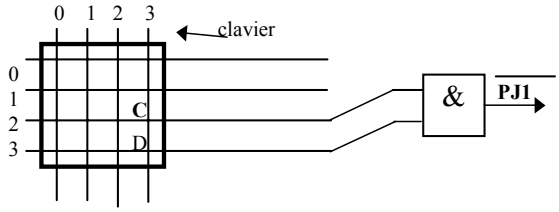
Cette fonction utilise évidemment la fonction lecture_clavier présente et gère l'écho des touches frappées, elle appelle donc une des fonctions d'affichage sur le panneau à cristaux liquides. Une simple fonction de temporisation entièrement logicielle assure les anti-rebonds.

Char acquisition_q1632(long *q1632) ;



20.4.4. Etude du fonctionnement en interruption possible

Selon le câblage et la valeur initiale du code colonne CC envoyé.



D'après ce câblage, les touches : C et D peuvent travailler ainsi en interruption sur PJ1.

Pour cela, **On devra avoir au repos (avant toutes frappes de clavier) CC = 11101111,** assurant un zéro sur la colonne de droite. Les lignes en l'air fournissant 1 en lecture, seules les 2 touches C et D provoqueront un passage à zéro de PJ1.

Une lecture clavier par la fonction précédente char lecture_clavier(char type) permet alors la récupération du code. Cette fonction replace d'ailleurs toujours ce code colonne 11101111 pour de nouveau pourvoir interrompre par C ou D.

Attention : Si on ne se sert pas de cette interruption, il ne faut pas valider PJ1 sur l'HC12, sinon plantage !

20.4.5. Drivers de base du port SPI0 sur interface SPI/parallèle CDP68HC68

Pour les comprendre, il faut se reporter au chapitre traitant du Bus SPI Motorola, et de l'exemple de programmation de l'interface SPI/parallèle.

Une fonction **void init_spi0_hc68_clavier(void) ;** est nécessaire pour initialiser ce port et l'interface. Elle fonctionnerait avec d'autres F_BUS. Elle est appelée dans la fonction ici vraiment nécessaire **ini_carte.c()** !!

On se sert ensuite d'une simple fonction mixte d'entrée_sortie **char io_spi0(char c) ;**

```
#define F_BUS    2     // dans ini_carte.h
void init_spi0_hc68_clavier(void)
{
// Initialisation de SPI0 pour compatible avec circuit IO série parallèle: CDP68HC68P1
 SPI0CR1=0x54; // 0 1 0 1   0 1 0 0    SPI Enable, Maître, Pol 0, phase 1 , Slave Select non géré automatiquement, MSB en tête
 SPI0CR2=0x00; // Slave Select non géré automatiquement, car doit être actif durant deux octets
 if(F_BUS == 2) SPI0BR= 0x01;  //  Spi Baud Rate = 2/4 = 500 KHz
 if(F_BUS == 4) SPI0BR= 0x01;  //  Spi Baud Rate = 4/4 = 1 MHz
 if(F_BUS == 8) SPI0BR= 0x01;  //  Spi Baud Rate = 8/4 = 2 MHz
 if(F_BUS ==16) SPI0BR= 0x02;  //  Spi Baud Rate = 16/8 = 2 MHz
 if(F_BUS ==20) SPI0BR= 0x40;  //  Spi Baud Rate = 20/10 = 2 MHz   - 1 0 0 - 0 0 0
 if(F_BUS ==24) SPI0BR= 0x21;  //  Spi Baud Rate = 24/12 = 2 MHz   - 0 1 0 - 0 0 1
 DDRS=0x80;PTS=0x80;  // Slave Select manipulé par le bit PTS7  (pour transmission par bloc de 16 bits)

// Programmtion du DDR du CDP68HC68P1
 PTS=PTS&0x7f;  // SSn à 0: Sélection du CDP68HC68P1, cablé numéro 0
 io_spi0(0x30);  // Accès à DDR en écriture
 io_spi0(0Xf0); // 4 bits MSB en sortie (scrutation des Colonnes) , 4 bits LSB en entrée (lecture des lignes)
 PTS=PTS|0x80;  // SSn à 1: Déselection de CDP68HC68P1
 // Port colonne à 1110 ---- pour activer travail possible en interruption sur dernière colonne
 // ecrire_colonne(0xE0); à écrire soi même dans application si besoin est
}
```

Remarque : tous les accès à ce circuit se font en deux temps : un octet de commande (avec indication du circuit concerné, 4 possibles sur le même bus, du registre, et du sens de transfert), suivi d'un octet de data. La sélection du circuit doit être maintenue durant les 2 deux accès.

```
// Fonction de lecture écriture du port SPI0
char io_spi0(char octet)
{
static char raz;  // sinon l'optimisateur peut le supprimer !
raz = SPI0SR;   // obligatoire juste avant envoi, sinon envoi ignoré: raz drapeau lecture
SPI0DR = octet;             // envoi data
while((SPI0SR & 0x80) ==0);  // attente sérialisation: drapeau
return(SPI0DR); // et raz drapeau lecture
}
```

20.4.6. Listing des Fonctions du clavier 16 touches, sur SPI0

```
/*****************************************************************************/
/*              FONCTIONS  d'ACQUISITION  au clavier 16 touches              */
/*  Partie entière acquise doit etre de module <  32767    sinon drapeau renvoyé  -1    */
/*  On spécifie ici tout signed char  car par défault les char sont parfois non signé ! */
/*       Cablé sur un CDP68HC68 port // sur port SPI0    Janvier 2004 G. Pallot et Omar  */
/*****************************************************************************/
/*    Colonnes  d7 d6 d5 d4  -  -  -  -     pour C0 C1 C2 C3 - - - - */
/*    Lignes    -  -  -  -  d3 d2 d1  d0    pour L0 L1 L2 L3 - - - -  */
/*    Suppose ports (SPI0 et circuit HC68 déjà initialisés par init_port_clavier()  */
#include "afficheur.h"
#include "registres_hc12.h"

extern char io_spi0(char c);  // fonction d'IO du port SPI0 dans inicarte.c
unsigned long conversion_entiers(signed char n,signed char *tab_entiers);
unsigned long conversion_decimaux(signed char m, signed char *tab_decimaux);
signed char lecture_clavier(signed char type);
signed char position(signed char numtouche);

signed char tab_codes_clavier[] ={1,2,3,0x0b,4,5,6,0x3b,7,8,9,0x3c,0x0f,0,0x0e,0x3d};

/* ========================================================================== */
    /*  Fonction  de lecture clavier, selon le type de touche et avec attente */
/* ========================================================================== */
signed char lecture_clavier(signed char type)
{
signed char NCMAX=3;                    /* numero max de colonnes */
unsigned char NC,NL,NT,CC,CL;    signed char CCSAV=0xff,CLSAV;
int k;  signed char code;    signed char codem,typem;
do
{
 CLSAV = 0xFF;
 while(CLSAV ==-1)                      // Tant que pas d'appui de touche
    {
    CC=0x7F;  ecrire_colonne(CC);
    for (k=0 ; k<=NCMAX ; k++)                  // scrutation de chaque colonne
      {         ecrire_colonne(CC); CL=lire_ligne();
                if (CL!=0xFF)                   // On appui sur une touche
                {         CLSAV=CL;
                          CCSAV=CC;   }
                 else
                {         CC=CC>>1;
                          CC=CC | 0x80;     }
       }
   }
   tempo(10);                                   // Anti rebond à l'appui

/* WAIT RELACHEMENT  ET CALCUL NUMERO DE TOUCHE …   *./
      while (CL!=0xFF)CL=lire_ligne();       // Attente relachement
```

```
      tempo(100);                          // Anti rebond au relachement
      NC = position(CCSAV);          // passage des codes type 1011… au numéro de 0 à 3 (ici 1)
      NL = position(CLSAV);
      NT=NC+(NCMAX+1)*NL;                  // Calcul numéro touche
      code = tab_codes_clavier[NT];  // Lecture du code dans le tableau
      codem = code & 0x70;           // Vérification du type attendu par masque
      typem = type & 0x70;
}
  while ( (codem!=typem)||(code<0));  // On recommence si pas le bon type

  ecrire_colonne(0xEF);   /*  au repos de nouveau 11101111 pour éventuellement colonne 3 en IT */
  return(code);                        // Retour du code de touché final
}

signed char position(signed char code)   // fonction de passage du code au numéro, par décalages gauche
{   signed char i=0;
     while (code < 0)    {   code=code<<1;i++;   }
     return(i);    }

/* ==========================================================================*/
/* Acquisition d'un nombre signé en Q16(32), Le nombre   doit etre de module <  32767,999  */
/*  ==========================================================================*/
signed char acquisition_q1632 (long *q1632)
{
signed char tab_entiers[5],tab_decimaux[4];    /* 5 chiffres entiers max mais faudra < 32767 */
signed char flagfrac = 0;
signed char signe=0;
signed char n=0,m=0,code;
unsigned long l,e,d,f;
signed char flagretour;     /* drapeau depassement */
tab_codes_clavier[15]=tab_codes_clavier[15]| 0x80;    /* inhiber return */
code=lecture_clavier(0);    /* lire une touche avec type numérique et avec attente */
efface_ligne();
tab_codes_clavier[15]=tab_codes_clavier[15] & 0x7f;    /* activer return */
     /* ************************************************** */
if (code==0x0f)          /* 0x0f = code touche signe moins */
   {   signe=-1;
       putch('-');
       tab_codes_clavier[12]=tab_codes_clavier[12] | 0x80;          /*  inhiber moins  */
       tab_codes_clavier[15]=tab_codes_clavier[15] | 0x80;          /*  inhiber return  pour eviter -return*/
       code=lecture_clavier(0);  /*Attente touche numérique */
       tab_codes_clavier[15]=tab_codes_clavier[15] & 0x7f;          /*  valider return  */
   }
do
{
if(code ==0x0e)        /* si code est  virgule */
   {   putch(',');
       tab_codes_clavier[14]=tab_codes_clavier[14] | 0x80;          /*  inhiber virgule  */
       tab_codes_clavier[12]=tab_codes_clavier[12] | 0x80;          /*  inhiber moins  */
       flagfrac = 1;
    }
else    putch(code + 0x30);                /* passage ASCII et envoi afficheur */

if( (code!=0x0e) && (code !=0x0b) )     /* si code different de virgule et return */
   {   tab_codes_clavier[12]=tab_codes_clavier[12] | 0x80;                    /*  inhiber moins  */
       if( (flagfrac ==0) && (n < 5)) { tab_entiers[n]=code; n++; }           /* limitation à 5 chiffres ent */
       if( (flagfrac ==1) && (m < 4)){ tab_decimaux[m]=code; m++; }  /* limitation à 4 chiffres frac */
   }
```

```
      code = lecture_clavier(0);
}
while(code != 0x0b);              // Tant que pas Return
      e=conversion_entiers(n,tab_entiers);   d=conversion_decimaux(m,tab_decimaux);
      f=e<<16;
      l=f+d;
if(signe==-1) l=-l;
tab_codes_clavier[14]=tab_codes_clavier[14] & 0x7f;  /* réactiver virgule */
tab_codes_clavier[12]=tab_codes_clavier[12] & 0x7f;  /* réactiver moins */
if(e <32768) { *q1632 = l; flagretour = 0;}
else flagretour = -1;    /*  cas de dépassement */
return(flagretour);
} /* ici pas de conversions possibles car complementation */

/* ================================================================== */
/* fonctions de conversion DCB  ⇒ BINAIRE */
/* ================================================================== */
/* conversion de la partie entière */
unsigned long conversion_entiers(char n, char *tab_entiers)
{
char k;   unsigned long x=0;      /* calcul en 32 bits pour detecter si depassement 16 bits */
for(k=0 ; k<n ; k++) x=10*x+tab_entiers[k];
return(x);
}
/* conversion de la partie fractionnaire */
unsigned long conversion_decimaux(char m, char *tab_decimaux)
{
char k;   unsigned long x=0;    unsigned int div=1;
for(k=0 ; k<m ; k++) x=10*x+tab_decimaux[k];
x = x <<16;          /*  x=x*pow(2,16); */
for(k=0 ; k<m ; k++)div=div*10;    /* création du terme en 10 puissance  */
return ( x/div +1);    ,-------- -------- en Q16(16) et 00000000 00000000,-------- -------- dans un long
}
/* ================================================================*/
/* Nécessairement en assembleur :
       la fonction void Tempo(unsigned int ms)  dèjà étudiée, s'y reporter      * /
/* ================================================================*/

 /* ================================================================== */
/*  Les Drivers de second niveau    (autre que char io_spi0(char) ;  déjà vu)
/* ================================================================== */
/*  ***** Les drivers du port utilisé  SPI0 et circuit d'interface SPI_Parallèle: CDP68HC68  **** */
 void ecrire_colonne(signed char c)
 {
 PTS=PTS&0x7f;                //sélection interface pour 2 accès
 io_spi0(0x10);                        /*C04(R/W|) = 1 => commande écrire une donnée  */
 io_spi0(c);  /* La donnée : C0 C1 C2 C3 1 1 1 1  on pourrait étendre à 8 colonnes avec autre matériel  */
 PTS=PTS|0x80;                //désélection interface
 }

 char lire_ligne(void)
 {
 char k;
 PTS=PTS&0x7f;                //sélection interface pour 2 accès
 io_spi0(0x00);                        /*C04(R/W|) = 0 => commande lire une donnée  */
 k=io_spi0(0x00);             /* lecture de la donnée */
 PTS=PTS|0x80;                         //désélection interface
 return(16*k + 15);      }          /* L0 L1 L2 L3 1 1 1 1  on pourrait étendre à 8 lignes avec autre matériel  */
```

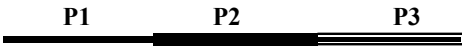
21. NOTIONS SUR MULTITACHE ET TEMPS REEL

21.1. Quelques définitions

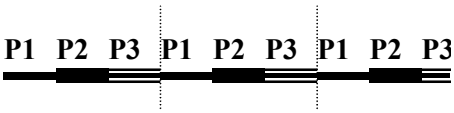
Multitache

Un programme traditionnel n'occupe jamais à 100 % le processeur (exemple : entrée sortie sur périphérique, attente lecture clavier …). D'où l'idée est d'utiliser ce temps perdu pour d'autres tâches. But final: occuper le plus efficacement possible le processus et toutes les ressources du système. Soit Pi divers processus, ou tâches:

Monotâche

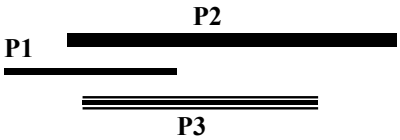


Multitâche Mono processeur



parallélisme virtuel (traitement de chacune à tour de rôle très rapidement). On a l'impression qu'elles s'exécutent simultanément.

