LA PROGRAMMATION DES PIC®

PAR BIGONOFF

PREMIERE PARTIE – Révision 24

DEMARRER LES PIC® AVEC LE PIC16F84

1. Introduction

Et voilà, nous sommes partis ensemble pour cette grande aventure qu'est la programmation des PIC®. Je vais tenter de rester le plus concret possible, mais, cependant, une certaine part de théorie est indispensable pour arriver au but recherché.

Je vais donc commencer ce petit cours par un rappel sur les systèmes de numération. Ca y est, j'en vois qui râlent déjà. Mais je suis sûr que vous comprendrez qu'il est impossible de programmer sérieusement un microcontrôleur sans savoir ce qu'est un bit, ou comment convertir les notations décimales en hexadécimales.

Rassurez-vous, je vais faire bref, et nous pourrons très rapidement aborder le sujet qui nous intéresse tant. Si vous êtes déjà un « pro », vous pouvez sauter le premier chapitre et passer directement au suivant.

N'hésitez jamais à me faire part de vos remarques, ni à me signaler les erreurs qui m'auraient échappées (www.abcelectronique.com/bigonoff ou www.bigonoff.org). Faites du copier/coller, répercutez les infos que vous trouverez ici, traduisez le document dans une autre langue ou un autre format. Simplement, dans ce cas, veuillez respecter les désirs de l'auteur en fin d'ouvrage et faites moi parvenir un exemplaire de votre travail. Ceci pour permettre d'en faire profiter le plus grand nombre (bigocours@hotmail.com).

J'attire votre attention sur le fait que ce cours, pour être efficace, doit être lu tout en réalisant les exercices que je vous propose. Les solutions des exercices sont disponibles sous forme de fichiers exemples fournis en annexe de ce cours.

Tout ce qu'il vous faudra, c'est un 16F84, un quartz de 4MHz, une petite platine d'essais, une LED, un bouton poussoir et le logiciel MPLAB®, mis gracieusement à votre disposition par la société Microchip® à l'adresse http://www.Microchip.com. A cette même adresse, vous pourrez vous procurer le datasheet du 16F84.

J'ai passé de nombreuses journées à réaliser ces exercices. Je les ai personnellement testés sur maquette un par un. Alors je vous demanderai de tenter de faire vous-même ces petits programmes avant de me poser par email des questions dont vous auriez eu les réponses si vous aviez réalisé cet effort. Croyez-moi sur parole, ce n'est qu'au moment de mettre en pratique qu'on s'aperçoit qu'on n'avait finalement pas bien compris quelque chose qui paraissait pourtant évident. Je réponds toujours au courrier reçu, mais il faut dire que c'est parfois légèrement énervant de recevoir une question de quelqu'un qui affirme avoir assimilé la totalité de cet ouvrage en une heure. Ca m'est arrivé !

J'ai utilisé personnellement la version 6.30 de MPLAB® à partir de la révision 13 du cours. Les précédentes révisions utilisaient la version 5.20. La version 6.60 est disponible section « archives » sur le site de Microchip®.

2. Les systèmes de numération

2.1 Le système décimal

Nous sommes habitués, depuis notre enfance à utiliser le système numérique décimal, à tel point que nous ne voyons même plus la manière donc ce système fonctionne, tant c'est devenu un automatisme.

Décimal, pourquoi ? Parce qu'il utilise une numération à 10 chiffres. Nous dirons que c'est un système en BASE 10. Pour la petite histoire, on a utilisé un système base 10 car nos ancêtres ont commencé à compter sur leurs 10 doigts, pas besoin d'aller chercher plus loin.

Mais la position des chiffres a également une grande importance. Les chiffres les moins significatifs se situent à droite du nombre, et leur importance augmente au fur et à mesure du déplacement vers la gauche. En effet, dans le nombre 502, le 5 a une plus grande importance que le 2. En réalité, chaque chiffre, que l'on peut appeler DIGIT, a une valeur qui dépend de son **RANG** . Quel est le multiplicateur à appliquer à un chiffre en fonction de sa position (rang) ? Il s'agit tout simplement de l'élévation de la BASE utilisée à la puissance de son RANG .

Cela a l'air complexe à écrire, mais est très simple à comprendre. Lorsque vous avez compris ceci, vous comprenez automatiquement n'importe quel système de numération.

Reprenons, par exemple notre nombre 502. Que signifie le 2 ? Et bien, tout simplement que sa valeur est égale à 2 multiplié par la base (10) élevée à la puissance du rang du chiffre, c'est à dire 0.

Remarquez ici une chose très importante : le comptage du rang s'effectue toujours de droite à gauche et en commençant par 0. Pour notre nombre 502, sa valeur est donc en réalité : $502 = 2*10° + 0*10^1 + 5*10^2$. Notez que le symbole * est utilisé pour indiquer « multiplié ». Et rappelez-vous que $10° = (10/10) = 1$, que $10^1 = 10$, et que $10^2 = 10*10 = 100$ etc.

2.2 Le système binaire

Vous avez compris ce qui précède ? Alors la suite va vous paraître simple. Cela ne pose aucun problème pour vous de compter sur vos 10 doigts, mais pour les ordinateurs, cela n'est pas si simple. Ils ne savent faire la distinction qu'entre 2 niveaux (présence ou absence de tension). Le système de numération décimal est donc inadapté.

On comprendra immédiatement que le seul système adapté est donc un système en base 2, appelé système binaire. Ce système ne comporte donc que 2 chiffres, à savoir 0 et 1. Comme, de plus, les premiers ordinateurs (et les PIC®) travaillent avec des nombres de 8 chiffres binaires, on a donc appelé ces nombres des octets (ou bytes en anglais). Le chiffre 0 ou 1 est appelé un BIT(unité binaire, ou BInary uniT) .

Pour nous y retrouver dans la suite de ce petit ouvrage, on adoptera les conventions suivantes : tout nombre décimal est écrit tel quel, ou en utilisant la notation D'xxx' ; tout nombre binaire est écrit suivant la forme B'xxxxxxxx' dans lesquels les 'x' valent ?… 0 ou 1 effectivement, vous avez bien suivi.

Analysons maintenant un nombre binaire, soit l'octet : B'10010101'. Quelle est donc sa valeur en décimal ?

Et bien, c'est très simple, on applique le même algorithme que pour le décimal. Partons de la droite vers la gauche, on trouve donc :

B'10010101' = $1*2^{\circ} + 0*2^{1} + 1*2^{2} + 0*2^{3} + 1*2^{4} + 0*2^{5} + 0*2^{6} + 1*2^{7}$

Comme, évidemment 0 multiplié par quelque chose = 0 et que 1 multiplié par un chiffre = le chiffre en question, on peut ramener le calcul précédent à :

B '10010101' = 1+4+16+128 = 149

Vous voyez donc qu'il est très facile de convertir n'importe quel chiffre de binaire en décimal. Et l'inverse me direz-vous ? Et bien, c'est également très simple. Il faut juste connaître votre table des exposants de 2. Cela s'apprend très vite lorsqu'on s'en sert.

On procède simplement par exemple de la manière suivante (il y en a d'autres) :

Quel est le plus grand exposant de 2 contenu dans 149 ? Réponse 7 (2^{7} = 128)
On sait donc que le bit 7 vaudra 1. Une fois fait, il reste 149-128 = 21
Le bit 6 représente 64, c'est plus grand que 21, donc b6 = 0
Le bit 5 représente 32, c'est plus grand que 21, donc b5 = 0
Le bit 4 représente 16, donc ça passe, b4 = 1, il reste 21-16 = 5
Le bit 3 représente 8, c'est plus grand que 5, donc b3 = 0
Le bit 2 représente 4, donc b2 = 1, reste 5-4 = 1
Le bit1 représente 2, c'est plus grand que 1, donc b1 = 0
Le bit 0 représente 1, c'est ce qu'il reste, donc b0=1, reste 0

Le nombre binaire obtenu est donc B'10010101', qui est bien notre octet de départ. Notez que si on avait trouvé un nombre de moins de 8 chiffres, on aurait complété avec des 0 placés à gauche du nombre. En effet, B'00011111' = B '11111', de même que 0502 = 502.

Pensez à toujours compléter les octets de façon à obtenir 8 bits, car c'est imposé par la plupart des assembleurs (nous verrons ce que c'est dans la suite de ces leçons).

Notez que la plus grande valeur pouvant être représentée par un octet est donc : B'11111111'. Si vous faites la conversion (ou en utilisant la calculette de Windows en mode scientifique), vous obtiendrez 255. Tout nombre supérieur à 255 nécessite donc plus d'un octet pour être représenté.

2.3 Le système hexadécimal

La représentation de nombres binaires n'est pas évidente à gérer, et écrire une succession de 1 et de 0 représente une grande source d'erreurs. Il fallait donc trouver une solution plus pratique pour représenter les nombres binaires. On a donc décidé de couper chaque octet en 2 (QUARTET) et de représenter chaque partie par un chiffre.

Comme un quartet peut varier de b'0000' à b'1111', on constate que l'on obtient une valeur comprise entre 0 et 15. Cela fait 16 combinaisons. Les 10 chiffres du système décimal ne suffisaient donc pas pour coder ces valeurs.

Plutôt que d'inventer 6 nouveaux symboles, il a été décidé d'utiliser les 6 premières lettres de l'alphabet comme CHIFFRES. Ce système de numération en base 16 a donc été logiquement appelé système hexadécimal.

Notez que ce système est simplement une représentation plus efficace des nombres binaires, et donc que la conversion de l'un à l'autre est instantanée. Dans la suite de ces leçons, nous noterons un nombre hexadécimal en le faisant précéder de 0x. Voyons si vous avez bien compris :

Tableau de conversion des différents quartets (un demi-octet)

Binaire	Hexadécimal	Décimal
B'0000'	0x0	0
B'0001'	0x1	1
B'0010'	0x2	2
B'0011'	0x3	3
B'0100'	0x4	4
B'0101'	0x5	5
B'0110'	0x6	6
B'0111'	0x7	7
B'1000'	0x8	8
B'1001'	0x9	9
B'1010'	0xA	10
B'1011'	0xB	11
B'1100'	0xC	12
B'1101'	0xD	13
B'1110'	0xE	14
B'1111'	0xF	15

Pour représenter un octet il faut donc 2 digits hexadécimaux. Par exemple, notre nombre B'10010101' est représenté en hexadécimal par 0x95. Si vous faites la conversion de l'hexadécimal vers le décimal, vous utilisez le même principe que précédemment, et vous obtenez $0x95 = 9*16^1 + 5*16° = 149$, ce qui est heureux.

Pour preuve, quel est le plus grand nombre hexadécimal de 2 digits pouvant être représenté ? Réponse : 0xFF, soit 15*16 + 15 = 255.

Si vous avez bien tout compris, vous êtes maintenant capable de convertir n'importe quel nombre de n'importe quelle base vers n'importe quelle autre. Vous trouverez également dans certaines revues, des allusions au système octal, qui est un système en base 8 qui a été largement utilisé.

2.4 Les opérations

Après avoir converti les nombres dans différents formats, vous allez voir qu'il est également très simple de réaliser des opérations sur ces nombres dans n'importe quel format. Il suffit pour cela d'effectuer les mêmes procédures qu'en décimal.

Petit exemple : Que vaut B'1011' + B '1110' ? Et bien, on procède exactement de la même façon que pour une opération en décimal.

```
    B'1011'
+   B'1110'
-------------
```

- On additionne les chiffres de droite, et on obtient 1+0 = 1
- On écrit 1
- On additionne 1 + 1 , et on obtient 10 (2 n'existe pas en binaire). On écrit 0 et on reporte 1
- On additionne 0+1+le report et on obtient 10. On écrit 0 et on reporte 1
- On additionne 1+1+le report et on obtient 11. On écrit 1 et on reporte 1
- Reste le report que l'on écrit, soit 1.

La réponse est donc B'11001', soit 25.

Les 2 nombres de départ étant B'1011', soit 11, et B'1110', soit 14. Vous procéderez de la même manière pour les nombres hexadécimaux, en sachant que 0xF + 0x1 = 0x10, soit 15+1 = 16.

2.5 Les nombres signés

Dans certaines applications, il est nécessaire de pouvoir utiliser des nombres négatifs. Comme les processeurs ne comprennent pas le signe « - » , et comme il fallait limiter la taille des mots à 8 bits, la seule méthode trouvée a été d'introduire le signe dans le nombre.

On a donc choisi (pas au hasard) le bit 7 pour représenter le signe. Dans les nombres signés, un bit 7 à '1' signifie nombre négatif. Si on s'était contenté de cela, on aurait perdu une valeur possible. En effet, B'10000000' (-0) serait alors égal à B'00000000' (0). De plus, pour des raisons de facilité de calcul, il a été décidé d'utiliser une notation légèrement différente.

Pour rendre un nombre négatif, il faut procéder en 2 étapes.

- On inverse la totalité du nombre.
- On ajoute 1

On obtient alors ce qu'on appelle le COMPLEMENT A DEUX du nombre.

Exemple : soit le nombre 5 : B'00000101' Comment écrire –5 ?

- on inverse tous les bits (complément à 1) B'11111010'
- on ajoute 1 (complément à 2) –5 = B'11111011'

Pour faire la conversion inverse, on procède de façon identique.

On inverse tous les bits B’00000100’
On ajoute 1 B‘00000101’

Et on retrouve notre 5 de départ, ce qui est logique, vu que –(-5) = 5.

Dans le cas des nombres signés, on obtient donc les nouvelles limites suivantes :

- La plus grande valeur est B’01111111’, soit +127
- La plus petite valeur devient B’10000000’, soit –128.

Remarquez que les opérations continuent de fonctionner. Prenons –3 + 5

```
    B  ’11111101’ (-3)
+   B  ’00000101’ (5)
----------------------------
=   B’100000010’ (2)
```

Et là, me direz vous, ça ne fait pas 2 ? Et bien si, regardez bien, il y a 9 bits , or le processeur n’en gère que 8. Le 9ème est donc tombé dans un bit spécial que nous verrons plus tard. Dans le registre du processeur, il reste donc les 8 bits de droite, soit 2, qui est bien égal à (–3)+ 5.

Maintenant, si vous avez bien suivi, vous êtes en train de vous poser la question suivante : Quand je vois B’11111101’, est-ce que c’est –3 ou est-ce que c’est 253 ? Et bien vous ne pouvez pas le savoir sans connaître le contexte.

Sachez que les nombres signifient uniquement ce que le concepteur du programme a décidé qu’ils représentent. S’il travaille avec des nombres signés ou non, ou si cet octet représente tout autre chose. La seule chose qui importe c’est de respecter les conventions que vous vous êtes fixées lors de la création de cet octet. C’est donc à vous de décider ce dont vous avez besoin pour tel type de données.

2.6 Les opérations booléennes.

Qu’est-ce que c’est que ça, me direz-vous ? Et bien, pour faire simple, disons que ce sont des opérations qui s’effectuent bit par bit sur un octet donné. Plutôt qu’une grosse théorie sur l’algèbre de Boole (j‘en vois qui respirent), je vais donner dans le concret en présentant les opérations indispensables à connaître dans la programmation des PIC® et autres microcontrôleurs.

2.6.1 Le complément

Que vous trouverez également sous les formes « inversion » ou « NOT » ou encore complément à 1. Elle est souvent notée « ! » Son fonctionnement tout simple consiste à inverser tous les bits de l’octet (0 devient 1 et 1 devient 0).

Exemple : NOT B’10001111’ donne B’01110000’.

Vous voyez ici que pour les opérations booléennes, il est plus facile de travailler en binaire. Traduisez l'exemple ci-dessus successivement en hexadécimal (on dira maintenant « hexa »), puis en décimal, et essayez de complémenter directement. Bonjour les neurones.

A quoi sert cette opération ? Par exemple à lire une valeur dont les niveaux actifs ont été inversés, à réaliser des nombres négatifs, ou autres que nous verrons par la suite.

2.6.2 La fonction « ET » ou « AND »

Appelée également multiplication bit à bit, ou « AND », et souvent notée « & »

Elle consiste à appliquer un mot sur un autre mot et à multiplier chaque bit par le bit de même rang. Pour faire une opération « ET », il faut donc toujours 2 octets.

Les différentes possibilités sont données ci-dessous (le tableau se lit horizontalement).

Première ligne : 0 AND 0 = 0. Ce type de tableau s'appelle « table de vérité »

Bit1	Bit2	AND
0	0	0
0	1	0
1	0	0
1	1	1

On voit donc que la seule possibilité pour obtenir un « 1 » est que le Bit1 ET le Bit2 soient à « 1 ». Ceci correspond à une multiplication. 1*1 = 1, 0*1 = 0, 1*0 = 0.

Exemple :

Soit B'11001100' AND B '11110000' donne B'11000000'

A quoi sert cette instruction ? Et bien, elle est utilisée pour MASQUER des bits qui ne nous intéressent pas.

Prenez l'exemple ci-dessus : Le 2ème octet contient 4 bits à 1 et 4 bits à 0. Regardez le résultat obtenu : Les 4 premiers bits de l'octet 1 sont conservés (1100), à l'emplacement des 4 autres nous trouvons des 0.

On peut donc à l'aide de cette instruction positionner un ou plusieurs bits dans un mot à 0 sans connaître son contenu précédent.

2.6.3 La fonction « OU » ou « OR »

Encore appelée OR, souvent notée « | « elle permet, comme son nom l'indique, de positionner un bit à 1 si le Bit1 OU le Bit2 est à 1.

La table de vérité suivante explique le fonctionnement de cette fonction.

Bit1	Bit2	OR
0	0	0
0	1	1
1	0	1
1	1	1

Petit exemple B'10001000' OR B'11000000' donne B'11001000'

A quoi sert cette instruction ? Et bien, tout simplement elle permet de forcer n'importe quel bit d'un mot à 1 sans connaître son contenu précédent.

Vous voyez que dans l'exemple précédent, les 2 premiers bits ont été forcés au niveau 1, indépendamment de leur niveau précédent.

2.6.4 La fonction « OU EXCLUSIF » ou « Exclusif OR » ou « XOR »

Voici la dernière fonction que nous allons aborder dans cette mise à niveau. Elle est souvent appelée XOR (eXlusif OR). Elle se comporte comme la fonction OR, à un détail près.

Pour obtenir 1, il faut que le Bit1 soit à 1 OU que le Bit2 soit à 1 à l'EXCLUSION des deux bits ensemble. Si les 2 bits sont à 1, alors le résultat sera 0.

Voici donc la table de vérité.

Bit1	Bit2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Petit exemple : B'10001000 ' XOR B'11000000' donne B'01001000 '

A quoi sert cette instruction ? Et bien tout simplement à inverser un ou plusieurs bits dans un mot sans toucher aux autres. Dans l'exemple précédent, vous voyez qu'à l'emplacement des 2 bits à 1 du 2ème octet, les bits correspondants du 1er octet ont été inversés.

Voilà, ainsi se termine le premier chapitre consacré aux PIC®. Je sais qu'il était particulièrement rébarbatif, mais, si vous ne maîtrisez pas parfaitement ce qui vient d'être expliqué, vous ne pourrez pas réaliser correctement vos propres programmes.

Notes :…

3. Composition et fonctionnement des PIC®

Enfin quelque chose de plus intéressant. Nous allons maintenant nous pencher sur un PIC®, et en particulier sur le 16F84. Rassurez-vous, tout ce que nous verrons sur le 16F84 pourra être directement utilisé sur les 16F876, qui ne sont rien d'autre que des 16F84 améliorés. Chaque PIC® dispose des fonctionnalités des modèles inférieurs, augmentées de nouvelles fonctions. Au moment de passer à la révision 13 du cours, les modèles que vous rencontrerez seront probablement des 16F84A, mais ça ne posera aucun problème pour l'étude de ce cours.

Tout d'abord, vous devez télécharger le datasheet du 16F84, car c'est un document que nous allons utiliser dans le reste de ces petites leçons. Je vous conseille vivement de l'imprimer, car vous en aurez toujours besoin quand vous vous lancerez dans la réalisation de vos propres programmes.

Ces datasheets sont mes livres de chevet. J'ai trouvé plus judicieux de travailler par la pratique et de les commenter, plutôt que de traduire « bêtement » les datasheets en question.

3.1 Qu'est-ce qu'un PIC® ?

Un PIC® n'est rien d'autre qu'un microcontrôleur, c'est à dire une unité de traitement de l'information de type microprocesseur à laquelle on a ajouté des périphériques internes permettant de réaliser des montages sans nécessiter l'ajout de composants externes.

La dénomination PIC® est sous copyright de Microchip®, donc les autres fabricants ont été dans l'impossibilité d'utiliser ce terme pour leurs propres microcontrôleurs.

Les PIC® sont des composants dits RISC (Reduced Instructions Set Computer), ou encore composant à jeu d'instructions réduit. Pourquoi ? Et bien, sachez que plus on réduit le nombre d'instructions, plus facile et plus rapide en est le décodage, et plus vite le composant fonctionne.

Vous aurez deviné qu'on trouve sur le marché 2 familles opposées, les RISC et les CISC (Complex Instructions Set Computer). Sur les CISC, on dispose de moins de vitesse de traitement, mais les instructions sont plus complexes, plus puissantes, et donc plus nombreuses. Il s'agit donc d'un choix de stratégie.

Tous les PIC® Mid-Range ont un jeu de 35 instructions, stockent chaque instruction dans un seul mot de programme, et exécutent chaque instruction (sauf les sauts) en 1 cycle. On atteint donc des très grandes vitesses, et les instructions sont de plus très rapidement assimilées. L'exécution en un seul cycle est typique des composants RISC.

L'horloge fournie au PIC® est prédivisée par 4 au niveau de celle-ci. C'est cette base de temps qui donne la durée d'un cycle. Si on utilise par exemple un quartz de 4MHz , on obtient donc 1000000 de cycles/seconde, or, comme le PIC® exécute pratiquement 1 instruction par cycle, hormis les sauts, cela vous donne une puissance de l'ordre de 1MIPS (1 Million d'Instructions Par Seconde).

Pensez que les PIC® peuvent monter à 20MHz. C'est donc une vitesse de traitement plus qu'honorable.

3.2 Les différentes familles des PIC®

La famille des PIC® était subdivisée au moment d'écrire la révision 1 de cet ouvrage en 3 grandes familles : La famille Base-Line, qui utilise des mots d'instructions (nous verrons ce que c'est) de 12 bits, la famille Mid-Range, qui utilise des mots de 14 bits (et dont font partie les 16F84 et 16F876), et la famille High-End, qui utilise des mots de 16 bits. Par la suite, d'autres familles sont apparues, comme la Enhanced family, et les choses ne devraient faire qu'évoluler.

Nous nous limiterons dans cet ouvrage à la famille Mid-Range, sachant que si vous avez tout compris, vous passerez très facilement à une autre famille, et même à un autre microcontrôleur. Du reste, d'autres cours sont prévus pour vous permettre d'évoluer plus facilement.

Notez dès à présent que le datasheet du 16F84 n'est qu'une petite partie de la documentation complète. Pour obtenir la documentation complète, vous ajoutez encore plus de 600 pages en téléchargeant chez Microchip® les datasheets pour la gamme Mid-Range.

Cependant, la documentation de base suffit pour 99,9% des applications, et, de plus, les datasheets Mid-Range sont disponibles sous la forme d'un fichier par chapitre. J'ai pour ma part presque tout imprimé, mais je vous conseille plutôt d'aller les chercher le jour où vous en aurez besoin.

3.3 Identification d'un PIC®

Pour identifier un PIC®, vous utiliserez simplement son numéro.
Les 2 premiers chiffres indiquent la catégorie du PIC®, 16 indique un PIC® Mid-Range.

Vient ensuite parfois une lettre L : Celle-ci indique que le PIC® peut fonctionner avec une plage de tension beaucoup plus tolérante.

Ensuite, vous trouvez :

C indique que la mémoire programme est une EPROM ou plus rarement une EEPROM
CR pour indiquer une mémoire de type ROM
Ou F pour indiquer une mémoire de type FLASH.

Notez à ce niveau que seule une mémoire FLASH ou EEPROM est susceptible d'être effacée, donc n'espérez pas reprogrammer vos PIC® de type CR. Pour les versions C, voyez le datasheet. Le 16C84 peut être reprogrammé, il s'agit d'une mémoire eeprom. Le 12C508, par exemple, possède une mémoire programme EPROM, donc effaçable uniquement par exposition aux ultraviolets. Donc, l'effacement nécessite une fenêtre transparente sur le chip, qui est une version spéciale pour le développement, et non la version couramment rencontrée.

Un composant qu'on ne peut reprogrammer est appelé O.T.P. pour One Time Programming : composant à programmation unique. Puis vous constatez que les derniers chiffres identifient précisément le PIC®. (84)

Finalement vous verrez sur les boîtiers le suffixe « -XX » dans lequel XX représente la fréquence d'horloge maximale que le PIC® peut recevoir. Par exemple –04 pour un 4MHz. En fait, il semble bien que cette donnée soit purement commerciale et que tous les PIC® d'un même modèle acceptent de tourner à la vitesse maximale de ce modèle, vitesse donnée en début du datasheet. Cette inscription semble donc dans la réalité des faits parfaitement inutile. Cette information n'engage pas ma responsabilité, à vous de juger et de vous renseigner.

Donc, un 16F84-04 est un PIC® Mid-Range (16) donc la mémoire programme est de type FLASH (F) donc réinscriptible de type 84 et capable d'accepter une fréquence d'horloge de 4MHz en théorie (probablement : 10Mhz pour un 16F84 et 20Mhz pour un 16F84A).

Une dernière indication que vous trouverez est le type de boîtier. Nous utiliserons pour nos expérience le boîtier PDIP, qui est un boîtier DIL 18 broches, avec un écartement entre les rangées de 0.3'' (étroit). La version 4MHz sera amplement suffisante.

Notez dès à présent que les PIC® sont des composants STATIQUES, c'est à dire que la fréquence d'horloge peut être abaissée jusqu'à l'arrêt complet sans perte de données et sans dysfonctionnement.

Ceci par opposition aux composants DYNAMIQUES (comme les microprocesseurs de votre ordinateur), dont la fréquence d'horloge doit rester dans des limites précises. N'essayez donc pas de faire tourner votre PIII/500 à 166MHz, car c'est un composant dynamique.

Donc, si vous voulez passer commande pour le PIC® que nous allons utiliser dans le reste de cet ouvrage, demandez donc un PIC® 16F84-04 en boîtier PDIP .

3.4 Organisation du 16F84

La mémoire du 16F84 est divisée en 3 parties. Page 4 du datasheet, vous trouverez la table 1-1 qui donne un aperçu de la famille 16F8X. Les numéros de pages peuvent varier en fonction des mises à jour de Microchip®. Vous devrez peut-être chercher un peu, ou alors utilisez le datasheet que je fournis avec le cours.

Pour ceux qui veulent tout comprendre, la figure 3-1 de la page 8 montre l'organisation interne d'un 16F84.

3.4.1 La mémoire programme

La mémoire programme est constituée de 1K mots de 14 bits. C'est dans cette zone que vous allez écrire votre programme. Ceci explique pourquoi vos fichiers sur PC font 2Kbytes.

En effet, il faut 2 octets pour coder 14 bits. Ceci explique également pourquoi, lorsque vous lisez un PIC® vierge, vous allez lire des 0x3FFF. Cela donne en binaire B'11111111111111', soit 14 bits. J'expliquerais plus loin d'où proviennent ces fameux 14 bits.

Notez à ce point qu'une instruction est codée sur 1 mot. Donc, 1K donne 1 bon millier d'instructions possibles (c'est déjà pas si mal). Quand vous en serez à écrire des programmes de 1K, vous serez sans aucun doute autonome pour vos applications.

3.4.2 La mémoire Eeprom

La mémoire EEPROM (Electrical Erasable Programmable Read Only Memory), est constituée de 64 octets que vous pouvez lire et écrire depuis votre programme. Ces octets sont conservés après une coupure de courant et sont très utiles pour conserver des paramètres semi-permanents. Leur utilisation implique une procédure spéciale que nous verrons par la suite, car ce n'est pas de la RAM, mais bien une ROM de type spécial. Il est donc plus rapide de la lire que d'y écrire. Si vous programmez souvent des eeproms (2416) vous aurez constaté déjà ce phénomène.

3.4.3 La mémoire Ram

La mémoire RAM (Random Access Memory) est celle que nous allons sans cesse utiliser. Toutes les données qui y sont stockées sont perdues lors d'une coupure de courant.

La mémoire RAM est organisée en 2 banques pour le 16F84. La RAM est subdivisée de plus en deux parties. Dans chacune des banques nous allons trouver des « cases mémoires spéciales » appelées REGISTRES SPECIAUX et des cases mémoires « libres » dont vous pouvez vous servir à votre guise.

Pour le cas du 16F84, vous disposerez de 68 octets libres. L'organisation de la RAM est montrée dans le tableau 4-2 page 13. Vous voyez la séparation verticale en 2 banques, et tout en bas vous voyez deux banques de 68 octets de RAM.

Malheureusement, l'indication « mapped in bank 0) vous indique qu'accéder à ces 68 octets depuis la banque 0 ou la banque 1 donne en fait accès à la même case mémoire.

Vous voyez dans la partie supérieure le nom de tous les registres spéciaux utilisés dans le PIC®. Nous les verrons tous, rassurez-vous.

Chaque registre provoque un fonctionnement spécial du PIC® ou la mise en service d'une fonction particulière. Vous remarquerez enfin que certains registres sont identiques dans les 2 banques (FSR par exemple). Cela signifie qu'y accéder depuis la banque 0 ou 1 ne fait pas de différence.

Remarquez que la banque 0 utilise les adresses de 0x00 à 0x7F, la banque 1 allant de 0x80 à 0xFF. Les zones en grisé sont des emplacements non utilisés (et non utilisables). L'emplacement 0x00 est un emplacement auquel on ne peut pas accéder.

Pour la grande majorité des registres, chaque bit a une fonction spéciale. Page 14, tableau 4-1, vous trouverez les noms des bits utilisés dans ces registres.

4. Organisation des instructions

4.1 Généralités

Allez, courage, cela devient de plus en plus concret. On va faire un petit survol du jeu d'instructions des PIC®. On saute directement page 55 du datasheet, au chapitre 9. Et oui, comme cet ouvrage n'est pas un manuel de référence technique, mais un apprentissage, il faut voir les chapitres du datasheet dans le désordre.

Sur cette page, vous trouvez un petit encadré grisé qui fait allusion à deux anciennes instructions qui ne sont plus utilisées. Nous ne nous en servirons donc pas. Par contre, vous trouvez un tableau 9-1 qui indique comment les instructions sont codées dans le PIC®. Et la, vous voyez enfin à quoi correspondent nos 14 bits de mémoire programme.

4.2 Les types d'instructions

Vous constaterez donc qu'il existe 4 type d'instructions :

4.2.1 Les instructions « orientées octet »

Ce sont des instructions qui manipulent les données sous forme d'octets. Elles sont codées de la manière suivante :

- 6 bits pour l'instruction : logique, car comme il y a 35 instructions, il faut 6 bits pour pouvoir les coder toutes

- 1 bit de destination(d) pour indiquer si le résultat obtenu doit être conservé dans le registre de travail de l'unité de calcul (W pour Work) ou sauvé dans l'opérande (F pour File).

- Reste 7 bits pour encoder l'opérande (File)

Aie, premier problème, 7 bits ne donnent pas accès à la mémoire RAM totale, donc voici ici l'explication de la division de la RAM en deux banques.

En effet, il faudra bien trouver une solution pour remplacer le bit manquant. Vous avez dit « un bit d'un des registres ? » BRAVO, je vois que vous avez tout compris. Il s'agit en réalité du bit RP0 du registre STATUS.

Ah, vous avez remarqué qu'il y a un RP1 ? Et oui, le 16F876 a 4 banques, ce bit sera utilisé pour certains autres PIC® que nous verrons dans la seconde partie. Vous veillerez à laisser RP1 à 0 pour le 16F84, afin de pouvoir « porter » votre programme sans problème vers un PIC® supérieur.

4.2.2 Les instructions « orientées bits »

Ce sont des instructions destinées à manipuler directement des bits d'un registre particulier. Elles sont codées de la manière suivante :

- 4 bits pour l'instruction (dans l'espace resté libre par les instructions précédentes)

- 3 bits pour indiquer le numéro du bit à manipuler (bit 0 à 7 possible), et de nouveau :

- 7 bits pour indiquer l'opérande.

4.2.3 Les instructions générales

Ce sont les instructions qui manipulent des données qui sont codées dans l'instruction directement. Nous verrons ceci plus en détail lorsque nous parlerons des modes d'adressage. Elles sont codées de la manière suivante :

- L'instruction est codée sur 6 bits

- Elle est suivie d'une valeur IMMEDIATE codée sur 8 bits (donc de 0 à 255).

4.2.4 Les sauts et appels de sous-routines

Ce sont les instructions qui provoquent une rupture dans la séquence de déroulement du programme. Elles sont codées de la manières suivante :

- Les instructions sont codés sur 3 bits

- La destination codée sur 11 bits

Nous pouvons déjà en déduire que les sauts ne donnent accès qu'à 2K de mémoire programme (2^{11}). Ceci ne pose aucun problème, le 16F84 ne disposant que de 1K mots de mémoire. Pour coder une adresse de saut à l'intérieur de la mémoire programme, il faut donc 10 bits ($2^{10} = 1024 = 1K$).

Par convention, en effet, 1Kbytes correspond à $2^{10} = 1024$ octets. Ce qui explique que si vous avez 16K de mémoire, en réalité vous avez 16*1024 = 16384 bytes. Par extension, 1Mbyte = 1024 Kbytes, donc 1048576 octets.

