Programmation en CALM sur Bimo (Dauphin 3e partie)

Apprenons à programmer le Microchip 16F630 du Bimo ou le 16F690 du BimoPlus

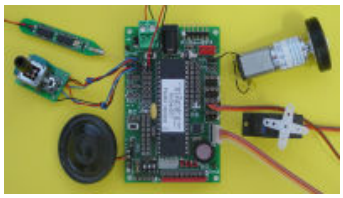
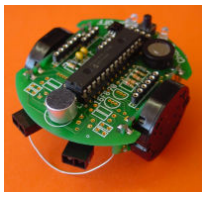
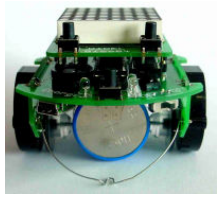
1- Introduction

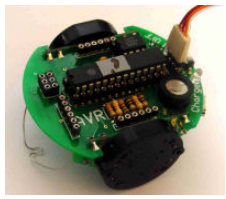
Microchip propose une centaine de processeurs avec différents brochages et performances. La famille PIC 16F est facile à mettre en œuvre avec des processeurs de toutes tailles : le 10F200 est un boitier de 3mm de long avec 6 pattes, donc 4 entrées-sorties programmables !
Pour faire une application avec un PIC, il faut

- un circuit imprimé avec le processeur et le connecteur pour le chargement du programme
- un éditeur/assembleur. SmileNG est certainement le plus facile à utiliser et l'assembleur CALM utilise les notations reprises pour le Dauphin simulé (Note 0 à la fin de ce document). Un compilateur C est préférable pour des grosses applications ou pour des applications avec du calcul et peu de temps réel, mais il implique un processeur plus puissant (16F877, 18Fxx). Note 1 sur les assembleurs, dévermineurs et outils professionnels.
- un programmateur qui transfère le binaire du programme dans le processeur. Le Pickit2 de Microchip est simple et bon marché (Bricoshop, Farnell). Note 2 sur les programmateurs de PIC.

On trouve sur internet quantité de kits pour microcontrôleurs, des programmateurs, des cours et livres expliquant le PIC dans le détail. Le but ici est d'expliquer ce qu'il faut savoir pour programmer des applications intéressantes, sans être exhaustif. La documentation de Microchip (en anglais) est très complète, on peut la lire quand on a bien compris les bases. Le matériel proposé est celui développé spécialement par Didel pour apprendre à écrire des programmes simples (la carte Bimo), piloter un robot avec le BimoPlus ou une application plus complexes avec des moteurs, servos, entrées analogiques (carte Dev877). La carte Dev877 n'a pas besoin d'un programmateur de PIC ; elle est documentée en http://www.didel.com/07dev877/indexF.
Cette documentation comporte plusieurs chapitres avec des exemples à charger. Les notions générales de représentation des nombres, exécution des instructions, constantes, variables, routines sont expliquées dans la brochure Dauphin http://www.epsitec.ch/dauphin/ et sa 2e partie http://www.epsitec.ch/downloads/documentation/dauphin/dauphin2.pdf

En préparation, le robot WellBot sera une jolie solution pour apprendre à programmer avec un robot en kit qui peut éviter des obstacles et suivre une ligne, et qui a un affichage couleur très attractif. Wellbot a une version Basic Stamp et une version AVR.

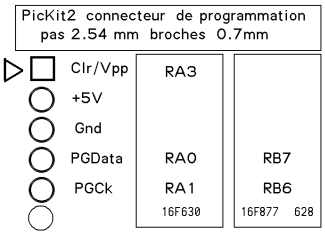




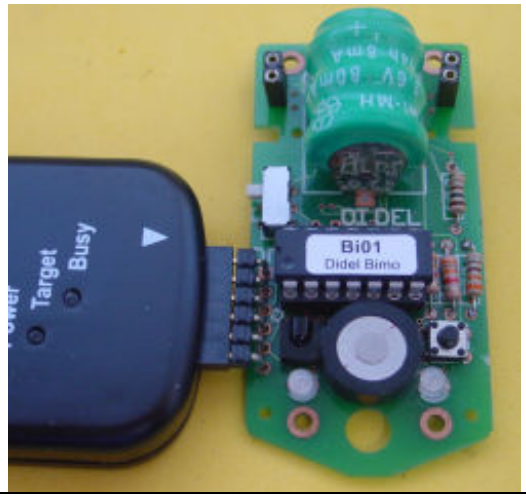


Cette brochure se concentre sur les cartes Bimo et BimoPlus. Le lecteur doit avoir étudié les brochures Dauphin 1ère et 2e partie et compris les exemples, sans nécessairement avoir fait tous les exercices. Par contre, on ne peut bien comprendre un PIC et savoir inventer un programme si on ne fait pas à fond tous les exemples de cette brochure.
Comme toujours en informatique, il y a beaucoup de notions mal expliquées au début, et petit à petit tout s'éclaire ! Patience et persévérance sont de mise.

Ce qu'il vous faut : une carte Bimo (enlevez le bloc moteur pour les premiers exercices), un Pickit2.

La carte BimoPlus est 100% compatible avec ce qui suit si un 630 est inséré. Avec un 690, l'initialisation est différente. D'autres programmateurs sont utilisables, il suffit de câbler une prise compatible.





Si vous êtes impatient de voir comment on programme le Bimo, sautez à la page 4 et faites les premiers programmes. Relisez le début plus tard.

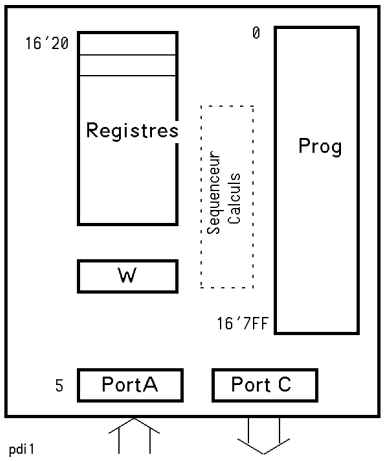
2- Le processeur 16F630

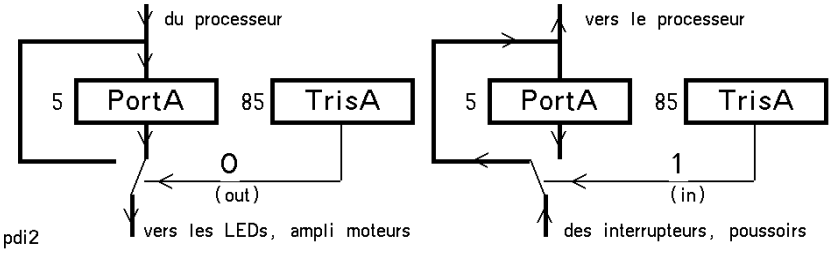
Le microcontrôleur 16F630 a tout dans le même boîtier : la mémoire qui contient le programme, le décodeur et séquenceur d'instructions, la mémoire des données (registres), les amplificateurs d'entrées et sorties et l'oscillateur qui va rythmer 1 million d'instructions par seconde.
Oublions la mémoire programme et le séquenceur d'instructions. Tout le travail de programmation se fait avec les registres (mémoire des données). Ils sont numérotés à partir de 20 hexa, mais on leur donnera des noms.

Les ports sont des registres qui ont des connexions avec l'extérieur ; Il y en a deux de 6 bits dans le 630, appelés PortA et PortC, ou RA et RC. Le BimoPlus a des lignes supplémentaires que l'on ignore ici.
Dans tous les microcontrôleurs PICs, un registre appelé W (work register, registre de travail) joue un rôle de plaque tournante. Avec l'instruction Move, on peut transférer un octet vers ou depuis un port, vers ou depuis un registre-mémoire de données, en passant par W. Par exemple :

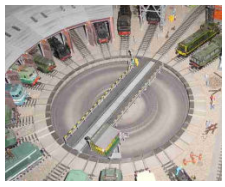
```
Move   PortA,W
Move   W,PortC
```

Ces deux instructions copient ce qui est lu sur le PortA sur le PortC.
Un port est plus compliqué qu'un registre. En sortie, c'est un registre et l'état 0 ou 1 de chaque bit est copié sur les fils de sortie selon une tension min ou max. En entrée c'est une sorte d'interrupteur qui laisse passer l'information vers le processeur quand l'instruction Move PortA,W est exécutée. L'aiguillage qui définit la direction est commandé par un registre dit registre de direction (Tris).





Le registre W fait penser aux plaques tournantes dans les anciennes gares de locomotives. La loco est une valeur qui passe par W pour aller dans le hangar (variables), vers la gare (entrées-sorties) ou dans le dépôt (test, modifications).



