Apprendre à programmer avec le 16F877A

Exemples et exercices sous www.didel.com/pic/Cours877-1.zip

Le but est d'apprendre à écrire des programmes en assembleur pour des applications utilisant un code efficace sur des microcontrôleurs en général plus petits. Notre démarche est de tester les programmes sur un 16F877 qui a 4 ports d'entrée-sortie, ce qui facilite la compréhension du fonctionnement. Le Microdule 16F877 est aussi idéal pour mettre au point des applications utilisant des processeurs en boîtier 6, 8, 14 pattes (voir www.didel.com/pic/Deverminage.pdf).
En plus de la carte 877, 4 cartes Microdules et des composants aident pour les tests.
La grande idée des Microdules est que vous ne les utilisez pas dans votre application. Ils restent toujours prêts pour tester des routines et préparer des nouveaux projets.

Notons encore que pour éviter d'allonger ce document avec des informations essentielles pour maîtriser tous les aspects d'un PIC, mais pas indispensables quand on n'a pas encore le besoin pour son application, plusieurs renvois sont faits à des documents plus spécialisés, dont une liste exhaustive est faite dans www.didel.com/pic/ToutSurLesPics.pdf

Matériel nécessaire

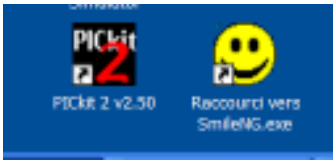
Pickit2	65.-
Kit Microdule M2840Eval	140.-
16F877A	10.-

Doc sous
www.didel.com/08micro/M2840Eval.pdf

La doc générale sur les Microdules est sous
www.didel.com/pic/Microdules.pdf

Table des matières

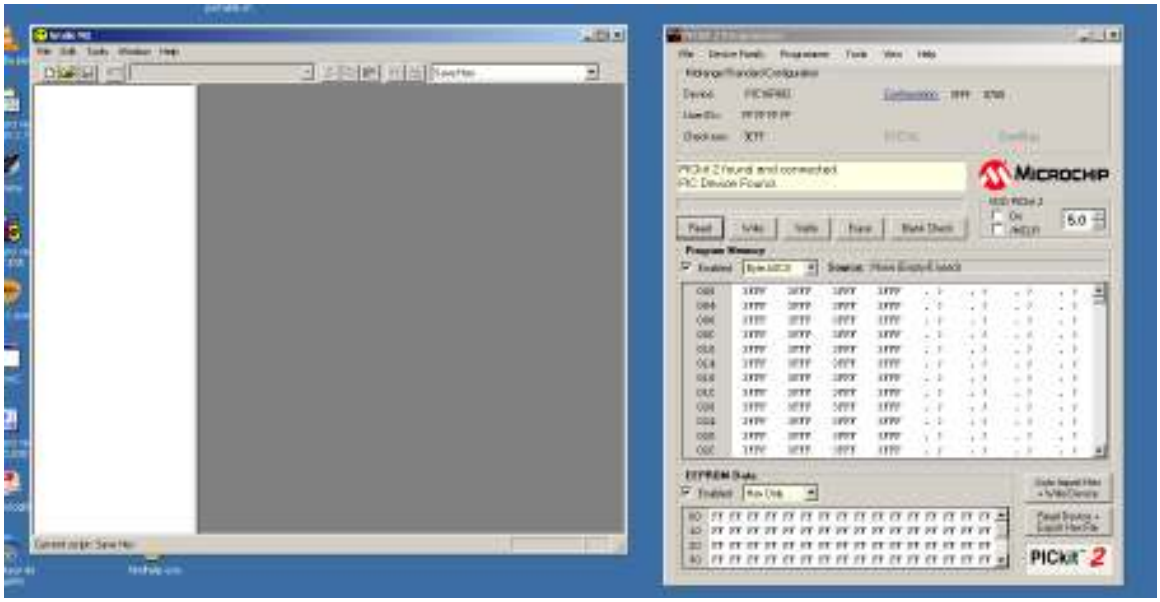
Introduction



Avant de pouvoir faire les exercices de ce document, il faut installer **SmileNG** www.didel.com/pic/SmileNG.zip extraire dans c: ce qui crée le dossier c:SmileNG Ouvrir et faire un raccourci à SmileNG.exe
Pickit2 : Chercher sous google ''Pickit2 install'' et choisir sur le site de Microchip **Pickit2 install, 3.9 MB**).