En fait, la réglementation officielle voudrait qu'on utilise le terme de « kilobinary » ou « kibi », abrégé en « Ki » pour exprimer 2 à la puissance 10. Il semble cependant que bien peu de monde ne l'utilise en pratique.

Maintenant vous saurez pourquoi vous voyez plus que votre mémoire théorique lors du test mémoire au démarrage de votre ordinateur.

Une petite parenthèse qui n'a rien à voir ici : les fabricants de disques durs considèrent que 1Mbytes = 1000000 bytes. Comme Windows indique la taille en Mbytes de 1048576 bytes, cela vous explique pourquoi la plupart de vos disques durs semblent plus petits que prévus.

4.3 Panoramique des instructions

Je vais maintenant vous montrer comment fonctionne le tableau de la figure 9-2 page 56. Ce tableau vous permet d'un simple regard de vous informer de la manière dont fonctionne chaque instruction

La première colonne indique le MNEMONIQUE et les OPERANDES pour chaque opération. Les mnémoniques sont des mots réservés (donc que vous ne pouvez utiliser que pour cet usage) compris et interprétés par le programme d'assemblage.

Notez ici la confusion de langage commune pour le terme ASSEMBLEUR, qu'on utilise à la fois pour indiquer le programme qui permet d'assembler le code (programme d'assemblage), et le langage utilisé dans l'éditeur (langage d'assemblage).

Pour ne pas se compliquer la tâche, et pour employer la même dénomination que le commun des mortels, j'adopterai le terme assembleur dans les 2 cas, même si je commets là une faute de langage. Autant rester simple. Le contexte distinguera les deux termes. S'il y a un doute, j'emploierai les termes explicites. En réalité, l'assembleur est le programme qui permet de réaliser l'assemblage.

Vous allez donc trouver à cet emplacement les instructions proprement dites que vous allez pouvoir encoder dans votre programme.

La syntaxe doit être la suivante pour l'assembleur MPLAB®, que nous utiliserons dès la prochaine leçon. Nous avons dans l'ordre :

- Etiquette (facultative)
- Espace(s) ou tabulation(s)
- Mnémonique (en majuscules ou minuscules)
- Tabulation ou Espace(s)
- Opérande ou la valeur
- Virgule éventuelle de séparation
- Bit de destination W ou F ou éventuellement numéro du bit de 0 à 7 si nécessaire
- Espace(s) ou tabulation(s)
- point-virgule. (facultatif si pas de commentaire)
- Commentaire. (facultatif)

Notez que le mnémonique ne peut pas se trouver en première colonne, et que tout ce qui suit le point-virgule est ignoré de l'assembleur (donc c'est de la zone commentaire).

La première colonne est réservée pour les étiquettes (repères).

Vous disposez également de la possibilité d'insérer un ou plusieurs espace(s) ou tabulation(s) de chaque côté de la virgule.

Voici à titre d'exemple deux lignes valides, les mots en vert sont des mots réservés, en bleu l'instruction, ceux en jaune étant libres, le reste est du commentaire :

```
Ma_ligne            ; Ceci est une étiquette
    MOVF  STATUS,W  ; charge le registre status dans le registre de travail
```

La seconde colonne du tableau donne un bref descriptif de l'instruction.

La troisième colonne donne le nombre de cycles nécessaires pour exécuter l'instruction.

Notez que toutes les instructions nécessitent un seul cycle, sauf les sauts qui en nécessitent 2, et les opérations de test avec saut, lorsque le résultat du test engendre le saut (instructions notées 1(2)).

La 4ème colonne donne ce qu'on appelle l' OPCODE, c'est à dire le mot binaire que MPLAB® va générer pour vous au départ du mnémonique.

Vous ne vous en servirez donc pas, mais sachez que vous pourriez programmer directement le PIC® sans passer par un assembleur, directement en construisant un fichier .hex et en entrant les valeurs trouvées ici.

Vous devriez alors tout calculer, y compris les sauts. C'est ce que j'ai fait à mes débuts sur un processeur 6502, car je ne disposais pas d'un assembleur. On pouvait également utiliser cette technique pour construire des programmes auto-modifiés pour cause de restriction mémoire.

Rassurez-vous, ces techniques appartiennent maintenant au moyen-âge de l'informatique. Tout au plus pouvez-vous mettre en corrélation les valeurs présentes ici avec les valeurs du tableau 9-1 à titre éducatif. Sinon, oubliez cette colonne.

La 5ème colonne est primordiale, car elle donne les INDICATEURS D'ETATS ou STATUS FLAGS affectés (modifiés) une fois l'instruction effectuée. Nous verrons ces indicateurs en détail, car ils constituent les clés de la programmation.

La dernière colonne renvoie à des notes en bas de page. La note 1 est très importante, elle fait allusion à la méthode « lecture/modification/écriture » propre aux ports d'entrées/sortie (I/O).

Nous y reviendrons au moment de la mise en œuvre des PORTS.

La note 2 indique qu'une modification d'un timer remet à zéro son prédiviseur. Nous y reviendrons en abordant le TMR0.

La troisième note indique que si vous vous servez de l'instruction pour modifier le compteur de programme (celui qui pointe sur la PROCHAINE instruction à exécuter), il y aura un cycle supplémentaire. C'est logique car cela équivaut à un saut. Nous verrons que cette technique est pratique pour aller chercher des valeurs dans une table construite en mémoire programme.

4.4 Les indicateurs d'état

Ces indicateurs sont indispensables pour la programmation. Il est donc absolument nécessaire d'avoir compris leur fonctionnement (du moins pour Z et C).

Lisez donc attentivement ce qui suit. Tous les indicateurs sont des bits du registre STATUS. Voyez le tableau page 15. Nous aborderons ici les flags Z et C. Les autres seront traités lors de l'étude des registres.

4.4.1 L'indicateur d'état « Z »

C'est l'indicateur Zéro, il fonctionne de la manière suivante :

Si le résultat d'une opération POUR LEQUEL IL EST AFFECTE, donne un résultat égal à 0, le flag Zéro passe à 1.

Donc, ne vous « mélangez pas les pinceaux ». Dire « si Z = 1 » correspond à dire « si résultat = 0 ». Le tableau 9-2, colonne 5 vous indique les instructions qui modifient Z.

Donc, si vous faites une addition avec ADDWF et que le résultat obtenu est 0, le bit Z sera à 1. Si le résultat est <>0 (différent de 0), le bit Z vaudra 0. Dans les 2 cas il est modifié.

Par contre, si vous stockez une valeur avec l'instruction MOVWF, le bit Z ne sera pas modifié, même si la valeur vaut 0. Ces remarques sont valables pour les autres flags, donc je n'y reviendrai pas.

4.4.2 L'indicateur d'état « C »

C'est l'indicateur pour Carry (report). Si le résultat d'une opération entraîne un débordement, le bit C sera positionné à 1. Il s'agit en fait du 9ème bit de l'opération.

Petit exemple :

Si vous ajoutez B'11111110' (254)
\+ B'00000011' (3)
Vous obtenez B'100000001', (257) donc 9 bits.

Comme les registres du PIC® ne font que 8 bits, vous obtiendrez B'00000001' (1) et C positionné à 1 (en fait le 9ème bit, donc le bit 8, donc 2^8 = 256). Donc le résultat final est de 256 + 1 = 257.

Remarquez que si vous aviez ajouté B'11111110' et B'00000010', vous auriez obtenu B'00000000'.

Dans ce cas, vous auriez eu C à 1 ET Z à 1, ce qui signifie résultat nul, mais avec report (donc résultat = 256).

Nous verrons les autres bits du registre d'état dans la suite de cet ouvrage, au moment où la nécessité se fera sentir.

Notes :…

5. Les débuts avec MPLAB®

Nous allons maintenant démarrer la grande aventure avec notre tout premier et modeste programme. J'expliquerai les instructions et les registres au fur et à mesure de leur utilisation. A partir de la révision 13 du cours, nous travaillerons avec MPLAB® 6.x, qui est la version qui succède à la version 5.x utilisée dans le cours jusque la révision 12.

5.1 Préparation à l'utilisation

La première chose à faire est d'aller chercher la version actuelle de MPLAB® 6.60 sur le site Microchip® : http://www.Microchip.com section « archives ». Vous pouvez également utiliser une version 7.xx assez semblable. Dans ce cours, les copies d'écrans ont été faites avec MPLAB® 6.3, mais les différences devraient être minimes pour les versions plus récentes.

Décompactez le fichier et procédez à son installation. Lors de l'installation, vous aurez plusieurs fenêtres explicatives concernant différents outils de Microchip®, comme le debugger et le simulateur. Comme je présume que vous n'avez pas (encore) ces outils, fermez toutes ces fenêtres en fin d'installation.

Personnellement, je n'aime pas laisser les données avec mes programmes dans la partition principale. Si vous êtes comme moi, créez un répertoire dans un endroit où vous rangez vos data et appelez-le, par exemple DataPIC.

Copiez y le fichier m16F84.asm fourni avec le cours. C'est un fichier que j'ai créé afin de pouvoir démarrer instantanément un nouveau programme. Je l'ai appelé « m16f84.asm », avec « m » pour « maquette ».

Si vous ne désirez pas créer un nouveau répertoire, copiez ce fichier dans le répertoire d'installation : par défaut c:\program files\MPLAB IDE.

Pour chaque nouveau programme que vous allez créer, faites un copier/coller du fichier m16F84.asm. Pour notre premier programme, copiez/collez ce fichier et renommez la copie obtenue en Essai1.asm.

5.2 Création de notre premier projet

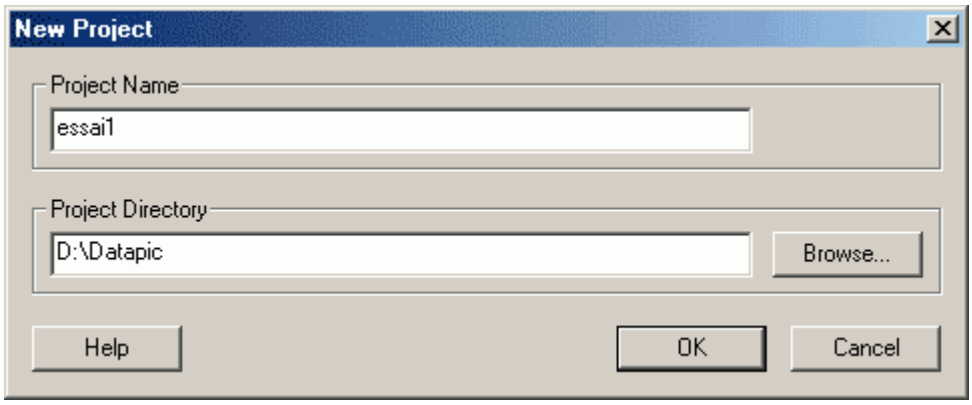
Vous pouvez maintenant lancer MPLAB® IDE à partir du menu démarrer ou de l'icône de votre bureau, si vous en avez accepté l'installation. Après quelques instants, vous vous retrouvez avec un écran vide avec menu et barres d'outils. S'il y a des fenêtres ouvertes dans le bureau de MPLAB® 6, fermez-les toutes, ainsi vous saurez comment les rouvrir, et tout le monde débutera avec la même configuration.

MPLAB® est un logiciel qui est construit sur la notion de projets. Un projet permet de mémoriser tout l'environnement de travail nécessaire à la construction d'un …. projet. Il vous permettra de rétablir tout votre environnement de travail lorsque vous le sélectionnerez.

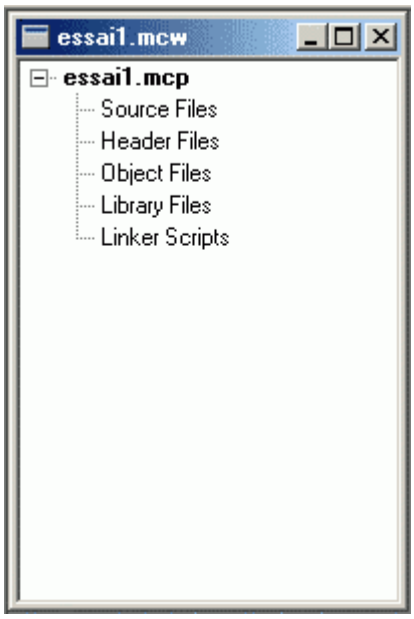
MPLAB® 6 dispose d'un « wizard » pour la création des projets, qui vous permet d'en créer automatiquement de nouveaux. Cependant, afin de vous permettre de localiser les principales options des projets, je ne m'en servirai pas dans ce petit tutorial. Si par la suite vous désirez l'utiliser, il se sélectionne à l'aide du menu « project->wizard »

Allez dans le menu « Project » et sélectionnez « new… ». La fenêtre qui s'ouvre vous permet d'introduire le nom du projet et le répertoire de travail du dit projet. Pour le répertoire, vous disposez du bouton « browser » qui vous permet de pointer sur le bon répertoire sans risque de vous tromper.

Entrez « essai1 » comme nom de votre nouveau projet, et sélectionnez le répertoire dans lequel vous avez placé votre fichier maquette et votre fichier essai1.asm. J'ai nommé le fichier asm de façon identique au projet, mais ce n'est nullement obligatoire.

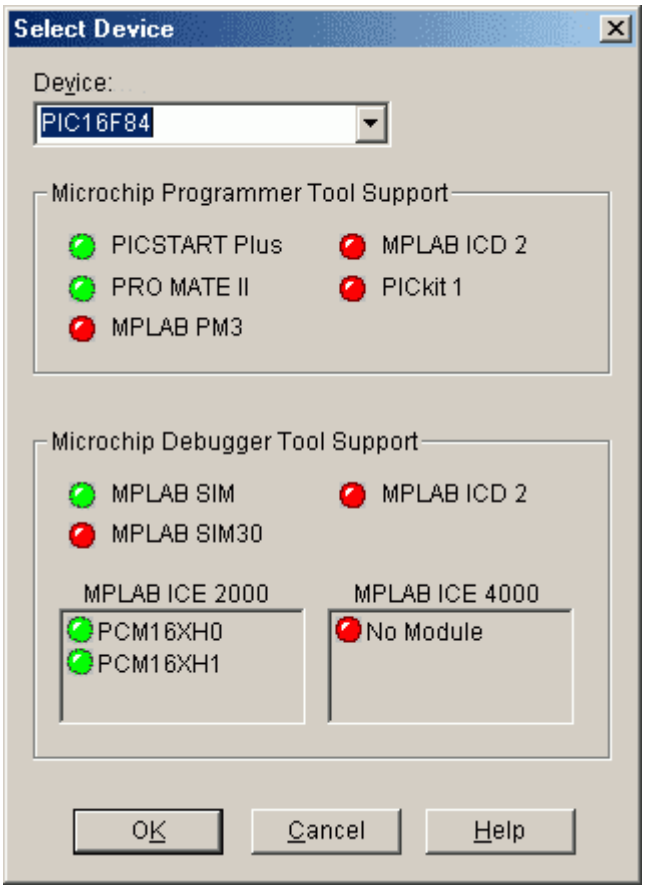


Une fois le bouton <OK> pressé, une nouvelle fenêtre apparaît dans le coin supérieur gauche du bureau de MPLAB® IDE.



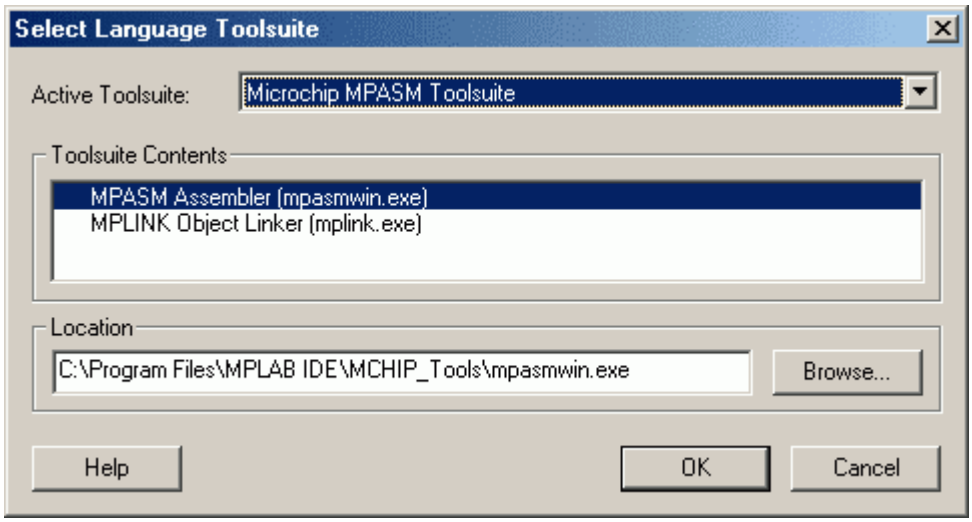
Dans cette fenêtre, vous voyez le nom de votre projet (essai1.mcp), ainsi que les fichiers liés à celui-ci. Pour l'instant, vous n'en avez aucun, c'est normal.

Nous allons commencer à préciser les paramètres importants de notre projet, et tout d'abord le type de PIC® que nous allons utiliser. Sélectionner le menu « configure->select device », une fenêtre apparaît pour vous proposer le choix du PIC®. Notez que par défaut, Microchip® vous propose un PIC® de la famille 18F, promotion de nouveau produit oblige… Sélectionnez le 16F84.



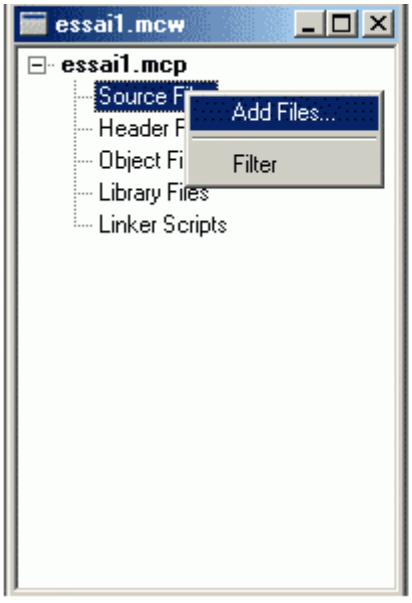
La fenêtre vous montre alors, à l'aide de leds vertes ou rouges quels outils sont supportés par le PIC® sélectionné. Vous voyez donc que le simulateur intégré (MPLAB® SIM) fonctionne avec ce PIC®, et ainsi de même pour les autres outils de développement disponibles chez Microchip®. Une fois cliqué <OK>, la fenêtre se referme.

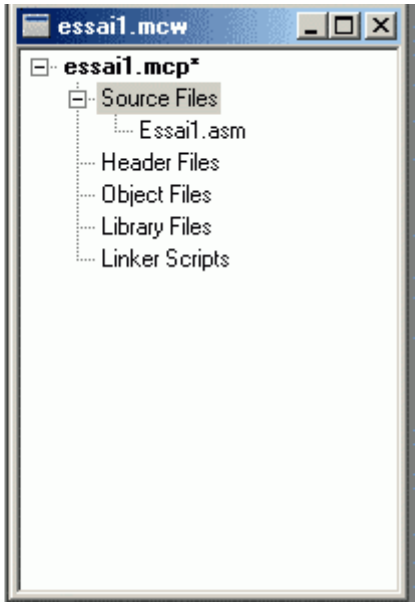
Nous allons maintenant préciser quel langage nous allons utiliser, sachant que nous travaillons en assembleur, mais que différents compilateurs sont proposés par Microchip® et d'autres sociétés. Sélectionnez le menu « project -> Select langage toolsuite ». Dans la fenêtre qui s'ouvre, sélectionnez dans le menu déroulant : « Microchip MPASM® toolsuite ». MPASM® est en effet l'assembleur par défaut de Microchip®.



Dans les fenêtres inférieures, vous voyez le nom des exécutables utilisés par MPASM®, ne vous en préoccupez pas. Cliquez <OK> pour fermer la fenêtre.

Il nous faut maintenant indiquer à MPASM® quel est notre ou nos fichiers source(s). Ceci s'effectue dans la fenêtre « essai1.mcw » qui est restée ouverte dans le coin supérieur gauche. Pour ajouter un fichier source, c'est très simple : cliquez avec le bouton droit sur « source files », puis sélectionnez « Add ». Une fois sélectionné le fichier, le nom de celui-ci apparaît dans l'arborescence. Notez que MPLAB® pointe par défaut sur votre répertoire de travail, précédemment choisi.





Ici, il est important de bien comprendre que le fichier source choisi sera celui qui sera assemblé ou compilé. Autrement dit, si dans ce fichier se trouve une instruction « include » qui inclut un autre fichier, vous ne devez pas le sélectionner en supplément, il sera joint au projet au moment de l'assemblage.

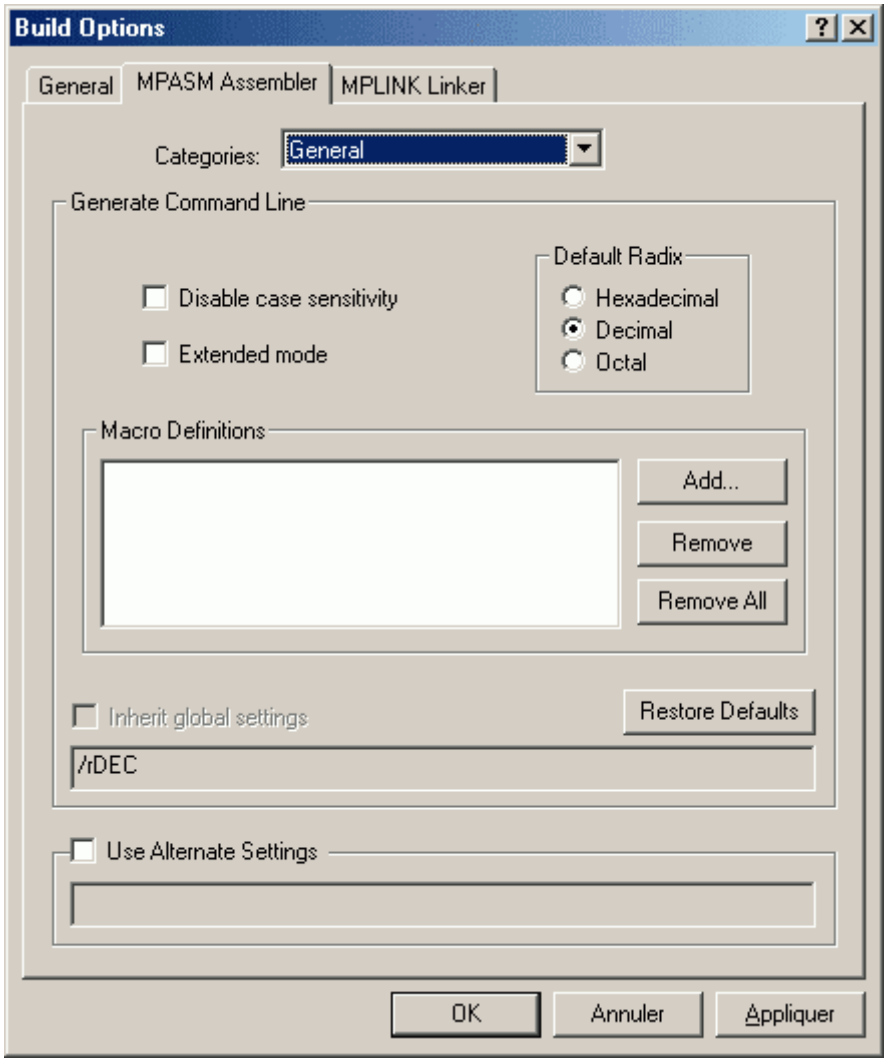
Par contre, si vous désirez assembler simultanément deux fichiers, et que votre premier fichier ne fait pas référence au second, alors vous devez ajouter ce fichier à l'arborescence.

Tous nos projets ne nécessiteront d'ajouter qu'un seul fichier. Si votre projet nécessite un ou plusieurs fichier(s) supplémentaire(s), la directive « include » sera ajoutée au fichier source afin que l'assembleur intègre automatiquement ce ou ces autre(s) fichier(s).

Vous n'avez besoin de rien d'autre, les autres éléments de l'arborescence ne sont pas nécessaires pour un projet assembleur du type dont nous allons parler.

Il nous reste un élément important à préciser, celui du système de numération utilisé par défaut. Sélectionnez le menu : « Project -> build options -> project ». Une fenêtre s'ouvre.

Sélectionnez l'onglet « MPASM assembler ».



Comme nous avons décidé de numéroter l'hexadécimal sous la forme 0x, et le binaire sous la forme B''. Tout nombre sans préfixe sera considéré comme décimal. Cochez donc la case « Decimal ». Ne touchez pas aux autres options, vous les utiliserez peut-être lorsque vous serez devenus un « pro » et que vous voudrez adapter certains paramètres. A ce moment, leur rôle vous semblera clair.

Afin d'éviter tout oubli, je vous conseille cependant de TOUJOURS placer un préfixe devant vos nombres, ceci vous évitera bien des déboires. Utilisez par exemple les syntaxes suivantes :

0x10, H'10', ou 10h	: notations hexadécimales
B'00010000'	: notation binaire
D'16 '	: notation décimale
.16	: autre notation décimale (n'oubliez pas le « . »)

Un nombre écrit sous la forme sans préfixe sera interprété selon le paramètre précédemment décrit, et donc risque de poser problème si vous intégrez votre fichier dans un autre projet, ou, simplement, si vous communiquez à autrui votre fichier source sans préciser la notation par défaut.

Maintenant, MPASM® est prêt à assembler votre programme. Mais, allez-vous me dire, il n'y a toujours rien à l'écran !

Effectivement, MPASM® ne se préoccupe pas des fichiers affichés à l'écran, seuls comptent le fichier inscrit dans l'arborescence et les fichiers inclus à partir de ce fichier.

Vous pouvez avoir autant de fichiers que vous voulez sur l'écran, ou ne pas en avoir du tout. MPLAB® s'en moque. Par contre, tous les fichiers ouverts lors de la sauvegarde de votre projet seront automatiquement rouverts lors de la prochaine ouverture du projet, même s'ils ne servent à rien. Cette propriété est pratique pour faire des copier/coller à partir d'un autre fichier ou pour voir tout en programmant ce qu'un autre a bien pu faire.

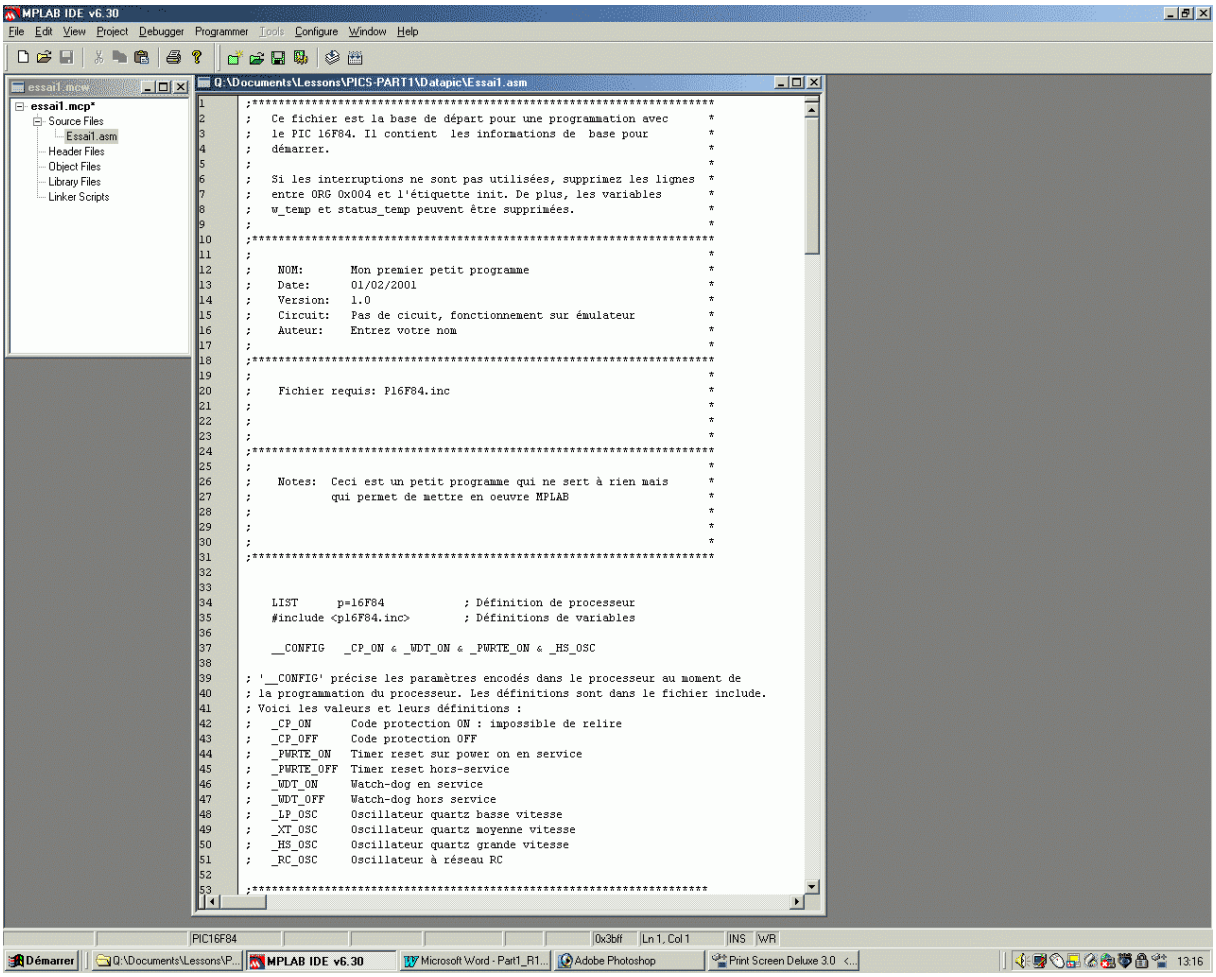
Bien entendu, nous allons afficher notre fichier source, pour voir ce qu'il contient. La méthode la plus simple est de double-cliquer sur son nom dans l'arborescence de la fenêtre essai1.mcw. Faites-le. Voici votre fichier de départ ouvert. Mettez-le à la taille appropriée en lui faisant occuper la moitié gauche de l'écran si vous manquez de place (vous n'aurez plus besoin de fenêtre du projet, vous pouvez la masquer ou la réduire).

Pour ouvrir un fichier qui n'est pas inclus dans l'arborescence, utilisez le menu « file->open » ou l'icône d'ouverture dans la barre d'outils.

Si vous voulez que vos écrans ressemblent au mien, changez la police de l'éditeur. Pour ceci, sélectionnez « Edit -> properties », puis l'onglet « text ». Ensuite, un clic sur le bouton «Select font… » vous permettra de choisir la police de caractères : « Courier New » de style « Standard » et d'imposer une taille de « 9 ».

Pour ceux que ça intéresse, je travaille sur un moniteur 22 pouces, et ma résolution est de 1280 X 1024, ça vous indique les proportions. Vérifiez également dans l'onglet « Tabs » que la longueur de la tabulation est fixée à 4 (« size » sur MPLAB® >7.5). Au besoin, modifiez-la.

Votre fichier asm est maintenant affiché à l'écran. Notez pour ceux qui viennent de MPLAB® 5 que la gestion des tabulations est différente sur MPLAB® 5 et MPLAB® 6, ce qui signifie que les mises en page d'un fichier écrit sur MPLAB® 5 ne seront plus respectées lors du passage à MPLAB® 6. J'ai donc modifié tous les fichiers en conséquence. Ceux qui disposent de la version précédente ont intérêt à utiliser les nouveaux fichiers.



Pour ceux qui viennent de MPLAB® 5, vous constatez que MPLAB® 6 est plus simple à manipuler, ceci va encore plus se confirmer lorsque nous verrons le simulateur.

Nous allons maintenant examiner ce fichier d’un peu plus près.

Notes…

6. Organisation d'un fichier « .asm »

Tout d'abord, cliquez n'importe où à l'intérieur de ce fichier. Vous êtes à l'intérieur d'un simple traitement de texte. Dans le coin inférieur gauche, vous verrez un numéro de ligne et de colonne. C'est la position actuelle de votre curseur. Je me servirai de cette position pour vous guider au travers du fichier. N'ajoutez donc pas de lignes pour l'instant, pour garder la correspondance correcte avec ce texte. Si vous ne voyez pas de numéro de lignes, allez dans le menu « edit -> properties » et cochez « line numbers » dans l'onglet « editor »

Si vous n'arrivez pas à effectuer des modifications dans votre fichier, et que votre clavier semble inactif (rencontré avec MPLAB® 5.x), c'est que vous avez utilisé un caractère étendu dans le nom de votre fichier. Au contraire de MPLAB® 5, MPLAB® 6 gère maintenant tous les caractères étendus, pour les fichiers et pour les dossiers.

6.1 Les commentaires

De la ligne 1 à la ligne 31 vous voyez un grand cadre. Si vous regardez attentivement le premier caractère de chaque ligne, vous verrez le symbole « ; ». Tout ce qui suit étant considéré comme zone de commentaire, vous pouvez y mettre tout ce que vous voudrez.

Prenez l'habitude de toujours commenter vos programmes. Soyez sûr que dans 6 mois, vous ne vous rappellerez plus ce que vous avez voulu faire, les commentaires vous seront alors d'une grande utilité si vous décidez de modifier votre programme.

Remplissons donc le cadre en indiquant les différentes références. Je joins avec cette leçon le fichier « essai1.asm », qui reprend le fichier tel qu'il sera à la fin de cette leçon.