3- La carte Bimo

Le schéma de la carte Bimo est expliqué dans la brochure de montage. Ce qu'il nous faut savoir c'est ce qui est câblé sur les deux ports du 16F630.

RA0 RA1 RA3 réservés comme extension et utilisés pour la programmation.
RA0 et RA1 commandent la diode bicolore de gauche, que nous allons ignorer (elle n'existait pas sur la série de Bimo du Festival 2008).
RA2 entrée pour le module infrarouge.
RA4 RA5 sorties vers la diode bicolore de droite.
RC0 RC1 sortie vers le moteur droite RC4 RC5 moteur gauche
RC2 entrée donnant l'état du poussoir (0 si pressé)
RC3 sortie vers le petit haut-parleur (0 si actif)

Ces lignes ont des numéros, mais on va leur donner des noms pour que le programme soit plus lisible. Pour dire que le haut-parleur est sur le bit 3 on déclarera bHP = 3. le ''b'' précise que c'est un numéro de bit etre 0 et 7 et pas un nombre 8 bits, entre 0 et 255 en décimal (entre 2'000000 et 2'11111111 en binaire, 0 et 16'FF en hexadécimal).

4- Un premier programme pour le Bimo ou le BimoPlus

Changeons la couleur de la LED à droite selon l'action sur le poussoir. Il faudra tester l'état 0 ou 1 en RC2 et agir sur RA4 RA5. Ces noms ne nous apprennent rien sur l'application, alors déclarons :

```
 ; PortA
 bRouge       = 4    ; état 1 pour allumer en rouge
 bVert        = 5    ; attention, si les deux sont à 1 c'est éteint
;PortC
 bPoussoir    = 2    ; état 0 si pressé
```

Si on veut allumer une diode biclore, il faut deux instructions pour imposer un ''0'' d'un côté et un ''1'' de l'autre :

```
  Set  PortA :#bVert
  Clr  PortA :#bRouge
```

Si on veut savoir si le poussoir est activé, on a l'instruction

```
  TestSkip,BS  PortC :#bPoussoir
```

Cette instruction Teste le bit dans variable après l'espace (test), et saute l'instruction suivante (skip) si(,) ce bit est à un (BS pour bit set).

Il faut savoir (le schéma le montre) que si le poussoir n'est pas pressé, la résistance tire le potentiel au +5V, donc on saute l'instruction suivante quand on ne presse pas sur le poussoir. Si on presse, il y a court-circuit avec le zéro, donc BC (bit clear) et l'instruction suivante saute (Jump) pour aux instructions qui changent de couleur.

Le cœur du programme est alors très simple :

```
 Boucle:
       TestSkip,BS  PortC :#bPoussoir
       Jump         OnPousse
       Jump         OnNePoussePas
OnPousse:
       Set          PortA :#bVert
       Clr          PortA :#bRouge
       Jump         Boucle
OnNePoussePas:
       Clr          PortA :#bVert
       Set          PortA :#bRouge
       Jump         Boucle
```

Ce programme ne sera pas accepté par l'assembleur, pour trois raisons :

1) on n'a pas dit pour quel processeur il est écrit,
2) l'assembleur ne sait pas ce que valent bPoussoir, bVert, bRouge
3) le processeur ne va pas afficher le vert ou le rouge, parce qu'on ne lui a pas dit que RA4 et RA5 sont des sorties

Il faut donc déclarer le processeur avec une pseudo-instruction, qui commence par un point (Note 2 sur les pseudos). Il faut ensuite déclarer les constantes (Ports) et les variables, dire que le programme commence en zéro et initialiser.

Un programme commence toujours par une initialisation des ports, ainsi que d'autres registres de configuration que l'on ne va pas expliquer tout de suite. Cette initialisation n'est pas la même pour le 16F630 et pour le 16F690.

```
.Proc       16F630       ; le type de processeur
.Ref        16F630       ; sa taille mémoire, etc
\b; PortA   (\b;fera joli dans l'éditeur SmileNG)
 bRouge     = 4
 bVert      = 5    ; attention, si les deux sont à 1 c'est éteint
\b;PortC
 bPoussoir  = 2

.Loc 0      ; adresse de début d'exécution
     Set    Status :#RP0
     Call   16'3FF      ; Routine qui calibre l'oscillateur
     Move   W,OscCal
     Clr    Status :#RP0

     Move   #2'001111,W ; RA5 RA4 en sortie, les autres en entrée
     Move   W,TrisA     ; Active les amplis de sortie
     Move   #2'001000,W ; RC5, 4, 2,1, 0 en sortie, RC2 en entrée
     Move   W,TrisC

Boucle:
     … voir plus haut
.End ; l'assembleur s'arrête ici.
```

```
.Proc   16F690
.Ref    16F690
\b; PortA
 bRouge       = 4
 bVert        = 5
\b;PortC
 bPoussoir  = 2

.Loc  0
      Clr      PortA
      Set      Status:#RP1
      Clr      AnSel
      Clr      Status:#RP1
; Neutralise le convertisseur A/D
      Move    #2'001111,W
      Move    W,TrisA
      Move    #2'001000,W
      Move    W,TrisC

Boucle:
     … voir plus haut
.End
```

Les instructions d'initialisation ne sont pas évidentes et nous pouvons retarder leur compréhension. Les noms ont été choisis par le fabricant, et sont déclarés dans le .Ref 16F630 quand elles dépendent du processeur. Les instructions Set (mets à un) Clr (mets à zéro) Move (copie) Call (apelle une routine) ont des noms qui disent bien leur effet.

A noter une petite différence avec l'assembleur du Dauphin, qui écrit H' pour un nombre hexa. Ici c'est 16' pour de l'hexa et 2' pour du binaire.

On voit que ce programme a une entête et une fin que l'on va trouver dans tous les programme écrits pour le Bimo : les déclarations et l'initialisation. On va donc «encapsuler» ces deux parties dans des fichiers qui seront insérés au moment de l'assemblage. La pseudo-instruction .Ins charge le fichier nommé à l'assemblage. Avec SmileNG, les erreurs dans ces fichiers sont aussi mises en évidence. Note 3 sur les erreurs d'assemblage et leur correction.

Le programme devient plus facile à lire, puisque les parties sur lesquelles on n'a pas besoin de réfléchir ont disparu.

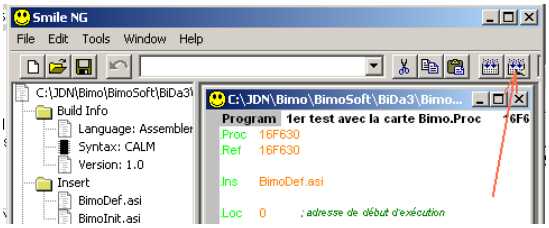
```
\prog:Bimo1.asm| 1er test avec la carte Bimo
; Couleur de la LED selon le poussoir.
.Proc         16F630
.Ref          16F630
.Ins          BimoDef.asi

.Loc   0        ; adresse de début
       Call   Init
Boucle:
       TestSkip,BS  PortC :#bPoussoir
       Jump         OnPousse
       Jump         OnNePoussePas
OnPousse:
       Set          PortA :#bVert
       Clr          PortA :#bRouge
       Jump         Boucle
OnNePoussePas:
       Clr          PortA :#bVert
       Set          PortA :#bRouge
       Jump         Boucle

.Ins   BimoInit.asi

.Loc   16'2007
.16    16'3F94
.End
```

```
\prog:Bplus1.asm| test avec la carte BimoPlus
; La couleur de la LED change quand on presse.
.Proc      16F690
.Ref       16F690
.Ins    BimoDef.asi

.Loc         0        ; adresse de début
           Call       Init
Boucle:
            TestSkip,BS  PortC:#bPoussoir
            Jump         OnPousse
            Jump         OnNePoussePas
OnPousse:
            Set          PortA:#bVert
            Clr          PortA:#bRouge
            Jump         Boucle
OnNePoussePas:
            Clr          PortA:#bVert
            Set          PortA:#bRouge

            Jump         Boucle

.Ins   BPlusInit.asi

.Loc      16'2007
.16       16'03D4
.End
```

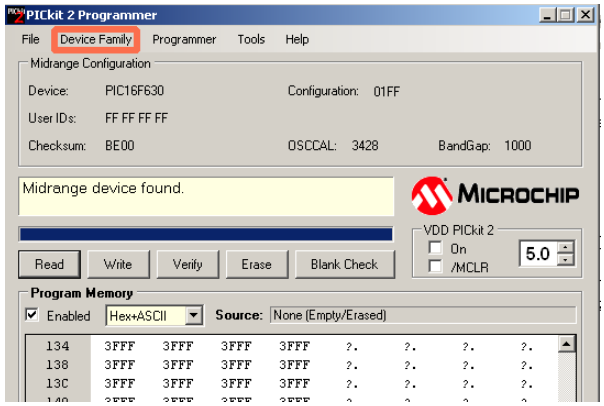
L'instruction Call apelle la routine Init, qui est cachée dans le fichier BimoInit.asi inséré.