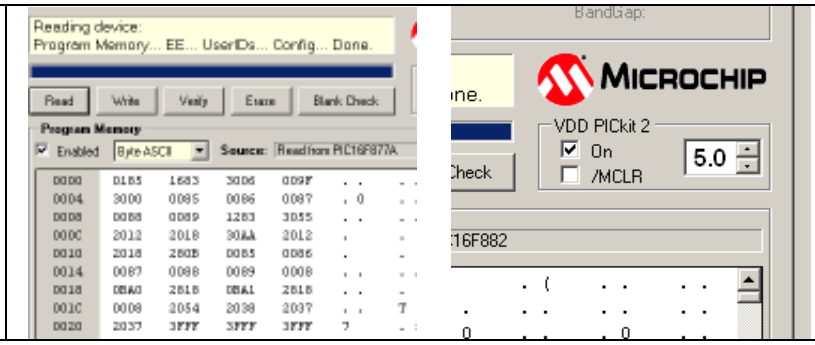
Les **exemples de programmes** associés à cette partie se trouvent sous www.didel.com/pic/Cours877.zip. Mettre ces fichiers dans un dossier Cours877.

Connectez le M2840 au Pickit2 relié au PC via USB. Exécutez SmileNG et Pickit2. Le Pickit2 doit avoir reconnu le 16F877.



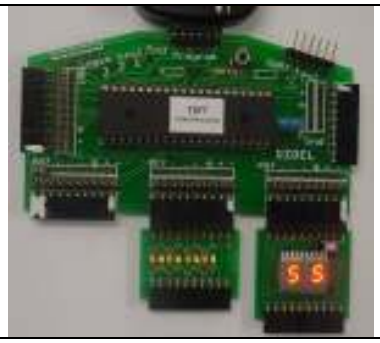
Lire le 16F877
Cliquez sur ''Read''. Le contenu s'affiche.

Vérifiez que la tension est 5V et cliquez sur la case On. La LED du. M2840 s'allume et le programme T877 s'exécute. Il alterne toutes les lignes.

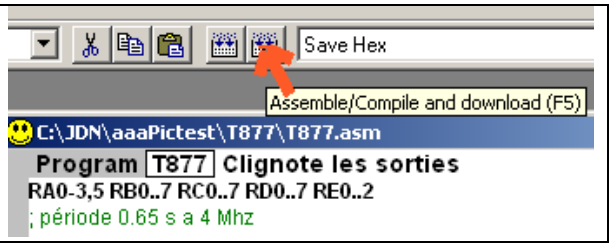


Déplacer l'afficheur sur les ports pour vérifier que cela clignote partout où cela a un sens (sauf RA7, 6, 4).

Le programme de test T877 est très utile pour vérifier en tout temps que le 16F877 est en ordre. On le recharge dans le Pickit2 en cliquant sous *File – Import Hex* – chercher le dossier *Cours877* et charger *T877.hex*.
Cliquer ensuite sur *Write*



Charger sous SmileNG le programme T877.asm. On voit qu'il initialise les ports en sortie et envoie des valeurs complémentaires avec une attente entre deux.
Assembler en cliquant à l'endroit indiqué ou avec la touche F5. Ceci recrée le fichier T877.hex que l'on peut recharger dans le Pickit2.



Ne vous inquiétez pas si vous ne comprenez pas tout ce programme, il sera expliqué par petites étapes plus loin.

Exercice : modifier la valeur Motif, ré-assembler et programmer.
Note : Pour chaque modification, une fois que l'on a vu ''Assembly successful'' on clique dans Write et le Pickit2 confirme ''Reloading Hex file''. Ceci est très efficace pour tester des variantes de programmes.
(Explications plus détaillées avec copies d'écran sous www.didel.com/kits/KiCalm1.pdf page 4)

Notions de base : bits, octets, constantes, variables, registres, …
Si le programme T877.asm ci-dessus est du charabia complet, le mieux est de lire la brochure Dauphin et de faire quelques exercices. http://www.epsitec.ch/Dauphin.zip Cette brochure vous apprend à programmer un processeur inventé, à la fois plus simple et plus riche que les PICs. Le notions de nombre, constantes, variables, etc. sont expliquées avec plus de détails dans le fichier www.didel.com/pic/Bases.pdf dont la lecture est recommandée maintenant ou plus tard, quand vous aurez fait quelques exercices.