6.2 Les directives

A la ligne 34, nous trouvons une DIRECTIVE destinée à MPASM® pour lui indiquer quel type de processeur est utilisé dans ce programme.

Les DIRECTIVES ne font pas partie du programme, elles ne sont pas traduites en OPCODE, elles servent à indiquer à l'assembleur de quelle manière il doit travailler. Ce sont donc des COMMANDES destinées à l'assembleur en lui-même.

Au contraire, les INSTRUCTIONS seront traduites en OPCODE et chargées dans le PIC®. Il est donc impératif de bien faire la distinction.

6.3 les fichiers « include »

La ligne 35 signale à l'assembleur que les ASSIGNATIONS sont dans le fichier P16F84.inc. Que contient ce fichier ? Et bien tout simplement la valeur de toutes les CONSTANTES que nous allons utiliser.

Pour voir ce qu'il contient, allez dans le menu « file ->Open », choisissez « all source files » dans le cadre inférieur, et ouvrez p16F84.inc. dans le répertoire C:\Program Files\MPLAB IDE\MCHIP_Tools pour MPLAB® 6 et C:\Program Files\Microchip\MPASM Suite pour MPLAB® 7

Une fois dépassée la zone de commentaires, vous verrez des lignes du style :

```
FSR     EQU     H'04'
```

Cette ligne signifie tout simplement que FSR est EGAL à 0x04. Autrement dit, lorsque vous utiliserez FSR dans une instruction, MPASM® interprétera FSR comme étant 0x04. 0x04 étant tout simplement l'adresse de FSR dans la mémoire du PIC®.

H'04' est une autre méthode autorisée pour exprimer un nombre hexadécimal, tout comme 04h

Si vous prenez votre tableau 4-2 page 13, vous constaterez que c'est bien le cas. Ce fichier est donc principalement destiné à vous éviter d'avoir à mémoriser toutes les adresses, un nom est bien plus simple à utiliser et à retenir. Fermez le fichier p16F84.inc pour ne pas encombrer votre fenêtre.

6.4 La directive _CONFIG

La ligne suivante, commence par « __CONFIG ». Cette ligne contient les fameux « fusibles » qui fixent le fonctionnement du PIC®.

Les valeurs écrites ici seront intégrées dans le fichier « .hex » pour signaler au programmateur les valeurs à encoder aux adresses spécifiques du PIC®. Nous y reviendrons.

Sachez donc que si un fichier « .hex » a été créé par un programmeur attentif qui a utilisé cette directive, vous n'aurez nul besoin de définir ces paramètres au moment de la programmation. Le logiciel du programmateur ira normalement chercher ces valeurs dans le fichier lui-même.

Malheureusement, on trouve nombre de personnes qui pensent que cette directive, commentaires, et autres facilités sont à réserver aux débutants et ne les utilisent pas. Grossière erreur. Je reçois régulièrement du courrier de personnes qui ont estimé cette procédure inutile, et qui se sont trompées par la suite.

J'ai placé dans le fichier toutes les valeurs possibles de ces paramètres, avec les explications correspondantes. Il suffit de remplacer une des valeurs par celle souhaitée.
Par exemple, activons le Code Protect (protection en lecture) :

On remplacera donc simplement la ligne :

```
__CONFIG    _CP_OFF & _WDT_ON & _PWRTE_ON & _HS_OSC
```

par

```
__CONFIG    _CP_ON & _WDT_ON & _PWRTE_ON & _HS_OSC
```

Faites-le. Remarquez que les différentes valeurs sont liées par le symbole « & » (AND) expliqué dans la leçon sur les systèmes de numérations. Ils fonctionnent donc en plaçant des bits à « 0 », si vous avez tout suivi. Donc, attention de toujours bien préciser tous les paramètres, même ceux que vous n'utilisez pas.

Les valeurs correspondantes sont de nouveau dans le fichier « P16F84.INC ». Donc, pas de magie, tout s'explique en réfléchissant un peu.

6.5 Les assignations

Aux lignes 57 et 62, vous trouverez des ASSIGNATIONS personnelles qui fonctionnent selon le même principe que dans le fichier « .inc ».

A quoi cela sert-il ? Et bien à faciliter la MAINTENANCE de votre programme. Il est en effet plus simple de retenir dans votre programme la valeur « MASQUE » que de manipuler la valeur 0x5B.

Les assignations se comportent comme une simple substitution. Au moment de l'assemblage, chaque fois que l'assembleur va trouver une assignation, il la remplacera automatiquement par sa valeur.

Un autre avantage est que si vous remplacez la valeur d'une assignation, le changement sera effectif pour tout le programme. Vous ne risquez donc pas d'oublier des valeurs en chemin.

Je vous encourage vivement à utiliser les ASSIGNATIONS et autres méthodes que nous allons voir plus bas. La syntaxe est simple puisqu'il s'agit de EQU (égal à)

Exemple d'assignation :

```
mavaleur     EQU    0x05
```

6.6 Les définitions

Descendons encore un peu. Nous découvrons, lignes 72 à 74 des exemples de DEFINE. Sachez que les « define » fonctionnent comme des ASSIGNATIONS. A ceci près que nous réserverons les assignations pour les valeurs, et les définitions pour remplacer un texte plus complexe.

Par exemple nous pourrons utiliser un PORT suivi d'un numéro de bit, ou bien carrément une instruction avec ses paramètres.

Une définition est construite de la manière suivante : La directive #DEFINE, suivie par le nom que l'on désire utiliser, puis la chaîne à substituer. Par exemple :

```
#DEFINE      monbit       PORTA,1
```

L'utilisation de cette macro s'effectue tout simplement en utilisant son nom dans le programme. Par exemple :

```
    bsf      monbit       ; mettre monbit à 1
```

Sera traduit par MPLAB® comme suit :

```
    bsf      PORTA,1      ; mettre monbit à 1
```

6.7 Les macros

Plus bas, lignes 82 à 85, nous trouvons une MACRO.

```
LIREIN macro
      comf PORTB,0
      andlw  1
      endm
```

La macro se compose d'un nom écrit en première colonne, suivi par la directive « macro ». Commence alors à la ligne suivant la portion de code qui constitue la macro. La fin de la macro est définie par la directive « endm ») (end of macro).

Une macro remplace donc un morceau de code que nous utilisons souvent. La macro fonctionne également uniquement comme un simple traitement de texte.

Dans notre exemple, chaque fois que la macro LIREIN sera rencontrée, elle sera remplacée au moment de l'assemblage par les 2 lignes :

```
    comf     PORTB , 0
    andlw    1
```

La macro simplifie donc l'écriture, mais ne raccourcit pas la taille du fichier .hex obtenu, puisque les 2 lignes seront écrites dans le PIC®.

Notez que l'on peut utiliser des macros plus complexes, avec passage de paramètres, mais nous n'entrerons pas dans ces fonctions particulières pour l'instant.

Notez également que vous disposez d'une aide dans le menu « help->MPASM Help ». En effet, l'aide de MPLAB® concerne l'utilisation du logiciel. Les aides concernant le langage sont dans MPASM®, puisque c'est ce langage que MPLAB® utilise (revoyez l'édition des nœuds).

6.8 La zone des variables

Toute zone définie par l'utilisateur commence avec la DIRECTIVE CBLOCK, suivie par l'adresse du début de la zone.

Pour placer nos variables, qui sont des emplacements mémoires auxquels on a donné un nom, nous consultons de nouveau le tableau 4-2. Nous voyons que la zone RAM librement utilisable commence à l'adresse 0x0C. Notre zone de variable contiendra donc la directive

```
    CBLOCK 0x00C          ; début de la zone variables
```

Ensuite, vous pouvez utiliser 68 emplacements mémoire, qui répondront à la syntaxe suivante : nom de la variable suivi du signe « : » suivi de la taille utilisée.
Par exemple :

```
w_temp :1       ; Zone de 1 byte
montableau : 8  ; zone de 8 bytes
```

Ensuite, vous devrez préciser la fin de la zone en cours à l'aide de la directive :

```
ENDC            ; Fin de la zone
```

6.9 Les étiquettes

Vous trouverez dans les programmes en 1ere colonne ce que nous appellerons des ETIQUETTES. Ce sont des noms que vous choisissez et qui sont des REPERES pour le programme.

L'assembleur les remplacera par l'adresse du programme à l'endroit où elles sont positionnées. Ceci vous évite de devoir calculer les emplacements programme. Nous en verrons plus loin le principe.

6.10 La directive « ORG »

La directive ORG, suivie de l'adresse, précise à quelle adresse les instructions qui suivent seront placées dans le PIC®. Il est important de savoir 2 choses :

Après un reset ou une mise sous tension, le PIC® démarre toujours à l'adresse 0x00. Le début de votre programme doit donc se situer là.

L'adresse 0x04 est l'adresse utilisée par les interruptions (nous verrons le principe plus tard). Il ne vous reste donc pas grand place pour placer votre programme. Nous commencerons donc par un saut vers l'emplacement du programme principal où nous aurons plus de place. Allons donc voir ligne 103 comment tout ceci fonctionne :

```
org 0x000           ; Adresse de départ après reset
goto    init        ; Adresse 0: initialiser
```

La première ligne est une DIRECTIVE qui indique que la ligne suivante sera placée à l'adresse 0x00.

La seconde ligne est une INSTRUCTION, expliquée page 62, qui indique au PIC® que le programme doit SAUTER à l'adresse « init ». « init » est une ETIQUETTE.

Après le reset, le PIC® exécute donc l'instruction goto init qui se trouve à l'adresse 0x00, suivie par l'instruction qui se trouve à l'adresse init plus bas dans le programme (donc juste en dessous de l'étiquette init).

6.11 La directive « END »

Cette directive précise l'endroit où doit cesser l'assemblage de votre programme. Elle est obligatoire dans tout programme, sous peine d'une erreur qui vous signalera que la fin de fichier (End Of File) a été atteinte avant de rencontrer la directive END.

Toutes les instructions situées après la directive END seront tout simplement ignorées.

7. Réalisation d'un programme

7.1 Création de notre premier programme

Avant de lancer notre premier programme, nous allons procéder à quelques modifications du fichier « essai1.asm », afin de ne conserver que ce qui nous intéresse.

Premièrement, allez ligne 104 (le numéro peut être différent chez vous) et remplacez la ligne goto init par goto start. Ne faites pas de faute d'orthographe.

```
goto    start          ; Adresse 0: démarrer
```

Descendez ensuite en ligne 225, vous trouverez là notre étiquette start, c'est donc là que notre programme va sauter. Effacez ensuite la ligne

```
clrwdt                 ; effacer watch dog
```

Rassurez-vous, nous verrons tout ceci par la suite.

Il reste donc une instruction que nous avons déjà vue, l'instruction « goto ».

```
goto start             ; boucler
```

Le datasheet, page 62, nous apprend que l'instruction « goto » est suivie d'une valeur IMMEDIATE (c'est à dire une valeur sous forme d'un nombre), codée sur 11 bits (rappelez-vous, le programme peut faire 1KMOTS, donc, avec 11 bits, on peut sauter n'importe où dans la mémoire programme.

La valeur peut bien sûr être remplacée par une étiquette, MPASM® se chargeant pour vous de calculer son emplacement. Le datasheet vous indique aussi qu'il s'agit d'un saut INCONDITIONNEL, c'est à dire qu'il s'effectuera toujours, sans condition. Il est rappelé qu'un saut prend 2 cycles.

Faisons de la place sous l'étiquette start, et ajoutons la ligne suivante (attention, pas en première colonne) :

```
clrf    mavariable   ; effacer mavariable
```

CLRF est une instruction détaillée dans le chapitre dédicacé aux instructions. Il y est dit que l'emplacement mémoire précisé derrière l'instruction (ou une variable) est effacé. Le bit Z est positionné à 1 ou à 0 selon le résultat de l'opération, comme toujours.

Mais comme le but de l'opération est de mettre la variable à 0, le bit Z vaudra toujours 1 après cette instruction. Notez que l'emplacement mémoire peut se situer de 0 à 127 (0x7F). C'est logique : si vous consultez le tableau 4-2, vous voyez que la RAM s'arrête au 127ème emplacements pour chacune des 2 banques.

Placez ensuite une étiquette (colonne1) que vous appellerez boucle. Sous cette étiquette, ajoutez l'instruction :
boucle

```
INCF        mavariable,f
```

Cette instruction est expliquée également dans le chapitre relatif aux instructions. Vous voyez que cette instruction incrémente (+1) le contenu de la variable mavariable, et que le résultat de l'opération est placé dans l'emplacement d. Pour toutes les instructions, d peut valoir soit « f », dans ce cas le résultat est stocké dans la variable en question, soit « w », dans ce cas le résultat est placé dans le registre de travail, et la variable n'est pas modifiée.

Vous voyez également que le bit de STATUS 'Z' est affecté par l'opération. Je rappelle une dernière fois : si le résultat de l'incrémentation donne '0', Z sera mis à 1. Il sera mis à 0 dans tous les autres cas.

Pour terminer, remplacez

```
    goto start
```

par

```
    goto boucle
```

Le programme doit toujours se terminer par la DIRECTIVE « END »
Vous devriez donc vous retrouver avec ceci :

```
start
    clrf mavariable       ; effacer mavariable
boucle
    incf mavariable,f     ; incrémenter mavariable
    goto boucle           ; boucler
END                       ; directive fin de programme
```

7.2 L'assemblage d'un programme

L'assemblage d'un projet peut s'effectuer de 2 façons. Soit on assemble uniquement le fichier sélectionné avec <F10>, soit on assemble tous les fichiers de l'arborescence en pressant <CTRL> + <F10>. Etant donné que nous n'avons qu'un seul fichier dans l'arborescence, nous utiliserons <F10>

Nous allons maintenant tenter d'assembler ce programme pour en faire un fichier .hex. Pressez la touche « F10 », des fenêtres s'ouvrent, MPLAB® passe les commandes à MPASM® qui tente d'assembler notre programme. L'assemblage s 'arrête en chemin, une barre rouge apparaît et nous nous retrouvons une nouvelle fenêtre («output ») qui contient les résultats de sortie de la commande. Si vous rencontrez un problème non prévu (erreur 173 par exemple) jetez un oeil en fin de cours, dans les annexes.

```
Deleting intermediary files... done.
Executing: "C:\Program Files\MPLAB IDE\MCHIP_Tools\mpasmwin.exe" /q /p16F84 "Essai1.asm"
/l"Essai1.lst" /e"Essai1.err" /rDEC
Message[302] D:\DATAPIC\ESSAI1.ASM 199 : Register in operand not in bank 0.  Ensure that bank bits are
correct.
Message[302] D:\DATAPIC\ESSAI1.ASM 215 : Register in operand not in bank 0.  Ensure that bank bits are
correct.
Error[113]   D:\DATAPIC\ESSAI1.ASM 227 : Symbol not previously defined (mavariable)
Error[113]   D:\DATAPIC\ESSAI1.ASM 229 : Symbol not previously defined (mavariable)
Halting build on first failure as requested.
```

Que s'est-il passé ? Et bien, examinons le rapport construit. Vous voyez d'abord des messages (warning). Ce sont des messages destinés à attirer votre attention, mais qui ne génèrent pas d'impossibilité d'assemblage. N'oubliez pas qu'il y a toute une partie du programme qui ne sert à rien dans notre exemple, mais qui est tout de même écrite (au dessus de l'étiquette start). Nous mettrons cette partie en pratique dans les chapitres suivants.

Le problème vient évidemment des lignes « error ». Il y est dit que le symbole « mavariable » n'a pas été défini par l'utilisateur. Cette erreur se retrouve dans les 2 lignes où nous avons utilisé notre variable. En fait, nous avons oublié de la DECLARER. Pour remédier à cela, fermons la fenêtre des messages d'erreurs, et retournons dans notre éditeur.

Faisons de la place sous la ligne 97, dans la zone des variables. Ajoutons alors

```
mavariable : 1          ; je déclare ma variable
```

Vous vous retrouvez donc avec ceci :

```
CBLOCK 0x00C              ; début de la zone variables
w_temp :1                 ; Zone de 1 byte
status_temp : 1           ; zone de 1 byte
mavariable : 1            ; je déclare ma variable
ENDC                      ; Fin de la zone
```

Relançons l'assemblage en pressant sur F10.

Cette fois, tout s'est bien passé, l'indicateur est resté vert, et, après les warnings, nous obtenons la phrase :

BUILD SUCCEEDED: construction achevée avec succès.

Nous venons de construire notre premier programme, et nous avons déjà appris 3 des 35 instructions que comporte le PIC®. Qui a dit que c'était compliqué ?

Vous pouvez laisser la fenêtre de sortie ouverte, la fermer ou la minimiser. Sortons de notre programme, et confirmez les demandes de sauvegarde. En cours d'édition, sauvez régulièrement votre fichier source à l'aide de <CTRL> + < s>

Allez voir maintenant dans votre répertoire de travail, et vous devriez y trouver 7 fichiers générés par votre application du type « essai1.xxx »

Remarquez surtout l'existence de votre premier fichier .hex. Le fichier tel qu'il devrait être à la fin de ce chapitre, est disponible dans les fichiers d'exemples, comme tous les autres créés par la suite.

Notes :…

8. La simulation d'un programme

Nous avons créé, dans le chapitre précédent, notre premier petit programme en 16F84. Bien sûr, ce programme ne sert strictement à rien. Cependant nous allons l'utiliser en vue d'expérimenter le simulateur de MPLAB®. Une fois ce chapitre vu, vous serez en mesure de :

- Créer et modifier un projet
- Assembler votre programme
- Le faire tourner sur simulateur pour le debugger

8.1 Lancement et paramétrage du simulateur

Commencez donc par lancer MPLAB®. Ce dernier connaît le nom de votre dernier projet (essai1.mcp) et l'ouvre automatiquement. Si vous avez procédé à des essais personnels depuis, chargez-le depuis le menu « project->open ». Vous vous retrouvez avec la fenêtre de votre dernière leçon.

Rappelez-vous que si vous avez un problème, le fichier « essai1.asm » est disponible dans les fichiers d'exemples fournis avec ce cours. Rappelez-vous alors de lancer la compilation par la touche <F10>.

L'intérêt d'un simulateur est de visualiser le fonctionnement du programme, nous allons donc signaler à MPLAB® ce que nous désirons observer. Sélectionnons d'abord la mise en service du simulateur de MPLAB®.

Sélectionnez le menu : « Debugger -> select tool -> MPLAB® Sim ». C'est tout. De nouveaux outils apparaissent dans la barre d'outils. Notez aussi l'apparition de nouveaux items dans le menu « debugger » Les connaisseurs de MPLAB® 5 constaterons que la procédure est une nouvelle fois beaucoup plus simple.

Si ce n'est déjà fait, placez la fenêtre du fichier source à fond vers la gauche. Nous allons donc faire apparaître à l'écran les informations à surveiller. Sélectionnez « View -> Special function registers ». Une nouvelle fenêtre s'ouvre. Agrandissez-la et placez la à droite de la fenêtre du fichier source (ou ailleurs si vous manquez de place).

Vous voyez dans cette fenêtre le contenu de tous les registres que vous avez découverts dans le tableau 4-2 de la page 13. Dans les chapitres suivants, nous les utiliserons progressivement tous.

Dans notre petit programme, nous utilisons une variable. Nous allons donc la faire apparaître. Allez dans le menu « View -> watch ». Une nouvelle fenêtre apparaît. Vous disposez de 4 onglets pour y placer 4 séries de variables, c'est très pratique pour debugger.

Pour visualiser une variable, vous disposez de plusieurs méthodes.

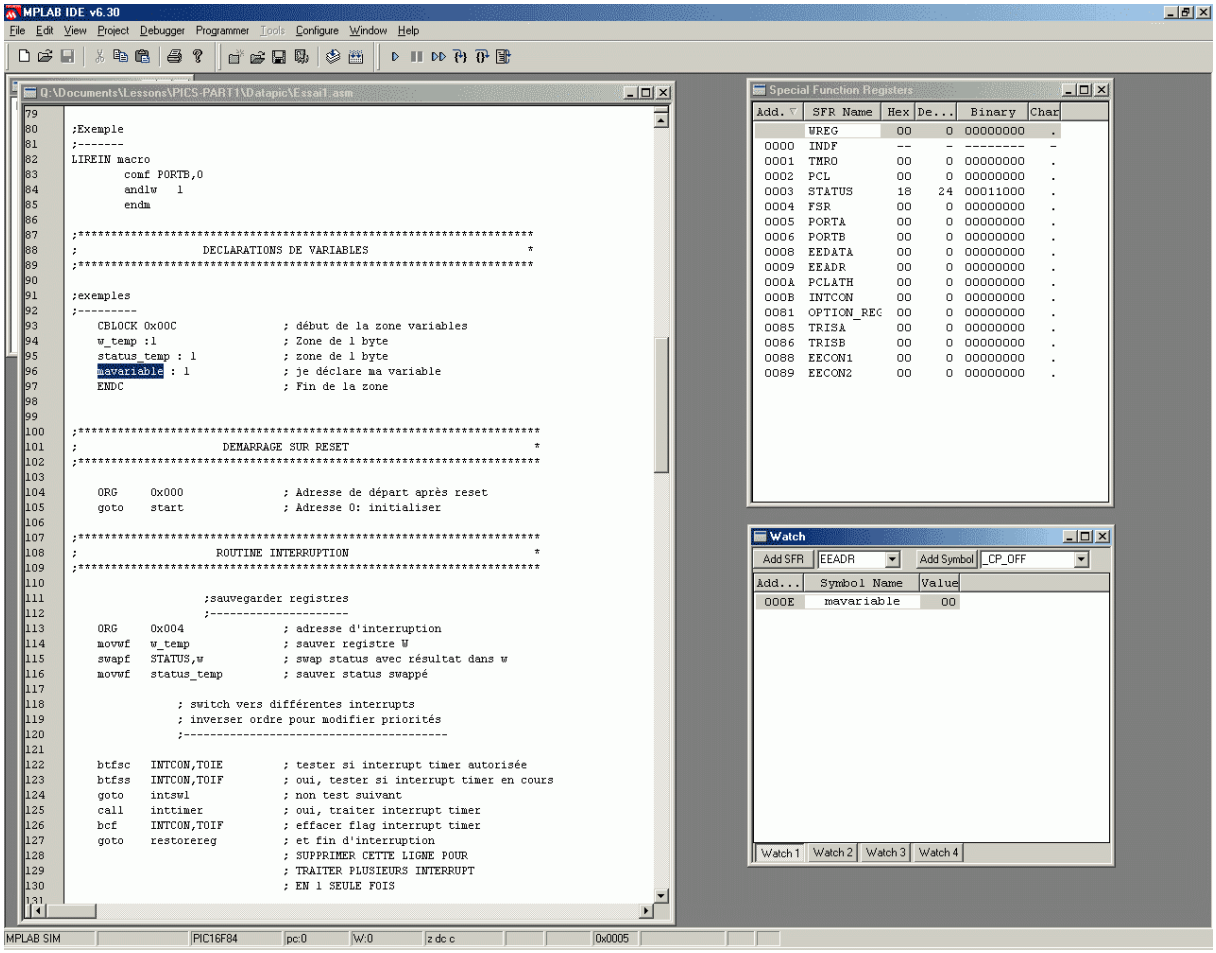
- Vous double-cliquez dans la case située juste sous « symbol name » et vous tapez le nom de la variable. Si la variable n'existe pas, la mention « not found » apparaît dans la colonne « value », dans le cas contraire, c'est la valeur actuelle qui apparaît.

- Vous double-cliquez dans la case juste sous la colonne « Add.. » (adresse), et vous entrez manuellement l'adresse (dans ce cas-ci, 0x00E, vous entrez donc « E »)

- Vous sélectionnez le nom dans le fichier source, puis vous cliquez une fois sur la sélection obtenue, et vous « tirez » la variable vers la fenêtre « Watch » en maintenant le bouton enfoncé. Une fois au-dessus, relâchez le bouton de la souris (« Drag and drop »).

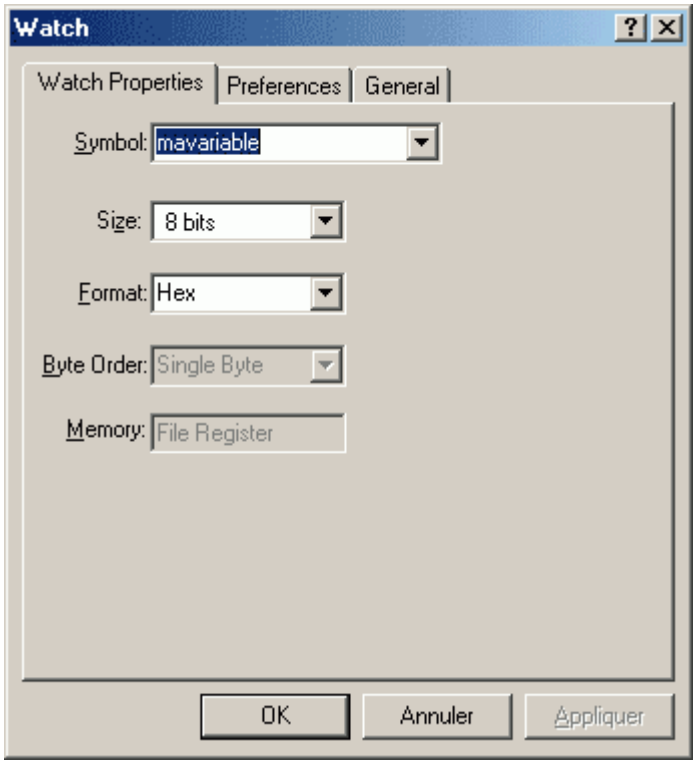
  Pour supprimer une variable, il suffit de la sélectionner et de presser <Del>.

  Votre environnement ressemble maintenant à ceci :



Notez qu'au démarrage, MPLAB® suppose que la valeur de votre variable vaut « 0 », ce qui n'est pas forcément le cas réellement dans le PIC®, puisque la RAM prend une valeur aléatoire lors d'une procédure de mise sous tension.

Cliquez avec le bouton droit de la souris, puis « properties » pour indiquer le style d'affichage des valeurs, soit 8 bits et format hexadécimal, mais ceci doit être fait par défaut.



Pressez ensuite sur <OK> pour fermer cette fenêtre.

8.2 Explication des registres fondamentaux

Vous voici prêt à lancer une simulation. Mais à quoi cela pourrait-il vous servir si vous ne comprenez pas les changements qui vont s'opérer dans les registres spéciaux ? Je vais donc commencer par vous expliquer les registres de base nécessaires à la compréhension du processus.

8.2.1 Les registres « PCL » et « PCLATH »

Un processeur, quel qu'il soit est un composant qui exécute SEQUENTIELLEMENT une série d'INSTRUCTIONS organisées selon un ensemble appelé PROGRAMME.

Il existe donc dans le processeur un SEQUENCEUR, c'est à dire un compteur qui permet de pointer sur la PROCHAINE instruction à exécuter. Ce séquenceur est appelé suivant les processeurs « compteur ordinal », « Pointeur de programme » etc. Dans le cas des PIC®, il s'appelle PC, pour Program Counter. Le PC n'est pas accessible directement par l'utilisateur.

Le principe de base est toujours le même. Dans les PIC®, les registres ne font que 8 bits, on ne peut donc stocker qu'une adresse maximale de 255. Il faudra donc 2 registres pour accéder à une adresse. Les PIC® ont un fonctionnement un peu particulier à ce sujet.

Nous trouvons tout d'abord un registre qui contient l'adresse basse du PC, c'est à dire les 8 bits de poids faibles. Ce registre est accessible en lecture et en écriture. Il est appelé PCL (PC Low)

Il existe un autre registre de 5 bits qui participe au fonctionnement du séquenceur. Il s'appelle PCLATH (PC LATch counter High). Il est accessible en lecture et en écriture par l'utilisateur.

Le PC complet étant codé sur 13 bits, il faudra donc compléter PCL avec 5 bits supplémentaires. Il existe deux cas possibles :

- Lors d'un saut, par exemple, le contenu du PC est chargé directement avec les 11 bits de destination contenus dans l'instruction en elle-même. Les 2 bits manquants sont extraits du registre PCLATH.. Les bits 3 et 4, qui doivent être positionnés par l'utilisateur, sont placés directement dans les bits 11 et 12 du PC afin de compléter l'adresse de destination. Comme le 16F84 ne gère que 1K de mémoire programme, nous n'aurons pas besoin de ce registre dans le cas des sauts. Rappelez-vous que le 16F84 ne gère que 10 des 13 bits du PC

- En cas de modification du PCL directement par l'utilisateur, comme pour un registre ordinaire, PCL est chargé dans PC et complétés par les 5 bits du registre PCLATH. Comme le 16F84 ne traite que 1K de mémoire programme, les bits b2, b3 et b4 de PCLATH seront inutilisés ici.

Remarquez que la limite du PC est de 13 bits, ce qui implique que les PIC® de la famille mid-range auront une capacité de mémoire programme de 8K mots maximum (soit 2^{13}).

Il est très important de se rappeler que le PC pointe toujours sur l'instruction suivante, donc l'instruction qui n'est pas encore exécutée. C'est indispensable de bien comprendre ceci pour analyser les programmes en cours de debuggage.

8.2.2 Le registre « W »

Ce registre est un registre utilisé par les PIC® pour réaliser toutes sortes de calculs. Souvenez-vous que la destination d'un résultat (d) peut en général être un emplacement RAM (f) ou le registre de travail (w). C'est un donc un registre fondamental.

8.2.3 Le registre « STATUS »

C'est un registre dont chaque bit a une signification particulière. Il est principalement utilisé pour tout ce qui concerne les tests. Il est donc également d'une importance fondamentale. Il est décrit dans le tableau de la page 15.

Voici les différents bits qui le composent, en commençant par le bit0 (b0), donc le bit le plus à droite, ou encore le moins significatif. Remarquez qu'on utilise le terme LSB, parfois comme byte le moins significatif, parfois comme bit le moins significatif. C'est également un abus de langage, mais le contexte permet très bien de les distinguer.

b0 : C Carry (report)	Ce bit est en fait le $9^{ème}$ bit d'une opération. Par exemple, si une addition de 2 octets donne une valeur >255 (0xFF), ce bit sera positionné à 1. .

b1 : DC Digit Carry — Ce bit est utilisé principalement lorsque l'on travaille avec des nombres BCD : il indique un report du bit 3 vers le bit 4. Pour info, un nombre BCD est un nombre dont chaque quartet représente un chiffre décimal. Nous n'aborderons pas ce principe ici.

b2 : Z Zéro — Ce bit est positionné à 1 si le résultat de la dernière opération vaut 0. Rappelez-vous cependant que ces flags ne sont positionnés que pour les instructions qui le précisent (Status bit affected).

b3 : *PD* Power down — Indique quel événement a entraîné le dernier arrêt du PIC® (instruction sleep ou dépassement du temps du watchdog). Nous y reviendrons plus tard. En réalité, vous verrez que PD est surmonté d'une petite barre qui signifie : actif à l'état bas. Donc que 0 = bit validé. Comme je n'arrive pas avec ce traitement de textes à écrire ces petites barres, j'écrirai les inversions en *italique*

b4 : *TO* Time-Out bit — Ce bit indique (si 0), que la mise en service suit un arrêt provoqué par un dépassement de temps ou une mise en sommeil. Dans ce cas, *PD* effectue la distinction.

b5 : RP0 Register Bank Select0 — Permet d'indiquer dans quelle banque de RAM on travaille.0 = banque 0.

b6 : RP1 Register Bank Select1 — Permet la sélection des banques 2 et 3. Inutilisé pour le 16F84, doit être laissé à 0 pour garantir la compatibilité ascendante (portabilité du programme).

B7 : IRP Indirect RP — Permet de décider quelle banque on adresse dans le cas de l'adressage indirect (que nous verrons plus tard).

8.3 Lancement de la simulation

Et voilà, vous connaissez maintenant 4 des registres. Nous allons pouvoir commencer à simuler notre programme.

Contrairement à MPLAB® 5, il n'y a pas besoin de sélectionner la fenêtre source pour les commandes de simulateur, elles sont toujours actives quelle que soit la fenêtre sélectionnée à l'intérieur de MPLAB. Si ce n'est pas fait, pressez <F10> pour assembler, puis pressez <F6>, La ligne

```
goto    start           ; Adresse 0: initialiser
```

Est maintenant indiquée par une flèche verte. La flèche verte indique la prochaine ligne qui sera exécutée par le simulateur. Vous avez en fait provoqué un Reset de votre programme. Au lieu d'utiliser les touches clavier (plus pratiques), vous pouvez également utiliser les nouveaux outils de la barre d'outils ou les nouveaux items du menu « debugger ». La correspondance est la suivante :

<F6> : reset
<F7> : step into (avancer d'un pas dans le programme)
<F8> : step over (idem, mais un appel de sous-programme est exécuté en une fois au lieu d'entrer à l'intérieur).

Rappelez-vous que le reset provoque le démarrage à l'adresse 0x00. Vérifions donc :

- Au dessus de la ligne sélectionnée, vous trouvez la directive « ORG 0x00 » qui précise que la ligne suivante est à l'adresse 0x00 : premier bon signe.

- Ensuite, examinons PCL et PCLATH, tous deux à 0 : Donc, la prochaine instruction exécutée sera celle située à l'adresse 0x00. C'est tout bon.

Examinons la ligne en question. Elle nous dit que la prochaine instruction, après son exécution, sera celle située à l'adresse « start ».