Les trois dernières lignes sont un peu mystérieuses. Elles dépendent du processeur et sont nécessaires pour que le Pickit2 programme le bon mode de fonctionnement interne.

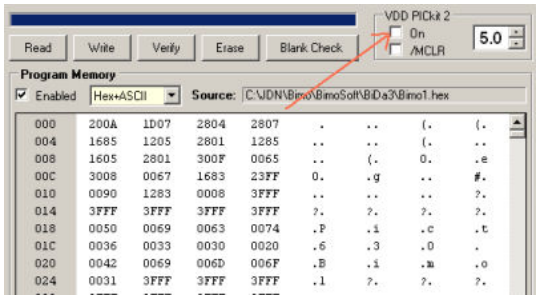
5- Assembler, télécharger et exécuter



Pour installer le programme SmileNG, créez un dossier SmileNG qui doit impérativement être un répertoire direct de C. Copiez dans ce répertoire les fichiers extraits de
www.bricobot.ch/programmer/SmileNG.zip
Créez un raccourci vers SmileNG.exe et exécutez-le.
Note 5 Les fichiers de SmileNG
Créez un dossier Bimo. Copiez les fichiers de
www. bricobot.ch/programmer/Bimo.zip
Chargez Bimo1.asm et créez le binaire en cliquant à droite en haut ou en tapant sur la touche F5.. Vous devez avoir la réponse «Assemblage correct ».
Note 4 Possibilités de SmileNG.



Créez un dossier Pickit2. Copiez les fichiers de
www. bricobot.ch/programmer/Pickit2.zip
Créez un raccourci vers Pickit2.exe et exécutez-le après avoir connecté le Pickit2.
Le lien http://pickit2.com/ a des informations utiles si on sait l'anglais. En français, on se perd dans plusieurs mauvais liens.



Connectez le Bimo.
Note 6 Options pour souder le connecteur de programmation du Bimo.
Si le Pickit n'est pas en mode ''Programmer-Manual select'' le processeur est reconnu. Autrement il faut choisir dans le menu ''Device Family – Midrange – Standard – 16F630''
On peut lire le binaire «Read» et le sauver «File etc».
Ce qui nous intéresse plus est de charger (Import) un fichier créé pas l'assembleur avec l'extension.hex et programmer le 630 avec l'ordre Write.
On peut alimenter le robot par Pickit2 (cliquer sur On).
Vérifions le programme en agissant sur le poussoir : la couleur change comme prévu.

Maintenant que les outils sont en place, ont peut apprendre à programmer le Bimo. Commençons par charger deux programmes sans les comprendre, pour se familiariser.

Chargez Bimo2.asm dans smileNG, puis après assemblage Bimo2.hex dans PicKit2. Ce programme clignote la Led.
Chargez Bimo4.asm. C'est une sirène, la plus simple possible.
Si vous avez un BimoPlus, les programmes s'apellent Bplus2 et Bplus4. On lne le redira plus par la suite.

6- Boucles d'attente

Avez-vous remarqué qu'au démarrage la LED pulse 2 fois avant que la Boucle s'exécute ?
C'est dans la routine `BimoInit.asi`, car c'est rassurant de voir que le processeur a démarré correctement. Si cela ne réagit pas juste après, il faut aller relire son programme !

Pour clignoter, il faut allumer, attendre, éteindre, attendre, etc. C'est très simple quand il n'y a rien d'autre à faire. Pour attendre, on décompte la valeur dans un registre. Le problème est que cela décompte très vite, à 1 million d'instructions par seconde. Le simulateur de Dauphin permettait de ralentir l'horloge, ce n'est plus possible.

Un registre est 8 bits. Utilisons un registre appelé Cx1 (on parle plus souvent de variable). En écrivant

```
      Move  #250,W      ; 1 µs
      Move  W,Cx1       ; 1 µs
 A$:  Nop               ; 1 µs
      DecSkip,EQ  Cx1   ; 1 µs, mais 2µs si saut
      Jump  A$          ; 2 µs
```

L'instruction DecSkip,EQ est nouvelle. Elle décompte (Dec) la variable, et saute (Skip) l'instruction suivante si (,) le résultat est égal à zéro (EQ).

On a un module qui met 1 milliseconde à s'exécuter (en fait 1002 microsecondes). Pourquoi ? Chaque instruction prend 1µs, sauf les Call et les Jumps qui prennent le double. La boucle va durer 4 microsecondes, on la fait 250 fois. Mais pour être précis, on a encore deux instructions d'initialisation.
On veut une attente de 0.1 seconde ? Il suffit de prendre une 2^{e} variable Cx2 et de répéter 100 fois le retard de 1 ms. La routine Del100ms (Del pour délai, retard) s'écrit :

```
\rout :Del100ms|Attente de 100ms
\mod ;Utilise Cx1 Cx2
Del100ms:
      Move  #100,W
      Move  W,Cx2
B$:   Move  #250,W
      Move  W,Cx1
  A$: Nop
      DecSkip,EQ  Cx1
      Jump  A$
      DecSkip,EQ  Cx2
      Jump  B$
      Ret
```

Quand le processeur a fait 250 fois la boucle A$, il décompte Cx2 et retourne en B$ pour refaire 250 fois la boucle A$, ceci 100 fois jusqu'à ce que CX2 soit nul.

C'est important de bien comprendre ce mécanisme. Vous faites à peu près la même chose si vous devez compter 70 objets : Vous faites des tas de 10 (compteur Cx1) et vous comptez 7 tas (compteur Cx2).

Il faut déclarer les variables CX1 et CX2, car le processeur ne connaît que des positions mémoire numérotées. Dans le 16F630 et le 16F690, il y a de la place à partir de l'adresse 16'20, donc on doit déclarer au début du programme

```
      Cx1   = 16'20
      Cx2   = 16'21
      Cx3   = 16'22
```

Ce n'est pas la meilleure façon de faire, on changera plus tard

Et maintenant on veut clignoter deux fois ?

```
      Set   PortA:#bRouge
      Call  Del100ms
      Clr   PortA:#bRouge
      Call  Del100ms
      Set   PortA:#bRouge
      Call  Del100ms
      Clr   PortA:#bRouge
      Call  Del100ms
```

Cela serait plus élégant d'utiliser une 3e variable comme décompteur

```
      Move  #2,W
      Move  W,CX3
 A$:  Set   PortA:#bRouge
      Call  Del100ms
      Clr   PortA:#bRouge
      Call  Del100ms
      DecSkip,EQ Cx3
      Jump  A$
```

C'est ce qui a été mis dans le fichier BimoInit.asi et c'est exécuté à la mise sous tension. Que faut-il changer pour clignoter 5 fois ?

Chargez et analysez le programme Bimo2. Il clignote en permanence et a une routine d'attente simplifiées qui ne réinitialise pas Cx1 et Cx2, car ils valent zéro en début du comptage, mais si on décompte on a des 1 partout, donc 255 (c'est comme pour une voiture neuve qui ferait 1km en arrière, elle afficherait 99999 km !). Si on décompte à partir de zéro, il faut 256 impulsions pour retouver le zéro. Joli truc pour économiser des instructions quand on veut le temps maximum.

7- Les Macros

Encore une petite amélioration que l'on fera à l'avenir. Ce que l'on veut c'est du Rouge ou du Vert. Il faut pas devoir se souvenir chaque fois sur quel Port cette LED bicolore a été câblée, on risque de faire faux ! Si on pense Rouge, il faut activer le bit rouge et désactiver le vert. Ceci se note dans une macro et s'écrit

```
.Macro      Rouge
      Set   PortA:#bRouge
      Clr   PortA:#bVert
.EndMacro
```

De même pour Vert et pour le haut-parleur. Il suffit maintenant de se souvenir du nom des macros Rouge, Vert, Eteint, HpOn et HpOff. Ces macros ont été ajoutées dans le fichiier BimoDef.asi Note 7 Autres possibilités des macros.