Multitache Multiprocesseurs



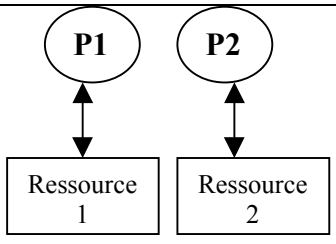
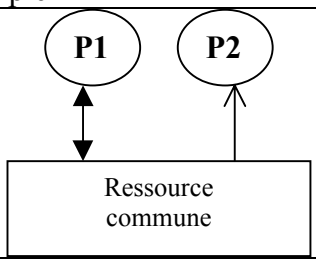
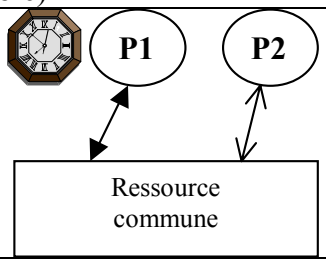
Vrai parallélisme possible ici.

Programme et Processus

Un processus est un programme qui une fois chargé en mémoire peut s'exécuter.
Un processus est dynamique par rapport un programme qui est statique.
Un processus peut être **crée**, **exécuté**, **mis en veille**, **détruit**.
Il possède:
Une zone programme (code et constantes).
Une zone de données.
Une zone de pile (variables locales, passage de paramètres, PC lors des sous programmes, etc …). La zone de donnée et la zone de pile peuvent être confondues.

Plusieurs processus peuvent être chargés en mémoire dans le système.

Partage de ressources (ce peut être des *périphériques*, mais aussi une *zone mémoire commune*, ou de *simples variables globales* communes à plusieurs fonctions.…)

Processus indépendants	Processus et ressource commune en relation simple	Processus liés, à Ressource commune (contraintes de temps possible)
		
Exemple: mesure d'une température et régulation d'une vitesse.	Exemple: gestion de stock par P1, P2 ne servant qu'à la consultation. P2 n'est pas bloqué si P1 travaille.	L'un des processus ou les deux peuvent être bloqués par l'autre, en attente d'information pour continuer.

Pour des cas très simple (mise à jour d'une variable par un processus, et lecture par l'autre) une gestion d'occupation de la ressource n'est pas nécessaire.

Le plus souvent, une gestion par **sémaphores** est indispensable.

Le plus simple des sémaphores est un simple **drapeau d'occupation de la ressource** (un peu comme dans une ressource partagée entre deux processeurs), mais plus généralement, un sémaphore se présente sous forme de compteurs et de files d'attente. Dans tous les cas il faut éviter le «dead lock », phénomène où les deux processus seraient tous les deux en attente de la ressource, sans déblocage possible !. Toujours prévoir un « time_out » (un message ou autre chose) signalant et ou corrigeant si cette situation anormale survient.

Synchronisation par **événements** :

Des variables booléennes (octets drapeaux) testées par les processus permettent de se synchroniser lorsque l'un des processus attend par exemple un résultat de l'autre.

Boites aux lettres :

Méthode courante de communication de données entre tâches. On associe deux files d'attente (tampons FIFO): pour les messages, pour les tâches. Un Handshake logiciel est géré.

Temps réel :

Un système temps réel doit pouvoir suivre toutes les contraintes temporelles en provenance de l'extérieur (débit d'information à traiter par exemple), il doit être prêt à tout moment à répondre à une requête de l'environnement extérieur dans un temps maximum déterminé. Exemple : pilotage de processus industriels, systèmes embarqués.

Le temps réel peut aller de quelques micro secondes pour certains traitements (calculateur de trajectoire en aviation, compression et décompression d'image ….) à quelques dizaines de millisecondes ou d'avantage (un traitement de texte est un système temps réel vis à vis de la cadence de frappe sur les touches du clavier !)

Cette notion est relative !

Condition du temps réel:

Processeur et logiciel **suffisamment rapides** pour suivre les exigence de l'environnement extérieur, pour les tâches principales ainsi que pour les requêtes d'urgence.

Pour les cas difficiles, possibilité de **préemption**: (interruption d'une tâche pour une autre plus urgente).

21.2. Bases d'un système multitâche et temps réel

21.2.1. Petits systèmes sans "exécutif"

On peut concevoir aisément des petits systèmes multitâche et temps réels gérant quelques tâches déterminées à l'avance, les priorité étant gérées par interruptions. Exemple : commande de moteur pas à pas simple avec en parallèle affichage sur écran de sa vitesse et d'une température.

On peut aborder le plus souvent de tels systèmes sans parler de processus ni de multitâche ! Le temps réel consiste simplement à assurer par exemple une vitesse maximale pour le moteur, et une vitesse régulière. Une étude doit évidemment s'effectuer pour chaque application.

Avantage: **simplicité,** et pour ces petits systèmes, on arrive à optimiser facilement le temps d'exécution si nécessaire.

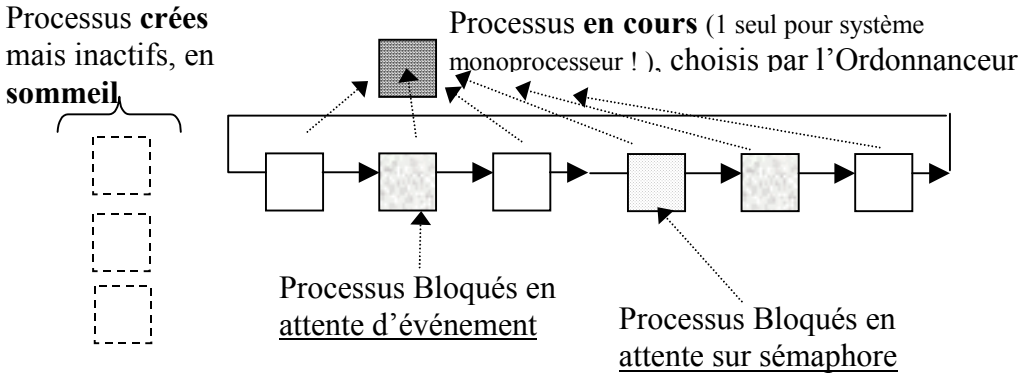
Le problème se complique pour de plus gros systèmes informatiques ou les tâches sont plus nombreuses et pouvant évoluer au cours du temps.

21.2.2. Exécutif multitâche et temps réel

Nous allons donc présenter une façon bien plus générale d'aborder une application informatique, et de gérer la parallélisme de plusieurs tâches. Cette méthode permet, à partir d'un exécutif temps réel, de concevoir de grosses applications, d'une manière très souple.

21.2.2.1. Les files d'attentes de tâches

Les tâches ou processus sont organisés en file d'attente en attente du processeur.



La **file des processeurs actifs** est une liste qui se modifie régulièrement au cours du temps:

Quand une tâche est volontairement ajoutée ou supprimée.

Quand une tache se termine d'elle même.

Le processus en cours : c'est celui qui est en cours de traitement à un instant donné. Il est choisi par l'**ordonnanceur** dans la liste des processus actifs dans l'ordre, ou en respectant une priorité.

Les processus bloqués en attente d'événement ou de sémaphore sont à priori gérés sans notion de priorité.

21.2.2.2. L'ordonnanceur

C'est une fonction qui fait partie de l'exécutif temps réel et qui est le chef d'orchestre du système. C'est lui qui choisit parmi les processus chargés et actifs ou « éligibles », lequel sera mis ¨en cours¨ (élu) prochainement.

Critères de sélection :

- Garantir à chaque tâche un temps d'allocation donné.
- Respecter des priorités et respecter un temps de réponse donné
- Eviter tout blocage du système !
- Attribuer un processeur à un processus donné (cas des systèmes multi processeurs).

Techniques non préemptives de choix d'un processus dans la file d'attente

On suppose que le processeur doit exécuter diverses taches de temps d'exécution plus ou moins longues et donc qui se terminent d'elles mêmes.

Techniques circulaire

Pas de priorité entre les processus ni de critères temporels.

Technique du plus court temps d'exécution (si on connaît la durée).

Si des processus courts arrivent sans arrêt, les processus longs ne sont jamais traités !

Technique du plus temps de traitement résiduel :

Si une tâche arrive avec un temps d'exécution plus court que le temps restant pour la tâche en cours, cette dernière est interrompue par « préemption » (réquisition), au profit de la nouvelle.

Inconvénients :

Pour les deux dernières techniques : il faut connaître les temps d'exécution des taches avant leur arrivée dans le système, ainsi que le temps restant, peu pratique dans les environnement temps réel qui ont des contraintes externes.

Si une tache plante ou restée bloquée en attente de ressource qui ne se libère pas, ou d'événement qui n'arrive jamais, ou s'il s'agit d'une tâche bouclant sans fin, le système est bloqué ! (L'ancien Windows 3.11 était un système pseudo multitâche non préemtif, une tâche pouvait bloqué tout le système!). Il faut au minimum une possibilité d'annuler de manière « préemtive » une tâche qui monopolise trop le processeur.

Technique du temps partagé (Time sharing)

Cette technique d'ordonnancement est **délibérément préemptive**, chaque processus est interrompu à une cadence régulière **Tcad** pour exécuter le suivant. Cette cadence est crée par une interruption extérieure (Timer par exemple, nommé Horloge temps réel).

On peut affecter un même quantum de temps pour chaque processus, ou des différents. Tous les processus peuvent être dans une seule file d'attente. **Un processus bloqué** (sur attente de sémaphore ou d'événement) **ne bloquera pas le système** car a la fin de son quantum de temps, il se retrouvera comme les autres dans la file et le suivant reprendra son exécution. **Une tache terminée disparaît toute seule** de la file d'attente.

Choix du quantum de temps:

Trop petit, on perd trop de temps dans les commutation de tâches.

Trop grand, on retarde trop la réponse des taches en attente dans la file.

Quelques millisecondes sont des valeurs fréquentes. On peut avoir des files différentes, des quantum différents. Le quantum de temps peut être réduit pour des taches bloquées en attente de libération.

C'est la technique utilisée pour de nombreux exécutifs. Elle nécessite évidemment des **fonctions** toutes **interruptibles** et **réentrantes** (une fonction interrompue peut très bien être utilisée par l'autre tache !)

21.2.2.3. Commutation des tâches

Il faut :

Sauver registres du processeur et pointeur de pile du processus courant.

Lancer un nouveau processus, en restaurant les registres et le pointeur de pile relatifs à celui ci, s'il s'agit d'une reprise d'exécution.

Attention aux interruptions durant la commutation de tâche, à gérer correctement !

Elle s'effectue :

Après choix par l'ordonnanceur (dans la file d'attente).

Sur phénomène externe.

Sur demande explicite de la tâche elle même (fin de tâche par exemple), elle doit alors d'elle même s'enlever de la file d'attente.

21.3. Système multitâche à temps partagé

21.3.1. Problème de temps réel

La **technique du temps partagé** est parfaite pour le multitâche, mais pour le temps réel, il faut assurer un déterminisme sur la vitesse d'exécution d'une part, mais aussi entre une requête et sa réponse, et selon le nombre de tâches dans la file d'attente, ces valeurs peuvent évoluer, et il faut en connaître les limites.

Par nature même du multitâche monoprocesseur, celui ci ne pouvant effectuer en réalité deux choses en même temps, il en résulte un **ralentissement général augmentant avec le nombres de taches** (calcul plus précis effectué ultérieurement).

21.3.1.1. Priorité de processus

On ajoute une **priorité pour chaque processus**, afin que ceux de priorités plus élevées soient choisis plus souvent que les autres par l'ordonnanceur.

On améliore ainsi la vitesse de certaines tâches, au détriment des autres …

Mais pour des tâches ou les contraintes du temps réel sont sévères, ce ne sera pas suffisant.

Etude plus détaillée de cette technique:

A la création de la tache, on lui décerne une **priorité** (de 1 à 100 par exemple).

On utilise ensuite l'**age** d'un processus :

Au départ, dans la file d'attente, on initialise l'age du processus à la valeur indiquée dans sa variable priorité (de 1 à 100 par exemple).

L'ordonnanceur à chaque fois incrémente l'age de tous les processus et lance celui dont l'age est le plus élevé.

Tout processus une fois lancé retrouve un age égal à sa priorité.

Exemple : soit trois processus P1, P2, P3 de priorité **30 45 25**

A chaque exécution de l'ordonnanceur, on aura successivement ages et actions suivantes :

P1	31	32	----	46	47→**31**	32	
P2	46→**45**	46→**45**	----	46→**45**	46	47→**45**	
P3	26	27	----	41	42	43	
	P2 lancé	P2 encore	-----	P2 encore	P1	P2 de nouveau	Etc.. Bientôt P3 passera !

Une priorité (c'est la **valeur relative** par rapport à celle des autres processus **qui compte**, et nom la valeur absolue) revient alors à un certain pourcentage de fois élu . Entre une priorité 100 et 90, il y a un rapport de 10.