Pressez <F7>. Votre programme est maintenant positionné sur la ligne qui suit l'étiquette « start ». Il a donc effectué un SAUT, ou encore, rupture de séquence.

```
start
    clrf      mavariable  ; effacer mavariable
```

Rappelez-vous qu'à ce stade, l'instruction n'est pas encore exécutée.
Le PCL vaut maintenant :

PCL 34 52 00110100, soit 0x34 ou 52 décimal, ou encore B'00110100'

MPASM® a donc calculé tout seul à quel emplacement se situe l'étiquette « start ». Si vous aviez voulu indiquer l'adresse vous-même, vous auriez dû compter toutes les lignes précédentes pour voir où vous en étiez. De plus, à chaque modification du programme, vous devriez recalculer toutes les adresses. Heureusement, MPASM® le fait pour vous.

Pressez de nouveau <F7> pour exécuter cette instruction : Effacer mavariable. Comme mavariable vaut déjà 0, il ne se passe rien au niveau de celle-ci.

ATTENTION, RAPPEL : à la mise sous tension, les cases mémoires (RAM) du PIC® se trouvent à une valeur quelconque. Or, MPLAB® ne peut savoir quelle est cette valeur. Il met donc par défaut 0. Si vous voulez qu'une case RAM du PIC® contienne effectivement 0, VOUS ETES OBLIGE DE LE FAIRE VOUS-MEME, sinon, le programme fonctionnera correctement sur simulateur, mais pas sur circuit réel.

Et oui, une simulation, ça reste une simulation.

La ligne pointée alors est :

```
incf      mavariable,f ; incrémenter mavariable
```

Le PCL pointe alors sur : PCL 35 53 00110101

C’est donc bien l’adresse suivante : pas de saut signifie pas de rupture de séquence, donc fonctionnement séquentiel du programme.

Profitons-en pour jeter un œil sur la fenêtre « Watch ». Vous voyez votre variable « mavariable ». Sa valeur est 0x00 (ou H’00’), et son adresse est 0x0E. Pourquoi ? Examinons la zone de déclaration des variables :

```
CBLOCK 0x00C              ; début de la zone variables
w_temp :1                 ; Zone de 1 byte
status_temp : 1           ; zone de 1 byte
mavariable : 1            ; je déclare ma variable
```

La zone commence en 0x0C. La variable w_temp se trouve donc à cette adresse et comprend un octet. Status_temp est donc en 0x0D, et mavariable en 0X0E. C’est tout simple.

Pressons encore <F7>. La variable « mavariable » vaut maintenant « 0x01 », car l’opération d’incrémentation a été exécutée. Vous l’avez déjà compris, l’exécution suivante va vous renvoyer à la ligne qui suit l’étiquette « boucle ». Amusez-vous à presser plusieurs fois la touche <F7> pour observer l’évolution de votre variable.

A ce stade, vous vous demandez ce qui va se passer lorsque votre variable atteindra la valeur 0xFF ? Bien entendu, vous n’avez pas envie de presser encore 500 fois la touche <F7>. On va donc accélérer le processus.

Nous allons donc modifier la valeur de « mavariable ». Pour ceci, rien de plus simple, il suffit de double-cliquer sur la case « value » située face au nom de « mavariable » et d’inscrire une nouvelle valeur (en hexadécimal, puisque nous avons choisi ce format.)

Double-cliquez, et indiquez « ff ». Ensuite cliquez n’importe où ailleurs pour valider la valeur.

Pressez maintenant <F7> jusqu’à ce que la ligne suivante soit exécutée :

```
incf mavariable,f  ; incrémenter mavariable
```

La ligne suivante est maintenant sélectionnée :

```
goto boucle        ; boucler
```

Examinons ce qui s’est passé :

La variable « mavariable » est passée à 0. Logique, car 0xFF + 1 = 0x100. Or, 0x100 nécessite 9 bits (256). On obtient donc 00 (et on reporte le bit 8).

Examinons maintenant le registre STATUS : que constate-t-on. ?

Les bits 2, 3 et 4 sont à 1, les autres sont à 0. Les bits 3 et 4 sont actifs à l’état bas (0). Comme ils sont à 1, ils sont donc inactifs.

Reste le bit2. Un coup d’œil sur le tableau 4-3 nous informe qu’il s’agit du bit Z (pour 0). Logique, car le résultat de l’opération est 0.

Si à ce niveau, vous vous demandez pourquoi le bit C (carry) n'est pas positionné à 1, bravo, car vous vous posez les bonnes questions, puisque le résultat de la dernière incrémentation est B'00000000' et on reporte B'1'. Mais alors, chapeau pour ceux qui ont pensé à vérifier le fonctionnement de l'instruction « incf » page 62 du datasheet. En effet, vous constaterez en vérifiant que le seul bit affecté (modifié) par cette instruction est le bit Z, à l'exclusion du bit C : « Status affected : Z ».

Nous allons clôturer par les autres méthodes d'exécution du programme par le simulateur. Pressez <F6> pour ramener le PC à 0. Dans le menu « Debugger->run » vous trouvez toutes les méthodes possibles : essayez ceci :

- Pressez <F9> pour lancer rapidement le programme sans visualisation. Pressez <F5> pour l'arrêter là où il est arrivé.

- Le menu « debugger -> animate » vous donne une exécution animée, plus lente à l'exécution, mais qui vous permet de suivre l'exécution du programme.

- L'avancée en pas à pas avec <F8> vous permet d'effectuer la même chose qu'avec <F7>, à la différence qu'une sous-routine, que nous verrons plus tard, est exécutée en une seule fois, comme si c'était une seule instruction. La touche <F7> entrera dans la sous-routine pour exécuter les instructions une à une. Encore une autre méthode : Allez sur la ligne

```
goto boucle       ; boucler
```

de votre petit programme. Positionnez votre souris en début de ligne et cliquez avec le bouton de droite. Un menu apparaît. Vous pouvez mettre des points d'arrêt dans votre programme. Vous pouvez également demander à votre programme de démarrer jusqu'au point spécifié (run to cursor), ainsi que d'autres options. La fenêtre « trace » menu view->trace vous donne des informations supplémentaires sur la trace du programme. Le menu « debugger » vous permet d'autres configurations. Lisez l'aide concernant le simulateur pour des informations détaillées. Le sous-menu « debugger->stimulus controler » vous permet par exemple de placer des points d'arrêts conditionnels.

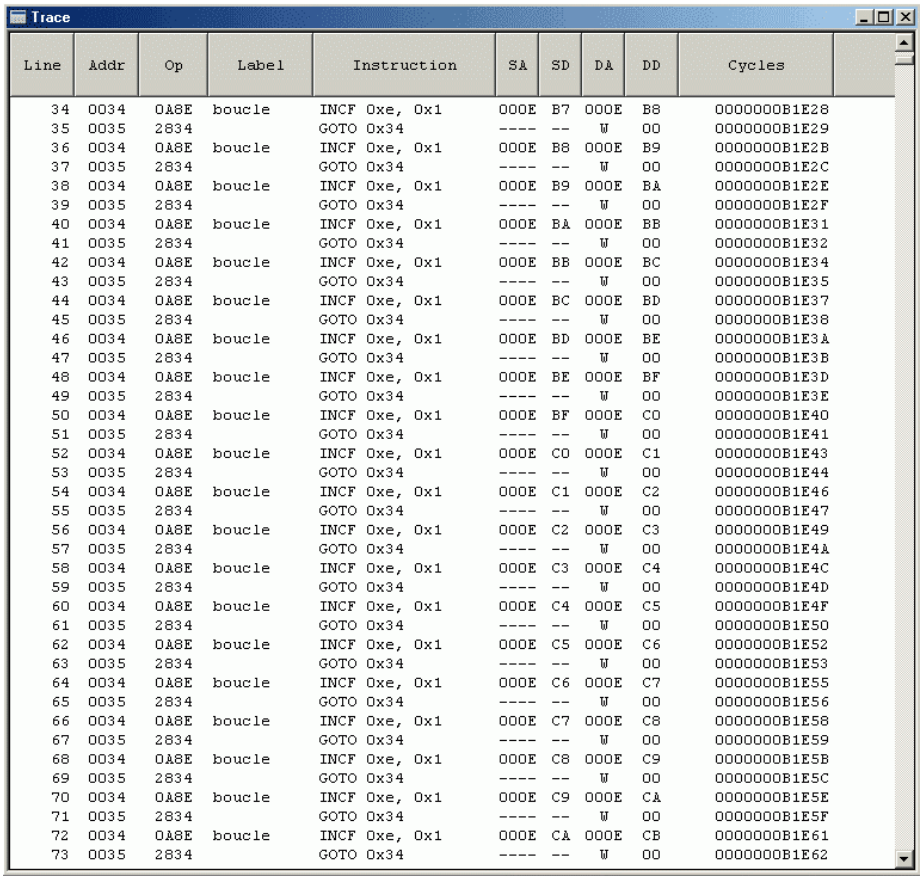
Un double-clic sur une ligne place également un point d'arrêt, que vous remarquerez grâce au symbole « B » qui apparaît à gauche de la ligne. Un autre double-clic l'annule.

Placez un point d'arrêt dans votre programme. Pressez <F6>, puis <F9>, le programme va s'exécuter jusque la ligne rouge, puis passer en stop. Voilà encore une nouvelle méthode pratique pour debugger les longs programmes.

Il existe cependant une nouvelle procédure encore plus rapide : la trace de votre programme.

Enlevez le point d'arrêt, pressez <F6>, puis sélectionnez : « view -> simulator trace ». Une nouvelle fenêtre s'ouvre, avec un message « no items to display ». Pressez maintenant <F9> puis quelques fractions de seconde plus tard, <F5>.

Dans la fenêtre «trace », vous obtenez le compte-rendu de toutes les instructions exécutées par votre programme depuis le lancement jusque l'arrêt. Ceci se révèle très pratique en cas de plantage inexpliqué par exemple.



Trace

Line	Addr	Op	Label	Instruction	SA	SD	DA	DD	Cycles
34	0034	0A8E	boucle	INCF 0xe, 0x1	000E	B7	000E	B8	0000000B1E28
35	0035	2834		GOTO 0x34	----	--	W	00	0000000B1E29
36	0034	0A8E	boucle	INCF 0xe, 0x1	000E	B8	000E	B9	0000000B1E2B
37	0035	2834		GOTO 0x34	----	--	W	00	0000000B1E2C
38	0034	0A8E	boucle	INCF 0xe, 0x1	000E	B9	000E	BA	0000000B1E2E
39	0035	2834		GOTO 0x34	----	--	W	00	0000000B1E2F
40	0034	0A8E	boucle	INCF 0xe, 0x1	000E	BA	000E	BB	0000000B1E31
41	0035	2834		GOTO 0x34	----	--	W	00	0000000B1E32
42	0034	0A8E	boucle	INCF 0xe, 0x1	000E	BB	000E	BC	0000000B1E34
43	0035	2834		GOTO 0x34	----	--	W	00	0000000B1E35
44	0034	0A8E	boucle	INCF 0xe, 0x1	000E	BC	000E	BD	0000000B1E37
45	0035	2834		GOTO 0x34	----	--	W	00	0000000B1E38
46	0034	0A8E	boucle	INCF 0xe, 0x1	000E	BD	000E	BE	0000000B1E3A
47	0035	2834		GOTO 0x34	----	--	W	00	0000000B1E3B
48	0034	0A8E	boucle	INCF 0xe, 0x1	000E	BE	000E	BF	0000000B1E3D
49	0035	2834		GOTO 0x34	----	--	W	00	0000000B1E3E
50	0034	0A8E	boucle	INCF 0xe, 0x1	000E	BF	000E	C0	0000000B1E40
51	0035	2834		GOTO 0x34	----	--	W	00	0000000B1E41
52	0034	0A8E	boucle	INCF 0xe, 0x1	000E	C0	000E	C1	0000000B1E43
53	0035	2834		GOTO 0x34	----	--	W	00	0000000B1E44
54	0034	0A8E	boucle	INCF 0xe, 0x1	000E	C1	000E	C2	0000000B1E46
55	0035	2834		GOTO 0x34	----	--	W	00	0000000B1E47
56	0034	0A8E	boucle	INCF 0xe, 0x1	000E	C2	000E	C3	0000000B1E49
57	0035	2834		GOTO 0x34	----	--	W	00	0000000B1E4A
58	0034	0A8E	boucle	INCF 0xe, 0x1	000E	C3	000E	C4	0000000B1E4C
59	0035	2834		GOTO 0x34	----	--	W	00	0000000B1E4D
60	0034	0A8E	boucle	INCF 0xe, 0x1	000E	C4	000E	C5	0000000B1E4F
61	0035	2834		GOTO 0x34	----	--	W	00	0000000B1E50
62	0034	0A8E	boucle	INCF 0xe, 0x1	000E	C5	000E	C6	0000000B1E52
63	0035	2834		GOTO 0x34	----	--	W	00	0000000B1E53
64	0034	0A8E	boucle	INCF 0xe, 0x1	000E	C6	000E	C7	0000000B1E55
65	0035	2834		GOTO 0x34	----	--	W	00	0000000B1E56
66	0034	0A8E	boucle	INCF 0xe, 0x1	000E	C7	000E	C8	0000000B1E58
67	0035	2834		GOTO 0x34	----	--	W	00	0000000B1E59
68	0034	0A8E	boucle	INCF 0xe, 0x1	000E	C8	000E	C9	0000000B1E5B
69	0035	2834		GOTO 0x34	----	--	W	00	0000000B1E5C
70	0034	0A8E	boucle	INCF 0xe, 0x1	000E	C9	000E	CA	0000000B1E5E
71	0035	2834		GOTO 0x34	----	--	W	00	0000000B1E5F
72	0034	0A8E	boucle	INCF 0xe, 0x1	000E	CA	000E	CB	0000000B1E61
73	0035	2834		GOTO 0x34	----	--	W	00	0000000B1E62

Il y a encore plein d'options concernant le debugger, je ne peux cependant pas toutes les expliquer dans le cadre de cet ouvrage. Vous les découvrirez petit à petit lors de vos expérimentations, n'hésitez pas à examiner tous les menus.

A l'heure actuelle, il y a cependant encore moins d'options que sur la version 5 de MPLAB®. Cependant, je pense que c'est dû à la jeunesse du produit. Microchip® a déjà ajouté des fonctions qui existaient sur la version 5, et qui n'existaient pas sur la version 6.0. Pour les anciens de MPLAB® 5, il faudra encore un peu de patience pour tout récupérer.

Notes :…

9. Le jeu d'instructions

Les instructions sont détaillées à partir de la page 57 du datasheet. Il peut donc sembler inutile de les reprendre dans ce cours. Mais j'ai finalement opté pour une explication pratique de ces instructions. J'ai choisi en effet de les expliquer avec des petits exemples concrets, plutôt que de simplement traduire le datasheet.

Bien entendu, vous pouvez expérimenter ces instructions avec MPLAB® et son simulateur, en insérant ces instructions après l'étiquette « start » de votre programme.

Je vais donc expliquer ces instructions dans un ordre qui me facilite les explications, en commençant par celles déjà vues, dans le but final d'obtenir toutes les références aux instructions dans le même document.

9.1 L'instruction « GOTO » (aller à)

Cette instruction effectue ce qu'on appelle un saut inconditionnel, encore appelé rupture de séquence synchrone inconditionnelle.

Rappelez-vous que l'instruction goto contient les 11 bits de l'emplacement de destination. Les 2 bits restants sont chargés depuis le registre PCLATH. Rappelez-vous en effet la manière dont est construit le PC dans le cas des sauts. On ne peut sauter qu'à l'intérieur d'une même PAGE de 2^{11}, soit 2048 Kmots.

Ceci n'a aucune espèce d'importance pour le 16F84, qui ne dispose que de 1Kmots de mémoire programme, mais devra être considéré pour les PIC® de plus grande capacité (16F876). Nous en reparlerons dans la seconde partie.

Voici donc en résumé le fonctionnement du goto :

- L'adresse de saut sur 11 bits est chargée dans le PC.
- Les 2 bits manquants sont chargés depuis PCLATH (b3 et b4), pas pour le 16F84.
- Le résultat donne l'adresse sur 13 bits (10 bits pour le 16F84)
- La suite du programme s'effectue à la nouvelle adresse du PC.

Souvenez-vous, que pour le 16F84 : Adresse de saut = adresse réelle. Vous ne devez donc vous préoccuper de rien. Pour les autres, en cas de débordement, MPLAB® vous le signalera.

Syntaxe

```
   goto     etiquette
```

Exemple

```
Start
   goto plusloin   ; le programme saute à l'instruction qui suit l'étiquette
                   ;  « plusloin »
      xxxxxxxx
plusloin
```

```
xxxxxxxx ; instruction exécutée après le saut : le programme se poursuit ici
```

Remarquez que vous pouvez sauter en avant ou en arrière.
goto nécessite 2 cycles d'horloge, comme pour tous les sauts

9.2 L'instruction « INCF » (INCrement File)

Cette instruction provoque l'incrémentation de l'emplacement spécifié (encore appelé File).

Syntaxe

```
incf  f,d
```

Comme pour toutes les instructions, « f » représente « File », c'est à dire l'emplacement mémoire concerné pour cette opération, « d », quant à lui: représente la Destination.

Sauf spécification contraire, d vaut toujours, au choix :

- f (la lettre f) : dans ce cas le résultat est stocké dans l'emplacement mémoire.
- W (la lettre w) : dans ce cas, le résultat est laissé dans le registre de travail, et le contenu de l'emplacement mémoire n'est pas modifié. Ce registre de travail est appelé également « accumulateur ».

La formule est donc (f) + 1 -> (d) : Les parenthèses signifient « le contenu de ». Soit, en français : Le contenu de l'emplacement spécifié est incrémenté de 1, le résultat est placé dans l'emplacement désigné par « d ». Emplacement qui pourra être soit l'emplacement spécifié par « f », soit l'accumulateur, (f) restant dans ce cas inchangé.

Bit du registre STATUS affecté

Le seul bit affecté par cette opération est le bit Z.

Etant donné que la seule manière, en incrémentant un octet, d'obtenir 0, est de passer de 0xFF à 0x00. Le report n'est pas nécessaire, puisqu'il va de soi. Si, après une incrémentation, vous obtenez Z=1 , c'est que vous avez débordé. Z vaudra donc 1 si (f) avant l'exécution valait 0xFF.

Exemples

```
incf mavariable , f      ; le contenu de ma variable est augmenté de 1
                         ; le résultat est stocké dans mavariable.
incf mavariable , w      ; Le contenu de mavariable est chargé dans w et
                         ; augmenté de 1. W contient donc le contenu de
                         ; mavariable + 1. mavariable n'est pas modifiée
```

9.3 L'instruction « DECF » (DECRement File)

Décrémente l'emplacement spécifié. Le fonctionnement est strictement identique à l'instruction précédente.

Syntaxe

```
decf  f , d           ; (f) – 1 -> (d)
```

Bit du registre STATUS affecté

Le seul bit affecté par cette opération est le bit Z.
Si avant l'instruction, (f) vaut 1, Z vaudra 1 après l'exécution (1-1 = 0)

Exemples

```
decf  mavariable , f  ; décrémente mavariable, résultat dans mavariable
decf  mavariable , w  ; prends (mavariable) – 1 et place le résultat dans w
```

9.4 L'instruction « MOVLW » (MOVe Literal to W)

Cette instruction charge la valeur spécifiée (valeur littérale, ou encore valeur immédiate), dans le registre de travail W.

Syntaxe

```
movlw k         ; k-> w : k représente une valeur de 0x00 à 0xFF.
```

Bit du registre STATUS affecté

Aucun (donc même si vous chargez la valeur '0'.)

Exemple

```
movlw 0x25            ; charge 0x25 dans le registre w
```

9.5 L'instruction « MOVF » (MOVe File)

Charge le contenu de l'emplacement spécifié dans la destination

Syntaxe

```
movf f , d            ; (f) -> (d)
```

Bit du registre STATUS affecté

Une fois de plus, seul le bit Z est affecté (si f vaut 0, Z vaut 1).

Exemple 1

Pour cette instruction, je vais me montrer beaucoup plus explicite. Vous allez comprendre pourquoi

```
movf mavariable,w           ; charge le contenu de mavariable dans w.
```

ATTENTION

Il est impératif ici de bien faire la distinction entre movlw k et movf f,w.

Dans le premier cas, c'est la VALEUR qui est chargée dans w, dans le second c'est le CONTENU de l'emplacement spécifié. Si nous nous rappelons, dans la leçon précédente que mavariable représentait l'adresse 0x0E de notre zone de RAM. Supposons maintenant que mavariable contienne 0x50.

Si nous exécutons l'instruction suivante :

```
movlw  mavariable
```

L'assembleur va traduire en remplaçant mavariable par sa VALEUR. Attention, la valeur, ce n'est pas le contenu. L'assembleur va donc réaliser la substitution suivante :

```
movlw  0x0E
```

Après l'instruction, nous aurons 0x0E dans le registre W. Nous parlerons dans ce cas d'un ADRESSAGE IMMEDIAT. Le déroulement est du type f -> (d) (f n'est pas entre parenthèses).

Si nous exécutons par contre l'instruction suivante :

```
movf mavariable , w
```

L'assembleur va traduire également en remplaçant mavariable par sa valeur. Nous obtenons donc :

```
movf 0x0E , w
```

Ce qui signifie : charger le CONTENU de l'emplacement 0x0E dans w. Nous parlerons ici d'un ADRESSAGE DIRECT. Je rappelle qu'il est primordial de ne pas confondre, sinon, vous ne comprendrez plus rien lors de la réalisation des programmes.
Après cette instruction, W contient donc 0x50. Le déroulement est du type (f) -> (d).

Exemple 2

```
movf     mavariable , f
```

Que fait cette instruction ? Si vous avez tout suivi, elle place le CONTENU de mavariable dans mavariable. Dire que cela ne sert à rien est tentant mais prématuré. En effet, si le contenu de mavariable reste bien inchangé, par contre le bit Z est positionné à 1. Cette instruction permet donc de vérifier si (mavariable) = 0.

9.6 L'instruction « MOVWF » (MOVe W to File)

Permet de sauvegarder le contenu du registre de travail W dans un emplacement mémoire.

Syntaxe

```
movwf     f                    ; (W) -> (f)
```

Bit du registre STATUS affecté

Aucun

Exemple

```
movlw     0x50                 ; charge 0x50 dans W
movwf     mavariable           ; « mavariable » contient maintenant 0x50.
```

9.7 L'instruction « ADDLW » (ADD Literal and W)

Cette opération permet d'ajouter une valeur littérale (adressage immédiat) au contenu du registre de travail W.

Syntaxe

```
addlw     k                    ; (W) + k -> (W)
```

Bits du registre STATUS affectés

Z Si le résultat de l'opération vaut 0, Z vaudra 1
C Si le résultat de l'opération est supérieur à 0xFF (255) , C vaudra 1
DC Si le résultat de l'opération entraîne en report du bit 3 vers le bit 4, DC vaudra 1

Ne vous inquiétez pas trop pour DC, il n'est utilisé que pour les opérations sur les quartets, par exemple, les nombres Binary Coded Decimal.

Exemple

```
movlw     253          ; charger 253 en décimal dans W
addlw     4            ; Ajouter 4. W contient 1, Z vaut 0, C vaut 1(déborde)
addlw     255          ; ajouter 255 W vaut 0, C vaut 1, Z vaut 1
```

9.8 L'instruction « ADDWF » (ADD W and F)

Ne pas confondre avec l'instruction précédente. Une nouvelle fois, il s'agit ici d'un ADRESSAGE DIRECT. Le CONTENU du registre W est ajouté au CONTENU du registre F

Syntaxe

```
addwf   f , d                ; (w) + (f) -> (d)
```

Bits du registre STATUS affectés

C, DC, et Z

Exemple

```
movlw 12              ; charger 12 dans W
movwf mavariable      ; mavariable vaut maintenant 12
movlw 25              ; charger 25 dans W
addwf  mavariable,f   ; résultat : (W) + (mavariable), donc 25+12
                      ; résultat = 37 sauvé dans mavariable (,f).
```

9.9 L'instruction « SUBLW » (SUBtract W from Literal)

Attention, ici il y a un piège. L'instruction aurait du s'appeler SUBWL. En effet, on soustrait W de la valeur littérale, et pas l'inverse.

Syntaxe

```
sublw     k           ; k – (W) -> (W)
```

Bits du registre STATUS affectés

C, DC, Z

Notez ici que le bit C fonctionne de manière inverse que pour l'addition. Ceci est commun à la plupart des microprocesseurs du marché. Les autres utilisent parfois un bit spécifique pour la soustraction, bit le plus souvent appelé « borrow » (emprunt).

Si le résultat est positif, donc, pas de débordement : C =1. S' il y a débordement, C est forcé à 0.

Ceci est logique, et s'explique en faisant une soustraction manuelle. Le bit C représente le 9ème bit ajouté d'office à la valeur initiale. Si on effectue une soustraction manuelle donnant une valeur <0, on obtient donc une valeur finale sur 8 bits, le report obtenu venant soustraire le bit C. Si le résultat est >0, il n'y a pas de report, le résultat final reste donc sur 9 bits.

La formule de la soustraction est donc : k précédé d'un neuvième bit à 1 – contenu de W = résultat sur 8 bits dans W avec 9ème bit dans C.

Exemple 1

```
movlw     0x01        ; charger 0x01 dans W
sublw     0x02        ; soustraire W de 2
                      ; résultat : 2 – (W) = 2-1 = 1
                      ; Z = 0, C = 1, donc résultat positif
```

Effectuons cette opération manuellement :

	C	b7	b6	b5	b4	b3	b2	b1	b0	Dec
	1	0	0	0	0	0	0	1	0^{10}	2
-	0	0	0	0	0	0	0	0^{1}	1	1
=	1	0	0	0	0	0	0	0	1	1

Comment procéder ? Et bien, comme pour une soustraction décimale. On commence par les bits de droite : 0-1, ça ne passe pas, donc on emprunte 10 (noté en exposant vert), et on soustraira évidemment une unité supplémentaire au bit b1. On a donc :B '10' – B'1', car souvenez-vous qu'on a emprunté 10 en BINAIRE. Résultat 1. On soustrait ensuite les b1 : on aura 1 – 0 – l'emprunt, donc 1-0-1 = 0. On continue de droite à gauche jusqu'au 9ème bit qui est le carry. Résultat final : B'00000001' et carry à 1, C.Q.F.D.

Exemple 2

```
movlw     0x02        ; charger 0x02 dans W
sublw     0x02        ; soustraire  2 – (w) = 2 –2 = 0
                      ; Z = 1 , C = 1 : résultat nul
```

	C	b7	b6	b5	b4	b3	b2	b1	b0	Dec
	1	0	0	0	0	0	0	1	0	2
-	0	0	0	0	0	0	0	1	0	2
=	1	0	0	0	0	0	0	0	0	0

On procède toujours de la même manière.

Exemple 3

```
movlw     0x03        ; charger 0x03 dans W
sublw     0x02        ; soustraire 2 – (W) = 2 – 3 = -1
                      ; Z = 0, C = 0, résultat négatif
```

	C	b7	b6	b5	b4	b3	b2	b1	b0	Dec
	1	0^{10}	0^{10}	0^{10}	0^{10}	0^{10}	0^{10}	1^{10}	0^{10}	2
-	0^{1}	0^{1}	0^{1}	0^{1}	0^{1}	0^{1}	0^{1}	1^{1}	1	3
=	0	1	1	1	1	1	1	1	1	-1

Procédons de la même manière, et nous obtenons B'11111111', avec le bit C à 0. Et là, me dites-vous, B'11111111', c'est FF, pas –1. Et bien, rappelez-vous ceci : Vous DEVEZ lire le bit C pour interpréter le résultat de votre soustraction.

Comme ce dernier vaut 0, vous êtes AVERTI que le résultat de l'opération est négatif. Or, comment savoir la valeur absolue d'un nombre négatif ? En prenant son complément à 2. Rappelez-vous :

Complément à 1 de B'11111111' = B'00000000' (on inverse tous les bits)

Complément à 2 = complément à 1 + 1, donc B'00000001'.

Donc, le complément à 2 de 0xFF vaut –0x01. C.Q.F.D.

La preuve, si vous ajoutez 1 à –1, vous obtenez B’11111111’ + B’00000001’ = B’00000000’ = 0.

Vous maîtrisez maintenant les soustractions. Certains auront sans doute pensé que j’expliquais trop en détail, mais mon expérience m’a appris que les soustractions représentaient souvent un écueil dans la réalisation de trop de programmes.

Encore un dernier détail : Pour effectuer une soustraction de 1, vous pouvez bien entendu effectuer une addition de –1. Le résultat sera strictement le même, à votre charge d’interpréter les bits Z et C. Je vous laisse le faire vous-même pour vous convaincre.

9.10 L’instruction « SUBWF » (SUBtract W from F)

Nous restons dans les soustractions, mais, cette fois, au lieu d’un adressage immédiat, nous avons un ADRESSAGE DIRECT.

Syntaxe

```
subwf     f , d        ; (f) – (W) -> (d)
```

Bits du registre STATUS affectés

C , DC , Z

Exemple

```
movlw     0x20             ; charger 0x20 dans w
movwf     mavariable       ; mettre w dans (mavariable) (0x20)
movlw     0x1F             ; charger 0x1F dans w
subwf     mavariable,w     ; (mavariable) -  (w) -> (w)
                           ; 0x20 – 0x1F = 0x01
                           ; résultat dans w, C=1, Z=0
movwf     autrevariable    ; sauver 0x01 dans une autre variable
```

9.11 L’instruction « ANDLW » (AND Literal with W)

Cette instruction effectue une opération « AND » BIT A BIT entre le contenu de W et la valeur littérale qui suit.

Syntaxe

```
andlw     k            ; avec k = octet : (w) AND k -> (w)
```

Bit du registre STATUS affecté

Z

Exemple

```
movlw      B’11001101’ ; charger w
andlw      B’11110000’ ; effectuer un ‘and’ (&)
```

	b7	b6	b5	b4	b3	b2	b1	b0
	1	1	0	0	1	1	0	1
And	1	1	1	1	0	0	0	0
=	1	1	0	0	0	0	0	0

Rappelez-vous qu’on effectue un AND entre chaque bit de même rang. Seuls restent donc positionnés à 1 les bits dont les 2 opérandes valent 1. Donc, le fait d’effectuer un AND avec la valeur B’11110000’ MASQUE les bits 0 à 3, et ne laisse subsister que les bits 4 à 7.
Tant que vous ne jonglerez pas avec l’hexadécimal, je vous conseille de toujours traduire les nombres en binaires pour toutes les instructions concernant les bits.

9.12 L’instruction « ANDWF » (AND W with F)

Maintenant, vous devriez avoir bien compris. De nouveau la même opération, mais en ADRESSAGE DIRECT. Je vais donc accélérer les explications.

Syntaxe

```
andwf      f , d       ; (f) AND (w) -> (d)
```

Bit du registre STATUS affecté

Z

Exemple

```
movlw 0xC8            ; charger 0XC8 dans w
movwf mavariable      ; sauver dans mavariable
movlw 0xF0            ; charger le masque
andwf mavariable,f    ; (mavariable) = 0xC0 (on a éliminé le quartet faible)
```

9.13 L’instruction « IORLW » (Inclusive OR Literal with W)

Et oui, les mêmes instructions, mais pour le OU inclusif. Inclusif signifie simplement le contraire d’exclusif, c’est à dire que le bit de résultat vaudra 1 si un des bits, OU LES DEUX BITS, opérandes =1.

Syntaxe

```
iorlw     k           ; (w) OR k -> (w)
```

Bit du registre STATUS affecté

Z

Exemple

```
movlw     0xC3        ; charger 0xC3 dans W
iorlw     0x0F        ; FORCER les bits 0 à 3
                      ; résultat ; (w) = 0xCF
```

	b7	b6	b5	b4	b3	b2	b1	b0
	1	1	0	0	0	0	1	1
OR	0	0	0	0	1	1	1	1
=	1	1	0	0	1	1	1	1

Donc, avec un ou inclusif (OR), on peut FORCER n'importe quel bit à 1 (pour rappel, avec AND, on peut forcer n'importe quel bit à 0).

9.14 L'instruction « IORWF » (Inclusive OR W with File)

Effectue un OR entre (w) et l'emplacement spécifié. C'est donc une instruction en ADRESSAGE DIRECT. Je ne donnerai pas d'exemple, vous devriez avoir compris.

Syntaxe

```
iorwf     f , d       ; (w) OR (f) -> (d)
```

Bit du registre STATUS affecté

Z

9.15 L'instruction « XORLW » (eXclusive OR Literal with W)

Par opposition au ou inclusif, voici maintenant le OU EXCLUSIF. Sa table de vérité est la même que le ou inclusif, excepté que lorsque les 2 bits sont à 1, le résultat est 0.

Cette instruction peut donc servir à INVERSER n'importe quel bit d'un octet. En effet, si vous effectuez « 1 xor 0 », vous obtenez 1, si vous effectuez « 0 xor 0 », vous obtenez 0.
Donc, si vous appliquez xor 0, la valeur de départ est inchangée.

Si par contre vous appliquez « 0 xor 1 » , vous obtenez 1, et avec « 1 xor 1 », vous obtenez 0. En appliquant xor 1, vous inversez le bit, quelque soit son état initial.

Maintenant, vous pouvez donc FORCER un bit à 1 avec OR, MASQUER un bit (le mettre à 0) avec AND, et l'INVERSER avec XOR.