8- Son continu

Pour faire un son continu, il suffit d'activer et désactiver le haut-parleur avec la bonne période. C'est comme pour clignoter, mais on va plus vite !
Pour du 500 Hz, qui s'entend bien, cela fait 1ms avec la membrane du haut-parleur attirée et 1ms avec la membrane relâchée.

```
\prog;Bimo3.asm | Son continu
.Proc 16F630
.Ref  16F630
.Ins     BimoDef.asi
Cx1  = 16'20      ; Expliqué plus loin
Cx2  = 16'21
Cx3  = 16'22
.Loc       0      ; adresse de début d'exécution
        Call      Init
Boucle:
      HpOn
      Call    Del1ms
      HpOff
      Call    Del1ms
      Jump    Boucle

Del1ms:
      Move    #250,W
      Move    W,Cx1
  A$: Nop
      DecSkip,EQ  Cx1
      Jump    A$
      Ret

.Ins  BimoInit.asi
.Loc     16'2007
```

```
.16      16'3F94
.End
```

En modifiant la valeur 250 qui fixe la demi-période on change la fréquence. Facile d'essayer !

9- Une sirène

Pour programmer une sirène, il suffit de varier la période, qui est une constante dans le programme précédent. On déclare une variable DemiPeriode, on l'initialise avec une valeur constante et on la change plus ou moins vite selon une constante (ou une variable si on veut faire des sirènes bizarres).

Le cœur du programme s'écrit

```
NouvelleSirene:
      Move    #DemiPeriodeMax,W
      Move    W,DemiPeriode
NouvellePeriode :
      Move    #NombreDePeriodes,W
      Move    W,Cx2
MemePeriode:
      HpOn
      Call    Delai
      HpOff
      Call    Delai
      DecSkip,EQ  Cx2
      Jump    MemePeriode
      Dec     DemiPeriode
      Move    #DemiPeriodeMin,W
      Xor     DemiPeriode,W   ; Compare
      Skip,EQ
      Jump    NouvellePeriode
; on pourrait ici faire un silence avec la routine Del100ms
      Jump    NouvelleSirene

\rout:Delai|Attente de DemiPeriode*4 microsecondes
Delai:
      Move    DemiPeriode,W
      Move    W,Cx1
  A$: Nop
      DecSkip,EQ  Cx1
      Jump    A$
      Ret
```

Il faut définir les constantes pour que la fréquence passe de 1 kHz (demi-période 500 µs) à 20 kHz (demi-période de 25 µs).

```
  DemiPeriodeMax= 500/4    ; 4 us par boucle
  DemiPeriodeMin= 25/4
Nombres de Periodes = 5    ; Ajuster selon la vitesse d'évolution
```

L'assembleur calcule en nombres entiers et va arrondir le 25/4 à la valeur 8.
Dans le programme complet, appelé Bimo4, il faut aussi déclarer la variablesupplémentaire DemiPeriode.

L'instruction XOR sera vue plus tard avec les instructions arithmétiques. Elle modifie W et agit sur un bit testé par l'instruction suivante, Skip,EQ (saute si égal à zéro).

On remarque dans ce programme qu'il y a des étiquettes, qui sont des noms donnés à des positions mémoires où le processeur devra sauter pour exécuter les boucles répétées le bon nombre de fois. Il y a des constantes, précédées du signe # dans les instructions. Il y a des variables, cases mémoire qui comme des tiroirs contiennent une information que l'on initialise avec une constante et qui varie ensuite. Cela apparaît probablement encore un peu compliqué, mais toute la puissance de la programmation est là et les exemples suivant vont permettre de mieux assimiler.

10- Notes

Nous avons vu comment générer un son continu de 500 Hz, mais on voudrait maintenant toute une gamme de fréquences. Une double boucle de 5 µs plus les instructions de préparation et d'action sur le HP, appelée 255 fois donnera une fréquence de 780 Hz. Appelée 1 seule fois, cela fera 70 kHz, appelée 2 fois 35kHz, trois fois 23 kHz, etc. Les fréquences ne sont pas espacées régulièrement, il y a peu de choix dans les ultrasons, mais la précision est acceptable autour du kHz.

Pour jouer une période complète à la bonne fréquence, il faut donc 14 instructions que l'on va encapsuler dans une routine:

```
\rout:OnePer|Joue une période
\in: Period → durée de la période
      ; min=1 →16 µs  max=0≡256 →2,566ms
OnePer: Move  Period,W
      Move  W,CX1
      HpOn
  B$: Nop   ; boucle 5us
      Nop
      DecSkip,EQ  CX1
      Jump  B$

      Move  Period,W
      Move  W,CX1
      HpOff
  C$: Nop
      Nop
      DecSkip,EQ  CX1
      Jump  C$
      Ret
```

Pour jouer cette période sans cesse, il suffit d'initialiser la période et boucler. C'est une autre façon d'écrire Bimo3.asm.

```
Deb:  Move  #100,W      1 kHz
      Move  W,Period
L$:   Call  OnePer
      Jump  L$
```

Une note a une ceraine durée, que l'on va exprimer en nombre de périodes. Pour la même durée, une note élevée a une période courte et il faut plus de périodes. Le produit est constant si la durée est la même.

La routine Note joue le bon nombre de périodes NPer selon la valeur de Period. Les variables des routines sont initialisées par les constantes qui correspondent aux notes. Choisissons pour la note ''Do'' une période de 223 et un nombre de période de 66.

Le ''Ré'' a une période de 201 (diminution de 2 fois racine 12eme de 2) et une période de 74. Le produit de la période par le nombre de période doit être constant.

```
\rout:JoueNote|Joue une note
\in:Perio Période
\in:W durée en multiple de 20 périodes
JoueNote:
A$:   Move  #20,W
      Move  W,CX2
 B$:  Call  Per
      DecSkip,EQ  CX2
      Jump  B$
      DecSkip,EQ Duree
      Jump  A$
      Ret
```

Le programme pour jouer la note doit préparer la période et la durée. Pour jouer une fois ''Do-Ré'' il faut écrire :

```
Deb:  Move  #223,W      ; Do
      Move  W,Period
      Move  #66,W
      Move  W,Duree
      Call  Note
      Move  #201,W       ; Ré
      Move  W,Period
      Move  #74,W
      Move  W,Duree
      Call  JoueNote

X:    Jump  X     ; On reste ici!
```

Ce n'est pas très élégant et on va faire mieux avec des tableaux de notes. Vous pouvez quand même tester le programme Bimo5.asm

11- Tableaux

Le PIC est assez spécial, mais très efficace pour accéder l'information dans un tableau.
L'instruction Add W,PCL permet de sauter par dessus le nombre d'instruction dans W.
Par exemple si on écrit

```
      Move  #3,W
      Add   W,PCL
      Jump  P0
      Jump  P1
      Jump  P2
      Jump  P3
      Jump  P4
```

Le programme va continuer en `P3:`

A noter que cette addition est 8 bits, et si le résultat de l'addition n'est pas inférieur à 256, il faut programmer de façon plus compliquée. Nos programmes pour apprendre seront toujours petits et en début de mémoire. La documentation sous xxx (à faire) explique comment gérer des tables qui ne sont pas en page 0.

Cet exemple montre que si, comme une routine du Bimo sait le faire, on compte le nombre d'actions sur le poussoir, c'est facile de sauter ensuite pour à des tâches différentes.

On trouve plus souvent dans les tables l'instruction RetMove #Val,W, qui comme son nom l'indique, se place à la fin d'une routine et retourne au programme appelant en transportant dans W la valeur mise dans l'instruction.

Si dans le programme on veut connaître la valeur dans la table qui correspond à la valeur dans W, on apelle la routine suivante :

```
  GetTable : Add  W,PCL
             RetMove  #Val0,W
             RetMove  #Val1,W
             RetMove  #Val2,W
             RetMove  #Val3,W
             RetMove  #Val4,W
```

Il ne faut pas demander une valeur supérieure à 4 !!
Prenons un exemple plus précis

Les notes successives d'un morceau nécessitent deux octets. On pointe avec une variable Note pour lire une paire et mémoriser la période et ne nombre de périodes. ''Do'' est en position 0, ''Ré'' en 2, etc, et on peut déclarer ces valeurs pour ensuite écrire le morceau avec des noms pour les notes.

```
GetNote:  ; Le no de la note est dans Note

      Call        TaNotes
      Move        W,Periode
      Inc         Note
      Call        TaNotes
      Move        W,NPer
      Ret
```

; Déclarations

```
Do    = 0
Re    = 2
Mi    = 4
Fa    = 6
Sol   = 8
La    = 10
```

```
TaNotes:    Move        Note,W
      Add   W,PCL
      RetMove     #223,W      ; Période (Do)
      RetMove     #66,W       ; Nb de périodes
      RetMove     #201,W      ; Ré
      RetMove     #74,W
      RetMove     #180,W      ; Mi
      RetMove     #84,W
      RetMove     #166,W      ; Fa
      RetMove     #88,W
      RetMove     #148,W      ; Sol
      RetMove     #100,W
      RetMove     #132,W      ; La
      RetMove     #111,W
      RetMove     #116,W      ; Si
      RetMove     #126,W
      RetMove     #111,W      ; Do2
      RetMove     #133,W
      RetMove     #96,W ; Ré2
      RetMove     #150,W
```

```
Si    = 12
Do2   = 14
Re2   = 14
```

Pour jouer un Sol :

```
      Move  #Sol,W
      Move  W,Note
      Call  GetNote
      Call  JoueNote
```

Voilà qui est clair et élégant.
Le programme Bimo6.asm vous évite de retaper tout cela.