21.3.1.2. Synchronisation sur événement

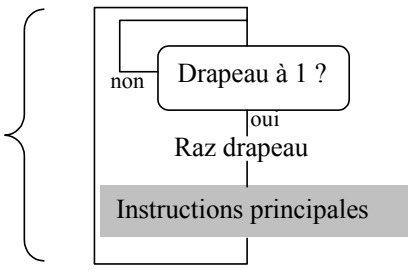
Le drapeau événement peut être positionné

Par une autre tache

Par un phénomène extérieur (drapeau de circuit d'entrée sortie, de Timer …)

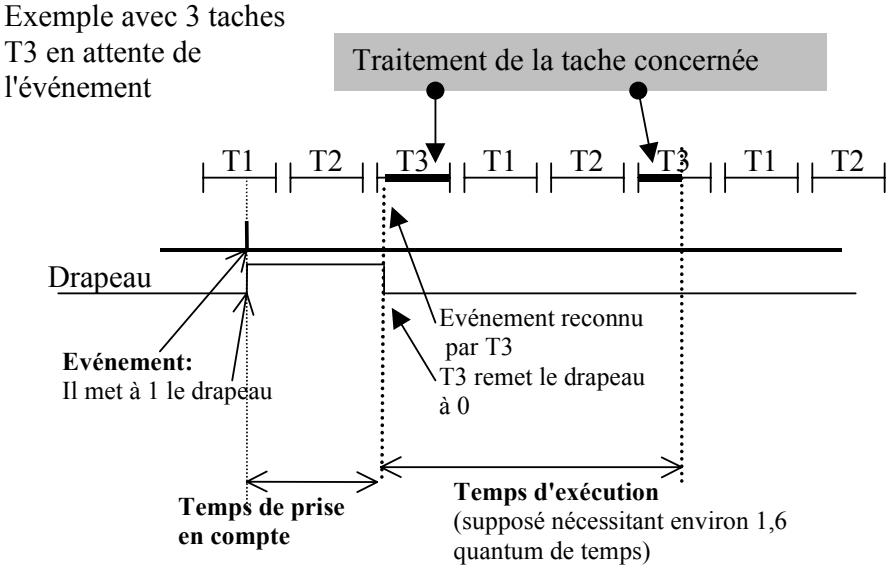
On travaille **sans interruption**

La **tache concernée** est à priori en attente de cet événement, d'ou son organigramme:



On observe deux phénomènes s'aggravant avec le nombre de taches:

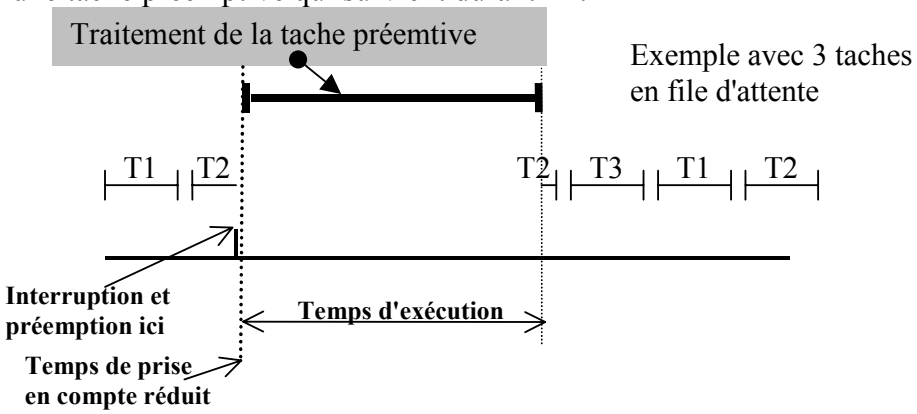
Un **retard de prise en compte**
Un **ralentissement de la tache** si celle ci dure plus d'un quantum de temps



Remarques dans le cas ou les **événements** deviennent **trop fréquents**:
Les autres taches du système ne sont pas ralenties.
Mais la tache concernée laissera passer des évènements sans pouvoir les traiter (perte de synchronisme).

21.3.1.3. contraintes de temps sévères

Recourt à une **préemption pure et simple** en dehors du choix normal de l'ordonnanceur, pour retrouver une action déterministe. On voit un exemple de 3 taches s'exécutant en temps partagé, et d'une tache préemptive qui survient durant T2.



Performances:
Bon déterminisme: temps de prise en compte très court et aucun ralentissement pour cette tache.
Mais tout le reste du temps partagé est ralenti !

Si plusieurs taches travaillent ainsi, on peut jouer sur les niveaux d'interruptions (facile sur certains processeurs), un tache d'interruption quelconque pouvant alors être elle même interrompue par des interruptions de niveau plus élevées.

Evidemment plus ces tâches sont longues et fréquentes, plus les autres taches travaillant en temps partagées sont ralenties. Les performances du système se dégradent forcément dès que la « charge de l'unité centrale » devient trop importante. Il y a une limite fonction de la puissance du processeur.

Remarque, interruption et évènement ?

On peut créer une tache préemptive très courte juste pour positionner un drapeau « événement interruption », cet événement est alors traité par la tache concernée qui doit être en attente de cet événement, comme précédemment .

Avantages :

Pratique pour des évènements ne pouvant pas travailler sans valider l'interruption correspondante

Le système n'est pas ralenti, mais le déterminisme est perdu : les temps de prise en compte et d'exécution sont identiques à la synchronisation sur événement vue précédemment.

21.3.1.4. Problème de la génération de cadences, ou d'intervalles de temps déterminés

En monotache, on pouvait générer des intervalles de temps précis soit par de simples boucles (fonctions écrites en assembleur !) ou par Timer sans interruption, le processeur ne faisant rien d'autre durant ce temps, soit aussi par Timer et interruptions.

En temps partagé, toute fonction de temporisation par boucle de programme est plus longue à s'exécuter, la durée de temporisation augmentera avec le nombre de taches. Modifier les priorités améliore un peu les choses, mais il est un peu absurde d'augmenter la priorité de taches qui attendent, au détriment de taches qui font effectivement quelque chose... et de toute façon, la durée de l'intervalle de temps généré n'est pas fiable.

Deux cas peuvent se présenter :

Pour des Intervalles de temps dont juste l'ordre de grandeur importe

Dans des cas très simples, et uniquement pour celles ci, on peut à la limite se servir de simples boucles de temporisations lorsque juste l'ordre de grandeur importe (anti rebond de clavier, messages à afficher durant quelques secondes sur un afficheur …), des augmentations de plus de 100% du retard peuvent en effet survenir.

Intervalles de temps plus précis : Timer travaillant sur « événement Timer »

Application à la génération d'une cadence régulière mais pas trop rapide :

Avantages :

On travaille sans tache préemptive, et sans ralentir le reste du système.

Erreur nulle sur la cadence moyenne, si celle ci n'est pas trop rapide.

Inconvénients :

Fluctuations (possibles de plusieurs dizaines de millisecondes) sur le retard de prise en compte.

La tache lancée est ralentie si elle dure plus d'un quantum de temps.

Entre deux cadence Timer, il faut assurer le temps de prise en compte et de traitement, sinon il y a perte de synchronisme.

Intervalles de temps très précis ou (et) plus courts : Timer et interruption

C'est le problème des taches à contraintes de temps sévères, elles doivent agir par préemption directe, comme pour les système travaillant sans temps partagé.

Application à la génération d'une cadence régulière :

Avantages :

Cette cadence peut être rendue prioritaire sur tout le reste.

Inconvénient :

Ralentissement des autres taches

Attention toutefois, il peut exister dans le système des interruptions de niveaux supérieurs pouvant interrompre un processeur déjà parti en interruption…

21.3.2. Essai d'estimation du déterminisme dans un système à temps partagé

On choisira comme exemple numérique :

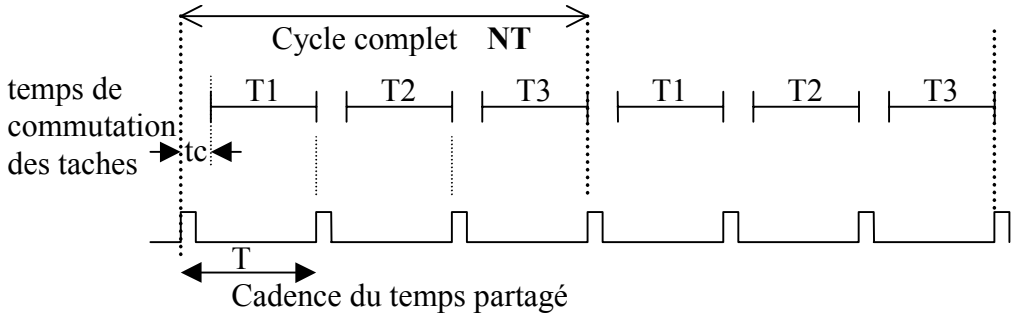
une cadence de temps partagé : **T = 4ms**

un temps de commutation de tache **tc = 0,18ms**

21.3.2.1. Estimation du ralentissement

Ordonnanceur sans priorité (ou priorités égales)

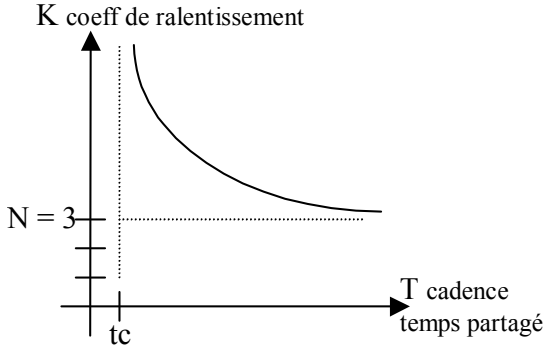
Considérons un **ordonnanceur sans priorité**, alternant N=3 taches T1, T2, T3 au rythme de T. Le temps perdu au moment la commutation non instantanée des taches est tc.



Durant un cycle complet de durée N.T une tache Ti s'exécute réellement pendant une durée T1 = T2 = T3 = T- tc.

Pour chaque tâche, on peut donc définir un coefficient d'allongement de temps **k** égal à

$$k = \frac{N.T}{T - tc}$$



On remarque la limite inférieur de la période T du temps partagé, il faut choisir T >> tc sinon on ralenti trop les taches.

Application numérique:

Pour notre système et pour 3 taches : Il vient **k** = $\frac{3T}{T - tc}$ = **3,14** !

Ordonnanceur avec priorité

Un calcul plus général est compliqué, on choisit ici 3 taches et les priorités:

T1 P=100

T2 P' =90

T3 P' = 90 (identique pour T2 et T3)

T1 est élue environ P- P' = 10 fois avant que les âges de T2 et de T3 atteignent 100, et que ces taches soient élues.

Un cycle complet donne T1 T1 T1 ….. T1 T2T3 puis recommence T1 T1 T1 …. T1 T2T3, et se déroule donc sur (P-P'+2)T = 12T

Exemple simple ci dessus:	T1	T2	T3
Durée réelle d'exécution	10(T-tc)	T-tc	T-tc
Coefficient k d'allongement	$\frac{(P-P'+2)T}{(P-P')(T-Tc)} = \frac{1{,}2T}{(T-tc)}$	$\frac{(P-P'+2)T}{T-tc} = \frac{12T}{(T-tc)}$	$\frac{12T}{(T-tc)}$

T1 tourne donc alors presque à la vitesse voulue **k = 1,25** au lieu de 3,14 !

Mais **T2 et T3** sont ralentis fortement ! **k = 12,5** ! ! !

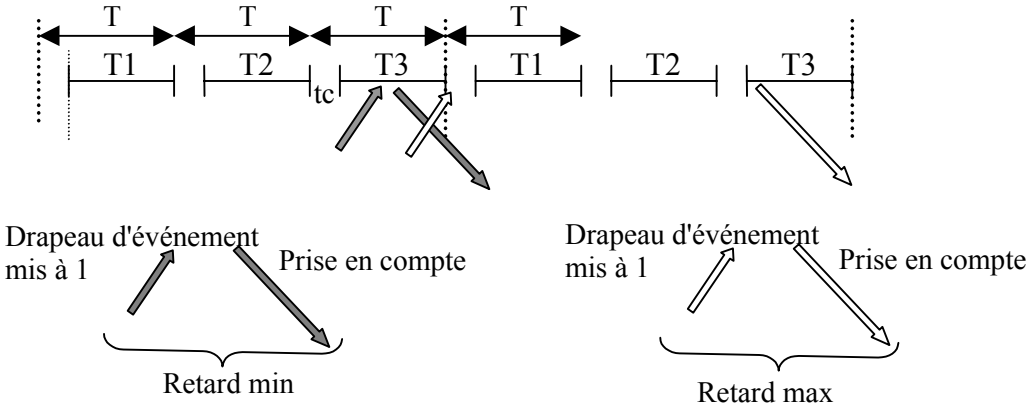
Remarque: si P-P' est encore plus grand, et avec tc petit, k se rapproche de 1 et donc T1 de sa vitesse max possible comme si la tache était seule. Mais en échange T2 et T3 ne s'exécutent pratiquement plus, donc c'est idiot et il y a un compromis à trouver!

21.3.2.2. Estimation du retard à la prise en compte d'un événement.

Supposons qu'un drapeau événement passe à 1, ce peut être un drapeau mis à un par une tache quelconque pour synchroniser une autre, ou un drapeau en provenance d'un circuit périphérique d'entrée sortie, travaillant **sans interruption** (par sondage). La tache concernée par cet événement est à priori en attente de celui ci, et doit s'en apercevoir le plus vite possible, si on veut assurer le "temps réel"

Cas de N processus à priorité identique.

On suppose la tache concernée (ici T3) en attente de cet événement (par boucle).



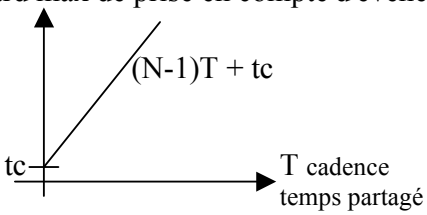
Pour N taches, de même priorité on a

$Retard_{min}$ = temps de reconnaissance par T3 (faible, quelques cycles d'exécution)

$Retard_{max}$ = (N-1)T + tc

(pour notre exemple 2T + tc = 8,2ms)

Retard max de prise en compte d'événement



Taches moins prioritaires

Il est évident que ce retard augmente si la tache T3 est moins prioritaire. Dans le cas simple de priorités P = 100 pour T1, P' = 90 pour T2 et T3, l'alternance des taches étant

T1 T1 T1 T1 T1 T1 T1 T1 T1 T1 T2 T3 T1 T1 T1 T1 T1 T1 T1 T1 T1 T1 T2 T3

Le retard max devient environ 11T.

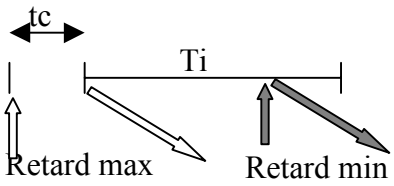
21.3.2.3. Estimation du retard lors de la préemption directe par interruption

On lance une tache sans passer par l'ordonnanceur, et celle ci arrête à priori le temps partagé le temps de son exécution.