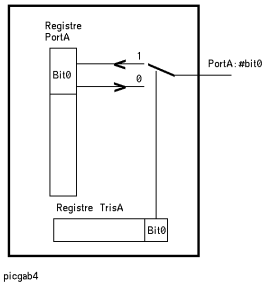
Architecture des PICs
Les PICs ont une architecture simple, avec une mémoire d'instructions 12 ou 14 bits, des variables 8 bits et des périphériques 8 bits dans le même *espace* que les variables. Cet espace des variables est coupé en 4 banques, ce qui complique la sélection : on travaille principalement en banque 0 en passant brièvement dans d'autres banques lorsque c'est nécessaire. Comprendre l'architecture interne n'est pas essentiel. La doc de Microchip est naturellement la référence. ww1.microchip.com/downloads/en/DeviceDoc/39582b.pdf. Le document www.didel.com/pic/Architecture.pdf présente une vue simplifiée avec mise en évidence des notions importantes.
Les périphériques ont plusieurs fonctions et il faut initialiser la bonne selon l'application. Cette richesse sera vue dans la partie 4, et les programmes PicTests www.didel.com/pic/PicTests.pdf évitent de se perdre dans la doc détaillée de Microchip pour les utilisations standard. Si vous voulez en savoir plus sur des bits qui définissent le comportement des périphériques, il faut chercher dans la documentation du fabricant ou utiliser une traduction en français, parfois un peu améliorée, par Bigonoff ou Oumnad. Les livres de Tavernier ont la même approche.

L'assembleur CALM
Les règles de l'assembleur et les pseudo-instructions apparaîtront petit à petit, pafois sans explications. Si une présentation systématique vous est nécessaire avant de voir des exemples de programme, lisez www.didel.com/pic/Calm.pdf

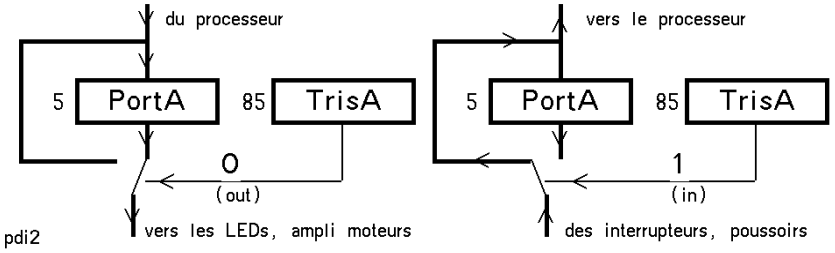
Entrées et sorties

L'intérêt des microcontrôleurs est qu'il ont des sorties sur lesquelles on peut brancher des diodes lumineuses, dans amplis pour commander des moteurs, un haut-parleur. Ils ont des entrées pour connecter des poussoirs et interrupteurs, pour lire des signaux analogiques, pour mesurer des durées d'impulsions. Ces fonctions sont programmables et c'est notre premier souci : configurer le microcontrôleur dans le mode compatible avec l'exercice.

Registre de direction
Pour décider si une broche du processeur est une entrée ou sortie, il faut que le contrôleur ait des aiguillages commandés par un registre de direction appelé Tris. Si le bit Tris est à 1 (input) la broche du circuit est connectée au bus et amène l'information dans le registre W, si l'instructionMove Port,W est exécutée. Si le bit Tris vaut 0 (output), c'est l'information qui vient du bus qui est mémorisée dans le registre et activée en sortie.



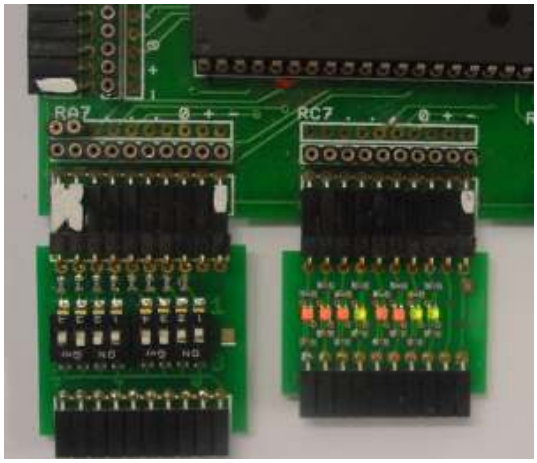
picgab4



pdi2

L'adresse du registre Tris est la même que pour le port correspondant, mais dans la banque 1, sélectionnée en activant le bit RP0 du registre Status.