12- Comment faire de la musique

On sait jouer une note, il suffit maintenant de créer une table avec les notes, avec un pointeur PtMelo qui avance dans la table. Comment s'arrêter ? On pourrait décompter le nombre de notes. C'est plus élégant de terminer la table par le code d'une note qui n'existe pas, 16'FF. On peut ainsi rajouter de notes sans devoir recalculer une longueur.

```
      Clr   PtMelo
Joue: Call  Melo1 ; lit une note
      Move  W,Note
      Call  GetNote
      Call  JoueNote
      Jump  Joue

\rout;Melo1|
      Move  PtMelo,W
      Inc   PtMelo
      Add   W,PCL
      RetMove  #Do,W
      RetMove  #Re,W
      RetMove  #Mi,W
      RetMove  #-1,W    ; fin
```

Le programme complet est Bimo7.asm

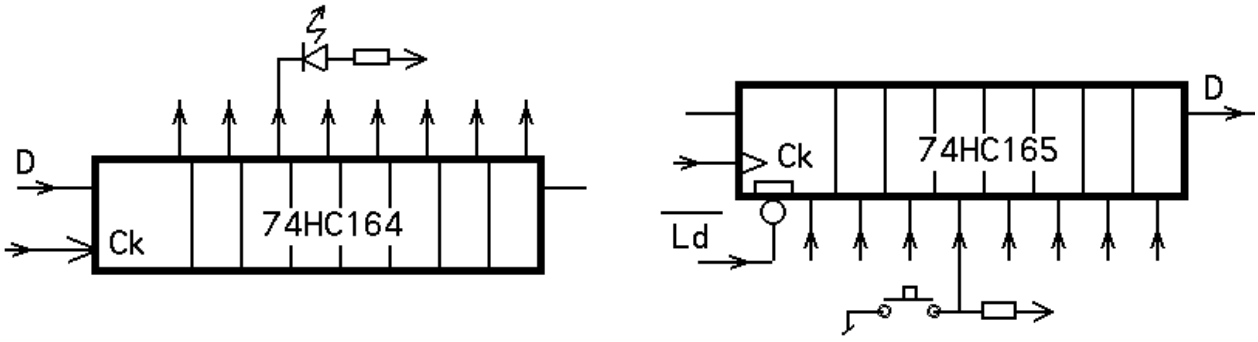
Dans la routine Melo,1, on remarque que l'écriture du morceau est très lourde. Une macro nommée par exemple " n4" permet de cacher ces RetMove, et écrire 4 notes par ligne (ou plus si on prolonge la macro) de la facon suivante

n4 do,re,mi,do

Le programme Bimo8.asm est préférable si vous voulez coder votre mélodie préférée. N'espérez pas trop de qualité. Votre téléphone joue mieux car il a un circuit spécial avec des compteurs 16 bits et des tables de notes permettant des accords et des longueurs de notes différentes.

Transferts série

Un registre à décalage est un circuit électronique formé de bacules (flip-flops) en cascade, avec un signal dit d'horloge (clock) qui pousse l'information d'une bascule à la suivante.



Il existe des circuits registres avec 8 ou 16 sorties simples ou amplifiées, ou avec 8 entrées. On

peut les cascader. L'intérêt est que quelque soit la longueur, il suffit de 2 ou 3 broches su le processeur pour en avoir beaucoup plus pour commander des LEDs ou lire des poussoirs.
A noter qu'avec le circuit 74HC164 pendant que l'information se décale, les sorties changent et un registre parallèle doit être ajouté sur les sorties si cela est gênant. Avec des LEDs il suffit de décaler le plus vite possible, d'où l'avantage d'une routine rapide en assembleur.

```
\rout:Aff8 bits sur BimoLeds – Un 0 allume
\in : Data  contient la valeur à afficher
Aff8:
        Move        #8,W
        Move        W,CX2
 Sb$:   ; MSB first
        RLC         Data     ; poids faible dans C
        Clr         PortA:#bDaOut ; Prepare un 0
        Skip,CC

\rout:Aff8 bits sur BimoLeds – Un 0 sur le registre allume
\in : Data  contient la valeur à afficher
Aff8:
        Move        #8,W
        Move        W,CX2
 Sb$:   ; MSB first
        RLC         Data     ; Pousse le bit de poids faible dans C
        Clr         PortA:#bDaOut ; Prepare un 0
        Skip,CC
        Set         PortA:#bDaOut ; Si C est à 1 on remplace pour un 1
        Set         PortA:#bCkOut ; Impulsion CK pour pousser dans le registre
        Clr         PortA:#bCkOut
        DecSkip,EQ  CX2         ; On compte 8 bits
        Jump        Sb$
        RLC         Data        ; Réaligne le Data pour qu'il ne soit pas modifié
        Ret
```

A noter que un bit à 1 dans le registre data doit amener un zéro sur la sortie du registre, si les diodes sont câblées avec leur anode au + 5V, ce qui est plus efficace.

Pour lire un registre, il faut une première impulsion pour charger le registre en parallèle, et ensuite on transfert dans le registre data un bit lu après l'autre.

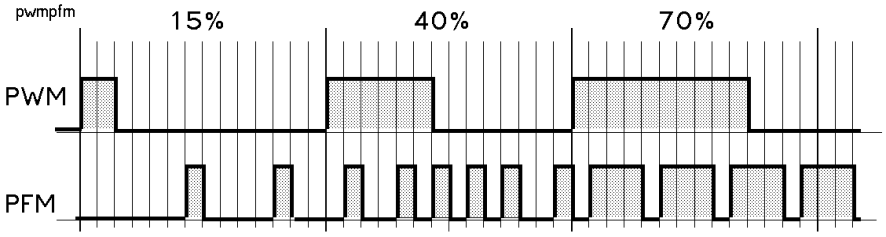
```
\rout:Get8 bits un 0 correspond à un poussoir actif et doit être mémorisé comme
un 1
\out: Data  contient la valeur lue
Get8:
        Set         PortA:#bLdOut ; Impulsion CK pour transférer dans le registre
        Clr         PortA:#bLdOut
        Move        #8,W
        Move        W,CX2
 Rb$:   ; MSB first
        SetC        ; Prépare si le bit lu ets 1
        TestSkip,BS  PortA:#bDaIn
        ClrC        ; Non, c'est un zéro puisque le bit lu est « set »
        RLC         Data     ; Pousse C dans le bit de poids fort
        Set         PortA:#bCkOut ; Impulsion CK pour pousser dans le registre
        Clr         PortA:#bCkOut
        DecSkip,EQ  CX2         ; On compte 8 bits
        Jump        Rb$
        Ret
```

L'interface SPI utilise exactement ce principe. I2C est une variante plus astucieuse.
http://www.didel.com/picg/doc/DocSPI.pdf http://www.didel.com/picg/doc/DocI2C.html

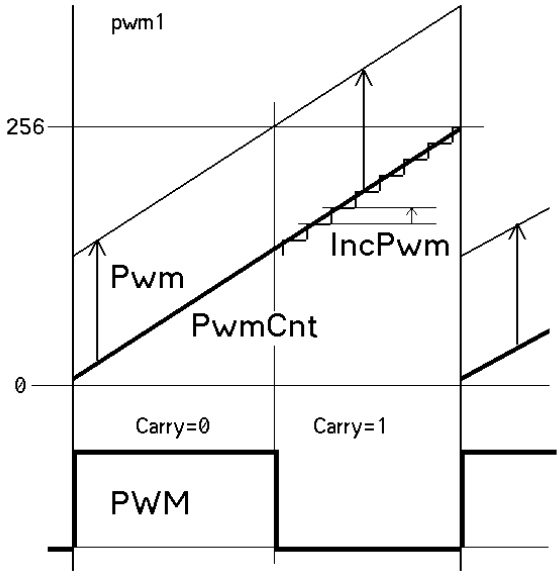
PWM et PFM
Un moteur demande en général trop de courant pour être commandé directement par le processeur. Si le moteur tourne dans un seul sens, c'est facile, il suffit d'un transistor.