$Retard_{min}$ presque immédiat (le temps de partir interruption, quelques micro secondes.

$Retard_{max}$ environ tc (moins fréquent mais à ne pas négliger !) (ici 0,18ms)

(sauf si le changement de tache peut être rendu lui même interruptible par niveau plus élevé ou des astuces de programmation)



Si ces interruptions sont trop fréquentes ou trop longues, le reste du système est évidemment ralenti.

21.3.3. Conclusion

Il n'existe pas de technique d'ordonnancement universelle et parfaite. Mais le technique du ***temps partagé*** est ***la plus performante***, et on adapte selon le cas :

Système multi utilisateur: on assure un partage équitable entre les utilisateurs.

Système de contrôle de processus industriel: nécessité d'une gestion correcte de priorité avec de techniques préemptives à priorité maximales.

Petits systèmes, ou le nombre de tâche est fixe ou très réduit : un exécutif temps réel n'est évidemment pas indispensable! et même parfois n'apporte rien (surtout si on veut minimiser la taille du code objet ..), et on peut très bien heureusement programmer au cas par cas, comme nous l'avons déjà fait remarquer !

21.3.4. Moniteur temps réel

Un moniteur (ou noyau) temps réel contient toute une série de fonctions assurant la gestion de files d'attente des tâches et l'ordonnancement, ainsi que des fonctions de gestion des ressources du système.

Exemple
- OS9 de microware (un peu moins utilisé maintenant)
- VME Exec motorola
- RTX de venturCom (pour Windows NT et CE)
- ……..

21.4. Mini exécutif "scolaire" multitâche à temps partagé

On choisit la technique du ***temps partagé***, avec un quantum de temps identique pour les tâches. Nous donnerons ici un petit logiciel en C, tournant **sur HC12**, et permettant un simple multi tâche, tout d'abord sans gestion de priorité.

Le lecteur y trouvera également des fonctions de gestion de listes chaînées (ajouter une fiche, retirer, visualiser …).

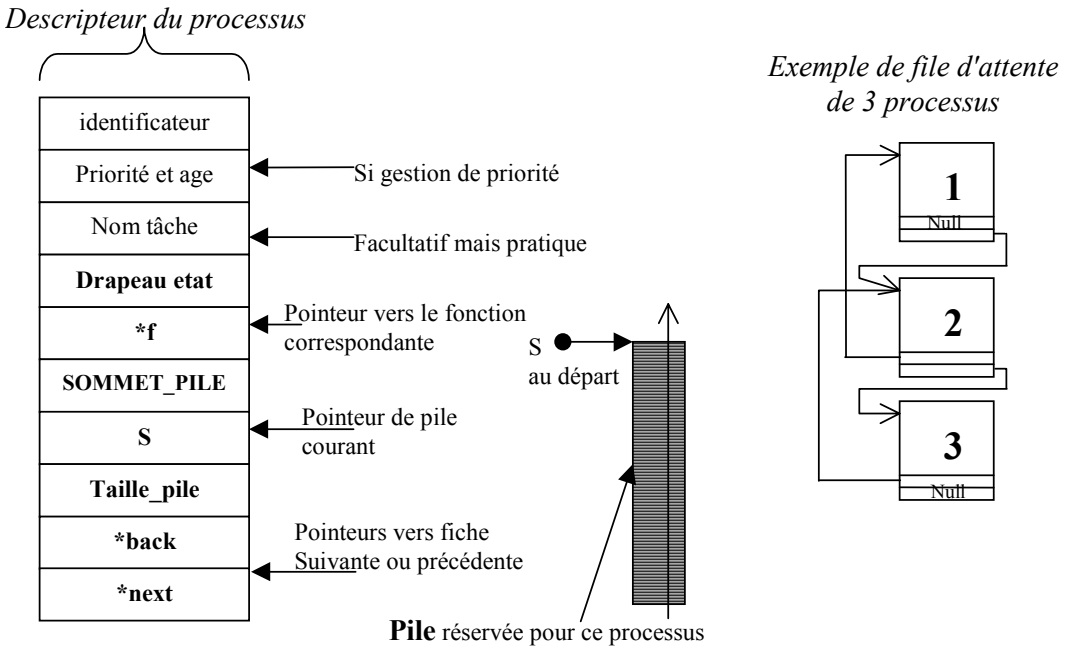
21.4.1. File d'attente, descripteur de processus.

Les renseignements correspondant à chaque tâche sont rassemblés dans un descripteur de processus, qui est en fait une fiche de renseignement. Cette fiche contient également la pile du processus, et donc toutes ses variables non partagées.

En C, une structure **tache** correspond au modèle de fiche. Elle est décrite dans le fichier multi.h.

Pour un processeur 8 bits de nombreuses valeurs sont des char, et les adresses des int. Elles seraient des int et les adresses des long int pour un 16 bits.

```
struct tache
{
   char     id;            /* identificateur de processus, unique à la création */
    char    prio;          /* priorité de 1 à 100 */
    char    age;           /* age */
    char    *nom;          /* pointeur vers chaîne de caractères: nom unique du processus */
    char    etat;          /*drapeau état du processus */
    void    (*f)(void);    /* pointeur sur le processus */
    unsigned int           SOMMET_PILE;     /* PILE sommet */
  unsigned int   s;                         /* pointeur de pile courant */
  unsigned int   taille_pile                /* taille de la pile du processus en octet */
    struct  tache *back;          /* vers la fiche suivante */
    struct  tache *next;          /* vers la fiche précédente */
};
```



La file d'attente des tâches actives est une liste doublement chaînée (usage des pointeurs **back** et **next).** Le double chaînage avant arrière permet un accès rapide à un processus.

La création d'une tache consiste à initialiser une **fiche** avec les renseignements de la tâche, la tache est alors en « sommeil » mais non active. Elle n'est pas encore dans la file d'attente.

On l'active alors en la plaçant dans la liste chaînée.

Le drapeau état sert pour différencier une tache encore jamais activée (état = 0) d'une tache déjà active (état = 1). La procédure sera un peu différente pour lancer la tache une première fois (pas de contexte reprendre) et pour la relancer pour un nouveau quantum de temps.

Une tache retirée de la file d'attente est remise en sommeil, on doit par contre replacer dans sa fiche de renseignement, ses caractéristiques initiales (age = priorité, pointeur de pile au sommet de sa zone réservée, état du processus : pas encore lancée), si on veut l'activer de nouveau !

L'allocation dynamique (par Malloc) permet de réserver de la place en mémoire progressivement selon les besoins. On peut détruire par ailleurs les fiches des taches non utiles pour libérer de la mémoire. On pourrait aussi retirer complètement de la mémoire les sections de type code et de type données correspondant à ces taches (possibilité non étudiée ici).

L'ordonnanceur correspond une fonction **struct tache * ordonn()** qui est lancée à la cadence d'une horloge temps réelle générée par le **Timer TC6 de l'HC12**, et retourne un pointeur vers la tâche à reprendre.

La **cadence** choisie du temps partagé est **T = 4 millisecondes**

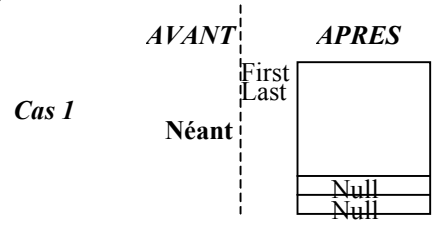
On mesurerait un **temps de commutation** de tache de **tc = 0,18ms**

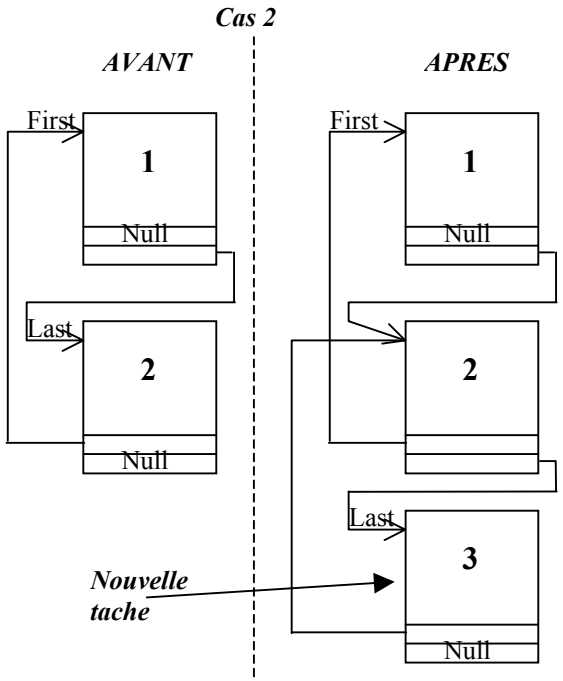
21.4.2. Fonctions du mini exécutif

21.4.2.1. Fonctions de création et de mise en file des processus

```
    #include <stdio.h>
#include <stdlib.h>
#include  "multi.h"
#include  <string.h>
#define hc11_reg 0x1000
#define TMSK1 *(char*)(hc11_reg + 0x22)                registres du TIMER de l'HC11
#define TCNT *(unsigned int*)(hc11_reg + 0xE)
#define TOC1 *(unsigned int*)(hc11_reg + 0x16)
#define TFLG1 *(char*)(hc11_reg + 0x23)

extern void valider_it(void);

char chercher_dans_file(struct tache  *tache);
void lancer_premiere(struct tache *tch);
extern void lancer(void);                      /* en  assembleur */
extern void sauver_s(void);                    /* en  assembleur */
extern void recuperer(void);                   /* en  assembleur */
```

```
 /********  variables globales de travail  propres à l'exécutif  **********/
extern struct tache *tache_courante;              /* pointe sur tache courante */
extern struct tache *tache_precedente;            /* pointe sur tache precedente */
extern struct tache *first;                       /* premiere de la file */
extern struct tache *last;                        /* derniere de la file */
extern char   id;                        /* numero identificateur de tache */
int snouv, ssauv,premier_pc;       /* variables globales de passage */
/* ------------------------------------------------------------------------------ */
/*                  Creation suppressions … des taches                            */
/* ------------------------------------------------------------------------------ */
struct  tache *creer( void (*f)(void), char prio, char *chaine, unsigned int taille_pile)
                                                  /* retourne un pointeur vers la fiche tâche  */
{
struct tache *p;
char *bas_pile;
p = (struct tache *)malloc( sizeof(struct tache));          /* allocation dynamique pour les données du
                                                            descripteur de tâche   */
p->id = id;
id=id+1;                                 /* l'identificateur s'incrémente à chaque tache crée */
p->prio = prio;
p->age = prio;
    p->nom = (char*)malloc(strlen(chaine)+1);    /* alloc. dynamique pour la description */
    strcpy(p->nom, chaine);
p->etat=0;      /* tache pas encore élue une seule fois  */
p->f = f;       /* adresse de la fonction */
bas_pile = (char )malloc(taille_pile);              /* allocation dynamique pour la pile  */
p->SOMMET_PILE = (unsigned int)(bas_pile+taille_pile-1);      /* calcul haut de la pile */
p->s = p->SOMMET_PILE ;                  /* S initialisé au sommet de la pile */
p->taille_pile = taille_pile ;                    /* taille pile rangée dans la fiche */
return(p);
}

/* ------------------------------------------------------------------------------------------------------------------------
--*/
                        /* Active la tâche par mise en file d'attente */
 void entrer(struct tache *p)
{
if (chercher_dans_file(p)== 0)
   {
      if(first==NULL)first=p;
      if(last==NULL) p->back = NULL;
      else
      { p->back = last;
      last->next=p;
      }
      last = p;
      p->next = NULL;
   }
}
```





```
                    /* Chercher si une tache est déjà en file d'attente */
char chercher_dans_file(struct tache  *tache)
{
struct tache *p;
p=first;
if( (p!=NULL) && (tache !=NULL) ;  /* si liste non vide et pointeur tache non NULL */
{
   while( (p !=tache)  && ( (p=p->next) !=NULL) ) ) ;          /* recherche de la fiche */
   if(p= =tache)return(1);     /* retourne 1 si trouve */
   else return(0);          /* pas trouve */
}
else return(0);          /* file vide */
}
```

Remarquer la ligne de recherche de fiches :

```
                        while( (p !=tache)  && ( (p=p->next) !=NULL)  ) ;
```

Elle signifie: attente tant que p est différent de tache **ET** p différent de NULL et à chaque fois p pointe la fiche suivante.