Un premier exercice : copier le portA sur le PortC

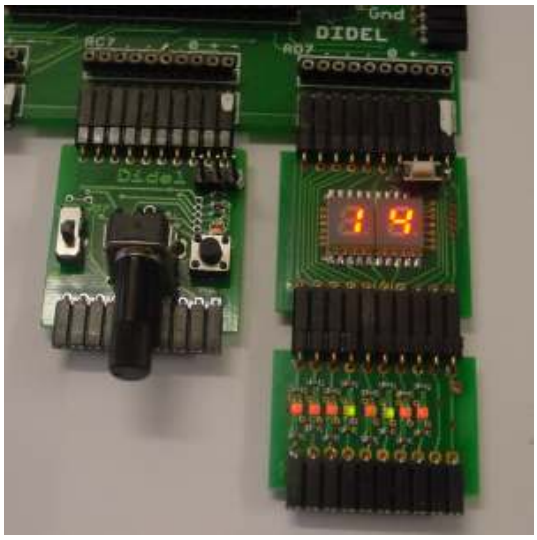


Le 16F877 a cinq ports, A et E sont incomplets. Le programme le plus simple que l'on puisse écrire copie l'état des lignes d'entrées du PortA sur les sorties du PortC. Le microdules Sw8 impose ses 0 et 1 sur le portA et transfère l'informations sur les diodes du module Lb8 : rouge pour l'état 0 et vert pour l'état 1.
Le cœur du programme **Ex877-101.asm** est

```
Loop:
      Move   PortA,W
      Move   W,PortC
      Jump   Loop
```

Chargez le programme et observez.RA6 RA7 ne sont pas disponibles sur le 877.

Peut-on imaginer plus simple ? Move est en fait une copie, on ne déplace pas l'information, on en fait une copie. On ne peut pas copier directement du portA vers le portB, il faut passer par le registre W, qui joue un rôle de plaque tournante.
Loop est un nom inventé (une étiquette pour dire où est la prochaine instruction) ; on aurait pu mettre Boucle, Toto, mais pas Jump ou PortA qui sont des noms réservés. Il faut mettre le : sans espace devant.



Pour copier C dans D, il faut adapter l'initialisation, et comprendre que le concepteur du Pic en 1975 n'a pas vu assez loin : il a réservé trois instructions Tris A B C et c'est tout. Pout agir sur TrisD et E, il faut passer par la banque 1, et utiliser les mêmes numéros d'adresse que ceuxdes ports D et E.
On pourrait écrire pour l'initialisation

```
 Set    Status:#RP0
Move    #16'06,W  ; Désactive
Move    W,AdCon1  ; le A/D
 Clr    Status:#RP0
Move    #-1,W     ; PortC en entrée
Move    W,TrisC
 Set    Status:#RP0
Move    #0,W      ; PortD en sortie
Move    W,PortD         ; TrisD
 Clr    Status:#RP0
```

C'est plus simple comme c'est fait dans **Ex877-102.asm**

Profitez de jouer avec le module PoToD8, doc sous www.didel.com/08micro/PoToD8.pdf , pour mieux comprendre les nombres positifs et négatifs (relire www.didel.com/pic/Bases.pdf).

Note Il peut y avoir problème pour lire sur le PortB. car le programmateur utilise RB6 et RB7 pour écrire dans le 16F877. Un interrupteur fermé peut perturber, malgré la résistance série.

Valeur fixe dans le portD

Si on regarde les affichages de la photo précédente, on lit 14 (hexa) et `00010100 (binaire)`. Tous les processeurs travaillent en binaire (base 2) que nous noterons 2'00010100. On préfère souvent l'hexa plus compact 16'14.
Assignons une valeur fixe, une constante dans le portD, qui sera en général utilisé pour afficher des valeurs. Le cœur du programme **Ex877-103.asm** est

```
Loop: Move   #16'6A,W    ; ou Move #2'01101010
      Move   W,PortD
      Jump   Loop
```

Faire les exercices proposés au bas du programme (ce qui suit le .end n'est pas assemblé).
N'oubliez-pas que c'est très rapide : modifier la valeur, *assembler*, déplacer la souris, *write*, cliquer *assemblage correct* pendant que cela programme, observer les microdules. Ce n'est que lorsque le nom du .hex change qu'il faut aller chercher sur le disque.

Question : ce programme copie sans arrêt sur le portD ; une fois suffit. Oui, mais il faut toujours dire au processeur ce qu'il doit faire. On pourrait écrire

```
            Move  #16'6A,W
            Move  W,PortD
      Ici:  Jump  Ici   ; tourne-toi les pouces !
```

Encore une modif au programme, enlevons le # qui veut dire constante, valeur immédiate :

```
            Move  16'6A,W
            Move  W,PortD
      Ici:  Jump  Ici
```

Quelle est la valeur affichée ? Peut-on la prévoir ?
Non car c'est une variable qui résulte d'un état imprévisible des bascules à la mise sous tension (ce n'est pas toujours la même chose si vous coupez).

```
Essayez     Move  16'7,W
            Move  W,PortD
```

Tiens, c'est la copie le portC ! Effectivement les ports sont comme des variables. Le portC est à l'adresse 7 s'il est initialisé en entrée on le lit.

```
Essayez     Move  16'85,W
            Move  W,PortD
```

L'assembleur signale une erreur ! bien sûr, il n'y a que 7 bits pour adresser les variables, l'adresse doit être inférieure à 16'7F.