Syntaxe

```
xorlw     k           ; (w) xor k -> (w)
```

Bit du registre STATUS affecté

Z

Exemple

```
movlw     B'11000101'       ; charger W
xorlw     B'00001111'       ; xor avec la valeur
                            ; résultat : B '11001010'
                            ; les 4 bits de poids faible ont été inversés
```

	b7	b6	B5	B4	b3	b2	b1	b0
	1	1	0	0	0	1	0	1
Xor	0	0	0	0	1	1	1	1
=	1	1	0	0	1	0	1	0

Remarquez que tous les bits de l'octet initial ont été inversés par chaque bit du second opérande qui était à 1.

9.16 L'instruction « XORWF » (eXclusive OR W with F)

C'est exactement la même opération que XORLW, mais en ADRESSAGE DIRECT.

Syntaxe

```
xorwf f , d           ; (w) xor (f) -> (d)
```

Bit du registre STATUS affecté

Z

9.17 L'instruction « BSF » (Bit Set F)

C'est une instruction qui permet tout simplement de forcer directement un bit d'un emplacement mémoire à 1.

Syntaxe

```
bsf f , b       ; le bit n° b est positionné dans la case mémoire (f)
                ; b est évidemment compris entre 0 et 7
```

Bit du registre STATUS affecté

Aucun

Exemples

```
bsf STATUS , C        ; positionne le bit C à 1 dans le registre STATUS
bsf mavariable , 2    ; positionne bit 2 de (mavariable) à 1
```

9.18 L'instruction « BCF » (Bit Clear F)

C'est une instruction qui permet tout simplement de forcer directement un bit d'un emplacement mémoire à 0.

Syntaxe

```
bcf f , b            ; le bit n° b est mis à 0 dans la case mémoire (f)
                     ; b est évidemment compris entre 0 et 7
```

Bit du registre STATUS affecté

Aucun

Exemples

```
bcf STATUS , C       ; positionne le bit C à 0 dans le registre STATUS
bcf mavariable , 2   ; positionne b2 de (mavariable) à 0
```

9.19 L'instruction « RLF » (Rotate Left through Carry)

Rotation vers la gauche en utilisant le carry.

Les opérations de décalage sont des opérations très souvent utilisées. Les PIC® ont la particularité de ne disposer que d'instructions de ROTATION. Vous allez voir qu'avec ces instructions, on peut très facilement réaliser des décalages.

L'opération de rotation effectue l'opération suivante : Le bit de carry C est mémorisé. Ensuite chaque bit de l'octet est déplacé vers la gauche. L'ancien bit 7 sort de l'octet par la gauche, et devient le nouveau carry. Le nouveau bit 0 devient l'ancien carry. Il s'agit donc d'une rotation sur 9 bits.

Syntaxe

```
rlf f , d       ; (f) rotation gauche avec carry-> (d)
```

Bit du registre STATUS affecté

C

Exemple1

Un petit exemple vaut mieux qu'un long discours.

```
bsf      STATUS,C          ; positionne le carry à 1
movlw    B'00010111'       ; charge la valeur dans w
movwf    mavariable        ; initialise mavariable
rlf      mavariable,f      ; rotation vers la gauche
```

	C	b7	b6	b5	b4	b3	b2	b1	b0
F	1	0	0	0	1	0	1	1	1
Rlf	0	0	0	1	0	1	1	1	1

Vous voyez que tous les bits ont été décalés vers la gauche. « C » a été réintroduit dans b0. Le résultat reste sur 9 bits.

Exemple 2

```
bcf       STATUS,C          ; positionne le carry à 0
movlw     b'00010111'       ; charge la valeur dans w
movwf     mavariable        ; initialise mavariable
rlf       mavariable,f      ; rotation vers la gauche
```

Si vous avez compris, le résultat sera B'00101110', avec le carry à 0. Si le carry était à 0 au départ, on effectue un simple décalage vers la gauche. Que se passe-t-il si, en décimal, on effectue ce type d'opération ?

Prenons le nombre 125 et décalons-le vers la gauche en décimal, nous obtenons 1250. Nous avons multiplié le nombre par sa base (décimal = base 10). Et bien c'est la même chose en binaire (une fois de plus).

Prenons B'00010111 ', soit 23 en décimal. Décalons-le, nous obtenons B'00101110', soit 46. Nous avons donc effectué une multiplication par 2. Retenez ceci, cela vous sera très utile par la suite.

9.20 L'instruction « RRF » (Rotate Right through Carry)

Rotation vers la droite en utilisant le carry

L'opération de rotation vers la droite effectue l'opération suivante : Le bit de carry « C » est mémorisé. Ensuite chaque bit de l'octet est déplacé vers la droite. L'ancien bit 0 sort de l'octet par la droite, et devient le nouveau carry. L'ancien carry devient le nouveau bit7. Il s'agit donc également d'une rotation sur 9 bits.

Syntaxe

```
rrf       f , d       ; (f) rotation droite avec carry-> (d)
```

Bit du registre STATUS affecté

C

Exemple1

```
bsf       STATUS,C          ; positionne le carry à 1
movlw     B'00010111'       ; charge la valeur dans w
movwf     mavariable        ; initialise mavariable
rrf       mavariable,f      ; rotation vers la droite
```

	b7	b6	b5	B4	b3	b2	b1	b0	C
F	0	0	0	1	0	1	1	1	1
Rrf	1	0	0	0	1	0	1	1	1

Vous voyez que tous les bits ont été décalés vers la droite. C a été réintroduit dans b7. Le résultat reste sur 9 bits.

Exemple 2

```
bcf       STATUS,C          ; positionne le carry à 0
movlw     b'00010111'       ; charge la valeur dans w
movwf     mavariable        ; initialise mavariable
rrf       mavariable,f      ; rotation vers la droite
```

Si vous avez compris, le résultat sera B'00001011', avec le carry à 1. Si le carry est à 0 au départ, on effectue un simple décalage vers la droite.

Que s'est-il passé ? Et bien notre nombre de départ, soit 23 en décimal est devenu 11. Le carry représente le bit « -1 », donc, la moitié du bit 0, donc ½. En effet, en décimal, le chiffre « 1 » derrière les unités a comme valeur 1/base, donc 1/10. En binaire ce sera donc ½.

Si nous regardons alors les 9 bits, nous obtenons 11 ½ . Nous avons donc effectué une DIVISION PAR 2. Retenez ceci, cela vous sera également très utile par la suite.

9.21 L'instruction « BTFSC » (Bit Test F, Skip if Clear)

Traduit littéralement, cela donne : Teste le bit de l'emplacement mémoire et saute s'il vaut 0. Il s'agit ici de votre premier SAUT CONDITIONNEL, ou RUPTURE DE SEQUENCE SYNCHRONE CONDITIONNELLE .

En effet, il n'y aura saut que si la condition est remplie.

Notez que dans ce cas l'instruction prendra 2 cycles, sinon, elle n'utilisera qu'un cycle. De plus, il faut retenir que pour tous ces types de saut, ON NE SAUTE QUE L'INSTRUCTION SUIVANTE. En effet, la syntaxe ne contient pas d'adresse de saut, comme nous allons le voir

Syntaxe

```
btfsc     f, b   ; on teste le bit b de la mémoire (f).
                 ; si ce bit vaut 0, on saute l'instruction suivante, sinon
                 ; on exécute l'instruction suivante.

   Instruction exécutée si faux      ; si le bit vaut 0, ne sera pas exécutée
(skip)
   Poursuite du programme            ; le programme continue ici
```

Bit du registre STATUS affecté

Aucun

Exemple1

Voici un exemple dans lequel on doit exécuter une seule instruction supplémentaire si le bit vaut 1.

```
btfsc     STATUS,C          ; tester si le bit C du registre STATUS vaut 0
bsf       mavariable,2      ; non (C=1), alors bit 2 de mavariable mis à 1
xxxx                        ; la suite du programme est ici dans les 2 cas
```

Exemple 2

Que faire si les traitements nécessitent plusieurs instructions ? Et bien, on combine les sauts conditionnels avec les saut inconditionnels (par exemple goto).

```
movlw     0x12              ; charger 12 dans le registre de travail
subwf     mavariable,f      ; on soustrait 0x12 de mavariable
btfsc     STATUS,C          ; on teste si le résultat est négatif (C=0)
goto      positif           ; non, alors au saute au traitement des positifs
xxxx                        ; on poursuit ici si le résultat est négatif
```

Ces procédures sont les mêmes pour tous les sauts inconditionnels, je ne les détaillerai donc pas avec autant d'explications.

9.22 L'instruction « BTFSS » (Bit Test F, Skip if Set)

Traduit littéralement, cela donne : Teste le bit de l'emplacement mémoire et saute s'il vaut 1. Toutes les remarques de l'instruction « BTFSC » restent valables.

Syntaxe

```
btfss     f, b              ; on teste le bit b de la mémoire (f).
                            ; si ce bit vaut 1, on saute l'instruction
                            ; suivante, sinon
                            ; on exécute l'instruction suivante.
xxxx                        ; si le bit vaut 1, ne sera pas exécutée (skip)
xxxx                        ; Le programme continue ici
```

Bit du registre STATUS affecté

Aucun

Exemple

```
btfss     STATUS,C          ; tester si le bit C du registre STATUS vaut 1
bsf       mavariable,2      ; non (C=0), alors bit 2 de mavariable mis à 1
xxxx                        ; la suite du programme est ici dans les 2 cas
```

9.23 L'instruction « DECFSZ » (DECrement F, Skip if Z)

Nous poursuivons les sauts conditionnels avec une instruction très utilisée pour créer des boucles. Cette instruction décrémente un emplacement mémoire et saute l'instruction suivante si le résultat de la décrémentation donne une valeur nulle.

Syntaxe

```
decfsz    f, d          ; (f) –1 -> (d). Saut si (d) = 0
```

Bit du registre STATUS affecté

Aucun

Exemple1

```
        movlw  3                  ; charger 3 dans w
        movwf  compteur           ; initialiser compteur
        movlw  0x5                ; charger 5 dans w
boucle                            ; étiquette
        addwf  mavariable , f     ; ajouter 5 à ma variable
        decfsz compteur , f       ; décrémenter compteur et tester sa valeur
        goto   boucle                    ; si compteur pas 0, on boucle
        movf   mavariable , w     ; on charge la valeur obtenue dans w
```

Comment écrit-on ce type de programme ? Et bien tout simplement de la manière suivante :

- on initialise le compteur de boucles.
- on place une étiquette de début de boucle
- on écrit les instructions qui doivent s'exécuter plusieurs fois
- l'instruction decfsz permet de déterminer la fin de la boucle
- l'instruction goto permet de localiser le début de la boucle.

ATTENTION

- Si vous aviez mis

```
decfsz compteur , w          ; décrémenter compteur et tester sa valeur
```

la boucle n'aurait jamais de fin, car la variable compteur ne serait jamais modifiée.

- Si vous placez 0 dans le compteur de boucles, elle sera exécutée 256 fois. Si vous ne désirez pas qu'elle soit exécutée dans cette circonstance, vous devez ajouter un test AVANT l'exécution de la première boucle, comme dans l'exemple suivant :

Exemple 2

```
        movf   compteur,f         ; permet de positionner Z
        btfsc  STATUS , Z         ; sauter si Z = 0, donc si compteur >0
        goto   suite              ; compteur = 0, ne pas traiter la boucle
boucle                            ; étiquette de début de boucle
        addwf  mavariable , f     ; ajouter 5 à ma variable
```

```
      decfsz compteur , f      ; décrémenter compteur et tester sa valeur
      goto   boucle            ; si compteur pas 0, on boucle
suite                          ; on saute directement ici si compteur = 0
      movf   mavariable , w    ; on charge la valeur obtenue dans w
```

9.24 L'instruction « INCFSZ » (INCrement F, Skip if Zero)

Je ne vais pas détailler cette instruction, car elle est strictement identique à la précédente, hormis le fait qu'on incrémente la variable au lieu de la décrémenter.

Syntaxe

```
incfsz   f , d       ; (f) + 1 -> (d) : saut si (d) = 0
```

Bit du registre STATUS affecté

Aucun

9.25 L'instruction « SWAPF » (SWAP nibbles in F)

Nous pouvons traduire cette instruction par «inverser les quartets dans F ». Cette opération inverse simplement le quartet (demi-octet) de poids faible avec celui de poids fort.

Syntaxe

```
swapf    f , d ; inversion des b0/b3 de (f) avec b4/b7 -> (d)
```

Bit du registre STATUS affecté

Aucun : cette particularité nous sera très utile lorsque nous verrons les interruptions.

Exemple

```
movlw  0xC5              ; charger 0xC5 dans w
movwf  mavariable        ; placer dans mavariable
swapf  mavariable , f    ; (mavariable) = 0x5C
```

9.26 L'instruction « CALL » (CALL subroutine)

Cette opération effectue un saut inconditionnel vers un sous-programme.
Voyons ce qu'est un sous-programme. Et bien, il s'agit tout simplement d'une partie de programme qui peut être appelé depuis plusieurs endroits du programme dit « principal ».

Le point de départ est mémorisé automatiquement, de sorte qu'après l'exécution du sous-programme, le programme continue depuis l'endroit où il était arrivé. Cela paraît un peu ardu, mais c'est extrêmement simple. Voyez les exemples dans la description de l'instruction RETURN.

Syntaxe

```
call     etiquette    ; appel de la sous-routine à l'adresse etiquette.
```

Mécanisme

Lors de l'exécution de l'instruction, l'adresse de l'instruction suivante est sauvegardée sur le sommet d'une pile (exactement comme une pile d'assiettes). Lorsque la sous-routine est terminée, l'adresse sauvegardée est retirée de la pile et placée dans le PC. Le programme poursuit alors depuis l'endroit d'où il était parti.

Notez que si le sous-programme (ou sous-routine) appelle lui-même un autre sous programme, l'adresse sera également sauvée au dessus de la pile.

Attention, cette pile a une taille limitée à 8 emplacements. Il n'existe aucun moyen de tester la pile, vous devez donc gérer vos sous-programmes pour ne pas dépasser 8 emplacements, sinon, votre programme se plantera.

Notez que lorsque vous sortez d'un sous-programme, l'emplacement est évidemment libéré. La limite n'est donc pas dans le nombre de fois que vous appelez votre sous-programme, mais dans le nombre d'imbrications (sous-programme qui en appelle un autre qui en appelle un autre) etc.

Bit du registre STATUS affecté

Aucun

9.27 L'instruction « RETURN » (RETURN from subroutine)

Retour de sous-routine. Va toujours de pair avec une instruction call. Cette instruction indique la fin de la portion de programme considérée comme sous-routine (SR). Rappelez-vous que pour chaque instruction « call » rencontrée, votre programme devra rencontrer une instruction « return ».

Syntaxe

```
return      ; fin de sous-routine. Le PC est rechargé depuis la pile, le
            ; programme poursuit à l'adresse qui suit la ligne call.
```

Bit du registre STATUS affecté

Aucun

Exemples

Comme ceci est un concept très important, je vais détailler un peu plus.

Imaginons un programme qui a besoin d'une petite temporisation (comme chaque instruction prend du temps, on peut s'arranger pour en faire perdre volontairement au programme afin de retarder son fonctionnement. Ecrivons-la :

```
        movlw  0xCA             ; valeur du compteur
        movwf  compteur         ; initialiser compteur de boucles
boucle
        decfsz compteur,f       ; décrémenter compteur, sauter si 0
        goto   boucle           ; boucler si pas 0
        xxx                     ; suite du programme
```

Imaginons maintenant que cette petite temporisation soit appelée régulièrement par notre programme principal. Ecrivons à quoi ressemble le programme principal :

```
        xxx                     ; instruction quelconque
        xxx                     ; instruction quelconque
        xxx                     ; instruction quelconque
        xxx                     ; instruction quelconque
        tempo                   ; ici, on a besoin d'une tempo
        xxx                     ; instruction quelconque
        xxx                     ; instruction quelconque
        xxx                     ; instruction quelconque
        tempo                   ; ici aussi
        xxx                     ; instruction quelconque
        xxx                     ; instruction quelconque
        tempo                   ; et encore ici
        xxx                     ; instruction quelconque
```

La première chose qui vient à l'esprit, est d'effectuer un copier/coller de notre temporisation. On obtient donc un programme comme ceci :

```
        xxx                     ; instruction quelconque
        xxx                     ; instruction quelconque
        xxx                     ; instruction quelconque
        xxx                     ; instruction quelconque
        movlw  0xCA             ; valeur du compteur
        movwf  compteur         ; initialiser compteur de boucles
boucle
        decfsz compteur,f       ; décrémenter compteur, sauter si 0
        goto   boucle           ; boucler si pas 0
        xxx                     ; instruction quelconque
        xxx                     ; instruction quelconque
        xxx                     ; instruction quelconque
        movlw  0xCA             ; valeur du compteur
        movwf  compteur         ; initialiser compteur de boucles
boucle2
        decfsz compteur,f       ; décrémenter compteur, sauter si 0
        goto   boucle2          ; boucler si pas 0
        xxx                     ; instruction quelconque
        xxx                     ; instruction quelconque
        movlw  0xCA             ; valeur du compteur
        movwf  compteur         ; initialiser compteur de boucles
boucle3
        decfsz compteur,f       ; décrémenter compteur, sauter si 0
        goto   boucle3          ; boucler si pas 0
```

Ceci n'est pas élégant, car, si nous devons changer la valeur de notre tempo, nous devons la changer partout dans le programme, sans oublier un seul endroit. De plus, ça prend beaucoup de place.

On peut également se dire : « utilisons une macro », qui, rappelez-vous, effectue une substitution au moment de l'assemblage. C'est vrai, qu'alors, il n'y a plus qu'un endroit à modifier, mais, dans notre PIC®, le code se retrouvera cependant autant de fois qu'on a utilisé la temporisation. Que de place perdue, surtout si la portion de code est grande et utilisée plusieurs fois.

Pour remédier à ceci, nous utiliserons la technique des sous-programmes. Première étape, modifions notre temporisation pour en faire une sous-routine :

```
tempo                           ; étiquette de début de la sous-routine
    movlw     0xCA              ; valeur du compteur
    movwf     compteur          ; initialiser compteur de boucles
boucle
    decfsz    compteur,f        ; décrémenter compteur, sauter si 0
    goto      boucle            ; boucler si pas 0
    return                      ; fin de la sous-routine.
```

Deuxième étape, nous modifions notre programme principal pour que chaque fois que nous avons besoin d'une tempo, il appelle le sous-programme. Nous obtenons :

```
    xxx               ; instruction quelconque
    xxx               ; instruction quelconque
    xxx               ; instruction quelconque
    call  tempo       ; appel du sous-programme
    xxx               ; instruction quelconque, le programme continue ici
    xxx               ; instruction quelconque
    xxx               ; instruction quelconque
    call  tempo       ; appel du sous-programme
    xxx               ; instruction quelconque, le programme continue ici
    xxx               ; instruction quelconque
    call  tempo       ; appel du sous-programme
    xxx               ; instruction quelconque, le programme continue ici
```

Dans ce cas, la routine tempo n'est présente qu'une seule fois en mémoire programme.

On peut améliorer : supposons que nous désirons une temporisation à durée variable.
On modifie la sous-routine en supprimant la valeur d'initialisation, et on place celle-ci dans le programme principal. Cela s'appelle un sous-programme avec passage de paramètre(s).

Exemple, notre sous-programme devient :

```
tempo                           ; étiquette de début de la sous-routine
    movwf     compteur          ; initialiser compteur de boucles
boucle
    decfsz    compteur,f        ; décrémenter compteur, sauter si 0
    goto      boucle            ; boucler si pas 0
return                          ; fin de la sous-routine.
```

Quant à notre programme principal :

```
xxx                 ; instruction quelconque
xxx                 ; instruction quelconque
xxx                 ; instruction quelconque
xxx                 ; instruction quelconque
movlw 0x25          ; charger w avec 0x25
call  tempo         ; appel du sous programme tempo d'une durée de 0x25
xxx                 ; instruction quelconque, le programme continue ici
xxx                 ; instruction quelconque
xxx                 ; instruction quelconque
movlw 0x50          ; charger w avec 0x50
call  tempo         ; appel du sous programme tempo d'une durée de 0x50
xxx                 ; instruction quelconque, le programme continue ici
xxx                 ; instruction quelconque
movlw 0x10          ; charger w avec 0x10
call  tempo         ; appel du sous programme tempo d'une durée de 0x10
xxx                 ; instruction quelconque, le programme continue ici
```

Voilà, maintenant vous savez ce qu'est un sous-programme. Enfantin, n'est-ce pas ?

9.28 L'instruction « RETLW » (RETurn with Literal in W)

Retour de sous-routine avec valeur littérale dans W. C'est une instruction très simple : elle équivaut à l'instruction return, mais permet de sortir d'une sous-routine avec une valeur spécifiée dans W.

Syntaxe

```
retlw     k      ; (w) = k puis return
```

Bit du registre STATUS affecté

Aucun

Exemple

```
test                        ; étiquette de début de notre sous-programme
    btfss   mavariable,0    ; teste le bit 0 de mavariable
    retlw   0               ; si vaut 0, fin de sous-programme avec (w)=0
    retlw   1               ; sinon, on sort avec (w) = 1
```

Le programme qui a appelé la sous-routine connaît donc le résultat de l'opération en lisant le registre «w ».

9.29 L'instruction « RETFIE » (RETurn From IntErrupt)

Cette instruction indique un retour d'interruption (nous verrons ultérieurement ce que sont les interruptions). Cette instruction agit d'une manière identique à RETURN, excepté que les interruptions sont remises automatiquement en service au moment du retour au programme principal.

Syntaxe

```
retfie                  ; retour d'interruption
```

Bit du registre STATUS affecté

Aucun

9.30 L'instruction « CLRF » (CLeaR F)

Cette instruction efface l'emplacement mémoire spécifié

Syntaxe

```
clrf      f             ; (f) = 0
```

Bit du registre STATUS affecté

Z : Vaut donc toujours 1 après cette opération.

Exemple

```
Clrf      mavariable    ; (mavariable) = 0
```

9.31 L'instruction « CLRW » (CLeaR W)

Cette instruction efface w

Syntaxe

```
clrw                    ; (w) = 0
```

C'est une instruction qui n'est pas vraiment indispensable, car on pourrait utiliser l'instruction « movlw 0 ». Cependant, à la différence de movlw 0, clrw positionne le bit Z.

Bit du registre STATUS affecté

Z : Vaut donc toujours 1 après cette opération.

9.32 L'instruction « CLRWDT » (CLeaR WatchDog)

Remet à 0 le chien de garde (watchdog) de votre programme. Nous aborderons la mise en œuvre du watchdog ultérieurement. Sachez cependant que c'est un mécanisme très pratique qui permet de provoquer un reset automatique de votre PIC® en cas de plantage du programme (parasite par exemple).

Le mécanisme est simple à comprendre : il s'agit pour votre programme d'envoyer cette instruction à intervalles réguliers. Si la commande n'est pas reçue dans le délai imparti, le PIC® redémarre à l'adresse 0x00. C'est exactement le mécanisme utilisé par les conducteurs de train qui doivent presser un bouton à intervalle régulier. Si le bouton n'est pas pressé, le train s'arrête. On détecte ainsi si le conducteur est toujours dans l'état d'attention requis.

Syntaxe

```
clrwdt                ; remet le timer du watchdog à 0
```

Bit du registre STATUS affecté

Aucun

9.33 L'instruction « COMF » (COMplement F)

Effectue le complément à 1 de l'emplacement mémoire spécifié. Donc, inverse tous les bits de l'octet désigné

Syntaxe

```
comf      f , d ; NOT (f) -> (d)
```

Bit du registre STATUS affecté

Z

Exemple

```
movlw  B'11001010'        ; charge valeur dans W
movwf  mavariable         ; initialise mavariable
comf   mavariable,w       ; charge l'inverse de mavariable dans W
                          ; (W) = B'00110101'
```

Astuce : en utilisant cette instruction, on peut également tester si mavariable vaut 0xFF. En effet, si c'est le cas, W vaudra 0 et Z vaudra donc 1.

9.34 L'instruction « SLEEP » (Mise en sommeil)

Place le PIC® en mode de sommeil. Il ne se réveillera que sous certaines conditions que nous verrons plus tard.

Syntaxe

```
sleep                     ; arrêt du PIC
```

Bit du registre STATUS affecté

T0, PD

9. 35 L'instruction « NOP » (No Operation)

Comme vous devez être fatigué, et moi aussi, je vous présente l'instruction qui ne fait rien, qui ne positionne rien, et qui ne modifie rien. On pourrait croire qu'elle ne sert à rien. En fait elle est surtout utilisée pour perdre du temps, par exemple pour attendre une ou deux instructions, le temps qu'une acquisition ai pu se faire, par exemple. Nous l'utiliserons donc à l'occasion.

Syntaxe

```
nop             ; tout simplement
```

Ceci termine l'analyse des 35 instructions utilisées normalement dans les PIC® mid-range.

Ceci peut vous paraître ardu, mais en pratiquant quelque peu, vous connaîtrez très vite toutes ces instructions par cœur. Pensez pour vous consoler que certains processeurs CISC disposent de plusieurs centaines d'instructions.

9.36 Les instructions obsolètes

Il reste 2 instructions qui étaient utilisées dans les précédentes versions de PIC®. Elles sont encore reconnues par le PIC16F84 mais leur utilisation est déconseillée par Microchip®. En effet, leur compatibilité future n'est pas garantie.

Il s'agit de l'instruction OPTION, qui place le contenu du registre « W » dans le registre OPTION_REG, et de l'instruction TRIS, qui place le contenu de « W » dans le registre TRISA ou TRISB suivant qu'on utilise TRIS PORTA ou TRIS PORTB.

Ces instructions ne sont plus nécessaires actuellement, car ces registres sont désormais accessibles directement à l'aide des instructions classiques. Je vous conseille donc fortement d'éviter de les utiliser, sous peine de rencontrer des problèmes avec les futures versions de PIC® de Microchip®.

Je ne vous les ai donc présentées que pour vous permettre de comprendre un éventuel programme écrit par quelqu'un d'autre qui les aurait utilisées.

10. Les modes d'adressage

Les instructions utilisent toutes une manière particulière d'accéder aux informations qu'elles manipulent. Ces méthodes sont appelées « modes d'adressage ».

Je vais simplement donner un petit exemple concret de ce qu'est chaque mode d'adressage. Supposons que vous vouliez mettre de l'argent dans votre poche :

10.1 L'adressage littéral ou immédiat

Avec l' ADRESSAGE IMMEDIAT ou ADRESSAGE LITTERAL, vous pouvez dire : 'je mets 100F en poche'. La valeur fait IMMEDIATement partie de la phrase. J'ai donné LITTERALlement la valeur concernée. Pas besoin d'un autre renseignement.

Exemple

```
movlw          0x55         ; charger la valeur 0x55 dans W
```

10.2 L'adressage direct

Avec l' ADRESSAGE DIRECT, vous pouvez dire : je vais mettre le contenu du coffre numéro 10 dans ma poche. Ici, l'emplacement contenant la valeur utile est donné DIRECTement dans la phrase. Mais il faut d'abord aller ouvrir le coffre pour savoir ce que l'on va effectivement mettre en poche. On ne met donc pas en poche le numéro du coffre, mais ce qu'il contient. Pour faire l'analogie avec les syntaxes précédentes, je peux dire que je mets (coffre 10) dans ma poche.

Exemple

```
movf     0x10 , W     ; charger le contenu de l'emplacement 0x10 dans W
```

10.3 L'adressage indirect

Avec l' ADRESSAGE INDIRECT , vous pouvez dire :

Le préposé du guichet numéro 3 va me donner le numéro du coffre qui contient la somme que je vais mettre en poche.

Ici, vous obtenez le numéro du coffre INDIRECTement par le préposé au guichet.

Vous devez donc allez demander à ce préposé qu'il vous donne le numéro du coffre que vous irez ouvrir pour prendre l'argent. On ne met donc en poche, ni le numéro du préposé, ni le numéro du coffre que celui-ci va vous donner. Il y a donc 2 opérations préalables avant de connaître la somme que vous empocherez.

Cet adressage fait appel à 2 registres, dont un est particulier, car il n'existe pas vraiment. Examinons-les donc :

10.3.1 Les registres FSR et INDF

Ceux qui suivent sont déjà en train de chercher dans le tableau 4-2 après INDF.

INDF signifie INDirect File. Vous le voyez maintenant ? Et oui, c'est le fameux registre de l'adresse 0x00. Ce registre n'existe pas vraiment, ce n'est qu'un procédé d'accès particulier à FSR utilisé par le PIC® pour des raisons de facilité de construction électronique interne.

Le registre FSR est à l'adresse 0x04 dans les 2 banques. Il n'est donc pas nécessaire de changer de banque pour y accéder, quelle que soit la banque en cours d'utilisation.

Dans l'exemple schématique précédent, le préposé au guichet, c'est le registre FSR. L'adressage indirect est un peu particulier sur les PIC®, puisque c'est toujours à la même adresse que se trouvera l'adresse de destination. En somme, on peut dire qu'il n'y a qu'un seul préposé (FSR) dans notre banque. Comment cela se passe-t-il ?

Premièrement, nous devons écrire l'adresse pointée dans le registre FSR.
Ensuite, nous accédons à cette adresse pointée par le registre INDF.

On peut donc dire que INDF est en fait le registre FSR utilisé pour accéder à la case mémoire. Donc, quand on veut modifier la case mémoire pointée, on modifie FSR, quand on veut connaître l'adresse de la case pointée, on accède également à FSR. Si on veut accéder au CONTENU de la case pointée, on accède via INDF. Nous allons voir tout ceci par un petit exemple, mais avant,

ATTENTION

Le contenu du registre FSR pointe sur une adresse en 8 bits. Or, sur certains PIC®, la zone RAM contient 4 banques (16F876). L'adresse complète est donc une adresse sur 9 bits.

L'adresse complète est obtenue, en adressage DIRECT, par l'ajout des bits 7 et 8 sous forme de RP0¨et RP1 (RP1 est inutilisé pour le 16F84 car seulement 2 banques), et par l'ajout du bit IRP dans le cas de l'adressage INDIRECT (inutilisé sur le 16F84). Veillez donc à toujours laisser IRP (dans le registre STATUS) et RP1 à 0 pour assurer la portabilité de votre programme.

Exemple

```
movlw  0x50         ; chargeons une valeur quelconque
movwf  mavariable   ; et plaçons-la dans la variable « mavariable »
movlw  mavariable   ; on charge l'ADRESSE de mavariable, par
                    ; exemple, dans les leçons précédentes, c'était
                    ; 0x0E. (W) = 0x0E
movwf  FSR          ; on place l'adresse de destination dans FSR.
                    ; on dira que FSR POINTE sur mavariable
movf   INDF,w       ; charger le CONTENU de INDF dans W.
```

LE CONTENU DE INDF EST TRADUIT PAR LE PIC® COMME ETANT LE CONTENU DE L'EMPLACEMENT MEMOIRE POINTE PAR FSR (W) = 0X50

10.4 Quelques exemples

Je vais me répéter, mais les modes d'adressages doivent impérativement être compris. En multipliant les exemples, j'espère que tout le monde pourra comprendre. Pour les habitués des processeurs divers, excusez ces répétitions. Les registres sont initialisés avec les valeurs précédentes.

```
movlw    mavariable
```

C'est de l'adressage immédiat ou littéral ; donc on charge la VALEUR de mavariable, ce qui correspond en réalité à son ADRESSE. Donc 0x0E est placé dans (W). Ceci se reconnaît au « l » de l'instruction movlw. Attention, la valeur de mavariable ce n'est pas son contenu.

```
movf     mavariable , w
```

Cette fois, c'est de l'adressage DIRECT, donc, on va à l'adresse mavariable voir ce qu'il y a à l'intérieur. On y trouve le CONTENU de mavariable, donc (w) = 0x50 (dans notre exemple). Pour l'assembleur, « mavariable » sera remplacée par « 0x0E », donc movf 0x0E,w

```
movf   INDF , w
```

Maintenant, c'est de l'adressage INDIRECT. Ce mode d'adressage se reconnaît immédiatement par l'utilisation du registre INDF. Le PIC® va voir dans le registre FSR, et lit l'adresse contenue, dans ce cas 0X0E. Il va ensuite à l'emplacement visé, et lit le CONTENU. Donc, dans (W) on aura le contenu de 0x0E, soit 0x50.

```
movf   FSR , w
```

Ceci est un piège. C'est en effet de l'adressage DIRECT. On placera donc dans (W) le CONTENU du registre FSR, donc 0X0E (l'adresse pointée) sera mis dans (W).

Je terminerai avec un petit mot pour les spécialistes des microprocesseurs qui lisent ce cours comme mise à niveau.

Vous aurez donc constaté que je ne parle nulle part des modes d'adressage de type indexé (pré et post, avec ou sans offset). En fait, c'est tout simplement parce que ce mode d'adressage n'existe pas dans les PIC® de type 16F.

Il vous faudra donc faire preuve d'astuce pour compenser ces lacunes.

Il est logique que la facilité d'apprentissage et de mise en œuvre se paye par une moins grande souplesse de création des applications.

Notes :…

11. Réalisation d'un programme embarqué

On désigne généralement sous la dénomination « logiciel embarqué » un programme destiné à tourner localement sur une carte disposant d'un microcontrôleur. Nous allons donc commencer la partie la plus amusante. Nous allons créer de petits programmes sur une carte à PIC®.