On sort de la boucle : Si la fiche est trouvée (p est égal alors à tache)
Ou si la fiche n'est pas trouvé en en fin de liste (p->next est égal alors à NULL)

```
/* ----------------------------------------------------------------------------------------------------------------*/
void extraire(char id)            /* mise en sommeil d'une tache de numéro id*/
{
struct tache *p;
char flag=0;
p=first;                    /* pointeur sur la première fiche */
if(p!=NULL)
while( (p->id != id)  && ( (p=p->next) !=NULL)  );   /* recherche de la fiche id  */

if(p!=NULL)
{
   p->etat = 0 ;             /* le processus extrait doit pouvoir être replacé en file, on le remet donc */
   p->s = p->SOMMET_PILE ;    /* dans son état initial, comme lors de la création de sa fiche */
   p->age = p->prio ;

   if(p->back !=NULL)                    /* cas  1  et 2  */
   {
      if(p->next !=NULL)
      { p->back->next = p->next;          /* cas 1 : c'est une fiche intermédiaire */
        p->next->back = p->back;          /* suppression de la fiche et raccordement des pointeurs */
      }
     else
      { p->back ->next =NULL;/* cas 2 c'est la dernière fiche, et elle n'est pas unique,
                                            on la supprime */
        last = p->back;
      }
   }
 else                                     /* cas 3 et 4 */
   {
     if(p->next != NULL)                  /* cas 3, c'est la première fiche, elle n'est pas seule  */
      {
      p->next->back = NULL;
      first = p->next;                    /* on la supprime */
      }
     else
      { first = NULL;                     /* cas 4 :  il n'y a qu'une fiche, on la supprime */
        last = NULL;
      }          } } }
```

void detruire(struct tache *p) ***/* détruire la fiche du processus de pointeur *p, récupérer la mémoire***

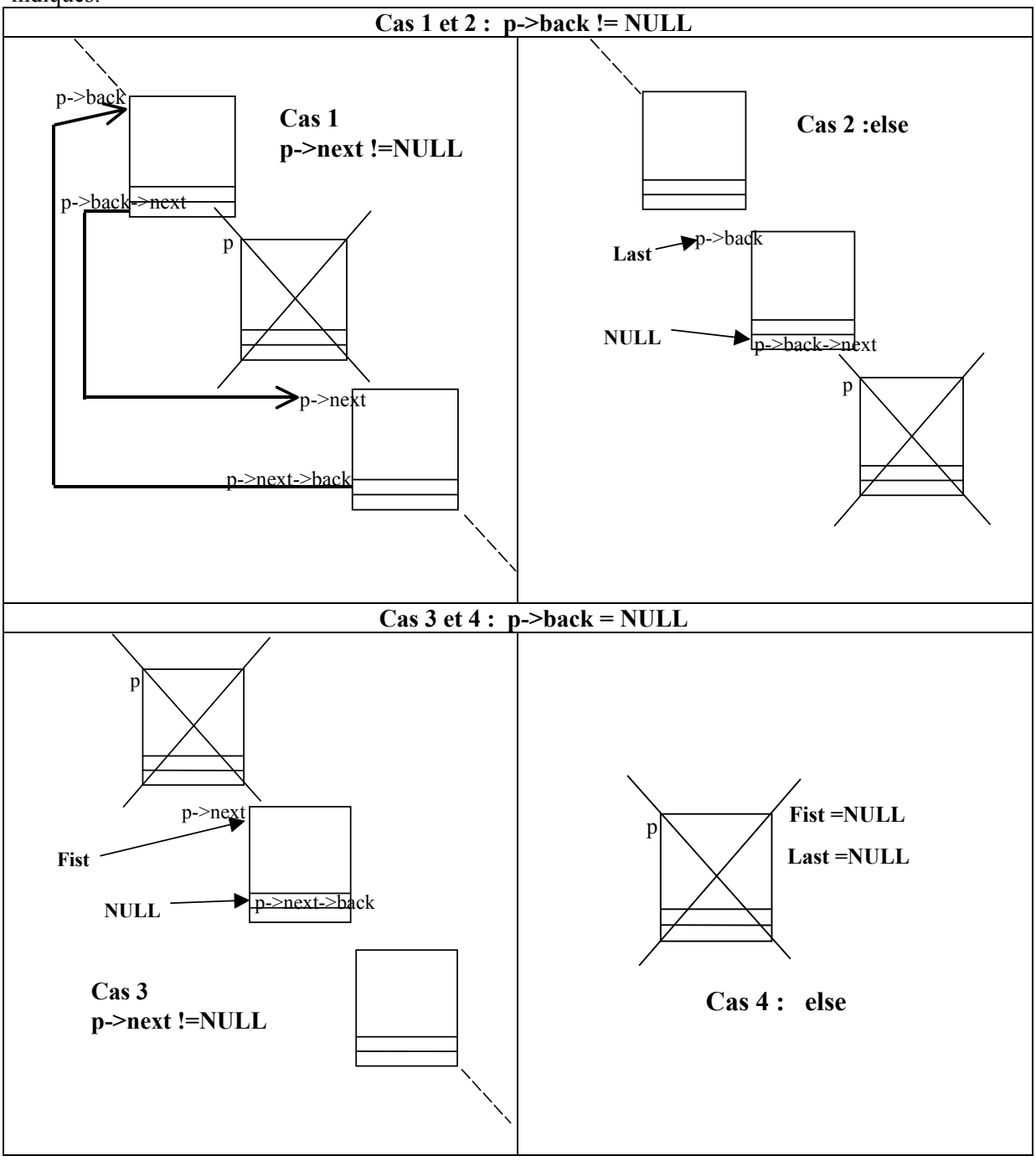
Cette fonction récupère aussi toute la place occupée par ses variable locales */

```
{
 free( (char*)( p->SOMMET_PILE  –  p->taille_pile + 1) ) ;
 free(p->nom) ;
 free(p) ;
}
```

→ Explication de la fonction extraire:

On peut dessiner les 4 cas possibles :

En **gras** on indique les **modifications**. Les autres liens évident pouvant exister ne sont pas indiqués.



On remarque aussi une sorte de réinitialisation dans sa fiche, condition indispensable si on veut de nouveau l'activer.

```
void voir_process(char id)      /* si printf marche sur le système de développement */
{                               /* afficher les caractéristiques d'un processus */
struct tache *p;  p=first;
if(p!=NULL)
    {   while( (p->id != id)  && ( (p=p->next) !=NULL)  );       /* recherche de la fiche */
    if(p !=NULL) {
                printf("\n  Processus numero %d",p->id); printf("\nom:    %s",p->nom);
                printf("\nPriorité:         %d",p->prio); printf("\nAdresse fonction:  %x",p->f);
                printf("\nPILE (sommet):     %d",p->SOMMET_PILE);
                }
    else   printf("\n  Processus numero %d  Non Trouvé ! ", id);
} }
```

21.4.2.2. Fonctions de gestion du multi_tache (Sans priorité)

interrupt[0xFFE2-0xFF80] void changer_de_tache(void)

```
                    /*Routine d'interruption assurant le temps partagé */
interrupt[0xFFE2-0xFF80] void changer_de_tache(void)      /* ou $62 décalage de  OC6   */
{  /* surtout pas de variables locale ici, sinon on décale la position du S à récupérer ... */
tache_precedente = tache_courante;     /* la tache précédente est maintenant la courante actuelle */
tache_courante=ordonn ();          /* recherche de la tache suivante par l'ordonnanceur */
if((tache_courante !=tache_precedente)&(first !=NULL))  /* test si 0 ou même tâche */
   {
   sauver_s();                  /* instructions assembleur de sauvegarde de S *vers ssauv */
   tache_precedente->s = ssauv;     /* rangement dans fiche  tache precedente */
   snouv=tache_courante->s;         /* mise en snouv de S nouvelle tache */

   TC6 = TC6 + 500;         /* cadence 4ms = 8*n microsecondes  ne pas changer Fbus 4MHz*/
   TFLG1 = 0x40;
   valid_it();
   if(tache_courante->etat ==0)lancer_premiere(tache_courante);    /* processus nouveau jamais
lance */
   else recuperer();     /* nouvelle tache courante, récupération d'un processus déjà lance */
    }
 else {
   TC6 = TC6 + 500;    /* cadence 4 ms = 8*n microsecondes  ne pas changer */
   TFLG1 = 0x40;
   valid_it();
   }  }
```

```
struct tache* ordonn(void)  /* l'ordonnanceur, retourne  tâche à lancer ou à reprendre */
                                    /* rudimentaire, sans gestion de priorité */
{
struct  tache *tch;
if(tache_courante->next !=NULL)tch=tache_courante->next;  /* on n'est pas en bout de liste des
                                                              taches */
else   tch = first;        /* en est en bout de liste */
return(tch);
}
```

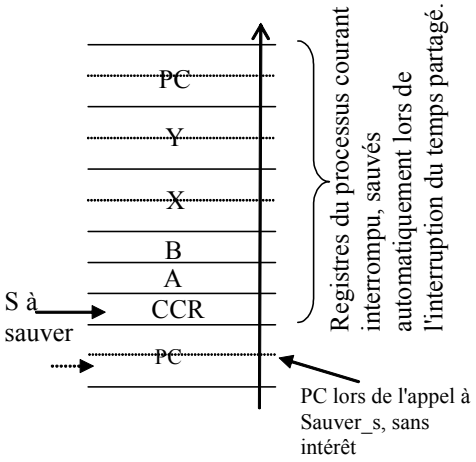
On quitte chaque tache par interruption, donc pour lancer une tache nouvelle, ou récupérer une tache déjà lancée, on se sert de la même instruction **RTI** (de retour d'interruption). Il suffit simplement de positionner correctement le pointeur S pour récupérer le PC (et tous les registres de la tache à lancer).

Lancer_première est une fonction C, **sauver_s, lancer**, et **récupérer** sont de petites routines assembleur. On se sert de deux variables globales de travail : **snouv** et **premier_pc**.

```
void lancer_premiere(struct tache  *tch)     /* lancer une tâche la première fois */
{
if(tch != NULL)
{
     tache_precedente = tache_courante;       /* ancienne tache courante devient la précédente */
     tache_courante = tch ;                   /* nouvelle tache courante */
     snouv  =  tch->s;                /* S et adresse fonction dans deux variables globales de travail */
     premier_pc = (int)tch->f;
     tch->etat=1;                /* drapeau état à 1: premier lancement effectué */
     lancer();          /* instructions assembleur initialisant S et PC */
   }
   else while(1) ;      /* pour éviter de planter si aucun tache au départ en file d'attente */
}

/* ------------------------------------------------------------------------------------------------ */
/*                  Routines supplémentaires assembleur HC11 ou HC12                                */
/* ------------------------------------------------------------------------------------------------ */
     PUBLIC       valider_it, inhiber_it , sauver_s, lancer, recuperer
     EXTERN       snouv, ssauv, premier_pc
     RSEG  RCODE
valider_it   cli
             rts
inhiber_it   sei
             rts
/*   Pour lancer une tâche pour la première fois */
lancer       lds     snouv          nouveau pointeur de pile dans registre S
             ldx     premier_pc
             pshx                   empilement du PC
             pshx                            |
             pshx    | ces 3 pshx équivalent à 6 des (en plus rapide)
             pshx    |
             tpa     reprendre l'ancien Code Condition (surtout pour le masque XIRQ)
             psha
             rti                    le rti lance le processeur sur le PC empilé
       /* ------------------------------------------------------------------------------ */
sauver_s     ins     Sauver S d'une tache
             ins
             sts ssauv      sauve S
             des
             des
             rts
```

(on pourrait supprimer les ins, les des, et le rts avec du code inline, sans le jsr d'appel)

```
recuperer lds  snouv   Reprendre S d'une tache
          rti     récupération des registres
                  et relance du processus
```



21.4.2.3. gestion d'une priorité multi_tache

On choisit le **système de l'age** expliqué précédemment.

Il suffit :

De choisir des priorité lors de la création des taches.

D'utiliser l'ordonnanceur à priorité suivant : à la place de struct tache * ordonn(void)

```
struct tache * ordonn_prio(void)    /* Ordonnanceur à gestion de priorité */
{
struct tache *p, *a_lancer;
char max_age = 0;
p=first;
while(p != NULL)
   {
   p->age = p->age + 1;    /* augmentation de l'age de chaque processus */
                           /*  en même temps que recherche de l'age max */
   if(p->age >= max_age)  {
                           max_age = p->age;   /* mis a jour max_age */
                           a_lancer = p;       /* mis a jour tache a lancer */
                           }
   p = p->next;
}
a_lancer->age = a_lancer->prio;     /* le processus à lancer, reprend son age initial prio */
return(a_lancer);
}
```

Remarque : cet ordonnanceur est équivalent à l'autre pour des taches de même priorité, on peut donc en fait toujours laisser celui ci.

21.4.3. Exemple d'une application simple en temps partagé

21.4.3.1. Cahier des charges

Soit un **moteur pas à pas**. Il doit tourner à **1 tour/seconde**

Sur un **panneau d'affichage** cristaux liquides, on veut afficher :

Sur la première ligne, et à chaque tour, des renseignements sur le moteur (nombre de pas effectués)

Sur la seconde ligne un chronomètre comptant les **1/10 s**

21.4.3.2. Etude préalable

On peut distinguer **3 taches ou processus** qui peuvent être à priori bien distincts :

-Processus « **moteur** » (moteur pas à pas)
-Processus « **affichage** » (affichage des tours)
-Processus « **chono** » (chronomètre)

Ces noms seront des noms de fonctions sans paramètres.

On peut déclarer des variables globale servant à l'application proprement dite:

nbrtours Comptage des tours
nbrpas Comptage des pas

Un **événement ev_tour** permettant de synchroniser l'affichage à chaque tour de moteur.

Cet événement sera aussi une variable globale 0 ou 1.

On choisit dans un premier temps un ordonnanceur simple sans priorité (ou ce qui revient au même des priorités égales sur un ordonnanceur avec priorité)

On se sert de **fonctions d'affichage** sur **panneau cristaux liquides**. Des fonctions standards ont déjà été étudiées, nous utiliserons ici des fonctions un peu modifiées, que nous décrirons plus loin. Elles permettent une écriture possible sur chaque ligne individuellement par des taches différentes.

Ligne 0 (16 caractères)
Ligne 1 (16 caractères)

On donne ici leur prototype et leur cahier des charges :
extern void ecrire_ligne(char ligne, char *texte);
Ecrire ligne complète ASCII sur ligne 0 ou 1 (max 16 caractères)

extern void ecrire_entier(char ligne, int valeur);
Ecrire en décimal un entier (signé 16 bits) sur ligne 0 ou 1 (le reste des espace)

extern void ecrire_q1632(char ligne,   long q1632, char *texte);
Ecrire en décimal un nombre signé en Q16(32) sur ligne 0 ou 1, suivi d 'un texte

On se sert d'une fonction **avance_pas** d'un moteur pas à pas (qui envoi les codes correct sur ses 4 enroulements)
Prototype : **void avance_pas(void) ;**
On tentera des améliorations successives.