Osciller le port D

Activer-désactiver le portD, donc créer une impulsion positive sur tous ses bits, s'écrit de plusieurs façons. La première et la 3e solution sont identiques (les mêmes instructions sont générées).

`Move  #2'11111111,W`	`Move  #2'11111111,W`	`Move  #-1,W`
`Move  W,PortD`	`Move  W,PortD`	`Move  W,PortD`
`Move  #2'00000000,W`	`Clr   PortD`	`Move  #0,W`
`Move  W,PortD`		`Move  W,PortD`

Pour répéter ces instructions d'activation/désactivation, on crée une boucle en mettant une étiquette au début de la boucle et en ajoutant une instruction qui saute à cette étiquette, c'est à dire à l'adresse correspondante dans la mémoire du processeur, comme dans le programme de test.

```
      Loop:    Move    #2'11111111,W    ; 1 µs
               Move    W,PortD
               Move    #2'00000000,W
               Move    W,PortD
               Jump    Loop
```

Les PICs à 4 MHz exécutent une instruction par microseconde (mais 2 µs pour les sauts). Ce programme va activer le port D pour 2 micro secondes seulement, et désactiver pour 3 micro secondes. On ne verra rien sans utiliser un oscilloscope.

Boucles d'attente

Pour que le processeur modifie les sorties à une vitesse ''humaine'', il faut insérer des boucles d'attente. On initialise un compteur et on décompte jusqu'à la valeur zéro, que le processeur sait bien reconnaître.

```
            Move  #250,W
            Move  W,Cx1
      Att:  Dec   Cx1   ; 1 µs
            Skip,EQ     ; 1 µs
            Jump  Att   ; 2 µs
```

L'instruction Dec diminue de 1 la variable Cx1. Cx1 est un nom inventé, et il faut dire où cette variable se trouve en mémoire. C'est libre à partir de l'adresse 16'20, donc il faut déclarer avant le début du programme

```
      Cx1   = 16'20
```

L'instruction `Skip,EQ` se prononce Skip If Equal, saute si c'est égal, si l'opération précédente a un résultat égal à zéro. Si ce n'est pas égal à zéro, on passe à l'instruction suivante, décompte à nouveau. A noter que le Skip,EQ, comme le Jump, s'exécute en 2 microsecondes.
Ce bout de programme va faire 250 fois la boucle qui dure 4 microsecondes, donc la durée totale est de 1 milliseconde (plus les 2 microsecondes d'initialisation du décompteur).

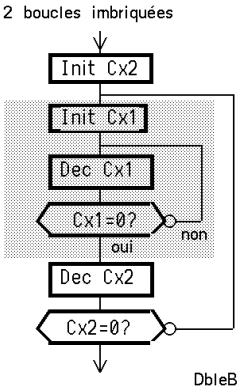
A noter les trois nouvelles instructions. Dec Cx1 décompte la variable mentionnée. L'instruction Skip,EQ permet de savoir si le compteur est à zéro (EQ equal). Si oui, le processeur saute l'instruction suivante. Si non, il exécute le saut qui lui fait attendre encore un tour.
Ce nouveau cœur de programme s'écrit (programme complet **Ex877-104.asm***)*

```
Loop:    Move     #2'11111111,W     ; on allume
         Move     W,PortD
; attente de 1ms
           Move  #250,W
           Move  W,Cx1
      At1: Dec   Cx1   ; 1 µs
           Skip,EQ     ; 1 µs
           Jump  At1   ; 2 µs

         Move     #2'00000000,W     ; on éteint
         Move     W,PortD
; attente de 1ms
           Move  #250,W
           Move  W,Cx1
      At2: Dec   Cx1   ; 1 µs
           Skip,EQ     ; 1 µs
           Jump  At2   ; 2 µs
         Jump     Loop
```

C'est encore trop rapide pour notre oeil. On ne peut pas augmenter la valeur 250 au delà de 255, car la variable Cx1 est un mot de 8 bits et les PICs n'ont pas d'instructions pour des compteurs ou décompteurs plus grands que 8 bits.
On fait alors une 2e boucle qui appelle la première et utilise une 2e variable. Vous voulez un dixième de secondes ? Il suffit de répéter 100 fois la boucle de 1ms .

```
; Boucle 100 millisecondes = 0.1s
      Move  #100,W
      Move  W,Cx2
At12: Move  #250,W
      Move  W,Cx1
At11: Dec   Cx1  ; Boucle 1ms
      Skip,EQ
      Jump  At11
      Dec   Cx2
      Skip,EQ
      Jump  At12
```



On pourrait naturellement compter et comparer quand on atteint la valeur 100 ou 250. C'est moins efficace avec le PIC.
Relisez plusieurs fois si ce n'est pas bien compris. Ces deux boucles imbriquées doivent être bien assimilées. On les teste dans la section suivante.