Si le moteur doit tourner dans les deux sens, il faut un montage en pont que l'on peut faire avec 4 transistors, mais c'est plus facile avec des circuits ''pont en H'', qui existent pour différentes tensions et courants.
Commander un moteur en tout ou rien de fait comme pour allumer une LED. Si le moteur doit tourner dans les deux sens, il faut 2 bits. Les combinaisons 01 et 10 font tourner dans un sens et dans l'autre. Les combinaisons 00 et 11 ne font pas tourner, avec parfois la différence que l'une des combinaison freine le moteur, et pas l'autre.

Pour agir sur la vitesse du moteur (et sur l'intensité d'une LED) le principe est d'envoyer des impulsions, à une fréquence suffisante pour que l'inertie du moteur ou la persistance lumineuse donne l'effet de continuité.
Avec le PWM (Pulse Width Modulation), la période est constante et le pourcentage actif varie.
Avec le PFM Pulse Frequency Modulation), la durée d'excitation du moteur est constante et l'espacement des impulsions varie.



Le PFM est très intéressant avec des moteur de mauvaise qualité ou qui tournent trop vite. On choisit une durée d'impulsion suffisament longue pour pour que le moteur démarre (1-10ms). On attend ensuite en fonction de la vitesse de rotation moyenne voulue. A faible vitesse, les a-coups sont visibles, mais on obtient des faibles vitesses que le PWM ne permet pas.



PWM unidirectionnel

```
; Incr Compteur PWM
          Move            #IncPWM,W
          Add             W,PwmCnt
          Skip,NE
          Add             W,PwmCnt
\b; Pwm1  helice
          Move            PwmCnt,W
          Add             Pwm1,W
          Skip,CC
          MotOn
          Skip,CS
          MotOff
```

PFM unidirectionnel

```
          Move            Pfm1,W
          Add             W,Pfm1Cnt
          Skip,CC
          MotOn
          Skip,CS
          MotOff
```

La commande bidirectionelle pose le problème de la représentation des vitesses positives et négatives. Le bit de signe peut être dans une variable séparée, comme bit de poids fort (2

solutions selon que les 7 autres bits qui donnent la vitesse sont en complément à 1 ou non) ou comme bit de poids faible. C'est cette dernière solution que nous préférons en général et donnons en exemple :

PWM bidirectionnel

```
          Clr             Prep1
\b;    Pwm1  bidir
          Move            #IncPWM,W
          Add             W,PwmCnt
          Skip,NE
          Add             W,PwmCnt
          TestSkip,BS     Pwm2:#bSens
          Jump            Pos2$
Neg2$:
          Move            PwmCnt,W
          Add             Pwm2,W
          Skip,CC
          Set             Prep1:#bN2
          Jump            Z2$
Pos2$:
          Move            PwmCnt,W
          Add             Pwm2,W
          Skip,CC
          Set             Prep1:#bP2
Z2$ :     ; suite
```

PFM bidirectionnel

```
          Clr             Prep1
\b; Pfm1
          TestSkip,BS Pfm1:#bSens
          Jump            Pos1$
Neg1$:
          Move            Pfm1,W
          Add             W,Pfm1Cnt
          Skip,CC
          Set             Prep1:#bN2
          Jump            Z1$
Pos1$:
          Move            Pfm1,W
          Add             W,Pfm1Cnt
          Skip,CC
          Set             Prep1:#bP2
Z1$:

; Prep1 est copié dans le port de
sortie quand tous les bits ont été
préparés
```

Les programmes Bimo10, BPlus10 font faire un aller et retour au robot. Adapter le programme pour le PWM et sauver sour Bimo11/BPlus11. A noter que l'attente de 10ms n'est pas nécessaire, mais si on supprime cette attente, l'évolution est trop rapide et on ne peut que pqrtiellement compenser en mettant IniCntCy au maximum et IncVit au minimum de 2.

Timers

Il y plusieurs timers qui facilitent la gestion des tâches avec des contraintes de temps. Leur maîtrise prend du temps. Contentons-nous ici de comprendre le Timer0 (TMR0) qui exister sur tous les PICs et est très primitif. C'est un compteur 8 bits avec un prédiviseur, c'est à dire un diviseur par 2,4,8.. 256 selon 3 bits du registre Option. Quand ce compteur déborde, une bacule est activée (bit TOIF dans le registre Option).
Si on veut allumer une LED pour 50ms, on peut faire une boucle d'attente. On peut aussi utiliser le timer0. Prédivisons par 256, donc à 4MHz 256 microsecondes par incrément du compteur. Pour 50ms, il faut 195 incréments. On charge TMR0 avec la valeur –195 et on attend que TOIF s'active.

```
          Clr             Option:#TIOF
          LedOn
          Move            # -195,W
          Move            W,TMR0
  At$:    TestSkip,BS     Option:#TIOF
          Jump            At$
          LedOff
```

Le programme Bimo12/BPlus12 clignote la Led du Bimo. Essayez de le faire seul avant de le charger. A noter que c'est trop rapide, le timer est d'abord fait pour les temps courts.

Ceci n'est pas un bon exemple d'utilisation. Le timer est plus intéressant pour mesurer la durée d'une impulsion (par exemple un signal infrarouge).
On mets le timer à zéro au début de l'impulsions, et il compte à la vitesse assignée pendant que le programme fait par exemple du PWM sur les moteurs en surveillant assez souvent si l'impulsions est terminée.

```
          TestSkip,BS     Port :#bImp
          Jump            PasEncore
          Clr             TMR0
 At$ :    … actions diverses
          TestSkip,BC     Port :#bImp
          Jump            At$
          Move            TMR0,W
```

; On pourrait tester TOIF pour vérifier qu'il n'y a pas eu dépassement de capacité.

Pour des explications et exemples sur les timers, voir www.didel.com/DocPic/PicTimers.html (en préparation).

Interruptions
Les interruptions permettent la ''simultanéité'' de plusieurs tâches avec naturellement une dégradation des performances qu'il faut savoir évaluer. Limitons-nous à un cas très simple.
Le Timer0 lance une interruption chaque fois qu'il déborde. Cette interuption peut par exemple surveiller plusieurs signaux et appeler la routine PWM. Le programme principal sera coupé pour 20 à 100 microsecondes toutes les millisecondes par exemple.

Une interruption qui survient n'importe quand doit pas perturber le programme principal. Il faut sauver, puis rétablir les registres W et F (qui contient en particulier le Carry C). Si des variables sont communes entre le programme principal et la routine d'interruption, il faut aussi réfléchir.

La routine d'interruption du PIC est à l'adresse 4. Pour que le débordement du Timer déclenche une interruption, il faut que l'interruption soit autorisée en général (bit GIE) et que celle du timer en particulier soit aussi autorisée (bit TOIE).
Au début de la routine d'interruption, il faut que la cause de l'interruption (ici TOIF activé) soit neutralisée. La roitine d'interruption se termine par l'instruction RETI qui rétablit la possibilité d'interruptions qui a été bloquée au début de la routine.

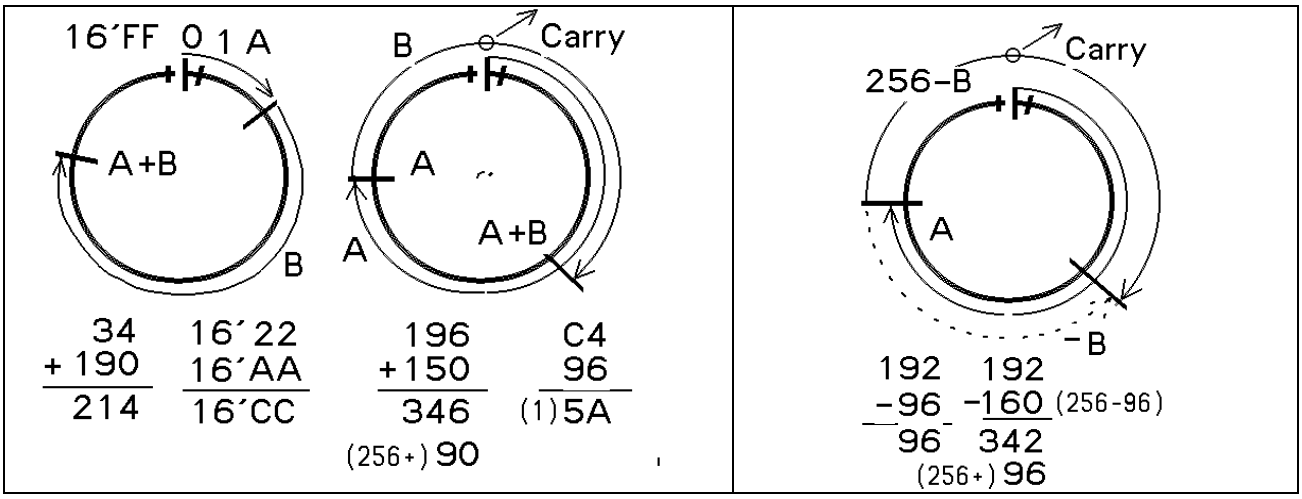
```
; les variables SaveW etc ont été déclarées
.Loc         0
            Jump          Deb

.Loc         4
; Interrupt tous les 256 x IniTim0 us
            Move          W,SaveW           ; Ne modifie pas F
            Swap          F,W               ; Truc !
            Move          W,SaveF
; On désactive le bit qui a demandé l'interruption
             Clr          Intcon:#TOIF
            Move          #-IniTmr0,W
            Move          W,TMR0
; On fait ici ce qu'il faut faire à chaque interruption
;            ...
            Swap          SaveF,W           ; Trucs !
            Move          W,F
            Swap          SaveW
            Swap          SaveW,W
            RetI

\b;Programme principal
Deb:
 ; Initialisation des ports
            Move          #2'00000111,W     ; Prescaler :256
            Move          W,Option
            Clr           IntCon:#TOIF      ; il est peut-être actif
            Move          #2'10100000,W     ; GIE and TOIE on
            Move          W,IntCon
Loop:        ;
```