21.4.3.3. Premier idée de programme

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "executif.h"
#include "registres_hc12.h"
#include "ini_carte.h"
extern void ini_carte(void);
/* @@@@@@@@@@@    POUR L'EXECUTIF   @@@@@@@@@@@@@@@@@ */
/*                    Les prototypes des fonctions externes sont  dans executif.h          */
/*                          Variables globales  de l'exécutif                                        */
struct tache *tache_courante = NULL ;  /* pointeurs sur descripteurs de tache */
struct tache *tache_precedente = NULL;
struct tache *last = NULL;
struct tache *first = NULL;
char id=0 ;                   /* N° d'identificateur de tache */
/*@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ */

/* ----------------------------   POUR L'APPLICATION   ----------------------------- */
/*                        Prototypes (externes ou non) de l'application                   */
extern void config_aff(char *tab);
extern void tempo(unsigned int ms);
extern void ecrire_ligne(char ligne, char *texte);
extern void ecrire_entier(char ligne, int valeur);
extern void ecrire_q1632(char ligne,   long q1632, char *texte);
void avancepas(void);
```

```
/*                    Les taches                                    */
void affichage(void);
void chrono(void);
void moteur(void);
/*              variables globales de l'application                 */
char tab_ini_aff[] = {0x28,0x0C,0x01,0x06,00};      /* chaine de config afficheur, 4 bits, pas de curseur*/
struct tache *t0,*t1;          /*  pointeurs de tache */
char ev_tour = 0;              // Drapeau Evénement tour
int nbrtours=0;
int nbrpas=0;

/*====================PROGRAMME============================ */
/* #define F_BUS ?    Selon la fréquence Bus utilisée, Voir obligatoirement dans  "ini_carte.h"  Car cette valeur sert pour plusieurs fichiers.  */
main(void)
{
/* ---Config de la carte, de l'afficheur, CREATION et MISE EN FILE des taches --- */
ini_carte();    // ini de la carte à F_BUS défini dans "ini_carte.h", ini port claier
config_aff(tab_ini_aff);
t1=creer(chrono,100,"chronometre",200);                  // priorités identiques 100
entrer(t1);                                              // Et taille de pile 200
t0=creer(affichage,100,"données sur le moteur",200);
entrer(t0);
t1=creer(moteur,100,"mot pas à pas",200);
entrer(t1);

DDRM = 0xF0;    /* Pour commande 4 enroulements du moteur, 4 MSB du port en sortie */
PTT = PTT & (0xFF-0x20);  // PT5 à 0
DDRT = 0x20;            // 0010 0000   PT5 en sortie pour test vitesse moteur

// @@@@    LANCEMENT  DE  L'EXECUTIF  Cadencé par TC6  à 4ms  @@@@
//      Validation It, fixation cadence du temps partagé, demarrage sur première tâche
//     Cadence_tcnt (MHz)= F_BUS(MHz) / 2^PREDIVISEUR = 0,125MHz
#define PREDIVISEUR  log(8*F_BUS)/log(2)     // Calculé à la compilation pour IT à 16 micro, ou pourrait mettre 4 directement si on garde toujours F_BUS à 2 MHz
TSCR1 = 0x80;   // Validation Timer général
TSCR2 = PREDIVISEUR;     // FBus/2**n: exemple avec Fbus = 2MHz, 4 pour Cadence TCNT de 8 micro secondes
TIOS = TIOS | 0x40;       // TC6 en « comparaison », sans toucher aux autres
TC6 = TCNT + 500;         /* cadence 4 ms, ne pas changer ...*/
TFLG1 = 0x40;             /* raz drapeau TC6, ordre important ici ! */
TIE = TIE | 0x40;         /* validation  interruption OC6, sans toucher aux autres */
valid_it();
lancer_premiere(t0);
}
```

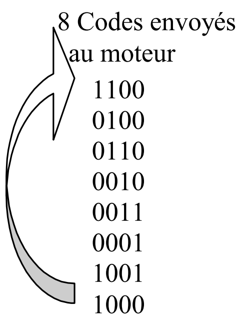
```
/* @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ */
 /* ------------------------   DESCRIPTION DES TACHES   ----------------------*/
```

```
//-------------- Tache N°1 :        Affichage renseignements moteur
void affichage(void)
{
long n =0;
while(1)
   {
   // Affichage Nombre de pas à chaque tour :
     if(ev_tour == 1){    ecrire_entier(0,nbrpas);
                          ev_tour = 0; }
   }
}
```

```
//-------------- Tache N°2 :        Chronomètre au 1/10 seconde
 void chrono(void)
{
long n =0;
while(1)
   {
  ecrire_q1632(1,n," s");
   n=n+65536/10;
   tempo(100);    /*   "100 ms"  evidemment ralenti par le temps partagé */
    }
}
```

```
//-------------- Tache N°3 :        Moteur pas à pas
void moteur(void)
{
while(1)
   {
   avancepas();
   tempo(5);       /* moteur 200 pas, pour vitesse 1 tour / seconde ?   évidemment ralenti*/
   }
}
```

```
/* ------------------------------   Autres fonctions ---------------------------------- */
void avancepas(void)             // Moteur 200 pas par tour
{
static char tab[10]={ 0xc0, 0x40, 0x60, 0x20, 0x30, 0x10, 0x90, 0x80, 0x99};
static char i = 0;
PTM=tab[i++];
if(tab[i]==(char)0x99)i=0;
// Et pour le nombre de tours:
if( (nbrpas++ % 200) == 0)
         { nbrtours++; ev_tour=1 ; }
       /* % donne le reste de la division */
}
```



Remarques :

Bien remarquer les 3 taches qui tournent en permanence et obligatoirement leurs boucles while(1) …

Remarquer la gestion du drapeau ev_tour de synchronisation de l'affichage à chaque tour.

Performances,

Pour ces 3 taches: N = 3 T = 4ms tc = 0,18ms

Le moteur est loin de tourner à la vitesse désirée.

Le chronomètre tourne lentement ….

Tout est ralenti.

Le coefficient de ralentissement est $k = \frac{NT}{T - tc} = 3{,}14$

Le moteur tourne donc à 0,32 tour par seconde au lieu de 1 !

Il est inutile de modifier T et tc, k restera toujours 3…

Causes : Les 3 taches ont même priorité, Tout est ralenti par le temps partagé, la fonction tempo effectue une temporisation mais de durée bien plus longue que celle désirée.

Le ralentissement augmenterait avec la charge du système.

Le **temps réel n'est pas assuré**.

21.4.3.4. Amélioration 1 : priorité de processus

On choisit la technique de l'age, étudiée précédemment.

On crée les processus avec des priorités, par exemple :

100 pour **moteur**

90 pour l'**affichage** et pour le **chrono**

Ce qui donne à peu près : un appel affichage et chrono pour 10 moteurs.

Il ne faut pas trop diminuer la priorité affichages, sinon celui-ci n'est plus vraiment visuellement synchrone de chaque rotation.

On constate une **nette amélioration** de la vitesse de rotation du moteur! Cependant encore un peu plus lente que 1 Tr/seconde. En effet on a cette fois ci (cf étude de cet exemple vue précédemment) $k = \frac{1{,}2T}{T - tc} = 1{,}25$

On constaterait aussi une succession non régulière des pas du moteur.

Le chronomètre par contre est encore plus lent, c'est normal (coeff $k = \frac{12T}{T - tc} = 12{,}5$!)

Le comptage du nombre de pas est OK.

C'est mieux, mais ce n'est pas parfait. Les performances du système se dégraderaient si on chargeait d'autres tâches en parallèle. Les performances sont évidemment inversement proportionnelles à la « charge de l'unité centrale ».

Conséquence : OK si pas trop de contraintes sur le temps réel. Une fonction telle que tempo est évidemment à éviter dans un tel système !

21.4.3.5. Amélioration N°2

On désire **garantir une vitesse de rotation fixe et régulière** égale à 1 Tr/seconde.

On doit envoyer une cadence de progression des pas de 200Hz soit toutes les 5ms. Pour cette cadence de même ordre de grandeur que le quantum de temps du temps partagé, on en revient à une technique préemptive pour faire avancer d'un pas le moteur à chaque interruption d'un Timer.

On choisit l'interruption It_TC0 du Timer de l'HC12

La tâche moteur n'est plus insérée dans la file d'attente des tâches. On la place directement dans la routine d'interruption adéquate. D'ou le nouveau programme (partie instructions) :

```
unsigned int delta ;                    // Variable globale pour le timer
main(void)
{
/* ---Config de la carte, de l'afficheur, CREATION et MISE EN FILE des taches ---
*/
ini_carte();    // ini de la carte à F_BUS défini dans "ini_carte.h", ini port claier
config_aff(tab_ini_aff);
t0=creer(affichage,90,"Nbr de pas",200);       /* taille pile 200 pour chacun */
entrer(t0);
t1=creer(chrono,90,"chronometre",200);
entrer(t1);
TIOS = TIOS | 1;          /* TC0 en Output compare pour le moteur */
TIE = TIE | 1;            /* IT TC0Validée pour le moteur */
delta = 625 ;             // Pour 1 tour/s, moteur 200 pas, cadence TCNT 125kHz
TC0 = TCNT + delta;   // cadence moteur initiale
DDRM = 0xF0;       // Port M en sortie pour Moteur, 4 bits MSB
/*@@@@@@@@@          LANCEMENT  DE  L'EXECUTIF    @@@@@@@@@@@ */
                      PAS DE CHANGEMENT
/*@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@*/

/* ------------------------     DESCRIPTION DES TACHES     ------------------------ */
void affichage(void)
{                                                IDENTIQUE
}

 void chrono(void)
{                                                IDENTIQUE
}

 /* ********   Tache préemptive sans passage par l'ordonnanceur   **************/
interrupt[0xFFEE-0xFF80] void it_moteur(void)  // par OC0  (ou décalage 0x6E)
{
avancepas();    // pour plus tard le cas Arret Moteur
TC0 = TC0 + delta;
TFLG1 = 0x01;      /* raz drapeau interruption TC0 */
}

/* ------------------------------   Autres fonctions ---------------------------------- */
 void avancepas(void)
{                                                IDENTIQUE
}
```

Performances :

Fonctionnement impeccable du moteur (pour cette vitesse).

Progression des pas pratiquement régulière.

Ok pour l'affichage des pas, l'affichage se fait correctement à une cadence assez lente (1 Tr/seconde) Pas de problème de temps réel.

Le chronomètre ne fonctionne évidemment toujours pas correctement !

21.4.3.6. Amélioration N°3 : Remède pour le chrono

Supprimer la fonction tempo, remplacer par le Timer, registre TC1 par exemple !

Pour compter les 0,1 secondes ou les secondes, on peut utiliser le Timer sans interruption, dans une tache scrutant son drapeau, en travaillant sur **événement Timer**. La cadence en moyenne sera toujours exacte même si de petites irrégularités existent.

Pour compter plus rapidement, par exemple les millisecondes, les retards de prise en compte et d'exécution dépasseront souvent la cadence à générer et des évènements timer ne seront pas pris en compte, d'ou perte de synchronisme. Il faudrait donc une technique par préemption directe, mais on ralentirait tout le reste du temps partagé. Et de plus si d'autres taches (comme ici le moteur) travaillent de cette façon, il y aura ralentissement de tout l'ensemble.

CONCLUSION : Aucune technique n'est parfaite si on est à la limite de puissance du processeur !

21.4.4. Modification des fonctions d'affichage et de lecture clavier en vue du temps partagé

Nous avons déjà étudié par ailleurs des fonctions permettant la gestion d'un petit clavier 16 touches et l'affichage sur un panneau de cristaux liquides, on pourra s'y reporter. Nous allons expliquer ici les modifications à apporter pour une utilisation dans des taches différentes en temps partagé.

On rappelle que le clavier n'est pas câblé sur des ports parallèles, mais sur une ligne série rapide SPI (3 fils) et un circuit spécial interface SPI/parallèle CDP66HC68. D'ou présence de **fonctions « Driver »** pour lire et écrire une ligne.

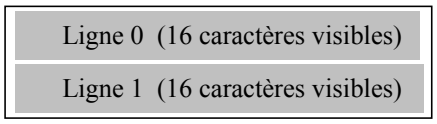
La fonction tempo de gestion de l'anti rebond reste encore par comptage et en assembleur, le temps partagé la ralentira, mais ce n'est pas grave ici….

21.4.4.1. Fonctions de gestion de l'afficheur

Comme l'afficheur possède deux lignes, il faut permettre des écritures quasi simultanées sur chaque ligne par deux processus différents.

On rappelle ici ses 3 registres : **IREG** Registre d'**instruction**
DREG Registre de **donnée**
BAC Son MSB = bit 'busy' de contrôleur

Le contrôleur utilisé (HD44780U) contient une mémoire nommée DDRAM (Display Data Ram) dans laquelle on écrit les caractères que l'on veut afficher. A une adresse correspond une position sur l'afficheur. Un compteur d'adresses lui est associé, et en mode normal il s'incrémente à chaque écriture d'un caractère.



Adresses sur 7 bits dans la Display Data Ram :

Ligne 0 : **000 0000 à 000 1111** caractères visibles sans décalage
001 0000 à 010 0111 autres (non utilisés ici)

Ligne 1 : **100 0000 à 100 1111** caractères visibles sans décalage
101 0000 à 110 0111 autres (non utilisés ici)

On peut écrire dans le compteur d'adresse en envoyant au registre d'instruction IREG l'octet: 1- - - - - - - (1 suivi de l'adresse sur 7 bits).

1) Modifications à envisager

Il est nécessaire :

De prévoir la possibilité d'écrire sur l'une des deux lignes au choix.

De créer un **curseur logiciel** propre à chaque processus. (le contrôleur de l'afficheur permet la gestion automatique d'un seul et unique curseur, il n'est donc plus utilisé).

L'écriture de tout caractère se fera alors en deux étapes :

Envoi de l'adresse (valeur du curseur logiciel) au compteur d'adresse de la DDRAM
Envoi du code ASCII du caractère (dans le registre de donnée DREG).

Il faut aussi créer un **drapeau d'occupation de ressource**, nommé ici **semaph_afficheur**, ainsi une tache quelconque pourra prendre la ressource pour y écrire à chaque fois un caractère, puis la libérera. (Sans ce drapeau, il y aurait fréquemment des mélanges entre des caractères envoyés par des processus différents).

Un « **Time out** » est nécessaire sur ce drapeau (avec un message d'alerte), dans les quelques cas ou il y aurait blocage. Si le message survient, il faudrait alors modifier quelque part son programme (ce serait le cas par exemple d'une tache qui écrirait dans l'afficheur et que l'on supprime de la file d'attente, il se pourrait que cette tache soit justement en attente de prise en compte d'un caractère par l'afficheur au moment ou on la supprime. Si par la suite une autre tache attend la libération de cette ressource, elle n'arrivera jamais ! un message évite le plantage. Le remède est évidemment de penser à remettre à zéro le sémaphore juste

avant de supprimer une tache si celle ci travaille avec l'afficheur. A faire aussi si on insère une telle tache dans la file.

2) Fonction correspondantes

Les registres de l'afficheur :