Programme complet
Il faut donc insérer nos boucles d'attente de 100ms après avoir écrit et après avoir modifié le motif sur le PortD. Le programme doit aussi initialiser le PortD en sortie, et déclarer l'adresse des variables Cx1 et Cx2.

```
\prog;Ex877-105|Clignote portD
.Proc   16F877
.Ref    16F877
.Loc    16'2007
.16     16'3F39
CX1     = 16'20 ; Début des var
CX2     = 16'21

Motif1  = 16'55 ; 2'01010101
Motif2  = 16'AA ; 2'10101010

Deb:    Clr     PortA
        Set     Status:#RP0
        Move    #16'06,W
        Move    W,AdCon1
        Move    #0,W    ; PortD
        Move    W,PortD ; sortie
        Clr     Status:#RP0

Loop:
        Move    #Motif1,W
        Move    W,PortC
; attente 200ms
        Move    #200,W
        Move    W,Cx2
At12:
        Move    #250,W
        Move    W,Cx1
At11:   Dec     Cx1
        Skip,EQ
        Jump    At11
        Dec     Cx2
        Skip,EQ
        Jump    At12

        Move    #Motif2,W
        Move    W,PortC
; attente 200ms
        Move    #200,W
        Move    W,Cx2
Att22:
        Move    #250,W
        Move    W,Cx1
At21:   Dec     Cx1
        Skip,EQ
        Jump    At21
        Dec     Cx2
        Skip,EQ
        Jump    Att22

        Jump    Loop
.End
```

Optimisation et simplifications
L'instruction DecSkip,EQ remplace les deux instructions de la boucle d'attente.

Si on n'a pas besoin d'une durée précise, on peut ne pas recharger les compteurs. Après être arrivés à zéro, ils recommencent un tour de 256 pas. La façon la plus simple de faire une attente de 256x256x3 µs =~ 0.2 secondes est d'écrire

```
A$: DecSkip,EQ  Cx1
    Jump  A$          ; A$ est une étiquette dite locale
    DecSkip,EQ  Cx2
    Jump  A$
```

Le programme **T877.asm** utilise cette attente simplifiée et clignote tous les ports en inversant le motif avec un ou-exclusif (partie 2). Le programme est nettement plus court.
T877.hex est un programme de test que l'on recharge toutes les fois que l'on veut vérifier que le processeur fonctionne et que toutes les sorties sont connectées. S'il ne tourne pas, c'est que l'oscillateur interne n'est pas initialisé.

Routines
Ce n'est pas très élégant, ni très efficace, de devoir écrire deux fois la même boucle d'attente, comme dans le programme **Ex877-105.asm**. On écrit une seule fois le module, appelé routine, et on peut l'appeler autant de fois que l'on veut. L'instruction Call fait l'appel et à la fin de la routine, l'instruction Ret (return) retourne à l'instruction qui suit le Call.
Ecrivons une routine **DelWx1ms** avec un paramètre en entrée qui est le contenu de W. DelWx1ms signifie ''Delai de contenu de W fois 1 milliseconde''. C'est la grande difficulté de tous les programmes informatiques : définir des noms qui expriment au mieux la fonction, et que l'on utilise ensuite sans hésitation.

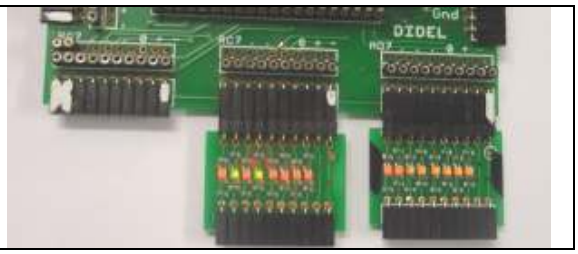
```
\rout:DelWx1ms|Attente de (W) fois 1ms
\in:W nombre de ms
\mod:W Cx1 Cx2
DelWx1ms:  Move    W,Cx2
A$:   Move  #250,W
      Move  W,CX1
  B$: Nop
      DecSkip,EQ  CX1   ; boucle 1ms
      Jump  B$
      DecSkip,EQ Cx2    ; boucle (W) fois 1ms
      Jump  A$
      Ret
```

Cette routine est le premier élément d'une bibiothèque de routines que chaque programmeur se constitue. Les 3 premières lignes contiennent toute l'information nécessaire pour l'utiliser.