11.1 Le matériel nécessaire

Utilisez une platine d'essais constituée de petits trous reliés ensemble par rangées. Les liaisons s'effectuent alors en fils volants. Vous trouverez ces platines chez tous les marchands d'électronique. Pour mettre ces leçons en pratique, il vous faudra le matériel suivant (récupérable et peu onéreux) :

- 1 PIC® 16F84 ou 16F84A en boîtier PDIP et de fréquence quelconque
- 2 supports « tulipe » 18 pins / écartement entre les rangées : 0.3''
- 1 Quartz de 4MHz
- 2 condensateurs de 27pF
- 1 Led rouge
- 1 Résistance de 330 ohms.
- 1 bouton-poussoir « normalement ouvert » (N.O.)
- 1 peu de fil rigide pour les connexions sur la platine d'essais
- 1 alimentation stabilisée
-

Si vous ne disposez pas d'une alimentation de 5V, vous pouvez, soit utiliser une pile plate de 4.5V, soit réaliser une petite alimentation dont le schéma est donné page suivante.

Il vous faudra dans ce dernier cas :

- 1 Bloc d'alimentation secteur de 9 à 15V, pas critique.
- 1 Condensateur de 10 à 100µF/35V.
- 1 Condensateur de 0.1 µF
- 1 diode 1N4007, ou autre diode permettant un courant de 1A (pas critique)
- 1 Régulateur de type 7805.

Cet ensemble ne vous ruinera pas. De plus, les composants pourront être récupérés pour vos prochaines applications. Vous verrez que les PIC® peuvent être utilisés dans des tonnes d'applications au quotidien. Bien entendu, n'omettez pas de vous construire un programmateur de PIC®. Voyez l'annexe A1.8 pour plus de détails.

11.2 Montage de la platine d'essais

- Insérez le PIC® dans un des supports tulipe.
- Insérez le support restant dans la platine d'essais, et procédez aux connexions suivant le schéma ci-joint.
- Vérifiez tout avant de connecter l'alimentation.
- Si vous n'avez pas d'alimentation stabilisée, utilisez une pile, ou connectez les composants suivant le schéma suivant.

Une fois tout ceci fait, vérifiez une dernière fois, sans placer le PIC® avec son support dans le support de la platine, puis mettez sous tension. Vérifiez si les tensions sont correctes au niveau des broches d'alimentation du PIC®.

Vous êtes maintenant prêt à commencer les expérimentations.

Effectuez une copie de votre fichier m16F84.asm, et renommez cette copie Led_cli.asm. Lancez MPLAB® et répondez « non » lorsqu'on vous demande si vous voulez charger Essai1.

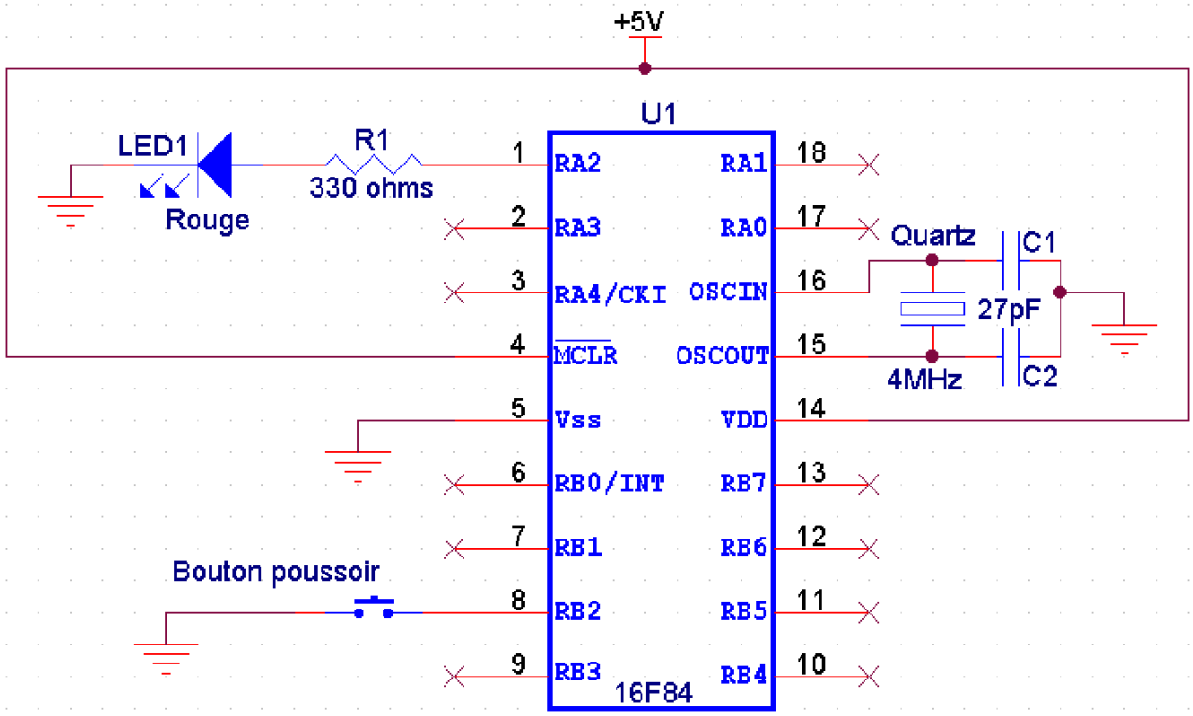
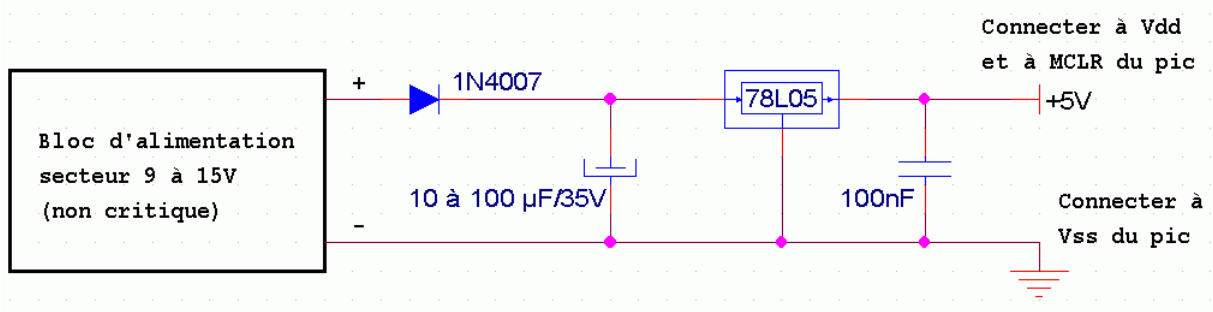


Schéma de connexion du PIC® (n'oubliez pas de relier *MCLR* à Vdd)

Schéma de la petite alimentation stabilisée



11.3 Création du projet

Dans MPLAB®, vous savez maintenant créer un projet. Créez donc le projet Led_cli dans votre répertoire de travail (project->new project). Si vous avez une fenêtre « untitled » à l'écran, fermez-la préalablement.

N’oubliez pas bien sûr d’ouvrir votre fichier « Led_cli.asm » .De nouveau, ce fichier est disponible en annexe.

11.4 Edition du fichier source

Complétez le cadre d’en-tête suivant votre désir. Je vous indique ci-dessous un exemple. Prenez l’habitude de toujours documenter vos programmes. Ce n’est pas un luxe, c’est impératif pour une maintenance efficace dans le temps.

```
;*****************************************************************************
;   PROGRAMME DE CLIGNOTEMENT D'UNE LED CONNECTEE SUR LE PORTA.2              *
;   D'UN PIC16F84. PROGRAMME D'ENTRAINEMENT AU FONCTIONNEMENT                 *
;   DES PICS.                                                                 *
;*****************************************************************************
;                                                                             *
;    NOM:      LED-CLI                                                        *
;    Date:     09/02/2001                                                     *
;    Version:  1.0                                                            *
;    Circuit:  Platine d'essais                                               *
;    Auteur:   Bigonoff                                                       *
;*****************************************************************************
;                                                                             *
;    Fichier requis: P16F84.inc                                               *
;                                                                             *
;*****************************************************************************
;                                                                             *
;    Notes: Ce petit programme permet de faire clignoter une LED              *
;           sur le port A.2 à une fréquence de 1Hz                            *
;           Ce programme fait partie de la leçon 6 des cours                  *
;                                                                             *
;*****************************************************************************
```

11.5 Choix de la configuration

Plus bas dans le fichier, vous trouverez ceci :

```
__CONFIG    _CP_OFF & _WDT_ON & _PWRTE_ON & _HS_OSC

; '__CONFIG' précise les paramètres encodés dans le processeur au moment de
; la programmation. Les définitions sont dans le fichier include.
; Voici les valeurs et leurs définitions :

;_CP_ON             Code protection ON : impossible de relire
;_CP_OFF            Code protection OFF
;_PWRTE_ON          Timer reset sur power on en service
;_PWRTE_OFF         Timer reset hors-service
;_WDT_ON            Watch-dog en service
;_WDT_OFF           Watch-dog hors service
;_LP_OSC            Oscillateur quartz basse consommation
;_XT_OSC            Oscillateur quartz moyenne vitesse ou externe
;_HS_OSC            Oscillateur quartz grande vitesse
;_RC_OSC            Oscillateur à réseau RC
```

J'ai inclus les commentaires dans le fichier de façon à ce qu'il soit plus rapidement modifiable sans devoir recourir au datasheet. Je vais donner un brin d'explication.

Remarquez qu'on effectue un AND (&) entre les différentes valeurs, les niveaux actifs sont donc des niveaux 0, puisqu'un « AND » ne permet d'imposer que des « 0 ». Vous devez donc préciser toutes les valeurs que vous n'utilisez pas.

Le premier paramètre précise si votre PIC® sera protégé ou non contre la lecture à la fin de la programmation. Laissez ici ce paramètre sur « CP_OFF » = non protégé.

Le second paramètre précise si le « chien de garde » (watchdog) est mis ou non en service. Dans un premier temps, remplacez WDT_ON par WDT_OFF pour le mettre hors-service.

Ensuite, laissez PWRTE sur ON pour préciser que vous utilisez un reset « sécurisé », donc avec un allongement du temps avant démarrage. Ceci vous met à l'abri des alimentations un peu lentes à démarrer. J'expliquerai ceci plus tard.

Enfin, vient le fonctionnement de l'oscillateur que vous allez utiliser. Le tableau 8-1 page 40 donne les valeurs recommandées en fonction des fréquences utilisées pour un PIC® de 10MHz.

Retenez que la valeur _HS_OSC convient pour les fréquences élevées, ce qui est notre cas, puisque nous utilisons la vitesse maximale de notre PIC®. Si vous avez acheté une version 10 ou 20MHz, pas de panique, le mode HS_OSC fonctionnera également.

Il est important de ne pas utiliser _RC_OSC si on utilise un quartz. Ce paramètre est réservé à un fonctionnement par réseau R/C tel que dessiné figure 8-7 page 41.

LE FAIT D'UTILISER LE PARAMETRE « RC » AVEC UNE HORLOGE EXTERNE PEUT ENTRAINER LA DESTRUCTION DU PIC®.

Même, si en pratique, les PIC® sont des composants très solides, évitez de vous tromper à ce niveau. Et voilà, vous connaissez parfaitement _Config. Vous avez maintenant la ligne suivante :

```
__CONFIG   _CP_OFF & _WDT_OFF & _PWRTE_ON & _HS_OSC
```

11.6 Le registre OPTION

Si vous regardez le tableau 4-2, vous constaterez que ce registre se trouve à l'adresse 0x81, donc dans la banque1. Dans les fichiers include de MPLAB®, ce registre est déclaré avec le nom OPTION_REG.

C'est donc ce nom que vous devrez utiliser. Nous allons le détailler ici. Ce registre est un registre de bits, c'est à dire que chaque bit a un rôle particulier :
Le tableau de la page 16 représente le contenu de ce registre :

b7 : *RBPU*

Quand ce bit est mis à 0 (actif niveau bas en italique), une résistance de rappel au +5 volts est placée sur chaque pin du PORTB. Nous verrons dans cette leçon le fonctionnement du PORTB. Si vous regardez notre schéma, vous constaterez que le bouton-poussoir connecté sur RB2 place cette pin à la masse si on le presse.

Il n'existe aucune possibilité pour envoyer du +5V dans notre schéma. En validant cette option, la résistance interne force la pin RB2 à 1 lorsque le bouton n'est pas pressé.

Notez que cette option valide les résistances sur toutes les pins du PORTB. Il n'est donc pas possible de choisir certaines résistances en particulier. De même cette fonction n'existe que pour le PORTB.

Vous constatez donc déjà que tous les ports ne sont pas identiques : il faut y penser au moment de la conception d'un circuit. Dans notre cas, nous mettrons donc les résistances en service avec b7 = 0

b6 : INTEDG

Donne, dans le cas où on utilise les interruptions sur RB0, le sens de déclenchement de l'interruption. Si b6 = 1, on a interruption si le niveau sur RB0 passe de 0 vers 1. Si b6 = 0, l'interruption s'effectuera lors de la transition de 1 vers 0.

Comme nous n'utilisons pas les interruptions dans cette leçon, nous pouvons laisser b6 = 0.

b5 : TOCS

Ce bit détermine le fonctionnement du timer0, que nous verrons bientôt. Retenez que le timer0 est incrémenté soit en fonction de l'horloge interne (synchronisé au programme) , dans ce cas b5 = 0, soit il compte les impulsions reçues sur la pin RA4, dans ce cas b5=1.

Comme ce dernier mode nécessite un circuit de génération d'impulsions externe, nous utiliserons pour le timer0 l'horloge interne, donc b5 = 0

b4 : TOSE

Donne, pour le cas ou le bit 5 serait 1, le sens de la transition qui détermine le comptage de tmr0. Si b4 = 1, on a comptage si le signal passe de 5V à 0V sur RA4, si on a b4 = 0, ce sera le contraire.

Comme nous avons placé b5=0, b4 est alors inutilisé. Nous laisserons donc b4 = 0.

b3 : PSA

Nous avons dans le PIC® un prédiviseur. Qu'est-ce que c'est ? Et bien tout simplement, ceci indique le nombre d'impulsions qui devront être reçues pour provoquer une incrémentation de la destination. Nous y reviendrons en détail avec le fonctionnement du tmr0.

A ce niveau, sachez simplement que ce prédiviseur peut servir à une des deux fonctions suivantes (et pas les deux) : soit il effectue une prédivision au niveau du timer du watchdog (b3 = 1), soit il effectue une prédivision au niveau du tmr0 (timer0) (b3=0). Dans notre cas, mettez b3 = 1 (nous verrons ci-dessous pourquoi).

b2, b1,b0 : PS2,PS1,PS0

Ces trois bits déterminent la valeur de prédivision pour le registre déterminé ci-dessus. Il y a donc 8 valeurs possibles, montrées dans le petit tableau de la page 16.

Remarquez que les valeurs sont différentes pour le watchdog et pour tmr0. En effet, il n'y a pas de 'division par 1' pour ce dernier registre.

Si vous désirez ne pas utiliser de prédiviseur du tout, la seule méthode est de mettre b3=1 (prédiviseur sur watchdog) et PS2 à PS0 à 0. Dans ce cas : pas de prédiviseur sur tmr0, et prédiviseur à 1 sur watchdog, ce qui correspond à pas de prédiviseur non plus. Nous mettrons donc b2=b1=b0= 0.

11.7 Edition du programme

Voilà encore un registre vu. Nous utiliserons donc la valeur B'00001000' pour notre programme, soit 0x08. J'ai l'habitude de ne pas traîner des valeurs fixes à travers mes programmes, afin d'en faciliter la maintenance. Je place ces valeurs en début de programme en utilisant des assignations ou des définitions.

L'assignation est déjà créée plus bas dans le programme. J'ai créé une CONSTANTE que j'ai appelé OPTIONVAL et qui contiendra la valeur à placer plus tard dans le registre OPTION_REG. Je rappelle que les CONSTANTES n'occupent pas de place dans le PIC®, elles sont simplement remplacées par l'assembleur au moment de la compilation. Elles servent à faciliter la lecture du programme.

Cherchez donc plus bas dans le programme après les assignations, et remplacez la valeur affectée à OPTIONVAL par celle que nous avons trouvée et ajoutez vos commentaires.

Supprimez l'assignation concernant INTERMASK, car nous ne nous servirons pas des interruptions dans ce premier programme. Dans la zone des assignations, il vous reste donc ceci :

```
;*****************************************************************
;                            ASSIGNATIONS                         *
;*****************************************************************

OPTIONVAL   EQU   H'08          ; Valeur registre option
                                ; Résistance pull-up ON
                                ; Pas de préscaler
```

Descendons encore jusqu'à la zone des définitions. Nous allons donner un nom à notre bouton-poussoir et à notre LED.

Les instructions bcf et bsf que nous allons utiliser pour mettre ou lire des 1 ou des 0 dans les registres ont la syntaxe suivante : « bsf f , n » et les registres d'accès s'appellant PORTA (pour le port A) et PORTB (pour le port B), nous utiliserons des DEFINE permettant d'intégrer f et n en même temps.

Nous voyons sur le schéma que la LED est connectée sur le bit 2 du port A. Le bouton-poussoir est connecté sur le bit 2 du port B. Nous effaçons donc les définitions d'exemple, et nous les remplaçons par les nôtres. Nous obtenons alors ceci :

```
;*****************************************************************
;                              DEFINE                             *
;*****************************************************************
#DEFINE LED         PORTA,2      ; Led rouge
#DEFINE BOUTON      PORTB,2      ; bouton-poussoir
```

Notez que LED et BOUTON sont des noms que nous avons librement choisis, à condition qu'il ne s'agisse pas d'un mot-clé. Pas question par exemple d'utiliser STATUS ou encore MOVLW, bien que ce dernier exemple soit tiré par les cheveux, cela pourrait vous arriver un jour d'utiliser un mot réservé par inadvertance. A quoi servent les définitions ?

Et bien supposons que vous décidez de connecter la LED sur le PORTB bit 1 (RB1), par exemple. Et bien, nul besoin de rechercher partout dans le programme, il suffira de changer dans la zone DEFINE.

On descend encore un peu, et on arrive dans la zone des macros. Nous n'en avons pas vraiment besoin ici, mais nous allons quant même les utiliser à titre d'exemple.
Effacez la macro donnée à titre d'exemple et entrons celles-ci.

```
;*****************************************************************
;                              MACRO                              *
;*****************************************************************
LEDON       macro
            bsf   LED
            endm

LEDOFF      macro
            bcf   LED
            endm
```

la première colonne donne le nom de la macro (ici, 2 macros, une LEDON et une LEDOFF). La directive macro signifie 'début de la macro' la directive endm signifie 'fin de la macro'. Notez que les macros peuvent évidemment comporter plusieurs lignes de code.

Prenons notre exemple : quand nous utiliserons la ligne suivante dans notre programme (attention, ne pas mettre la macro en première colonne) :

```
LEDON
```

Au moment de la compilation, notre assembleur remplacera LEDON par :

```
bsf       LED
```

Il remplacera également LED par PORTA,2. Ce qui fait qu'en réalité nous obtiendrons :

```
bsf       PORTA , 2
```

Nous avons donc obtenu une facilité d'écriture et de maintenance. Gardez à l'esprit que les macros sont des simples substitutions de traitement de texte. Si votre macro se compose de 50 lignes de code, les 50 lignes seront copiées dans le PIC® à chaque appel de la macro.

Nous arrivons dans la zone des variables. Nous ajouterons celles-ci au fur et à mesure de leur nécessité. Effacez donc les 2 variables présentes, car elles sont utilisées dans les routines d'interruption que nous n'utiliserons pas ici.

```
;*****************************************************************
;                     DECLARATIONS DE VARIABLES                  *
;*****************************************************************
   CBLOCK 0x00C    ; début de la zone variables
   ENDC            ; Fin de la zone
```

A l'ORG 0x00, laissons l'appel vers la routine d'initialisation. Tout programme comporte en effet une étape d'initialisation des variables et des registres. Prenez l'habitude de séparer cette initialisation du reste du programme.

Comme nous n'utiliserons pas les interruptions, supprimez tout ce qui suit jusqu'à la routine d'initialisation, vous obtenez :

```
*****************************************************************
;                       DEMARRAGE SUR RESET                      *
;*****************************************************************

   org      0x000        ; Adresse de départ après reset
   goto  init            ; Adresse 0: initialiser

;*****************************************************************
;                        INITIALISATIONS                         *
;*****************************************************************

suite du programme
```

A ce stade, avant de poursuivre, nous allons étudier les registres dont nous allons nous servir, et tout d'abord :

11.8 Le registre PORTA

Ce registre est un peu particulier, puisqu'il donne directement accès au monde extérieur. C'est en effet ce registre qui représente l'image des pins RA0 à RA4, soit 5 pins. Si vous suivez toujours, c'est ce registre qui va servir à allumer la LED.

Ce registre se situe à l'adresse 05H, dans la banque 0. Chaque bit de ce registre représente une pin. Donc, seuls 5 bits sont utilisés. Pour écrire sur une pin en sortie, on place le bit correspondant à 1 ou à 0, selon le niveau souhaité.

Par exemple :

```
bsf PORTA , 1   ; envoyer niveau 1 sur RA1
```

place un niveau +5V sur la pin RA1. Notez qu'il faut pour cela que cette pin soit configurée en sortie (voir TRISA). Pour tester une entrée, on pourra par exemple utiliser

```
btfss    PORTA,3     ; tester RA3 et sauter si vaut 5V
```

Pour les électroniciens, vous avez le schéma interne des bits RA0 à RA3 figure 5-1 page 21. Vous voyez que la sortie peut être placée au niveau haut ou bas grâce aux deux transistors de sortie (montage push-pull). Quand ces pins sont programmées en entrée, elles sont sensibles à des niveaux 0/5V. Le niveau de transition dépend du modèle de PIC®, mais se situe généralement à la moitié de la tension d'alimentation. Une entrée « en l'air » est vue comme étant au niveau 0. Evitez cependant ce cas pour les problèmes de grande sensibilité aux parasites.

Pour la pin RA4, figure 5-2, en sortie elle est de type drain ouvert, et en entrée elle comporte un trigger de Schmitt.

Les PORTS disposent d'une diode de protection vers le 0V et vers le 5V. De sorte qu'avec une simple résistance série, vous pouvez envoyer des signaux vers ces pins qui sortent de la gamme de tension Vss/Vdd.

La pin RA4 fait exception à cette règle, car elle ne dispose que d'une diode vers la masse.

Pour tous, retenez que les pins RA0 à RA3 peuvent être utilisées en entrée avec des niveaux 0/5V ou en sortie en envoyant du 0V ou du 5V. Quant à la pin RA4, en sortie, elle ne peut pas envoyer du 5V.

Elle ne peut que mettre la sortie « en l'air » sur niveau 1 et la forcer à la masse sur niveau 0. En entrée, elle comporte un dispositif permettant d'empêcher les hésitations lorsque le niveau présenté n'est pas franc. Par exemple lorsque le niveau varie lentement de 0V à 5V.

Tenez compte de ces spécificités lorsque vous construirez votre montage. Par exemple, si nous avions placé la LED telle quelle sur RA4, elle ne se serait jamais allumée.

A partir de la page 75, chapitre 11, vous avez les caractéristiques maximales de fonctionnement du PIC®. Vous pouvez retenir simplement de vous limiter à utiliser 20mA maximum par pin de sortie, avec un maximum total de 80mA pour le PORTA, et 150mA pour le PORTB.

Si vous utilisez des caractéristiques proches de cette limite, analysez les tableaux avec plus de précision pour obtenir les limites en puissance et t°. Nous n'aurons pas ce genre de problème dans nos exercices. La consommation de notre LED sur RA2 répond à la formule suivante :

Intensité : (5V – Tension Led) / 330 ohms, soit (5V – 1,75V) / 330 ohms = 9,85mA.

11.8.1 Fonctionnement particulier des PORTS

Les PORTS fonctionnent toujours sur le principe lecture->modification->écriture.
Par exemple, si vous écrivez bsf PORTA,1, le PIC® procède de la manière suivante :

1) Le PORTA est lu en intégralité (pins en entrée et en sortie)
2) Le bit 1 est mis à 1
3) Tout le PORTA est réécrit (concerne les pins en sortie).

Conséquences

Supposons par exemple que le RA4 soit en sortie et mis à 1. Comme il est à drain ouvert, supposons qu'une électronique externe le force actuellement à 0.

Si vous effectuez l'opération suivante :

```
bsf      PORTA  , 1         ; mettre RA1 à 1
```

Lorsque le PORTA va être lu, RA4 sera lu comme 0, bien qu'il soit à 1. RA1 sera forcé à 1, et le tout sera replacé dans PORTA. RA4 est donc maintenant à 0, sans que vous l'ayez explicitement modifié.

C'est vrai que ce type de cas n'est pas courant (par exempe, utilisation de RA4 pour piloter une ligne I²C), mais il est bon de le savoir, si un jour une de vos interfaces vous pose problème, et si vous avez l'impression que vos pins en sortie changent de niveau sans que vous le désiriez.

11.9 Le registre TRISA

Ce registre est situé à la même adresse que PORTA, mais dans la banque 1. Son adresse complète sur 8 bits est donc 0x85.

Ce registre est d'un fonctionnement très simple et est lié au fonctionnement du PORTA.

Chaque bit positionné à 1 configure la pin correspondante en entrée. Chaque bit à 0 configure la pin en sortie.

Au reset du PIC®, toutes les pins sont mises en entrée, afin de ne pas envoyer des signaux non désirés sur les pins. Les bits de TRISA seront donc mis à 1 lors de chaque reset.

Notez également que, comme il n'y a que 5 pins utilisées sur le PORTA, seuls 5 bits (b0/b4) seront utilisés sur TRISA.

Exemple d'utilisation

Prenons en exemple le schéma de notre petit circuit : que devons-nous faire pour allumer la LED ? (PS : ne mettez pas pour l'instant ces instructions dans votre projet).
Premièrement, nous devons configurer TRISA et mettre le bit2 (RA2) en sortie.

Comme le registre TRISA se trouve en banque1, et que l'adressage DIRECT n'utilise que 7 bits, nous devrons donc commuter le bit RP0 du registre STATUS (voir chapitres précédents).

Ensuite, nous pourrons envoyer un niveau ‘1’ sur PORTA, correspondant à 5V sur la pin RA2. La séquence correcte sera donc :

```
bsf STATUS , RP0            ; on passe en banque 1
bcf TRISA , 2               ; bit 2 de TRISA à 0 = sortie pour RA2
bcf STATUS , RP0            ; on repasse en banque 0
bsf PORTA , 2               ; on envoie 5V sur RA2 : la LED s’allume
.
.
.
bcf PORTA , 2               ; 0V sur RA2, la LED s’éteint
```

Notez que comme RA2 restera en sortie durant tout le programme (dépend du schéma), nous placerons la valeur de TRISA dans la routine d’initialisation. Quand nous voudrons allumer ou éteindre la LED dans le programme principal, nous n’aurons donc plus qu’une seule instruction à utiliser. On commence à entrer dans le concret .

11.10 Les registres PORTB et TRISB

Ces registres fonctionnent exactement de la même manière que PORTA et TRISA, mais concernent bien entendu les 8 pins RB. Tous les bits sont donc utilisés dans ce cas.

Voyons maintenant les particularités du PORTB. Nous en avons déjà vu une, puisque les entrées du PORTB peuvent être connectées à une résistance de rappel au +5V de manière interne.

La sélection s’effectuant par le bit 7 du registre OPTION. Le schéma interne visible figures 5-3 et 5-4 page 23 du datasheet vous montre que les bits b0 et b4/b7 peuvent être utilisés comme source d’interruption, le bit 0 peut de plus être utilisé de manière autonome pour générer un autre type d’interruption.

Nous verrons le fonctionnement des interruptions dans un autre chapitre. Sachez cependant déjà que les schémas que vous concevrez vous-même dans le futur devront tenir compte des particularités des PORTs.

Note :

Après un reset, vous vous demandez peut-être quel est l’état de tel ou tel registre ?

Vous trouverez ces explications dans le tableau de la page 14. Vous voyez qu’après un reset, le registre OPTION_REG voit tous ses bits mis à 1. Vous devez donc spécifier l’effacement du bit7 pour valider les résistances de rappel au +5V.

11.11 Exemple d’application

Toujours en partant de notre schéma, nous désirons allumer la LED lorsque nous pressons le bouton, et l’éteindre lorsque nous le relâchons. Voici un exemple du programme nécessaire (attention, prenez garde que le niveau sur RB2 passe à 0 lorsque le bouton est enfoncé (connexion à la masse)

```
        bsf     STATUS , RP0            ; on passe en banque 1
        bcf     OPTION_REG, NOT_RBPU    ; résistance de rappel en service
        bcf     TRISA , 2               ; bit 2 de TRISA à 0 = sortie pour RA2
                                        ; inutile de configurer TRISB, car il est
                                        ; en entrée par défaut au reset du PIC
        bcf    STATUS , RP0             ; on repasse en banque 0
boucle                                  ; étiquette début de la boucle principale
        btfss  PORTB , 2                ; tester RB2, sauter si vaut 1
        bsf    PORTA , 2                ; RB2 vaut 0, donc on allume la LED
        btfsc  PORTB , 2                ; tester RB2, sauter si vaut 0
        bcf    PORTA , 2                ; RB2 vaut 1, donc LED éteinte
        goto   boucle                   ; et on recommence
```

11.12 La routine d'initialisation

Examinons les instructions contenues à partir de l'étiquette « init »

Les 2 premières lignes ,

```
clrf      PORTA         ; Sorties portA à 0
clrf      PORTB         ; sorties portB à 0
```

mettent les sorties des PORTs à 0. Comme ça, lorsque vous configurerez les ports en sortie, ils enverront par défaut des niveaux 0V. Si votre électronique demande, pour être sécurisée, d'autres valeurs, à vous de vous en charger.

Ensuite nous trouvons la ligne :

```
clrf      EEADR         ; permet de diminuer la consommation
```

Ne vous inquiétez pas pour cette ligne. Sachez seulement qu'une valeur différente dans ce registre augmente légèrement la consommation des PIC® version 16C84. Cette ligne est donc un petit plus pour les montages alimentés par pile. Le registre EEADR sera vu lors des accès à l'EEPROM interne. Notez que ce bug a été corrigé sur le 16F84. Cette ligne n'est donc pas nécessaire dans ce cas.

La ligne suivante permet de se connecter sur la banque1. Jusqu'à nouvel ordre, les instructions suivantes utilisent des registres situés banque1.

```
bsf STATUS,RP0              ; passer en banque 1
```

Ensuite, nous trouvons :

```
movlw OPTIONVAL             ; charger masque
movwf OPTION_REG            ; initialiser registre option
```

Rappelez-vous que OPTIONVAL est une constante. Sa valeur a été définie précédemment par nos soins à 0x08. Cette valeur sera envoyée ici dans le registre OPTION (OPTION_REG). Souvenez-vous que OPTION_REG est le nom déclaré dans MPLAB® pour le registre OPTION.

Suit la petite routine suivante, destinée à effacer la RAM. Souvenez-vous, et c'est très important, qu'à la mise sous tension, la RAM contient des valeurs aléatoires. Pour éviter les mauvaises surprises, j'ai intégré cette routine à chaque démarrage, qui assure que la RAM ne contienne que des 0.

```
    ; Effacer RAM
    ; ------------
   movlw    0x0c        ; initialisation pointeur
   movwf    FSR         ; pointeur d'adressage indirect
init1
   clrf     INDF        ; effacer ram pointée par FSR
   incf     FSR,f       ; pointer sur suivant
   btfss    FSR,6       ; tester si fin zone atteinte (>=0x40)
   goto     init1       ; non, boucler
   btfss    FSR,4       ; tester si fin zone atteinte (>=0x50)
   goto     init1       ; non, boucler
   xxxxx                ; ici se trouve la suite du programme
```

Voilà donc un exemple concret de l'utilisation de l'adressage indirect.

En premier lieu, on initialise le pointeur vers la zone à manipuler (ici, la zone RAM utilisateur). Cette zone commence à l'adresse 0x0C (voir tableau 4-2) et se termine à l'adresse 0x4F incluse. La zone située dans la banque 1 n'existe pas pour ce PIC® (image de la banque 0) et n'a donc pas besoin d'être initialisée.

Puis vous trouvez l'instruction clrf INDF, qui signifie donc, si vous avez suivi, effacer l'emplacement mémoire dont l'adresse se trouve dans le registre FSR. On efface donc la mémoire située à l'emplacement 0x0C.

Vient ensuite l'incrémentation de FSR, donc, maintenant, FSR POINTE sur 0x0D.

On trouve alors 2 tests. Pour sortir de cette routine et arriver à la ligne que j'ai indiqué (ici se trouve la suite), il faudra donc éviter les 2 lignes goto.

Vous tombez sur le premier goto si le bit 6 de FSR vaut 1. Pour éviter ce goto, FSR devra donc valoir au moins B'01000000', ou encore 0x40. A ce moment, les adresses 0x0C à 0x3F sont donc mises à 0.

On arrive au deuxième test. Le goto suivant sera donc sauté si le bit 4 de FSR est à 1. On arrivera donc à la ligne 'ici se trouve la suite' uniquement lorsque les bits 4 et 6 de FSR seront égaux à 1. Ceci donne l'adresse suivante : B'01010000', soit 0x50.

Notez pour ceux qui ont suivi que vous auriez pu également utiliser un compteur de boucles et vous servir de l'instruction decfsz. Ceci aurait cependant nécessité l'utilisation d'une variable, variable qui se serait effacée elle-même par la présente routine, avec plantage du programme. Donc, la méthode présentée ici est donc la plus simple.