```
#define  IREG  *(char *)0x6000          Registre d'instruction
 #define DREG *(char*)0x6001                Registre de donnée
 #define BAC *(char*)0x6000           Son MSB = bit 'busy' de contrôleur
```

Variables globales :

```
signed char semaph_afficheur=0;       /*  Sémaphore de l'afficheur, libre au départ */
```

→Ecrire une ligne complète ASCII sur ligne 0 ou 1 (max 16 caractères)

```
#include <string.h>
void ecrire_ligne(char ligne, char *texte)     /* deux lignes indépendantes, pour multitache, avec curseur logiciel, limitée à 16 caractères. Fonction de base, appelant Driver W_IREG et W_DREG */
 {
 char k;char *ch; int curseur;char l;     /* le curseur logiciel est une simple variable locale */
 ch = texte;
 l = strlen(texte)-1;    /* longueur chaine sans 00 fin de chaine */
 if(ligne == 0) curseur = 0x80;   /*   1   000 0000       première ligne  */
 if(ligne == 1) curseur = 0xC0;   /*   1   100 0000       seconde ligne   */
 k=1;
 while((*ch != 0x00) & (k++ <=16)) )    /* tant que pas fin de chaîne ni plus de 16 caractères */
      {
      attente_liberation();     /* attente afficheur libre de tout process */
      semaph_afficheur = 1;   /* Ce process prend cette ressource pour écrire 1 caractère */
      W_IREG(curseur);            /* envoi curseur , driver W_IREG */
      tempo40micro();    //attente prise en compte, simple tempo, plus simple ici, et interromptible
      W_DREG(*ch++);             /* envoi caractère driver W_DREG, pas de tempo */
       semaph_afficheur = 0;
       curseur++;                          // Gestion du curseur propre à une tache
       }
 for(k= l;k<=16;k++){          /* on va compléter la ligne par des espaces */
      attente_liberation();     /* attente afficheur libre de tout process */
      semaph_afficheur = 1;
      W_IREG(curseur);
      tempo40micro();       //attente prise en compte par simple tempo, plus simple
      W_DREG (' ');         /* envoi espace, pas de tempo */
      semaph_afficheur = 0;
      curseur++;
       }
 }

void attente_liberation(void)    // Avec message d'erreur de sémaphore sur ligne 1
{
 int time_out = 0;
 while ( (time_out++ <= 5000) & (semaph_afficheur !=0));   /* attente libération */
 if (time_out >=5000 ){ semaph_afficheur =0; ecrire_ligne(1," Pb Semaph aff !");
                  tempo(250);   );      /* fonction tempo en fait de plus de 500 ms car sera ralenti, mais peu importe, le but est de permettre de voir ce message !*/
                   ecrire_ligne(1,"");      // suppression du message
                  }
}
```

→Ecrire en décimal un entier (signé 16 bits) sur ligne 0 ou 1 (le reste des espace)

On fabrique petit à petit toute la chaîne de caractères à écrire (avec suppression des zéros en tête), et on l'affiche par la fonction précédente sur la ligne voulue. Sinon les techniques de conversion sont classiques.

```
void ecrire_entier(char ligne,   int n)
 {
 signed char signe = 0;   unsigned int q,r;
 char k;
 char flag = 0; char ascii[7];char asciisanszeros[7];     /* Signe 5 chiffres et NULL fin de chaine */
 char *pt=asciisanszeros;                          /*      S x x x x x NULL */
 ascii[0]=' ';
 if(n < 0){ n = -n; signe = -1;}
 if(n==0){*pt++=0x20; *pt++=0x30; *pt=0;}     /* cas affichage 0  donc 3 caractères:  espace 0 NULL  */
 else
   {
   for(k=5;k>=1;k--)  {  q=n/10;  r=n-10*q;   ascii[k]=r+0x30;    n=q;  }
   if(signe ==0) *pt++ = ' ';
   else *pt++ = '-';
   for(k=1;k<6;k++)
      {   if(ascii[k]!=0x30 )flag=1; if(flag==1)*pt++=ascii[k];  }
    *pt = 0;  /* fin de chaine */
   }
 ecrire_ligne(ligne, asciisanszeros);          /* affichage de la ligne */
}
```

→Ecrire en décimal un nombre signé en Q16(32) sur ligne 0 ou 1, suivi d'un texte

Même technique, on fabrique la chaîne complète ASCII pour l'afficher à la fin.

```
void ecrire_q1632(char ligne,  long q1632, char *texte)
 {
 unsigned int entiere; unsigned int frac; unsigned long int val32;
 signed char signe = 0;   unsigned int q,r;    char k;  char l;
 char flag = 0; char ascii[11];
                          /* signe, 5 chiffres, virgule, 3chiffres et NULL fin de    chaîne     Sxxxxx,xxxx0*/
 char asciisanszeros[17];     /* max ligne de 16 caractères */
 char *pt=asciisanszeros;
 char *pttexte=texte;
 if(q1632 < 0){ q1632 = -q1632; signe = -1; ascii[0]='-'; }
 else ascii[0]=' ';
 entiere = q1632 >>16;             /* ici la partie entière est donc toujours > 0 */

for(k=5;k>=1;k--)
       { q=entiere/10;   r=entiere-10*q;   ascii[k]= r+0x30;  entiere=q;  }
       ascii[6]=',';
 frac = 0xffff & q1632;   /* partie fractionnaire   dans frac */
   for(k=7;k<=9;k++)       /* on multiplie 3 fois par 10 */
      {
      val32 =(unsigned long)10*(unsigned long)frac;       /* val32 sert de variable de travail de 32 bits */
      entiere = val32 >> 16 ;  /* chaque chiffre (dixième, centième .., se trouve dans la partie entière */
      ascii[k]=entiere+0x30;     /* on trouve le caractère ASCII de chaque valeur de 0 à 9 */
      frac = (unsigned int) (0xffff & val32);    /* en enlève à chaque  fois la partie entière */
      }
```

```
 /*  traitement pour enlever les zéros en tête  sauf si 1 seul,  pt  pointe le début de la chaîne finale à écrire
asciisanszeros */
      if(signe ==0) *pt++ = ' ';
      else *pt++ = '-';
      l=1;       /* longueur asciisanszeros, on vient déjà d'écrire le signe */
      for(k=1;k<=9;k++)
         {
         if(ascii[k]!=0x30 )flag=1;
         if((flag==1) || (k==5)) { *pt++=ascii[k]; l++; }
         }
      for(k=l;k<=15;k++)  *pt++ = *pttexte++;
      *pt = 0;  /* fin de chaine */

   ecrire_ligne(ligne, asciisanszeros);    /* on ecrit enfin la ligne complète  */
   }

   /*************************** LES  DRIVER ****************************************
   ******       ROUTINES  DE BASE pour le C, AVEC LCD MODE 4 BITS SUR SPI1    ***********
   ***   Format  8 bits  EN RS RWn -  Db7 Db6 Db5 Db4      RS = 0: command   RS = 1:  Data   **/
   #define EN (char)0x80
   #define RS (char)0x40
   #define RWn (char)0x20
   void Write( char);
   char Read(void);
   void W_4(char c);
   extern void tempo(unsigned int ms);
   char R_BAC(void)                // Lecture du registre BAC
   {
   char c;
   inhib_it();
   PTH=0;
   DDRH = 0xE0;    // bits utiles du Port LSB en entrée
   Write(RWn);    // envoi de 001-(0000)
   Write(RWn+EN); // envoi de 101-(0000)
   c = Read()<<4;          // Lecture D7 D6 D5 D4 0 0 0 0
   Write(RWn);    // envoi de 001-(0000)
   Write(RWn+EN); // envoi de 101-(0000)
   c = c + (Read()& 0x0F);   // Lecture D7 D6 D5 D4 D3 D2 D1 D0
   Write(RWn);    // envoi de 001-(0000)
   valid_it();
   return(c);
   }

   void W_IREG(char c)                    // Ecriture registre d'instruction
   {
   unsigned cc;
   inhib_it();
   PTH = 0;
   DDRH = 0xEF;    // Port en sortie tout à zéro
   cc = (unsigned char)c >> 4;
   W_4(cc);                    //  envoi de 0 0 0 0 D7 D6 D5 D4
   cc = (unsigned char)c & 0x0F;
   W_4(cc);                    //  envoi de 0 0 0 0 D3 D2 D1 D0
   valid_it();
   }

   void W_DREG(char c)                    // Ecriture registre de données
   {
   unsigned cc;
```

```
inhib_it();
PTH = 0;
DDRH = 0xEF;    // Port en sortie tout à zéro
cc = (unsigned char)c >> 4;
W_4(cc + RS);                   //  envoi de 0 1 0 0 D7 D6 D5 D4
cc = (unsigned char)c & 0x0F;
W_4(cc+RS);                     //  envoi de 0 1 0 0 D3 D2 D1 D0
valid_it();
}

void W_4(char c)
{   Write(c);  Write(c + EN);   Write(c);  }

void ini_mode_4bits(void)
{
W_4(3);       // Nécessaire si le Reset Interne de mise sous tension de l'afficheur n'existait pas
tempo(5);    // > 4.1 ms
W_4(3); tempo(1);     // > 100micro
W_4(3); tempo(1);
W_4(2); tempo(1);
}

void attente_libre(void)    // non utilisé ici, on attend prise en compte par tempo
{   while( (R_BAC() & 0x80) != 0);   }   // tant que busy = 1

void Write(char c)   { PTH = c; }

char Read(void)  {  return(PTH&0x0F);   }   // les 4 LSB seulement
```

21.4.4.2. Fonctions de gestion du clavier (matriciel 4x4)

L'écho des caractères de chaque touche s'effectue sur la ligne au choix. Pour cela on concatène petit à petit chaque caractère frappé et on utilise la fonction précédente affich_ligne à chaque fois. Ces fonctions sont très voisines des fonctions standards étudiées par ailleurs (se reporter à ces pages pour explications supplémentaires et organigramme).

On rappelle que le clavier est câblé sur un bus série SPI rapide avec un interface SPI/Parallèle CDP68HC68. D'ou présence de Driver de lecture ligne et écriture colonne.
Pas de gestion de sémaphore.

```
/*************************************************************************/
/*  FONCTIONS  ACQUISITION POUR TEMPS PARTAGE   clavier 16 touches       */
/*  Partie entière acquise doit être de module <  32767    sinon drapeau renvoyé  -1    */
/*  On spécifie ici tout signed char  car par défaut les char sont parfois non signé ! */
/*        Cablé sur un CDP68HC68 port // sur port SPI0    Janvier 2004 G. Pallot et Omar  */
/*************************************************************************/
/*    Colonnes  d7 d6 d5 d4  -  -  -  -     pour C0 C1 C2 C3 - - - -  */
/*    Lignes    -  -  -  -  d3 d2 d1  d0    pour L0 L1 L2 L3 - - - -  */
/*    Suppose ports (SPI0 et circuit HC68 déjà initialisés par init_port_clavier()  */
#include "afficheur.h"
#include "registres_hc12.h"
extern char io_spi0(char octet);
void ecrire_colonne(char c);
char lire_ligne(void);
#include <string.h>
extern void ecrire_ligne(char ligne, char *texte);
extern void ecrire_entier(char ligne, int valeur);
extern void tempo(unsigned int ms);
unsigned long conversion_entiers(signed char n,signed char *tab_entiers);
```

```
unsigned long conversion_decimaux(signed char m, signed char *tab_decimaux);
signed char lecture_clavier(signed char type);   signed char position(signed char numtouche);

void echo(char ligne, char c);   /*  pour faire l'écho des caractères tapés sur la ligne désirée */
signed char tab_codes_clavier[] ={1,2,3,0x0b,4,5,6,0x3b,7,8,9,0x3c,0x0f,0,0x0e,0x3d};      /* 3-
autres fonctions, touches B C D */
char chaine_echo[20];     /* pour former les chaînes à afficher à chaque frappe */
```

Ces variables sont globales, ce qui est à priori incompatible avec le multi tache, mais à priori il est absurde d'avoir deux taches lisant en même temps le même clavier !

→Acquérir un nombre signé en Q16(32), écho sur la ligne numéro ligne. Renvoi 0 si OK, -1 si dépassement

signed char acquerir_q1632 (char ligne, long *q1632)

```
{
signed char tab_entiers[5],tab_decimaux[4];    /* 5 chiffres entiers max mais faudra < 32767 */
signed char flagfrac = 0;    signed char signe=0;   signed char n=0,m=0,code;
unsigned long l,e,d,f;
signed char flagretour;     /* drapeau depassement */
chaine_echo[0]=NULL;      /* vide au départ */
            *-------------------------------------------------------------------------*
   tab_codes_clavier[15]=tab_codes_clavier[15]| 0x80;    /* inhiber return */
   code=lecture_clavier(0);    /* lire une touche avec type numérique et avec attente */
   tab_codes_clavier[15]=tab_codes_clavier[15] & 0x7f;    /* activer return */

if (code==0x0f)            /* 0x0f = code touche signe moins */
   {   signe=-1;
      echo(ligne, '-');   /* au lieu de putch('-') qui écrirait sur ligne courante */
      tab_codes_clavier[12]=tab_codes_clavier[12] | 0x80;  /*  inhiber moins  */
      tab_codes_clavier[15]=tab_codes_clavier[15] | 0x80;   /*  inhiber return  pour eviter -return*/
      code=lecture_clavier(0);
      tab_codes_clavier[15]=tab_codes_clavier[15] & 0x7f;  /*  valider return  */
   }
do
{
if(code ==0x0e)         /* si code est  virgule */
   {   echo(ligne,',');
      tab_codes_clavier[14]=tab_codes_clavier[14] | 0x80;  /*  inhiber virgule  */
      tab_codes_clavier[12]=tab_codes_clavier[12] | 0x80;  /*  inhiber moins  */
      flagfrac = 1;
    }
else    echo(ligne,code + 0x30);              /* passage ASCII et envoi afficheur */
if( (code!=0x0e) && (code !=0x0b) )      /* si code different de virgule et return */
   {   tab_codes_clavier[12]=tab_codes_clavier[12] | 0x80;  /*  inhiber moins  */
      if( (flagfrac ==0) && (n < 5)) { tab_entiers[n]=code; n++; }     /* limitation à 5 chiffres ent */
      if( (flagfrac ==1) && (m < 4)){ tab_decimaux[m]=code; m++; }     /* limitation à 4 chiffres frac */
   }
      code = lecture_clavier(0);
}
while(code != 0x0b);
      e=conversion_entiers(n,tab_entiers);
      d=conversion_decimaux(m,tab_decimaux);
      f=e<<16;   l=f+d;
if(signe==-1) l=-l;
tab_codes_clavier[14]=tab_codes_clavier[14] & 0x7f;  /* réactiver virgule */
tab_codes_clavier[12]=tab_codes_clavier[12] & 0x7f;  /* réactiver moins */
```