Compteur
Compter ou décompter en 8 bits est trivial. Comment compter plus loin que 256 ?
Comptons sur le portD, toutes les quelques ms. Toute les fois que le compteur D passe par zéro, le bit Z du registre Status s'active (le Carry dont on parlera plus tard n'est pas activé). On teste ce bit pour décider si on incrémente le portC.
Les microdules visualisent ce compteur, charger et exécuter le programme **Ex877-106.asm.**

```
Loop:  (attente)
       Inc    PortD
       Skip,EQ
       Jump   Loop
       Inc    PortC
       Jump   Loop
durée 5 ou 4 us
```

```
Loop:  (attente)
       IncSkip,EQ PortD
       Dec    PortC
       Inc    PortC
       Jump   Loop
durée 4 us
```



On voit que la 2e solution est astucieuse. Cela ne semble pas rationnel de décompter et compter la même variable à chaque cycle, mais le résultat est que le programme est plus court, plus rapide et sa durée est toujours la même.
La décrémentation, à faire en exercice, a aussi une jolie solution.

Saturation
Comme un compteur de voiture, un compteur binaire recommence à zéro après la valeur maximum. Si on veut bloquer un compteur 8 bits à 16'FF, on écrit

```
Inc   Cnt
Skip,NE     ; saute si différent de zéro
Dec   Cnt   ; on revient en arrière à 16'FF si =0
```

Pour bloquer à zéro un décompteur, on refuse de décompter s'il est à zéro. L'instruction Test ne fait rien d'autre que d'activer Z si le contenu de la variable est zéro.

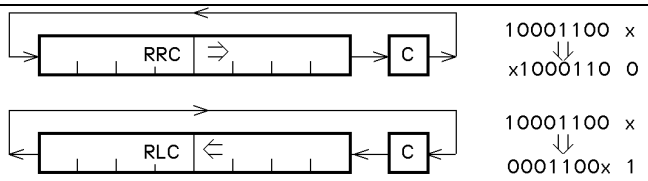
```
      Test  Cnt
      Skip,EQ     ; saute si égal à zéro
      Dec   Cnt
```

Exercice : bloquer un compteur ou décompteur 16 bits.

Décalages

Il n'y a que deux instructions de décalage dans les PICs, qui sont en fait des rotations à travers le Carry. Le Carry est un bit dans un registre de fanions, d'état (status) que nous retrouverons à propos d'arithmétique. Pour le moment, c'est un bit dans le processeur, que l'on peut mettre à un ou à zéro avec les instructions SetC et ClrC.

RRC Var
Rotate Right through Carry
RLC Var
Rotate Left through Carry



Pour retrouver l'information, il faut donc 9 décalages.
On peut tester facilement avec le portD et les instructions du programme **Ex877-108.asm.** On remarque que pour avoir une délai d'observation suffisant, il a fallu définir une attente multiple de 0.1 seconde, donc un délai maximum de 25.5 secondes

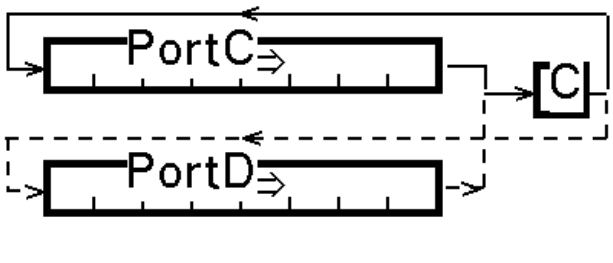
```
Loop:  Move      #10,W
       Call      DelWx01s ; 10x0.1s = 1 seconde
       RRC       PortD
       Jump      Loop
```

Regardons les programmes faits. Il contiennent beaucoup d'instructions qui reviennent chaque fois. Créons un module qui ne change plus, et est importé depuis le disque à chaque assemblage. Ce module contient en plus des deux routines de délai une routine d'initialisation qui met le portA en entrée, et les ports B, C, D, E en sortie. Ce fichier **Ex877Init1.asi** est appelé par le programme **Ex877-109.asm** qui fait la même chose que le **Ex877-108.asm** précédent. Les fichiers avec l'extension **.asi** ne peuvent pas être assemblés, il sont toujours appelés (insérés) dans des fichies exécutables avec l'extension **.asm**

Jouons encore avec les décalages

Pour se faire la main, commençons à décaler sur 16 bits, en observant les LEDs sur les ports C et D. Initialisons ces deus registres (ports) et décalons à droite en vidant, c'est-à-dire en injectant des zéros, en mettant le carry à zéro avant chaque décalage.

```
       Move   #2'10110111,W
       Move   W,PortC
       Move   #2'10101101,W
       Move   W,PortD
Loop:  Move   #10,W
       Call   DelWx01s
       ClrC
       RRC    PortC
       RRC    PortD
       Jump   Loop
```



On voit dans ce programme **Ex877-110.asm** que le Carry transporte le bit ''poussé dehors'' du PortC vers le portD.