Effacez ensuite les lignes suivantes :

```
bcf   TRISA,0          ; Bit PORTA.0 en sortie (exemple)
bcf   STATUS,RP0       ; Sélectionner banque 0
movlw INTERMASK        ; masque interruption
movwf INTCON           ; charger interrupt control
```

car elles ne nous intéressent pas pour notre programme.

Voyons maintenant ce que nous devons ajouter :

Tout d'abord, nous devons initialiser RA2 en sortie, pour pourvoir agir sur la LED. L'instruction sera donc

```
bcf TRISA , 2
```

Or, nous avons déclaré dans les DEFINE, LED comme étant égal à PORTA,2. Pour permettre un remplacement facile de la LED sur une autre pin, que se passe-t-il si nous écrivons ?

```
bcf LED
```

Certains vont dire « on éteint la LED ». Et bien, pas du tout. Cette instruction sera remplacée par l'assembleur par

```
bcf PORTA , 2
```

Autrement dit par

```
bcf 0x05 , 2
```

Or, RP0 est toujours positionné à 1 au moment de l'exécution de cette ligne. On pointe donc toujours sur la banque 1. Si vous regardez ce qu'il y a à l'adresse 0x05 dans la banque1, c'est à dire à l'adresse 0x85, vous trouvez TRISA. Ainsi, l'opération a placé RA2 en sortie.

En changeant simplement le DEFINE LED au début du programme, vous pouvez changer toutes les références à RA2 dans tout le programme, même pour TRISA. Placez un petit commentaire d'en-tête, et vous obtenez :

```
; initialisations spécifiques
; --------------------------

bcf LED         ; LED en sortie (banque1)
```

Repassez ensuite en banque 0 avant de quitter l'intialisation. C'est une bonne pratique, car beaucoup d'erreurs sont provoquées par des erreurs de banque, ajoutez donc :

```
bcf STATUS , RP0      ; Repasser en banque 0
```

Terminez avec la ligne

```
goto start
```

qui branche le programme principal. Pour le moment, cela peut paraître inutile, car le programme principal suit directement cette instruction, mais nous allons ajouter plus loin un sous-programme qui sera intercalé entre les deux.

Voici à quoi devrait maintenant ressembler votre routine d'initialisation, que vous devriez maintenant avoir entièrement comprise (pas besoin d'une aspirine?).

Rassurez-vous, cela a l'air dur au début, mais, une fois assimilé, c'est très simple. De plus, vous avez vu pratiquement tout ce qu'il faut savoir pour réaliser un programme sans routines d'interruptions.

```
;*****************************************************************
;                       INITIALISATIONS                          *
;*****************************************************************
;
init
    clrf      PORTA             ; Sorties portA à 0
    clrf      PORTB             ; sorties portB à 0
    clrf      EEADR             ; permet de diminuer la consommation
    bsf       STATUS , RP0      ; sélectionner banque 1
    movlw     OPTIONVAL         ; charger masque
    movwf     OPTION_REG        ; initialiser registre option

       ; Effacer RAM
       ; ------------
    movlw     0x0c              ; initialisation pointeur
    movwf     FSR               ; pointeur d'adressage indirect
init1
    clrf      INDF              ; effacer ram
    incf      FSR,f             ; pointer sur suivant
    btfss     FSR,6             ; tester si fin zone atteinte (>=0x40)
    goto      init1             ; non, boucler
    btfss     FSR,4             ; tester si fin zone atteinte (>=0x50)
    goto      init1             ; non, boucler

       ; initialisations spécifiques
       ; ---------------------------
    bcf       LED               ; LED en sortie (banque1)
    bcf       STATUS , RP0      ; repasser banque 0
    goto      start             ; sauter au programme principal
```

Descendons maintenant dans le programme principal, et supprimons la ligne :

```
    clrwdt                      ; effacer watchdog
```

11.13 Les résultats de la compilation

Lancez la compilation par la touche <F10>.

Dans la fenêtre de résultat de compilation, vous devez trouver une ligne du type

Message[302] D:\DOCUME~1\LESSONS\DATAPIC\LED_CLI.ASM 101 : Register in operand not in bank 0. Ensure that bank bits are correct.

Ceci est un message d'avertissement (warning) qui vous signale qu'à la ligne 101 (dans mon cas, mais suivant vos espaces ou commentaires, vous aurez un autre numéro de ligne), vous avez accédé à un registre qui n'est pas situé en banque 0.

MPASM® vous demande de vérifier que votre bit RP0 est positionné correctement. Allez dans l 'éditeur sur la ligne dont vous avez le numéro. Vous tombez sur la ligne :

```
    movwf          OPTION_REG   ; initialiser registre option
```

Un coup d'œil sur le tableau 4-2 vous indique que le registre OPTION est dans la banque1. Mais comme votre RP0 a été positionné à 1 à cet endroit, il n'y a pas de problème. Quand vous aurez réalisé de gros programmes, vous aurez énormément de warning de ce type . Si vous voulez éviter ces warnings de banque, ajoutez la ligne

Errorlevel –302

directement sous la ligne #include de votre fichier source. Le « - » signifie que vous retirez ce message des messages actifs, et le « 302 » est le numéro du warning à éviter.

11.14 Le programme principal

Nous voulons faire clignoter notre LED à une fréquence de 1Hz (1 clignotement par seconde) . Nous allons donc créer un programme de la forme :

debut
- J'allume la LED
- J'attends ½ seconde
- J'éteins la LED
- J'attends ½ seconde
- Je retourne au début

Nous voyons que nous utilisons 2 fois la temporisation de ½ seconde. Nous créerons donc un sous-programme que nous appellerons « tempo ». Notre programme principal aura donc l'aspect suivant :

```
start
    bsf     LED         ; allumer la LED
    call    tempo       ; appeler la tempo de 0.5s
    bcf     LED         ; éteindre LED
    call    tempo       ; appeler la tempo de 0.5s
    goto    start       ; boucler
```

On aurait pu écrire également (en utilisant nos macros)

```
start
    LEDON               ; allumer la LED
    call    tempo       ; appeler la tempo de 0.5s
    LEDOFF              ; éteindre LED
    call    tempo       ; appeler la tempo de 0.5s
    goto    start       ; boucler
```

Choisissez la méthode que vous préférez. L'important, c'est d'avoir compris les deux méthodes.

11.15 La sous-routine de temporisation

Nous n'avons pas encore vu le timer0, ni les interruptions. Le but ici est de vous faire comprendre le fonctionnement du 16F84. Pour réaliser une tempo, il suffit dans notre cas de faire perdre du temps au 16F84 entre chaque inversion de la LED.

Nous devons donc perdre approximativement 0.5s. Les secondes ne sont pas appropriées pour les PIC®, qui travaillent à une vitesse beaucoup plus élevée. Nous utiliserons donc des unités de temps compatibles avec les PIC®.

Notre PIC® est cadencé à la fréquence de notre quartz, soit 4MHz. Or, le PIC® exécute un cycle d'instruction tous les 4 cycles de l'horloge principale. Le PIC® exécutera donc (4.000.000/4) = 1 million de cycles par seconde.

La plupart des instructions (hormis les sauts) s'exécutent en 1 cycle, ce qui vous donne approximativement un million d'instructions par seconde. Vous verrez parfois la dénomination MIPS. Ceci signifie Million d'Instructions Par Seconde. Notre PIC® avec ce quartz a donc une puissance de traitement de près de 1MIPS.

Chaque cycle d'instruction dure donc 1 millionième de seconde, ou encore une microseconde (µs). Voilà donc l'unité pour travailler avec notre PIC®.

Donc, notre tempo de 0.5s est donc une tempo de 500.000 microsecondes. Autrement dit, nous devons « perdre pour rien » 500.000 cycles dans notre routine de temporisation. Vous voyez donc que votre PIC®, pour notre application, va passer son temps à ne rien faire d'utile.

La première idée qui vient à l'esprit est de réaliser une boucle qui va incrémenter ou décrémenter une variable.

Réalisons-la. Commençons donc par déclarer notre variable (cmpt1) dans la zone de RAM. Ajoutons donc cette déclaration . Nous obtenons :

```
;*****************************************************************
;                    DECLARATIONS DE VARIABLES                    *
;*****************************************************************
   CBLOCK 0x00C          ; début de la zone variables
   cmpt1 : 1             ; compteur de boucles 1
   ENDC                  ; Fin de la zone
```

Créons maintenant l'ossature de notre sous-routine, que nous placerons entre la routine d'initialisation et le programme principal. N'oubliez pas de toujours utiliser des commentaires :

```
;*****************************************************************
;                      SOUS-ROUTINE DE TEMPORISATION              *
;*****************************************************************
;-----------------------------------------------------------------
; Cette sous-routine introduit un retard de 500.000 µs.
; Elle ne reçoit aucun paramètre et n'en retourne aucun
;-----------------------------------------------------------------
tempo
                         ; nous allons placer notre code ici
        return           ; retour de la sous-routine
```

Réalisons maintenant notre boucle.

```
tempo
   clrf     cmpt1       ; effacer compteur1
boucle1
   decfsz   cmpt1       ; décrémenter compteur1
   goto     boucle1     ; si pas 0, boucler
   return               ; retour de la sous-routine
```

Lançons la compilation (F10) :

Nous obtenons dans la fenêtre des résultats une ligne supplémentaire de la forme :

Message[305] D:\DOCUME~1\LESSONS\DATAPIC\LED_CLI.ASM 134 : Using default destination of 1 (file).

Le numéro de ligne peut varier suivant votre code source. Comme la compilation s'est effectuée correctement, il s'agit une fois de plus d'un message de type warning. Positionnez-vous dans l'éditeur sur la ligne incriminée (ligne 134 pour moi). Vous êtes sur la ligne :

```
   decfsz   cmpt1       ; décrémenter compteur1
```

En effet, l'instruction decfsz est de la forme « decfsz f , d ». Nous avons donc oublié d'indiquer la destination de l'opération. MPLAB® vous signale qu'il a utilisé pour vous ',f'

Faites donc attention, car si vous aviez voulu obtenir le résultat dans w, votre programme était faux. Modifiez donc votre commande en ajoutant ',f'

```
   decfsz   cmpt1 , f   ; décrémenter compteur1
```

Maintenant, nous allons calculer la durée de cette tempo.

```
tempo
   clrf     cmpt1       ; 1 cycle
boucle1
   decfsz   cmpt1 , f   ; 1 cycle si on ne saute pas, 2 si on saute. Donc, on
                        ; ne sautera pas 255 fois et on sautera 1 fois
   goto     boucle1     ; 2 cycles multiplié par 255 passages
   return               ; 2 cycles.
```

Le temps total est donc de :

2 cycles pour l'appel de la sous-routine (call tempo)
1 cycle pour le reset de la variable
257 cycles pour les 256 décrémentations
510 cycles pour les 255 goto
2 cycles pour le return.

Soit un total de 772 cycles. On est loin des 500.000 cycles nécessaires. Pour la suite des calculs, nous allons négliger les 2 cycles du call et les 2 cycles du return (comparés aux 500.000 cycles, c'est effectivement dérisoire).

Bon, nous allons allonger notre routine, en réalisant une seconde boucle qui va forcer la première boucle à s'exécuter 256 fois. Commençons par déclarer une nouvelle variable cmpt2 :

```
    cmpt1 : 1       ; compteur de boucles 1
    cmpt2 : 1       ; compteur de boucles 2
```

Ecrivons donc les 2 boucles imbriquées :

```
tempo
    clrf    cmpt2       ; effacer compteur2
boucle2
    clrf    cmpt1       ; effacer compteur1
boucle1
    decfsz  cmpt1 , f   ; décrémenter compteur1
    goto    boucle1     ; si pas 0, boucler
    decfsz  cmpt2 , f   ; si 0, décrémenter compteur 2
    goto    boucle2     ; si cmpt2 pas 0, recommencer boucle1
    return              ; retour de la sous-routine
```

Vous voyez que notre première boucle est toujours là, mais au lieu d'effectuer le return une fois terminée, nous recommençons la boucle tant que cmpt2 ne soit pas également égal à 0. Nous allons donc exécuter 256 fois notre boucle 1.

Quelle est la temporisation obtenue ? Calculons approximativement :

Durée de la boucle 1 : 257 cycles + 510 cycles + 1 cycle (clrf cmpt1) = 768 cycles.
Or cette boucle va être exécutée 256 fois, donc 768*256 = 196608 cycles, auquel il convient d'ajouter les quelques cycles d'initialisation etc.

Or, nous désirons 500.000 cycles. Nous devrons donc utiliser cette boucle (500.000/196608) = 2,54 fois. Nous ne savons pas faire de demi boucle. Nous effectuerons donc 2 boucles. Nous allons nous arranger pour que nos deux premières boucles durent 500.000/2 = 250.000 cycles.

Chaque instruction ajoutée dans la boucle1 est exécutée 256*256 fois. Chaque instruction ajoutée dans la boucle 2 est exécutée 256 fois. Chaque cycle extérieur aux boucles est exécuté 1 fois. Nous avons donc la possibilité de réaliser des temporisations très précises. Ce n'est pas nécessaire ici. Cependant, nous allons quand même améliorer la précision.

Si nous ajoutons 1 cycle inutile dans la boucle1, nous ajouterons 256*256 = 65536 cycles. Nous devons en ajouter 250.000 – 196608 = 53392. Cela nous donnerait une erreur de 12000 cycles, soit 12 millièmes de seconde (12ms). La précision est largement suffisante pour notre application, mais je rappelle que vous pouvez affiner cette précision en effectuant les calculs précis. Je vous conseille même de le faire vous-même comme exercice. Vous pourrez vérifier vos résultats avec un chronomètre.

Pour perdre un cycle, nous ajouterons simplement l'instruction NOP, qui ne fait rien.

Reste donc à réaliser la dernière boucle de 2. Créons une troisième variable cmpt3 et une troisième boucle. (Rassurez-vous, il y a des méthodes plus simples, ceci est expliqué dans un but didactique : il importe simplement d'avoir compris).

```
;*****************************************************************
;                     DECLARATIONS DE VARIABLES                  *
;*****************************************************************
   CBLOCK 0x00C    ; début de la zone variables

   cmpt1 : 1             ; compteur de boucles 1
   cmpt2 : 1             ; compteur de boucles 2
   cmpt3 : 1             ; compteur de boucles 3

        ENDC             ; Fin de la zone
```

Voici le code final :

```
;*****************************************************************
;                       SOUS-ROUTINE DE TEMPORISATION             *
;*****************************************************************
;-----------------------------------------------------------------
; Cette sous-routine introduit un retard de 500.000 µs.
; Elle ne reçoit aucun paramètre et n'en retourne aucun
;-----------------------------------------------------------------
tempo
   movlw    2            ; pour 2 boucles
   movwf    cmpt3        ; initialiser compteur3
boucle3
   clrf     cmpt2        ; effacer compteur2
boucle2
   clrf     cmpt1        ; effacer compteur1
boucle1
   nop                   ; perdre 1 cycle *256 *256 *2
   decfsz   cmpt1 , f    ; décrémenter compteur1
   goto     boucle1      ; si pas 0, boucler
   decfsz   cmpt2 , f    ; si 0, décrémenter compteur 2
   goto     boucle2      ; si cmpt2 pas 0, recommencer boucle1
   decfsz   cmpt3 , f    ; si 0, décrémenter compteur 3
   goto     boucle3      ; si cmpt3 pas 0, recommencer boucle2
   return                ; retour de la sous-routine
```

Lancez la compilation avec <F10>. Si tout s'est bien passé, vous obtenez dans votre répertoire de travail le fichier LED_CLI.HEX

Envoyez-le dans votre PIC® à l'aide de votre programmateur.

Placez le PIC® sur la carte HORS TENSION, envoyez l'alimentation et regardez bien :

VOTRE LED CLIGNOTE A LA FREQUENCE DE 1HZ.

Félicitations, voici votre premier programme embarqué. La porte est ouverte à des tonnes de nouvelles applications.

CONSEIL IMPORTANT

Si, à ce stade, vous sautez au plafond en criant « CA MARCHE » (j'ai connu ça), ne vous précipitez pas en hurlant pour faire admirer votre réalisation à votre épouse. Je doute qu'elle partage votre enthousiasme pour « cette petite lampe qui clignote ».

Le fichier tel qu'il devrait être à la fin de cette leçon est disponible dans les fichiers joints.

Notes…

12. Les interruptions

Voilà un chapitre qui fera certainement peur à beaucoup de futurs programmeurs. Pourtant, le mécanisme des interruptions est très aisé à comprendre, et également à mettre en œuvre, pour autant que l'on travaille de manière propre et structurée.

12.1 Qu'est-ce qu'une interruption ?

Imaginez une conversation normale :

- Chaque interlocuteur prend la parole quand vient son tour de parler.
- Survient alors un événement extérieur dont le traitement est urgent. Par exemple, un piano tombe du 3ème étage de l'immeuble au pied duquel vous discutez.
- Vous imaginez bien que votre interlocuteur ne va pas attendre la fin de votre phrase pour vous signaler le danger. Il va donc vous INTERROMPRE durant le cours normal de votre conversation., afin de pouvoir TRAITER IMMEDIATEMENT l'EVENEMENT extérieur.

Les interlocuteurs reprendront leur conversation où elle en était arrivée, sitôt le danger écarté (s'ils ont évité le piano, bien sûr).

Et bien, pour les programmes, c'est exactement le même principe :

- Votre programme se déroule normalement.
- Survient un événement spécifique.
- Le programme principal est interrompu (donc, subit une INTERRUPTION), et va traiter l'événement, avant de reprendre le programme principal à l'endroit où il avait été interrompu.

L'interruption est donc une RUPTURE DE SEQUENCE ASYNCHRONE, c'est à dire non synchronisée avec le déroulement normal du programme.

Vous voyez ici l'opposition avec les ruptures de séquences synchrones, provoquées par le programme lui-même (goto, call, btfss…).

12.2 Mécanisme général d'une interruption

Nous pouvons dire, sans nous tromper de beaucoup, qu'une routine d'interruption est un sous-programme particulier, déclenché par l'apparition d'un événement spécifique. Cela a l'air un peu ardu, mais vous allez voir que c'est très simple.
Voici donc comment cela fonctionne :

- Le programme se déroule normalement
- L'événement survient
- Le programme achève l'instruction en cours de traitement
- Le programme saute à l'adresse de traitement de l'interruption
- Le programme traite l'interruption

- Le programme saute à l'instruction qui suit la dernière exécutée dans le programme principal.

Il va bien sûr de soi que n'importe quel événement ne peut pas déclencher une interruption. Il faut que 2 conditions principales soient remplies :

- L'événement en question doit figurer dans la liste des événements susceptibles de provoquer une interruption pour le processeur sur lequel on travaille
- L'utilisateur doit avoir autorisé l'interruption, c'est à dire doit avoir signalé que l'événement en question devait générer une interruption.

Organigramme général de l'exécution d'une interruption.

Que pouvons-nous dire en voyant cet ordinogramme ? Et bien, nous pouvons déjà nous dire que le programme principal ne sait pas quand il est interrompu, il est donc crucial de lui remettre ses registres dans l'état où ils étaient avant l'interruption.

En effet, supposons que l'instruction xxx ait positionné un flag (par exemple, le bit d'indicateur Z). Si par malheur, la routine d'interruption a modifié ce bit, le programme ne pourra pas se poursuivre normalement.

Nous voyons également que l'instruction xxx termine son exécution avant de se brancher sur la routine d'interruption. Une instruction commencée n'est donc jamais interrompue.

12.3 Mécanisme d'interruption sur les PIC®

Bien entendu, les PIC® répondent au fonctionnement général ci-dessus, mais ils ont également leurs particularités. Voyons maintenant le principe des interruptions sur les PIC®

- Tout d'abord, l'adresse de début de toute interruption est fixe. Il s'agit toujours de l'adresse 0x04.

- Toute interruption provoquera le saut du programme vers cette adresse. Toutes les sources d'interruption arrivant à cette adresse, si le programmeur utilise plusieurs sources d'interruptions, il faudra qu'il détermine lui-même laquelle il est en train de traiter.

- Les PIC® en se connectant à cette adresse, ne sauvent rien automatiquement, hormis le contenu du PC, qui servira à connaître l'adresse du retour de l'interruption. C'est donc à l'utilisateur de se charger des sauvegardes.

- Le contenu du PC est sauvé sur la pile interne (8 niveaux). Donc, si vous utilisez des interruptions, vous ne disposez plus que de 7 niveaux d'imbrication pour vos sous-programmes. Moins si vous utilisez des sous-programmes dans vos interruptions.

Le temps de réaction d'une interruption est calculé de la manière suivante :

- Le cycle courant de l'instruction est terminé.

- Le flag d'interruption est lu au début du cycle suivant.

- Celui-ci est achevé, puis le processeur s'arrête un cycle pour charger l'adresse 0x04 dans PC.

- Le processeur se connecte alors à l'adresse 0x04 où il lui faudra un cycle supplémentaire pour charger l'instruction à exécuter.

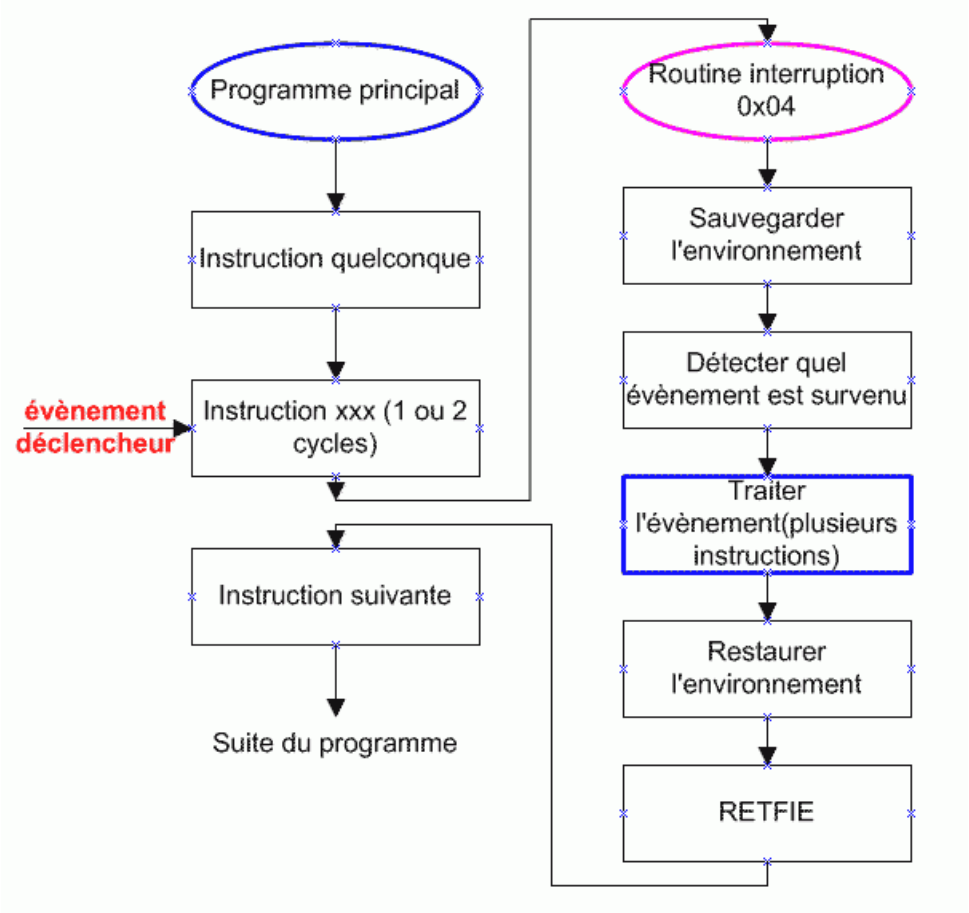
Le temps mort total sera donc compris entre 3 et 4 cycles.

Remarquez que :

- Une interruption ne peut pas être interrompue par une autre interruption. Les interruptions sont donc invalidées automatiquement lors du saut à l'adresse 0x04 par l'effacement du bit GIE (que nous allons voir).

- Les interruptions sont remises en service automatiquement lors du retour de l'interruption. L'instruction RETFIE agit donc exactement comme l'instruction RETURN, mais elle repositionne en même temps le bit GIE.

Les interruptions sur les PIC®



12.4 Les sources d'interruptions du 16F84

Le 16F84 est très pauvre à ce niveau, puisqu'il ne dispose que de 4 sources d'interruptions possibles (contre 13 pour le 16F876 par exemple). Les événements susceptibles de déclencher une interruption sont les suivants :

- TMR0 : Débordement du timer0 (tmr0). Une fois que le contenu du tmr0 passe de 0xff à 0x00, une interruption peut être générée. Nous utiliserons ces propriétés dans le chapitre sur le timer 0.

- EEPROM : cette interruption peut être générée lorsque l'écriture dans une case EEPROM interne est terminée. Nous verrons ce cas dans le chapitre sur l'écriture en zone eeprom.

- RB0/INT : Une interruption peut être générée lorsque, la pin RB0, encore appelée INTerrupt pin, étant configurée en entrée, le niveau qui est appliqué est modifié. Nous allons étudier ce cas ici.

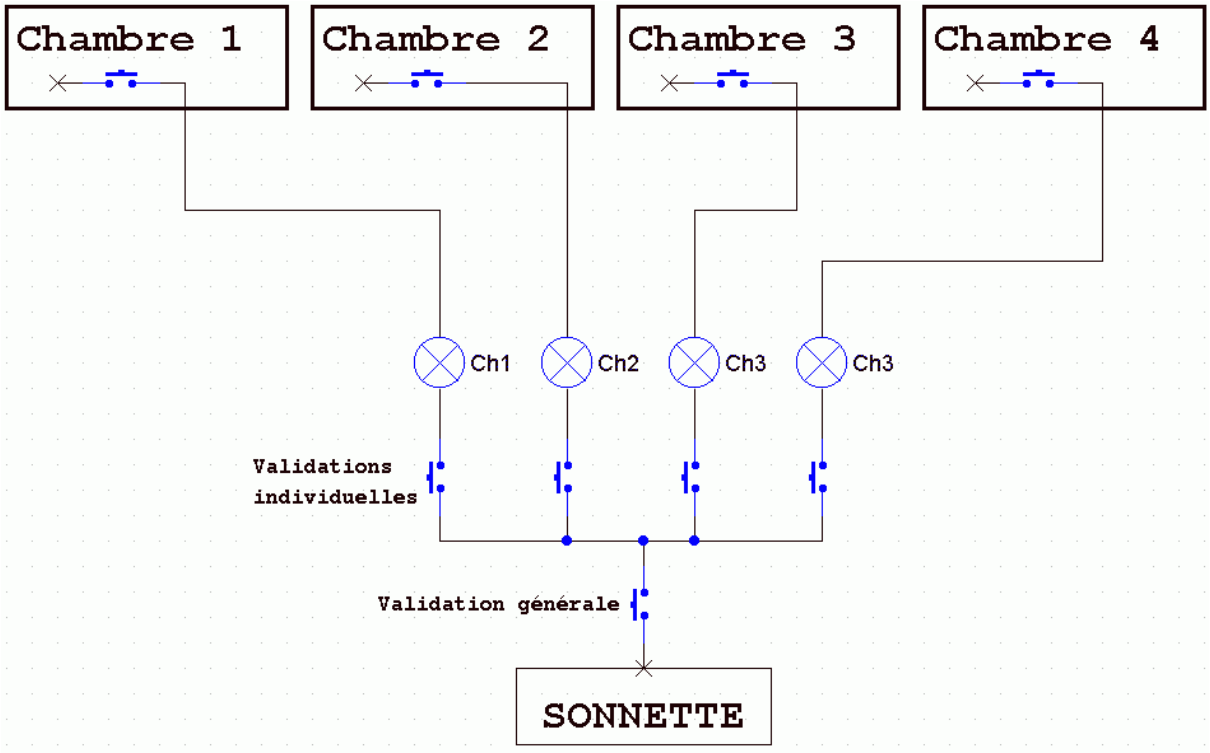
- PORTB : De la même manière, une interruption peut être générée lors du changement d'un niveau sur une des pins RB4 à RB7. Il n'est pas possible de limiter l'interruption à une seule de ces pins. L'interruption sera effective pour les 4 pins ou pour aucune.

12.5 Les dispositifs mis en œuvre

Comment interdire ou autoriser les interruptions, comment détecter quel est l'événement déclencheur, et comment les gérer ? Nous allons aborder ceci d'abord de manière symbolique, pour vous aider à bien visualiser la procédure.

Imaginons un hôtel. Le groom de service représente notre programme.
L'hôtel comporte 4 chambres, et chaque chambre est équipée d'un bouton-poussoir. Chaque bouton-poussoir servant à appeler le groom, est relié à une lampe à l'accueil de l'hôtel.

Chaque lampe a la possibilité de faire résonner une sonnette si l'interrupteur général de la sonnette est positionné sur ON, et si l'interrupteur particulier reliant chaque lampe à la sonnette est enclenché. Voilà donc le schéma obtenu (attention, c'est un schéma symbolique, pas électrique, les lignes représentent la circulation de l'information, pas un courant électrique) :



Quand vous comprendrez bien ce petit schéma symbolique, vous aurez compris les interruptions. Vous voyez tout de suite qu'il y a deux méthodes pour que le groom soit prévenu.

Soit via la lampe, soit via la sonnette.

- Pour la première méthode, le groom doit venir voir les lampes à intervalles réguliers pour vérifier si personne n'a appelé. Il doit donc venir SCRUTER le tableau de signalisation. C'est la méthode de SCRUTATION.

- Avec la sonnette, le groom est INTERROMPU dans son travail par celle-ci. Il n'a pas besoin de venir scruter inutilement, mais il sera dérangé dans son travail par la sonnette. C'est la méthode des INTERRUPTIONS Notez déjà les points suivants :

- Le locataire de la chambre ne peut décider quelle méthode sera utilisée, c'est le groom qui décide de valider ou non les sonnettes.

- Le groom peut inhiber toutes les sonnettes en une seule fois, ou décider quelle chambre va pouvoir actionner la sonnette.

- Les interrupteurs de validation n'agissent pas sur les lampes.

- Si le groom est interrompu dans son travail par la sonnette, il doit de toute façon aller regarder les lampes pour voir qui a sonné. Sauf s'il sait qu'il n'a autorisé qu'une seule chambre à actionner la sonnette.

- Ce qui n'apparaît pas sur ce schéma simplifié, mais qu'il faut savoir, c'est que, une fois la lampe allumée, elle le reste. C'est le groom qui doit l'éteindre manuellement.

Mettons donc tout ceci en pratique sur le 16F84. Vous avez déjà compris que lampes et interrupteurs étaient, dans le 16F84, des bits de registres particuliers. Voici maintenant le registre principal de gestion des interruptions pour le 16F84.

12.6 Le registre INTCON (INTerrupt CONtrol)

Ce registre se situe à l'adresse 0x0B, dans les 2 banques. Il est donc toujours accessible. Il est détaillé figure 4-5 page 17. C'est un registre de bits, donc, chaque bit a une fonction particulière. Voici le détail de ces bits :

b7 : GIE

Global Interrupt Enable bit. Il permet de valider ou d'invalider toutes les interruptions d'une seule fois. Ce bit correspond donc à notre interrupteur de validation générale.

b6 : EEIE

Eeprom write complete Interrupt Enable bit. Ce bit permet de valider l'interruption de fin d'écriture en eeprom (nous étudierons plus tard le mécanisme d'écriture eeprom).

b5 : T0IE

Tmr0 Interrupt Enable bit : Valide l'interruption générée par le débordement du timer0.

b4 : INTE

INTerrupt pin Enable bit : Valide l'interruption dans le cas d'une modification de niveau de la pin RB0.

ATTENTION : rappelez-vous le bit 6 du registre OPTION, qui détermine quel est le sens de transition qui provoque l'interruption. On pourra donc choisir si c'est une transition 0->1 ou 1->0 qui provoque l'interruption, mais pas les deux ensemble.

b3 : RBIE

RB port change Interrupt Enable bit : Valide les interruptions si on a changement de niveau sur une des entrées RB4 à RB7.

b2 : T0IF

Tmr0 Interrupt Flag bit. C'est un Flag, donc il signale. Ici c'est le débordement du timer0

b1 : INTF

INTerrupt pin Flag bit : signale une transition sur la pin RB0 dans le sens déterminé par INTEDG du registre OPTION (b6)

b0 : RBIF

Port Interrupt Flag bit : signale qu'une des entrées RB4 à RB7 a été modifiée.

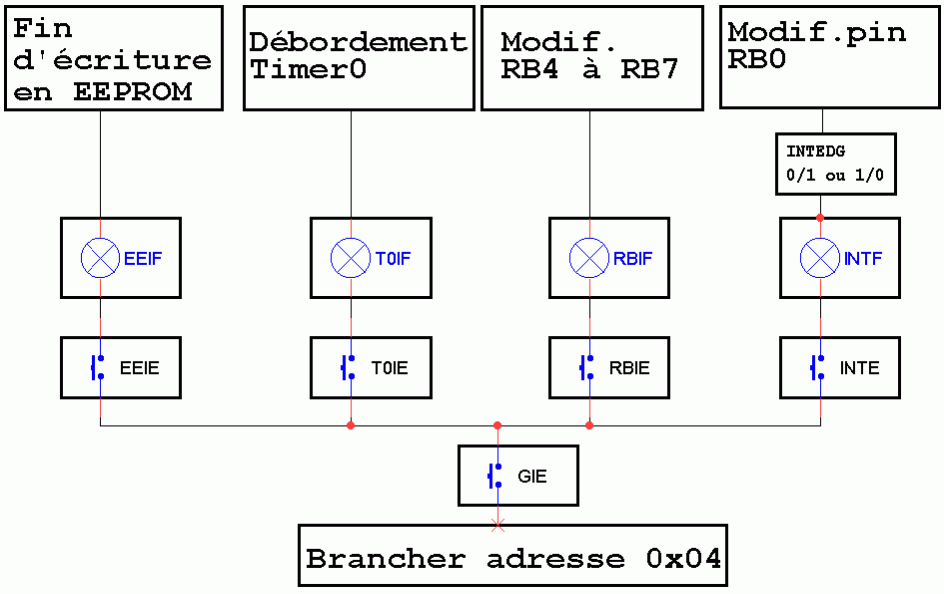
Remarque

Rappelez-vous que les flags ne se remettent pas à 0 tout seuls. C'est votre programme qui doit s'en charger, sous peine de rester indéfiniment bloqué dans une routine d'interruption. Nous dirons que ces flags sont des FLAGS REMANENTS. Prenez garde que le reset de RBIF doit être précédé d'une lecture ou d'une écriture du PORTB afin de mettre fin à la condition de génération du flag d'interruption.