Le programme Bimo13/Pplus13 gère le Pfm par interruption. Dans le programme principal, il suffit de modifir la valeur du Pfm.

Pour des explications et exemples sur les interruptions, voir www.didel.com/DocPic/PicInter.html (en préparation).

Arithmétique
Les calculs même très simples sont difficiles avec le PIC. Les choses se passent en binaire d'abors, sur 8 bits à la fois ensuite, avec une instruction de soustraction inhabituelle.

Avec des nombres négatifs, il faut avoir biebn compris. Donc si votre application doit fire des calculs précis, des interpolations, acheter un 18F, 24F ou 32F avec des kilobytes de mémoire et écrivez en C. Les 16F sont des microcontrôleurs, il sont fait pour agir sur des bits en entrée-sortie, pas sur des nombres de plus de 8 bits. Mais quand on sait, c'est très joli ce que l'on peut faire. L'addition est facile: on ajoute deux nombres de 8 bits. Si le résultat est nul, le flag Z (Equal) est activé et si l'instruction Skip,EQ suit, l'instruction suivante ne sera pas exécutée. Si le résultat déborde, le flag C (Carry) est activé. Les instruction Skip,CS et Skip,CC permettent de prendre une décision appropriée en cas de débordement.



Par exemple, si on veut ajouter 2 nombres de 16 bits, qui ont été déclarés comme variable Nb1H Nb1L Nb2H Nb2L, avec le résultat dans Nb1H Nb1L, on doit écrire

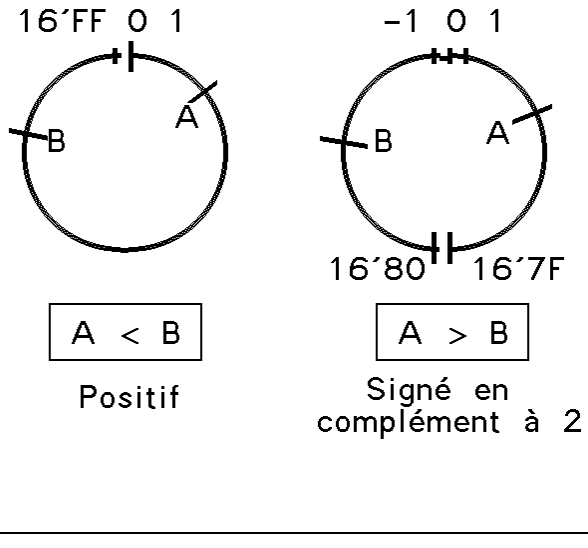
```
Move        Nb2L,W      ; Ou  Move #ValeurL,W
Add         W,Nb1L
Skip,CC
Inc         Nb1H
Move        Nb2H,W      ; Ou  Move #ValeurH,W
Add         W,Nb1H
```

La soustraction a deux anomalies par rapport à tous les autres processeurs. D'une part au lieu de soustraire, le processeur ajoute le complément à 2, ce qui revient au même pour le résultat, mais pas pour le Carry (figure). D'autre part, les opérandes sont inversés, ce que la notation CALM mets heureusement bien en évidence :

Sub #Val,W n'existe pas, mais on peut utiliser Add #-Val,W
Sub W,#Val,W existe et calcule #Val-W, résultat dans W
Sub W,Reg existe et calcule W-Reg, résultat dans Reg
Sub W,Reg,W existe et calcule W-Reg, résultat dans W

Les nombres négatifs dont difficile à maîtriser, car il existe plusieurs représentation et le processeur n'as pas comme d'autre processeurs un flag overflow et des instructions de comparaisons.
La représentation naturelle des nombres négatif est en complément à 2. Le cercle arithmétique permet de bien voir la différence. Mais on voit sur la figure que la notion de plus grand, plus petit impose des instruction différentes.

En résumé, tant qu'il n'y a pas de dépassement de capacité, c'est relativement simple. Un simulateur est très utile pour tester des modules de programmes avant de les mettre dans l'environnement temps réel de l'application.



Pour des explications et exemples sur les opérations arithmétiques fréquentes (avec des macros qui évitent de comprendre les détails, voir www.didel.com/DocPic/PicArith.html (en préparation).

Il n'y a pas de programme exemple sur le disque pour les calculs arithmétiques. Le lecteur doit lire la documentation plus complète sur les instructions qui contiennent des exemples d'applications
http://www.didel.com/picg/picg87x/CoursPicg87x.html
En anglais : http://www.didel.com/picg/doc/PicSoft.pdf
Il faut ensuite se lancer pour écrire ses propres programmes et étudier les documents de http://www.didel.com/picg/doc/ pour acquérir plus d'expérience.

Le sottisier du programmeur PIC
Mon nouveau programme ne réagit pas
- Les .Loc pour les variables et le code ont été oubliés ou sont sont faux
- L'initialisation des périphériques est incomplète ou incorrecte
- Une table est à cheval ou en dehors de la page définie par le PcLatH
- Le programme est piégé dans une boucle où il ne fait rien
- Une macro derrière un saut conditionnel fait plus d'une instruction
- Un Ret a été oublié à la fin d'une routine

Le programme marchait bien il y a 5 minutes, je n'ai rien changé (ou si peu) et cela ne marche plus!
- Un # a été oublié
- Un 16' a été oublié pour un nombre BCD ou hexa
- La ligne du port n'a pas la bonne direction
- Un Set, Clr bit agit sur le mauvais port, sur les bits d'à côté
- Le compteur de boucle est réinitialisé dans la boucle
- La variable compteur de boucle est utilisée par une routine appelée dans la boucle
- Un Skip,CS devrait être un Skip,CC, idem TestSkip,BS ou BC
- Une macro de plus d'une instruction a été ajoutée derrière un saut conditionnel
- La table a débordé de sa page (ne pas oublier les .If pour signaler cette erreur)
- Les variables débordent la zone mémoire (ne pas oublier le .If..)
- Une variable n'est pas accessible car on n'est pas dans la bonne banque.

Pour une analyse des instructions avec des exemples
http://www.didel.com/picg/picg84/PicgDoc.html
http://www.didel.com/picg/picg87x/CoursPicg87x.html

Pour une liste des instructions commentées, en anglais, avec à la fois les notations Microchip et CALM : http://www.didel.com/picg/doc/PicSoft.pdf

Notes

Note 0 Remerciements
L'assembleur CALM a été essentiellement développé par Patrick Faeh en 1980-83. C'est un programme remarquablement compact et efficace, difficile à adapter à de nouveaux processeurs. Des essais de réécriture en Java dans le cadre de projets d'étudiants de l'EPFL n'ont pas été satisfaisants.
L'environnement SmileNG est un travail de vacances de Sebastian Gerlach (étudiant Microtechnique 3e année) en 1998, transposant et améliorant sur PC l'environnement Smile des Smakys utilisés depuis 1978.
Les notations CALM ont été développées au LCD/LAMI EPFL dès 1974 pour éviter aux étudiants de devoir travailler avec la gabegie de notations différentes de chaque fabriquant. Des assembleurs CALM ont été écrits pour plus de 15 processeurs (les PICs comptent pour un).