```
if(e <32768) { *q1632 = l; flagretour = 0;}
else flagretour = -1;    /*  cas de dépassement */
return(flagretour);
} /* ici pas de conversions possibles car complémentation */

/*********************************************************************** */
 /* sous programme de conversion des entiers DCB en binaire */
unsigned long conversion_entiers(signed char n, signed char *tab_entiers)
{
signed char k;
unsigned long x=0;      /* calcul en 32 bits pour détecter si dépassement 16 bits */
for(k=0 ; k<n ; k++) x=10*x+tab_entiers[k];
return(x);
}

/* sous programme de conversion des décimaux en binaire */
unsigned long conversion_decimaux(signed char m, signed char *tab_decimaux)
{
signed char k;
unsigned long x=0;
unsigned int div=1;
for(k=0 ; k<m ; k++) x=10*x+tab_decimaux[k];
x = x <<16;          /*  x=x*pow(2,16); */
for(k=0 ; k<m ; k++)div=div*10;    /* création du terme en 10 puissance  */
return ( x/div +1);
}
```

→ Fonction de lecture clavier, selon le type de touche et avec attente

Type de touches : **0x30** Touches **B, C ou D** **0x00** **Numériques**

Retour : Pour les chiffres de 0 à 9, valeurs 0x00 à 0x09
Pour B, C, D codes 0x3B, 0x3C, 0x3D
Pour , - return autres codes (voir listing)

Tous les codes sont dans le tableau global tab_codes_claver[]

```
signed char lecture_clavier(signed char type)
{
signed char NCMAX=3;      /* numero max de colonnes */
unsigned char NC,NL,NT,CC,CL;  signed char CCSAV=0xff,CLSAV;
int k;  signed char code;  signed char codem,typem;
do
{
 CLSAV = 0xFF;
 while(CLSAV ==-1)
   {
    CC=0x7F;
    ecrire_colonne(CC);          // Appel  à Driver
    for (k=0 ; k<=NCMAX ; k++)
      {
    ecrire_colonne(CC);          // Appel  à Driver
        CL=lire_ligne();
        if (CL!=0xFF)
           {
             CLSAV=CL;  CCSAV=CC;
           }
        else
           {
           CC=CC>>1;  CC=CC | 0x80;
           }
      }
    }
```

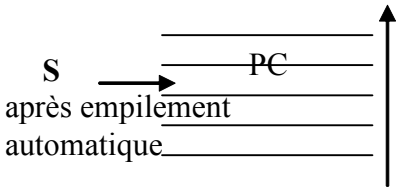
```
    tempo(5);
    /* WAIT RELACHEMENT  */
      while (CL!=0xFF)CL=lire_ligne();
      tempo(20);

      NC = position(CCSAV);   NL = position(CLSAV); NT=NC+(NCMAX+1)*NL;
      code = tab_codes_clavier[NT];
      codem = code & 0x70;
      typem = type & 0x70;
}
  while ( (codem!=typem)||(code<0));
  ecrire_colonne(0xEF);    /*  au repos de nouveau 11101111 pour éventuellement colonne 3 en IT */
  return(code);
}

signed char position(signed char code)
{
      signed char i=0;
      while (code < 0)
      { code=code<<1;i++; }
   return(i);
}
void echo(char ligne, char c)
{                    /* la chaine echo en global  s'allonge sans cesse d'un caractère et est affichée */
char minichaine[2];
minichaine[0]=c;
minichaine[1]=0;
strcat(chaine_echo,minichaine);
ecrire_ligne(ligne,chaine_echo);
}

/*****************************************************************/
/*    Les drivers du port utilisé SPI0 et interface SPI_Parallèle: CDP68HC68  */
/*****************************************************************/
 void ecrire_colonne(signed char c)
 {
 PTS=PTS&0x7f;
 io_spi0(0x10); /*C04(R/W|) = 1 => écrire une donnée  */
 io_spi0(c);  /* C0 C1 C2 C3 1 1 1 1  on pourrait étendre à 8 colonnes avec autre matériel  */
 PTS=PTS|0x80;
 }

 char lire_ligne(void)
 {
 char k;
 PTS=PTS&0x7f;
 io_spi0(0x00);    /*C04(R/W|) = 0 => lire une donnée  */
 k=io_spi0(0x00);
 PTS=PTS|0x80;
 return(16*k + 15);   /* L0 L1 L2 L3 1 1 1 1  on pourrait étendre à 8 lignes avec autre matériel  */
 }
```

→ La Fonction écriture lecture du port SII0 **char io_spi0(char c) ;** est la même que pour la fonction sans temps partagé. Non redonnée ici.

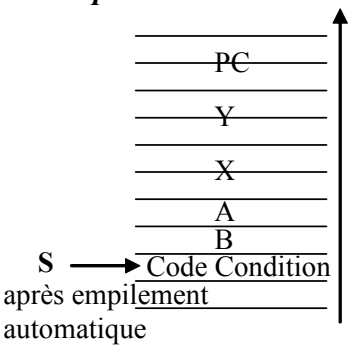
22. ANNEXES SUR LE HC12 : PILE, INSTRUCTIONS, ET VECTEURS D'INTERRUPTION

22.1. Evolution automatique du pointeur de pile en HC12

Après JSR :



Après interruption:



Un peu différent de l'HC11 : ici la pile fonctionne par pré-décrémentation.

Le pointeur de pile se place donc à chaque fois **sur le dernier octet empilé**.

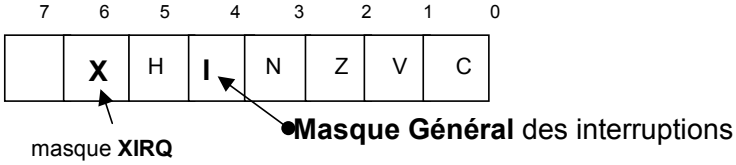
22.2. Vecteurs d'interruption HC12

Les vecteurs d'interruptions sont en FLASH, donc reprogrammables.

Avec l'outil de développement IAR, le C programme de la même façon les vecteurs d'interruptions en Mise au point et pour l'Application finale : le **vecteur** contient **l'adresse du programme d'interruption** à lancer.

Ne pas s'occuper l'indication HPRIO (qui permettrait de modifier éventuellement l'ordre de prise en compte des interruptions.

- ***Lignes d'interruptions Hard Ware***
  IRQ Sur front
  XIRQ Sur niveau
  XIRQ est non masquable si on l'a démasquée une fois ! dans le code condition:



(Par transfert par A : tpa, anda #%10111111, tap)

- ***Table de JMP ?***
  N'existe pas pour l'HC12, avec la carte utilisée de Axiom, et le Debugger IAR

- ***Table de vecteurs 1 :***

Vector Address	Interrupt Source	CCR Mask	Local Enable	HPRIO Value to Elevate
$FFFE, $FFFF	Reset	None	None	–
$FFFC, $FFFD	Clock Monitor fail reset	None	PLLCTL (CME, SCME)	–
$FFFA, $FFFB	COP failure reset	None	COP rate select	–
$FFF8, $FFF9	Unimplemented instruction trap	None	None	–
$FFF6, $FFF7	SWI	None	None	–
$FFF4, $FFF5	XIRQ	X-Bit	None	–
$FFF2, $FFF3	IRQ	I-Bit	IRQCR (IRQEN)	$F2
$FFF0, $FFF1	Real Time Interrupt	I-Bit	CRGINT (RTIE)	$F0
$FFEE, $FFEF	Enhanced Capture Timer channel 0	I-Bit	TIE (C0I)	$EE
$FFEC, $FFED	Enhanced Capture Timer channel 1	I-Bit	TIE (C1I)	$EC
$FFEA, $FFEB	Enhanced Capture Timer channel 2	I-Bit	TIE (C2I)	$EA
$FFE8, $FFE9	Enhanced Capture Timer channel 3	I-Bit	TIE (C3I)	$E8
$FFE6, $FFE7	Enhanced Capture Timer channel 4	I-Bit	TIE (C4I)	$E6
$FFE4, $FFE5	Enhanced Capture Timer channel 5	I-Bit	TIE (C5I)	$E4
$FFE2, $FFE3	Enhanced Capture Timer channel 6	I-Bit	TIE (C6I)	$E2
$FFE0, $FFE1	Enhanced Capture Timer channel 7	I-Bit	TIE (C7I)	$E0
$FFDE, $FFDF	Enhanced Capture Timer overflow	I-Bit	TSRC2 (TOF)	$DE
$FFDC, $FFDD	Pulse accumulator A overflow	I-Bit	PACTL (PAOVI)	$DC
$FFDA, $FFDB	Pulse accumulator input edge	I-Bit	PACTL (PAI)	$DA
$FFD8, $FFD9	SPI0	I-Bit	SP0CR1 (SPIE, SPTIE)	$D8
$FFD6, $FFD7	SCI0	I-Bit	SC0CR2 (TIE, TCIE, RIE, ILIE)	$D6
$FFD4, $FFD5	SCI1	I-Bit	SC1CR2 (TIE, TCIE, RIE, ILIE)	$D4
$FFD2, $FFD3	ATD0	I-Bit	ATD0CTL2 (ASCIE)	$D2
$FFD0, $FFD1	ATD1	I-Bit	ATD1CTL2 (ASCIE)	$D0
$FFCE, $FFCF	Port J	I-Bit	PTJIF (PTJIE)	$CE
$FFCC, $FFCD	Port H	I-Bit	PTHIF(PTHIE)	$CC
$FFCA, $FFCB	Modulus Down Counter underflow	I-Bit	MCCTL(MCZI)	$CA

Remarque :

Les vecteurs interruptions « **Enhanced Capture** » sont **les mêmes** que les interruptions « **Output Compare** ».

➢ ***Table de vecteurs 2 :***

$FFC8, $FFC9	Pulse Accumulator B Overflow	I-Bit	PBCTL(PBOVI)	$C8
$FFC6, $FFC7	CRG PLL lock	I-Bit	CRGINT(LOCKIE)	$C6
$FFC4, $FFC5	CRG Self Clock Mode	I-Bit	CRGINT (SCMIE)	$C4
$FFC2, $FFC3	BDLC	I-Bit	DLCBCR1(IE)	$C2
$FFC0, $FFC1	IIC Bus	I-Bit	IBCR (IBIE)	$C0
$FFBE, $FFBF	SPI1	I-Bit	SP1CR1 (SPIE, SPTIE)	$BE
$FFBC, $FFBD	SPI2	I-Bit	SP2CR1 (SPIE, SPTIE)	$BC
$FFBA, $FFBB	EEPROM	I-Bit	EECTL(CCIE, CBEIE)	$BA
$FFB8, $FFB9	FLASH	I-Bit	FCTL(CCIE, CBEIE)	$B8
$FFB6, $FFB7	CAN0 wake-up	I-Bit	CAN0RIER (WUPIE)	$B6
$FFB4, $FFB5	CAN0 errors	I-Bit	CAN0RIER (CSCIE, OVRIE)	$B4
$FFB2, $FFB3	CAN0 receive	I-Bit	CAN0RIER (RXFIE)	$B2
$FFB0, $FFB1	CAN0 transmit	I-Bit	CAN0TIER (TXEIE2-TXEIE0)	$B0
$FFAE, $FFAF	CAN1 wake-up	I-Bit	CAN1RIER (WUPIE)	$AE
$FFAC, $FFAD	CAN1 errors	I-Bit	CAN1RIER (CSCIE, OVRIE)	$AC
$FFAA, $FFAB	CAN1 receive	I-Bit	CAN1RIER (RXFIE)	$AA
$FFA8, $FFA9	CAN1 transmit	I-Bit	CAN1TIER (TXEIE2-TXEIE0)	$A8
$FFA6, $FFA7	CAN2 wake-up	I-Bit	CAN2RIER (WUPIE)	$A6
$FFA4, $FFA5	CAN2 errors	I-Bit	CAN2RIER (CSCIE, OVRIE)	$A4
$FFA2, $FFA3	CAN2 receive	I-Bit	CAN2RIER (RXFIE)	$A2
$FFA0, $FFA1	CAN2 transmit	I-Bit	CAN2TIER (TXEIE2-TXEIE0)	$A0
$FF9E, $FF9F	CAN3 wake-up	I-Bit	CAN3RIER (WUPIE)	$9E
$FF9C, $FF9D	CAN3 errors	I-Bit	CAN3RIER (TXEIE2-TXEIE0)	$9C
$FF9A, $FF9B	CAN3 receive	I-Bit	CAN3RIER (RXFIE)	$9A
$FF98, $FF99	CAN3 transmit	I-Bit	CAN3TIER (TXEIE2-TXEIE0)	$98
$FF96, $FF97	CAN4 wake-up	I-Bit	CAN4RIER (WUPIE)	$96
$FF94, $FF95	CAN4 errors	I-Bit	CAN4RIER (CSCIE, OVRIE)	$94
$FF92, $FF93	CAN4 receive	I-Bit	CAN4RIER (RXFIE)	$92
$FF90, $FF91	CAN4 transmit	I-Bit	CAN4TIER (TXEIE2-TXEIE0)	$90
$FF8E, $FF8F	Port P Interrupt	I-Bit	PTPIF (PTPIE)	$8E
$FF8C, $FF8D	PWM Emergency Shutdown	I-Bit	PWMSDN (PWMIE)	$8C
$FF80 to $FF8B	Reserved			

Remarque

→ La table des vecteurs débute donc à l'adresse : **debut_table_vecteur = $FF80**

22.3. Instructions assembleur

→ Pas de place dans ce poly, se reporter en fin du ploy de TP ←