On veut maintenant faire l'équivalent de Move PortC,PortD par décalage. Il faut 8 décalages, et on va utiliser un décompteur, comme dans les boucles d'attente. Voir **Ex877-111.asm**

On peut tester le Carry avec l'instruction Skip,CC qui saute l'instruction suivante si le Carry est Clear (à zéro). On a aussi l'instruction Skip,CS.
Comme exercice, comptons les bits à un. L'algorithme est simple : on décale. et toutes les fois que le carry est à un, on compte. En parallèle, on affiche le résultat en binaire sur le portB.

```
; initialisation
       Move   #2'10110111,W
       Move   W,PortC
       Move   #2'10101101,W
       Move   W,PortD
       Clr    PortB  ; servira de compteur
```

```
Loop:  Move   #10,W
       Call   DelWx01s
       ClrC
       RRC    PortC
       RRC    PortD
       Skip,CC
```

	Inc PortB Jump Loop

Ce programme **Ex877-112.asm** n'arrête pas de décaler, mais comme le registre est vide et que cela ne compte plus, on ne s'en rend pas compte. Dans un programme réel, il faudrait ajouter un décompteur pour s'arrêter après 16 cycles.

Ping-pong – à vous d'inventer
On veut que la led Verte s'allume à guche, se décale seule et rebondisse à droite. Comment faire ? On part avec les ports C et D à zéro. On injecte la Led verte avec le Carry. On décale en testant si le carry. Quand il passe à un on continue dans une 2e partie de programme. Envoyez votre programme pour correction à info@didel.com.

Résumé du chapitre
On a montré la différence entre constante (valeur immédiate) et variable.
Les variables sont numérotées de 0 à 16'7F ; de 0 à 16'20, on a des ports avec des noms réservés (PortA, etc). On a utilisé les ports B, C, D comme variables avec des boucles d'attente pour avoir le temps d'observer. On déclare l'emplacement des variables en donnant leur adresse. **Cx1 = 16'20**. On verra plus tard une meilleure façon pour ces déclarations.

Les variables ont 8 bits de large, mais des ports peuvent être incomplets. On initialise une variable en passant la valeur par W d'abord. L'instruction Move #Valeur,Variable n'existe pas, elle nécessiterait plus des 14 bits des instructions.

La plupart des instructions agissent sur les fanions (flags) **Z** et **C**, qui sont deux bits du registre Status. Dans ce registre on trouve aussi un bit **RP0** qui sélectionne un 2e groupe de variables, qui contient en particulier les registres de direction. Les regitres de direction **TrisA, TrisB, TrisC** peuvent être modifiés directement avec les instructions **Move W,TrisA**, etc. C'est plus compliqué pour les registres D et E.

Les instructions **Inc Var** et **Dec Var** activent le fanion Z quand le résultat est nul.
Les instructions **IncSkip,EQ Var** et **DecSkip.EQ Var** économisent une instruction. Elles ne modifient pas Z ! IncSkip,NE et DecSkip,EQ n'existent pas.
Les instructions **RRC Var** et **RLC Var** font tourner les 8 bits à travers le carry.

Enfin, on a compris l'intérêt des routines dont on peut oublier le contenu, il suffit de savoir ce qu'elles font et leur passer un ou plusieurs paramètres. Dans les cas simples, ce paramètre est un mot de 8 bits dans W. Pour varier un délai, 8 bits est insuffisant et on a deux routines différentes pour les délais courts **DelWx1ms** et les délais longs **DelWx01s.**
Ces routines ''de librairie'' sont dans un fichier avec l'extension **.asi**. Le fichier Ex**877Init1.asi** contient aussi la routine Init qui initialise le Ports A en entrée et les ports B, C, D, E en sortie. Chaque application a des entrées-sorties spécifiques, donc une routine **Init:** spécifique.

A quoi servent ces .Ascii ''T e x t e '' à la fin des programmes ? A rien, mais ils laissent une trace dans le fichier hex, que l'on peut lire dans le Pickit2.

Vos autres questions à info@didel.com

La 2e section va continuer le petit jeu d'observation de l'effet de nouvelles instructions sur des variables et ports et que l'on peut visualiser.
La section **i** inventoriera toutes les instructions et est une annexe sans exercices que l'on lit une première fois et consulte quand on n'est plus sûr de sa mémoire.
La 3e section présente les périphériques internes du 877, indispensables pour beaucoup d'applications.
La 4e section montre comment on gère les périphériques externes les plus courants avec des exemples un peu plus corsés. Elle donne quelques conseils pour le développement des programmes.

jdn 100928