Note 1 sur les assembleurs, dévermineurs et outils professionnels.
Vous ne voulez pas travailler avec SmileNG ? La conversion des programmes en assembleur Microchip est facile si on mémorise leurs mnémoniques parfois cabalistiques. En sens inverse, c'est plus facile puisque si vous savez de que fait l'instruction, sa traduction en CALM est logique.

```
Microchip                                          CALM
PROC   16F630                                      .Proc 16F630
                                                   .Ref   16F630
BINTER EQU    0      ; Interupteur en RA0          bInter = 0    ; RA0
DIRA   EQU    B'11111'     ; RA4..0 en entrée      DIRA     = 2'011111
BLED   EQU    7      ; Led en RC7                  BLED     = 7
DIRC   EQU    B'00000000'  ; Tout en sortie       DirC   = 2'000000
```

```
DEBVAR EQU    0x20                                  ; DebVar défini dans 16F630.ref
ORG    DEBVAR                                       .Loc   DebVar
OLDPORT   RES1                                      OldPort : .Blk.16  1
;Début du programme
       ORG    0                                     .Loc   0
DEBUT                                               Debut :
       MOVLW  DIRA                                         Move   #DirA,W
       TRIS   5                                            Move   W,TrisA
       MOVLW  DIRB                                         Move   #DirC,W
       TRIS   6                                            Move   W,TrisC
BOUCLE                                              Boucle:
       BTFSS  5,BINTER      ; Si 0 on allume               TestSkip,BS  PortA:#bInter
       BCF    6,BLED ; 0 allume                            Clr    PortC:#bLed
       BTFSC  5,BINTER      ; dans l'autre cas, BSF       TestSkip,BC PortA:#bInter
       6,BLED ; 1 éteint                                   Set    PortC:#bLed
       GOTO   BOUCLE                                       Jump   Boucle
       END                                          .End
```

L'avantage de l'environnement Microchip est qu'un dévermineur est disponible, l'environnement C est efficace et permet l'insertion de routines en assembleur Microchip uniquement.

SmileNG est un environnement idéal pour débuter et faire des projets avec des petits processeurs dans des applications avec des contraintes de temps réel (capteurs, moteurs). Si des applications plus ambitieuses sont visées, avec des gestions de messages, des interfaces bus CAN ou autre, du traitement d'images ou de signaux analogiques, des opérations mathématiques, il est évident qu'il faut se mettre au C avec un système de développement et des processeurs plus performants, donc plus coûteux.

Note 2 sur les pseudos

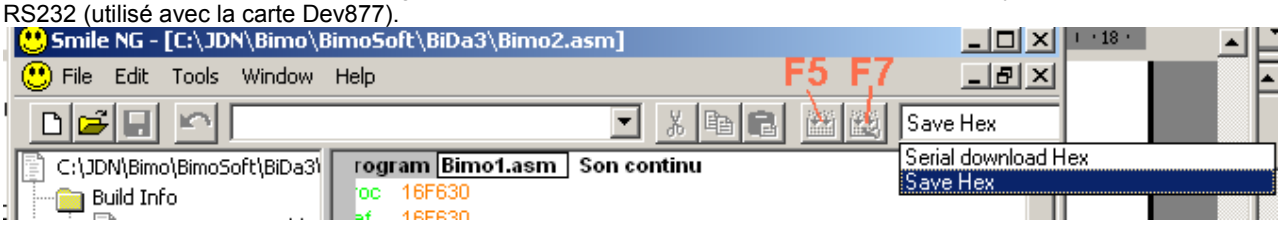
Les pseudo-instructions (pseudos pour être plus court) sont des commandes à l'assembleur, elles agissent sur l'insertion des fichiers, la mise en page et la traduction des instructions. Toutes les pseudos commencent par un point. Elles sont composées soit d'une pseudo-opération seulement, soit d'une pseudo-opération suivie par une ou plusieurs expressions (séparées par des virgules). Une liste adaptée au PICs se trouve en
www.**didel**.com/dev877/cours/2ass/2Ass.doc

Note 3 sur les erreurs d'assemblage et leur correction

A la fin de l'assemblage d'un programme incorrect, les erreurs sont insérées dans le source et la première erreur apparaît au centre de l'écran (dans certains cas, juste en dessous de l'écran descendre le pointeur). Il suffit de corriger les erreurs et relancer l'assemblage, pas besoin d'effacer le message d'erreur. Si l'erreur est dans un fichier inséré, il faut corriger et sauver sur disque. Parfois, l'erreur n'est signalée que à gauche du listage. Il faut cliquer sur la mention erreur en rouge pour que l'erreur apparaisse.
Les espaces et tab sont assez libres pour la mise en page. Mais devant le : (deux-points) d'une étiquette, il n'en faut pas.
Les fichiers édités avec l'éditeur SmileNG ne passent pas sans autre dans Word. Dans l'autre sens, il faut ajouter des retours à la ligne et enlever l'espace devant les deux-points.

Note 4 Possibilités de SmileNG.

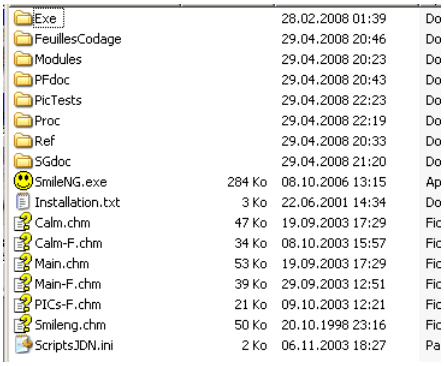
File : Dans le menu «Page Setup»on peut configurer l'apparence sur l'écran et l'imprimante. Très pratique d'avoir des impressions compactes sur 2 ou 3 colonnes.
Edit : Il est mentionné que la touche F8 active ou annule le décodage des séquences LILA. Ces séquences commencent par un \ en début de ligne, suivi d'un mot clé et d'un ; Pour la liste, voir le lien ci-dessous.
Tools : Permet de configurer ce que fait l'assembleur après l'assemblage (ne pas modifier sans comprendre) et de changer l'affichage sur l'écran.
Touche F7 ou icone : Assemble seulement
Touche F5 ou icône : Exécute la séquence sélectionnée dans la fenêtre, modifiable dans Tools. Save Hex sauve le binaire sur disque pour pouvoir le programmer avec le PicKit2. Serial download permet d'envoyer le binaire sur un COM RS232 (utilisé avec la carte Dev877).



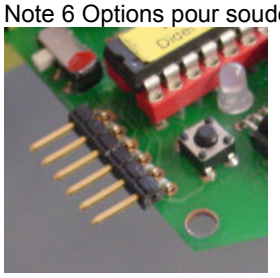
On consultera www.didel.com/dev877/PdSmileNG.doc pour plus de détails.

Note 5 Les fichiers de SmileNG07.zip

En bref, on a dans les répertoires
Exe – l'assembleur Calm et d'autres essais non doccumentés
FeullesCodage – Les instructions et qq autres info résumées
Modules – des fichiers auxiliaires
PFdoc – La doc CALM de Patrick Faeh
PicTest – Des programmes de test pour 16F630, 676, 870, 877
Proc – Description des processeur PIC (Z80, 68000 etc sur demande)
Ref – Noms réservés pour les registres et bits des processeurs PICs
SGdoc – La doc SmileNG de Sebastian Gerlach
SmileNG.exe - L'exécutable
xxx.chm - Un help, malheureusement pas assez complet
ScriptsJDN.ini – Des lignes de commandes pour différentes fonctionnalités, si on veut compléter le fichier Module\Script.ini

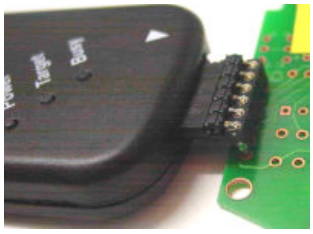


Note 6 Options pour souder le connecteur de programmation du Bimo.

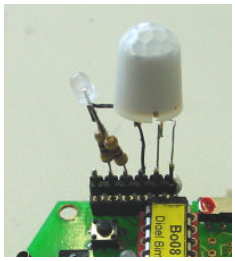


On pourrait souder directement le connecteur coudé sur le bimo, mais cela déborde et peut gêner le mouvement

La solution conseillée est d'utiliser un connecteur intermédiaire male coudé, qui reste attaché au PicKit2.



Un connecteur femelle 0.7mm (fourni avec les PicKit2 de BricoShop) est soudé sur le circuit du Bimo.



On peut monter des LEDS ou capteur sur le connecteur pour tester des extensions. Trois lignes du processeur sont disponibles.

Note 7 Autres possibilités des macros

Les macros sont très utiles avec les .If … .Endif pour écrire des programmes qui sont compatibles avec différents processeurs. Par exemple, .. ?
Pour toutes les possibilités, voir www.**didel**.com/dev877/cours/2ass/2Ass.doc

Jdn 090316