Remarquez déjà que tous les bits dont le nom se termine par E (Enable) sont en fait des commutateurs de validation (ce sont les interrupteurs de notre schéma symbolique). Les bits dont le nom se termine par F sont des Flags (indicateurs). Ils sont représentés par des lampes sur notre schéma symbolique. Sur celui-ci nous avons 5 interrupteurs et 4 ampoules. Or nous n'avons que 8 bits dans INTCON0. Que nous manque-t-il ?

Si vous regardez attentivement, il nous manque le bit EEIF. Mais, rassurez-vous, ce bit existe bien, il est tout simplement dans le registre EECON1, que nous verrons dans la leçon sur les accès EEPROM. Nous allons donc transformer notre schéma symbolique pour qu'il corresponde à la réalité d'un PIC® 16F84.

Après ces explications détaillées, trop diront les habitués des processeurs, vous devez maintenant avoir compris le fonctionnement des interruptions sur le 16F84. Analysons maintenant quelques points de la routine d'interruption en elle-même. Pour rester concrets, nous allons mélanger théorie et pratique.



12.7 Sauvegarde et restauration de l'environnement

Si vous regardez de nouveau l'organigramme de la routine d'interruption, vous constatez que vous devez procéder à la sauvegarde et à la restauration de l'environnement de votre programme. En quoi cela consiste-t-il ?

Et bien, comme nous l'avons déjà dit, votre programme interrompu ne sait pas qu'il l'a été. Vous devez donc remettre les registres dans l'état où ils se trouvaient au moment de l'interruption, et ce pour permettre à votre programme de continuer à fonctionner correctement une fois le traitement de l'interruption terminé.

Petit exemple :

Supposons votre programme principal interrompu entre les 2 instructions suivantes :

```
movf     mavariable , w     ; charger mavariable et positionner Z
btfss    STATUS , Z         ; tester bit Z et sauter si vaut 1
```

Il est plus que probable que votre routine d'interruption va utiliser au moins une instruction qui modifie le bit Z. Vous devrez donc restaurer le registre STATUS dans l'état qu'il était avant de sortir de la routine d'interruption. Sinon le test ne se fera pas sur la valeur de mavariable, mais sur une valeur de Z modifiée par la routine d'interruption.

De plus, il est à peut près certain que le registre « w » va être modifié également, donc il vous faudra veiller à le sauvegarder également.

12.7.1 Les registres à sauvegarder

Pour commencer, le plus simple : le PC est sauvegardé automatiquement, le programme revient donc à la bonne adresse tout seul.

Par contre, STATUS est, comme nous venons de voir, à restaurer par la routine d'interruption.

W sera également modifié par la routine d'interruption. S'il y a d'autres registres à sauvegarder, ce sera en fonction du fonctionnement de votre programme. Les registres à sauver obligatoirement dans 99,9% des cas sont donc STATUS et W. Si votre routine d'interruption ne contient que des instructions ne modifiant ni STATUS ni W (bsf, etc), vous pourrez vous passer de la sauvegarde, mais il s'agit d'un cas particulier.

Si, dans la routine d'interruption et dans le programme principal, nous utilisons l'adressage indirect, nous devrons également restaurer FSR.

12.7.2 La méthode de sauvegarde

Pour sauver les registres W et STATUS on utilise une méthode tout à fait classique. Il importe de commencer par sauver W avant STATUS, puisque la sauvegarde de STATUS impose de d'abord charger ce registre dans le registre de travail, ce qui induit donc la perte du registre W.

La sauvegarde se résume donc à :

```
movwf     w_temp            ; sauve W dans un emplacement de sauvegarde
movf      STATUS,w          ; transfère STATUS dans W
movwf     status_temp       ; sauvegarde de STATUS
```

Notez que les datasheets et les documents de référence, comme le mid-range reference manual, indiquent la méthode suivante :

```
movwf     w_temp            ; sauve W dans un emplacement de sauvegarde
swapf     STATUS,w          ; transfère STATUS dans W
movwf     status_temp       ; sauvegarde de STATUS
```

Vous constatez qu'au lieu d'un movf, Microchip recommande l'utilisation d'un swapf, qui ne modifie aucun bit de STATUS dans la manœuvre. Evidemment, l'octet chargé est « swappé » et donc il faudra également utiliser un swap lors de la restauration.

Après interrogation sur ce point précis, je n'ai pas pu obtenir d'information auprès de Microchip sur la raison précise de cette méthode conseillée (peut-être le résidu d'un ancien bug concernant les manipulations de STATUS ?). Il semble même que cette méthode ait disparu des recommandations officielles Microchip. Néanmoins, les premières versions de ce cours ont utilisé cette méthode recommandée, ce qui fait que c'est cette façon de procéder que vous retrouverez dans les sources et documents. Sachez simplement que vous pouvez utiliser la méthode qui vous convient, ça ne change strictement rien.

Attention : ceci ne vaut QUE pour le registre STATUS. En ce qui concerne la restauration de W, les remarques qui vont suivre restent d'application.

12.7.3 La méthode de restauration

Vous allez me dire : mais que dire à ce sujet, il suffit de procéder de façon inverse, à savoir :

```
movf      status_temp,w     ; charge le STATUS sauvegardé
movwf     STATUS            ; restaurer STATUS
movf      w_temp,w          ; restaurer W
```

Tout semble simple, et pourtant se cache ici un piège grossier. En effet, la dernière instruction « movf » modifie les flags de STATUS. Donc, en restaurant le registre w, vous détruisez votre restauration de STATUS. Ceci est évidemment inacceptable, c'est pourquoi il est nécessaire d'utiliser une autre méthode.

L'astuce est d'utiliser pour restaurer W une instruction qui ne modifie aucun bit du registre STATUS. Un coup d'œil sur les instructions disponibles nous permet de voir que l'instruction « swapf » amène la variable dans W SANS modifier aucun bit de STATUS.

Cependant, cette instruction présente l'inconvénient d'inverser les 4 bits de poids fort avec les 4 bits de poids faible de l'octet désigné, ce qui conduirait à une restauration incorrecte. Il sera donc nécessaire pour annuler cette inversion d'utiliser 2 instructions « swapf » consécutives, ce qui remettra l'octet dans le bon ordre.

La routine modifiée se présente alors comme suit :

```
    movf      status_temp,w      ; charge le STATUS sauvegardé
    movwf     STATUS             ; restaurer STATUS
    swapf     w_temp,f           ; inverse l'emplacement de sauvegarde
    swapf     w_temp,w           ; réinverse et amène le résultat dans W
```

De cette façon, W est bel et bien restauré sans avoir modifié le registre STATUS précédemment restauré.

Bref, voici les deux façons de procéder, qui sont strictement équivalentes,
La méthode « intuitive » :

```
sauvegarde
    movwf     w_temp             ; sauve W dans un emplacement de sauvegarde
    movf      STATUS,w           ; transfère STATUS dans W
    movwf     status_temp        ; sauvegarde de STATUS
traitement_interrupt
    …
    …
    …
restauration
    movf      status_temp,w      ; charge le STATUS sauvegardé
    movwf     STATUS             ; restaurer STATUS
    swapf     w_temp,f           ; inverse l'emplacement de sauvegarde de W
    swapf     w_temp,w           ; réinverse et amène le résultat dans W
```

Ou la méthode préconisée à l'origine, et utilisée dans le cours :

```
sauvegarde
    movwf     w_temp             ; sauve W dans un emplacement de sauvegarde
    swapf     STATUS,w           ; transfère STATUS dans W
    movwf     status_temp        ; sauvegarde de STATUS
traitement_interrupt
    …
    …
    …
restauration
    swapf     status_temp,w      ; charge le STATUS sauvegardé
    movwf     STATUS             ; restaurer STATUS
    swapf     w_temp,f           ; inverse l'emplacement de sauvegarde
    swapf     w_temp,w           ; réinverse et amène le résultat dans W
```

Il n'y a en fait aucune différence pratique entre les deux, mais il importe de vous souvenir que l'utilisation de 2 swapf pour restaurer W reste obligatoire.

Voici donc la structure de base d'une routine d'interruption :

```
;*********************************************************************
;                     ROUTINE INTERRUPTION                           *
;*********************************************************************

      ;sauvegarder registres
      ;---------------------
   org      0x004             ; adresse d'interruption
   movwf    w_temp            ; sauver registre W
   swapf    STATUS,w          ; swap status avec résultat dans w
   movwf    status_temp       ; sauver status swappé

      ; switch vers différentes interrupts
      ; ----------------------------------
; ici, on teste éventuellement de quelle interruption il s'agit

      ; traitement des interruptions
      ; ----------------------------
; ici, on peut traiter  interruption puis effacer son flag

      ;restaurer registres
      ;-----------------------
   swapf    status_temp,w     ; swap ancien status, résultat dans w
   movwf    STATUS            ; restaurer status
   swapf    w_temp,f          ; Inversion L et H de l'ancien W
                              ; sans modifier Z
   swapf    w_temp,w          ; Ré-inversion de L et H dans W
                              ; W restauré sans modifier status
   retfie                     ; return from interrupt
```

Et voilà votre squelette de routine d'interruption : vraiment pas compliqué, n'est-ce pas ? Il n'y a plus qu'à compléter, car le fichier m16F84.asm contient déjà les routines de sauvegarde et restauration (que nous venons d'expliquer) et les tests de type d'interruption (que nous allons détailler).

Comme je suis partisan de la programmation structurée, la routine de switch branche en réalité sur des sous-programmes séparés. Rien n'empêche en effet d'utiliser des sous-programmes dans une routine d'interruption. Attention cependant à votre limite de pile disponible.

12.7.4 Opérations sur le registre STATUS

Je ne voudrais pas terminer cette partie sans attirer votre attention sur un point particulier, à savoir les instructions dont la destination est le registre STATUS.

Toute instruction dont le résultat est susceptible de modifier un ou plusieurs bit(s) du registre STATUS, et dont la cible est le registre STATUS lui-même, induit que chaque bit susceptible d'être modifié par l'instruction se retrouve en lecture seule.

Pour être plus explicite, quelques exemples :

```
movwf    STATUS       ; aucun problème, movwf ne modifie aucun flag
movf     STATUS,w     ; aucun problème, STATUS n'est pas la destination
andwf    STATUS       ; le bit Z sera positionné en fonction du résultat
addwf    STATUS       ; idem pour C, DC, et Z.
clrf     STATUS       ; modifie Z, et donc ne donne pas le résultat escompté
```

Soyez donc très prudent, et si la destination de votre instruction est le registre STATUS, utilisez de préférence un movwf, un swapf, ou les instructions de manipulation de bits (bcf, bsf).

12.7.5Particularité de l'instruction « RETFIE »

A ce niveau de l'exposé, une remarque pertinente serait la suivante :
Pourquoi existe-t-il une instruction RETFIE, alors qu'on pourrait utiliser RETURN ?

Et bien, vous ne pouvez pas interrompre une interruption par une autre. Si c'était le cas, les sauvegardes des registres W et STATUS seraient « écrasées » par une seconde opération (mais c'est possible sur d'autres processeurs et même d'autres PIC®). Donc, dès que le programme est branché sur l'interruption, le bit GIE est mis à 0 automatiquement.

Pour qu'une nouvelle interruption puisse avoir lieu une fois celle en cours terminée, il faut remettre GIE à 1. Ceci est exécuté automatiquement par RETFIE.

Vous allez alors me dire : et si je fais ceci ?

```
bsf       INTCON , GIE       ; remettre GIE à 1
return                       ; et sortir de la routine d'interruption
```

Et bien, c'est exactement ce que fait RETFIE, mais à un détail près, et ce détail est de la plus grande importance : RETFIE est une seule et même instruction, donc ne peut pas être interrompu par une interruption.

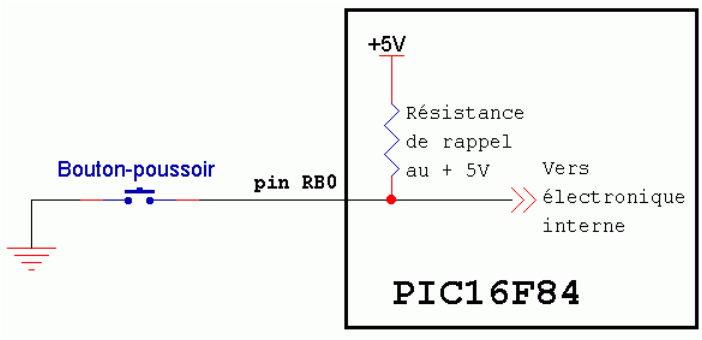
Dans le cas où les 2 instructions précédentes seraient utilisées, une fois GIE mis à un, si un des flags d'interruption est toujours à 1 (autre interruption, où la même qui se reproduit une autre fois), le programme se reconnecterait sur la routine d'interruption avant d'avoir exécuté le RETURN, donc avant d'avoir restauré le PC.

Celui-ci continuerait à occuper un emplacement sur la pile. En sachant que la pile est limitée à 8 emplacements, il y aurait de nombreuses chances de plantage du programme par débordement de la pile.

12.8 Utilisation d'une routine d'interruption

Effectuez une copie de votre fichier m16f84.asm. Renommez-la en « myinter.asm ».
Créez un nouveau projet MPLAB® avec le nom « myinter » suivant le processus habituel.

Nous allons construire un programme qui inverse l'allumage d'une LED à chaque pression sur un bouton-poussoir. Modifiez votre petite platine d'expérimentation de façon à connecter le bouton-poussoir sur l'entrée RB0 (désolé, quand j'ai dessiné le schéma, je pensais expliquer les interruptions avec le timer).



Ceci vous montre cependant que l'électronique et la programmation avec les microcontrôleurs sont très dépendants. Quand vous construisez un circuit, vous devez déjà penser à la manière dont vous allez réaliser la programmation. A l'inverse, si vous disposez d'un circuit déjà existant, des contraintes vous sont déjà imposées au niveau de la programmation. Il est par exemple impossible de traiter notre bouton-poussoir par interruption s'il est câblé sur l'entrée RB2.

La résistance est interne au PIC®, il s'agit d'une des résistances de rappel au +5V (pull-up) qu'on peut activer par logiciel.

Comme d'habitude, remplissez le cadre d'en-tête, et supprimez les lignes inutilisées dans les variables, macro, et DEFINE. Attention, ne supprimez pas les variables w_temp et status_temp..

```
#include <p16F84.inc>        ; Définitions des constantes
```

Voici l'en-tête tel que vous pourriez le compléter :

```
;*****************************************************************
;    Ce programme est un programme didactique destiné à monter     *
;    le fonctionnement des interruptions                           *
;                                                                  *
;*****************************************************************
;                                                                  *
;    NOM:       Interruption par bouton-poussoir sur RB0           *
;    Date:      13/02/2001                                         *
;    Version:   1.0                                                *
;    Circuit:   Platine d'essais                                   *
;    Auteur:    Bigonoff                                           *
;                                                                  *
;*****************************************************************
;                                                                  *
;    Fichier requis: P16F84.inc                                    *
;                                                                  *
;                                                                  *
;                                                                  *
;*****************************************************************
;    Notes:  Ce programme transforme un bouton-poussoir en         *
;            télérupteur. Un pulse allume la LED, un autre         *
;            l'éteint                                              *
;*****************************************************************
```

Vous allez dire que j'exagère en vous faisant mettre des commentaires partout. Croyez-moi sur parole, les commentaires permettent une maintenance aisée du programme. Ils vous feront gagner à coup sûr beaucoup plus de temps qu'ils ne vous en feront perdre.

Modifiez la config pour supprimer le fonctionnement du watch-dog :

```
__CONFIG    _CP_OFF & _WDT_OFF & _PWRTE_ON & _HS_OSC
```

Calculons la future valeur à envoyer dans le registre OPTION

- b7 à 0, car on a besoin de la résistance de rappel au +5V pour le bouton-poussoir.
- b6 à 0, car on veut une interruption quand presse le bouton, donc quand le niveau passe de 1 à 0 (flanc descendant)
- b5/b0 à 0, aucune importance dans cette application.

Donc, nous placerons notre assignation à B'00000000', donc :

```
;*****************************************************************
;                              ASSIGNATIONS                      *
;*****************************************************************

OPTIONVAL           EQU    H'00'        ; Valeur registre option
                                        ; Résistance pull-up ON
                                        ; Interrupt flanc descendant RB0
```

Pour le registre INTCON, nous devrons avoir :

- b7 à 1 : pour valider les interruptions
- b6 à 0 : pas d'interruption EE
- b5 à 0 : pas d'interruption tmr0
- b4 à 1 : interruption RB0 en service
- b3 à 0 : Pas d'interruption RB
- b2/b0 à 0 : effacer les flags

Cela donne : B'10010000', soit 0x90
Assignons une constante pour cette valeur ;

```
INTERMASK           EQU    H'90'        ; Masque d'interruption
                                        ; Interruptions sur RB0
```

Ensuite, dans la zone des DEFINE, nous définirons notre LED et notre bouton-poussoir :

```
;*****************************************************************
;                              DEFINE                            *
;*****************************************************************
#DEFINE     Bouton  PORTB , 0   ; bouton poussoir
#DEFINE     LED     PORTA , 2   ; LED
```

Dans la zone macros , nous pouvons écrire 2 macros que nous utiliserons souvent. Ce sont les instructions pour passer en banque 0 et banque1. Elles seront incluses dans le nouveau fichier « m16f84.asm » nommé « m16f84_new.asm ». A la fin de cette leçon, supprimez l'ancien fichier et renommez « m16F84_new » en « m16f84 ».

```
;*****************************************************************
;                              MACRO                              *
;*****************************************************************
BANK0  macro
      bcf     STATUS , RP0       ; passer en banque 0
      endm

BANK1  macro
      bsf     STATUS , RP0       ; passer en banque1
      endm
```

Venons-en à la zone des variables. Nous devons garder w_temp et status_temp, car ces variables sont utilisées dans la routine d'interruption pour sauver les registres W et STATUS.

Nous allons également récupérer notre petite routine de tempo de notre fichier « led_cli.asm ». Nous aurons donc besoin des variables utilisées dans cette routine. Tout ceci nous donne donc pour l'instant :

```
;*****************************************************************
;                   DECLARATIONS DE VARIABLES                     *
;*****************************************************************
   CBLOCK 0x00C          ; début de la zone variables

   w_temp :1             ; Sauvegarde de W dans interruption
   status_temp : 1       ; Sauvegarde de STATUS dans interrupt
   cmpt1 : 1             ; compteur de boucles 1 dans tempo
   cmpt2 : 1             ; compteur de boucles 2 dans tempo
   cmpt3 : 1             ; compteur de boucles 3 dans tempo

   ENDC                  ; Fin de la zone
```

12.9 Analyse de la routine d'interruption

Nous avons déjà vu la première partie, qui est la sauvegarde des registres utilisés

```
;*****************************************************************
;                     ROUTINE INTERRUPTION                       *
;*****************************************************************

                  ;sauvegarder registres
                  ;---------------------
   org 0x004                  ; adresse d'interruption
   movwf    w_temp            ; sauver registre W
   swapf    STATUS , w        ; swap status avec résultat dans w
   movwf    status_temp       ; sauver status swappé
```

Ensuite, nous devons déterminer quelle est l'origine de l'interruption en cours. Dans le cas présent, il s'agit obligatoirement de l'interruption RB0/INT, car nous n'avons autorisé que celle-là. Dans le cas présent, nous pourrions donc nous passer de ce test. Mais le but poursuivi est de vous expliquer les méthodes à utiliser. Nous garderons donc la totalité des explications afin que vous puissiez utiliser n'importe quelle combinaison. Examinons donc cette partie :

```
            ; switch vers différentes interrupts
            ; inverser ordre pour modifier priorités
            ;---------------------------------------

    btfsc   INTCON,T0IE         ; tester si interrupt timer autorisée
    btfss   INTCON,T0IF         ; oui, tester si interrupt timer en cours
    goto    intsw1              ; non test suivant
    call    inttimer            ; oui, traiter interrupt timer
    goto    restorereg          ; et fin d'interruption
                                ; SUPPRIMER CETTE LIGNE POUR
                                ; TRAITER PLUSIEURS INTERRUPT
                                ; EN 1 SEULE FOIS
```

Les 2 premières instructions examinent si nous avons affaire à une interruption tmr0. Vous allez me dire : pourquoi 2 lignes ? Il suffit d'examiner T0IF, tout simplement.

Et bien, ce n'est pas si simple. Imaginons en effet que le tmr0, que nous n'utilisons pas, ait débordé. Le bit T0IF est donc mis à 1 et n'a pas généré d'interruption, car T0IE est à 0. De même, si nous avions accepté les interruptions timer et que l'interruption soit due à une autre cause, nous aurions T0IE à 1 et T0IF à 0.

Nous ne devons donc traiter l'interruption que si l'interruption est en service, et que si le flag est positionné, d'où le double test. Examinez ce double test et observez le fonctionnement des btfsc et btfss :

Si T0IE vaut 0, on saute directement au test suivant par la ligne goto. S'il vaut 1, on teste ensuite T0IF. Si celui-ci vaut 0, on arrive à la ligne goto qui passe au test suivant. S'il vaut également 1, on appelle la sous-routine de traitement de l'interruption timer0.

La dernière ligne permet de sauter à la fin de la routine d'interruption, donc de restaurer les registres et de sortir. Donc, dans le cas où on aurait deux sources d'interruptions simultanées, une seule serait traitée à la fois. Si par contre on supprime cette ligne, l'interruption suivante sera traitée dès celle en cours terminée. J'ai donc étudié cette ossature pour vous laisser toutes les variantes possibles, et directement utilisables.

Remarques

- Si on n'utilise jamais l'interruption tmr0, on peut supprimer cette partie de code

- Si l'interruption tmr0 était en service tout au long du programme, on pourrait supprimer le test de T0IE (car il serait tout le temps à 1)

- En cas de 2 interruptions simultanées de 2 événements distincts, la première interruption traitée sera celle testée en premier. L'ordre des tests modifie donc la priorité des interruptions.

Ensuite, nous trouvons la même procédure pour les interruptions de type RB0 (dont nous allons nous servir) et d'interruption RB4/RB7. Remarquez l'emplacement prévu pour ajouter le test pour l'interruption eeprom. Nous compléterons m16F84.asm à ce niveau en étudiant les procédures d'écriture en eeprom. Toutes ces modifications ont été incluses dans le fichier « m16f84_new.asm ».

```
intsw1
   btfsc    INTCON , INTE      ; tester si interrupt RB0 autorisée
   btfss    INTCON , INTF      ; oui, tester si interrupt RB0 en cours
   goto     intsw2             ; non sauter au test suivant
   call     intrb0             ; oui, traiter interrupt RB0
   bcf      INTCON,INTF        ; effacer flag interupt RB0
   goto     restorereg         ; et fin d'interruption
                               ; SUPPRIMER CETTE LIGNE POUR
                               ; TRAITER PLUSIEURS INTERRUPT
                               ; EN 1 SEULE FOIS
intsw2
   btfsc    INTCON,RBIE        ; tester si interrupt RB4/7 autorisée
   btfss    INTCON,RBIF        ; oui, tester si interrupt RB4/7 en cours
   goto     intsw3             ; non sauter
   call     intrb4             ; oui, traiter interrupt RB4/7
   bcf      INTCON,RBIF        ; effacer flag interupt RB4/7
   goto     restorereg         ; et fin d'interrupt
intsw3
```

; ici, la place pour l'interruption eeprom

Enfin, nous trouvons la partie servant à la restauration des registres sauvegardés. Nous avons déjà vu cette procédure :

```
                        ;restaurer registres
                        ;-------------------
restorereg
   swapf    status_temp , w    ; swap ancien status, résultat dans w
   movwf    STATUS             ; restaurer status
   swapf    w_temp , f         ; Inversion L et H de l'ancien W
                               ; sans modifier Z
   swapf    w_temp , w         ; Ré-inversion de L et H dans W
                               ; W restauré sans modifier status
   retfie                      ; return from interrupt
```

12.10 Adaptation de la routine d'interruption

Nous allons maintenant modifier cette routine d'interruption pour l'adapter à notre cas précis. Nous n'avons qu'une seule source d'interruption validée, donc, si nous entrons dans cette interruption, ce sera forcément pour traiter INT/RB0. Supprimons donc les tests.

Il nous reste donc :

```
;*****************************************************************
;                     ROUTINE INTERRUPTION                      *
;*****************************************************************

      ;sauvegarder registres
      ;---------------------
   org 0x004                  ; adresse d'interruption
   movwf    w_temp            ; sauver registre W
   swapf    STATUS , w        ; swap status avec résultat dans w
   movwf    status_temp       ; sauver status swappé
   call     intrb0            ; traiter interrupt RB0

               ;restaurer registres
               ;-------------------

   swapf    status_temp , w   ; swap ancien status, résultat dans w
   movwf  STATUS              ; restaurer status
   swapf  w_temp , f          ; Inversion L et H de l'ancien W
                              ; sans modifier Z
   swapf  w_temp , w          ; Ré-inversion de L et H dans W
                              ; W restauré sans modifier status
   retfie                     ; return from interrupt
```

Nous pourrions également nous passer de la ligne call intrb0, et placer la procédure de traitement directement à sa place. Conservons cependant cet appel de sous-routine, car nous ne sommes pas à quelques instructions près. En cas où vous devriez optimiser une application à l'extrême, vous pourriez y penser. Dans le cas contraire, je vous conseille de donner priorité à la lisibilité de votre programme (oui, j'insiste encore).

Voyons maintenant la suite du programme. Nous allons trouver les 3 routines de traitement des interruptions appelées à l'origine par notre routine d'interruption. Comme nous avons supprimé 2 de ces appels, inutile de conserver les sous-routines correspondantes. La seule dont nous aurons besoin sera donc :

```
;*****************************************************************
;                     INTERRUPTION RB0/INT                      *
;*****************************************************************
intrb0
   return           ; fin d'interruption RB0/INT
                    ; peut être remplacé par
                    ; retlw pour retour code d'erreur
```

Elle ne contient que le retour de sous-routine correspondant à l'appel du call intrb0. Attention, à ce niveau vous n'utiliserez pas RETFIE, car vous ne sortez pas de l'interruption, vous sortez d'une sous-routine appelée par la routine d'interruption.

Si vous utilisiez RETFIE à ce moment, vous remettriez les interruptions en service avant d'en sortir, donc plantage de votre programme.

Remarquez que vous pourriez utiliser retlw pour retourner une valeur prédéfinie traitée dans votre routine d'interruption. Par exemple, retlw0 si on veut traiter d'autres interruptions et retlw 1 si on sort de la routine. Dans ce cas, la ligne

```
   goto     restorereg        ; et fin d'interruption
```

correspondante pourrait être précédée du test de w retourné par la sous-routine. Ceci est un exemple un peu complexe à traiter ici, mais parfaitement envisageable lorsque vous jonglerez avec les PIC®.

12.11 L'initialisation

Poursuivons : nous trouvons la routine d'initialisation que nous avons déjà expliquée la leçon précédente. Nous nous contenterons de remplacer les changements de banque par les macros que nous avons écrites , et à configurer RA2 en sortie pour la LED.

Nous obtenons :

```
;*****************************************************************
;                       INITIALISATIONS                          *
;*****************************************************************

init
    clrf    PORTA               ; Sorties portA à 0
    clrf    PORTB               ; sorties portB à 0
    clrf    EEADR               ; permet de diminuer la consommation
    BANK1                       ; passer banque1
    movlw   OPTIONVAL           ; charger masque
    movwf   OPTION_REG          ; initialiser registre option

            ; Effacer RAM
            ; ----------------
    movlw   0x0c                ; initialisation pointeur
    movwf   FSR                 ; pointeur d'adressage indirect
init1
    clrf    INDF                ; effacer ram
    incf    FSR,f               ; pointer sur suivant
    btfss   FSR , 6             ; tester si fin zone atteinte (>=0x40)
    goto    init1               ; non, boucler
    btfss   FSR , 4             ; tester si fin zone atteinte (>=0x50)
    goto    init1               ; non, boucler

            ; configurer PORTS
            ; ---------------
    bcfLED                      ; RA2 en sortie (TRISA)

    BANK0                       ; passer banque 0
    movlw   INTERMASK           ; masque interruption
    movwf   INTCON              ; charger interrupt control
    goto    start               ; sauter programme principal
```

Il ne nous reste plus qu'à supprimer la ligne

```
    clrwdt                      ; effacer watch dog
```

du programme principal, puisque nous avons désactivé le watchdog.

Remarque

Une grande partie des erreurs dans les programmes est le fait d'erreurs de sélection de banques (surtout pour les PIC® à 4 banques). Je vous conseille d'adopter comme convention de toujours entrer dans une routine avec la banque 0, et de toujours s'assurer que vous êtes en banque 0 avant d'en sortir. En cas de dérogation, indiquez-le clairement dans l'en-tête de votre sous-routine.

Lancez la compilation pour vérifier que vous n'avez pas fait d'erreurs. Vous devriez obtenir ceci (au numéro de ligne près) :

```
Message[302] D:\DOCUME~1\LESSONS\DATAPIC\MYINTER.ASM 143 : Register in operand
not in bank 0.  Ensure that bank bits are correct.

Build completed.
```

Je ne reviendrai pas ici sur les warnings.

12.12 Construction du programme principal

Nous allons maintenant réaliser un télérupteur. Qu'est-ce à dire ? Et bien nous allons réaliser la fonction suivante : Une pression sur le bouton-poussoir allume la LED, une seconde pression l'éteint. Je vais vous guider pas à pas dans cette petite réalisation, en essayant de vous montrer les problèmes pratiques rencontrés dans une réalisation de ce type.

Je rappelle que ceci est un programme didactique. Nous réaliserons ce télérupteur de manière un peu plus élégante dans la leçon sur le timer0.

Puisque nous désirons utiliser les interruptions, l'inversion de l'allumage de la LED se fera dans la routine d'interruption du bouton-poussoir. Dans un premier temps, nous pourrions penser que le programme principal n'a rien à faire.

Remarque très importante

En aucun cas vous ne pouvez laisser un programme se « balader » hors de la zone de votre programme. Si le programme n'a plus rien du tout à faire, vous devez alors le boucler sur lui-même. Car le programme ne s'arrête pas (sauf passage en mode sleep que nous verrons plus tard).

Notre programme principal pourrait donc être de la forme :

```
start
   goto start              ; boucler
```

Mais, pour faciliter la visualisation de ce qui se passe sous l'émulateur, je vous demanderai d'ajouter quelques 'NOP' inutiles.

Voilà donc notre programme principal :

```
;*****************************************************************
;                       PROGRAMME PRINCIPAL                       *
;*****************************************************************
start
   nop                  ; instruction inutile
   nop                  ; instruction inutile
   nop                  ; instruction inutile
   nop                  ; instruction inutile
   nop                  ; instruction inutile
   goto start           ; boucler
   END                  ; directive fin de programme
```

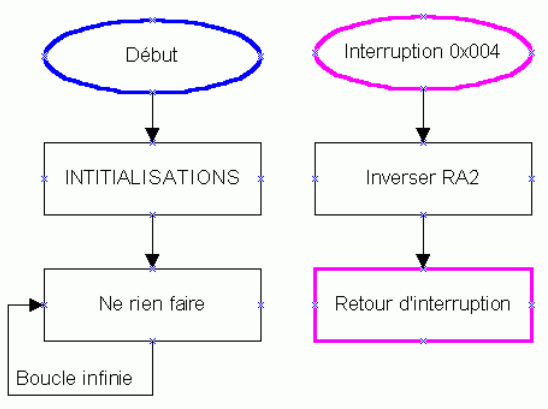
12.13 Construction de la routine d'interruption

Nous avons programmé le PIC® de façon à ce qu'une interruption sur flanc descendant de INT/RB0 provoque une interruption. Il nous suffit donc d'exécuter dans cette routine d'interruption l'inversion du niveau de la LED. Ceci est réalisé très simplement avec un « ou exclusif ». Nous pouvons donc écrire :

```
;*****************************************************************
;                     INTERRUPTION RB0/INT                        *
;*****************************************************************
;---------------------------------------------------------------------
; inverse le niveau de RA2 à chaque passage
;---------------------------------------------------------------------
intrb0
   movlw    B'00000100' ; bit positionné = bit à inverser
   BANK0                ; car on ne sait pas sur quelle banque
                        ; on est dans une interruption (le programme
                        ; principal peut avoir changé de banque). Ce n'est
                        ; pas le cas ici,mais c'est une sage précaution
   xorwf    PORTA , f   ; inverser RA2
   return               ; fin d'interruption RB0/INT
```

Voilà, notre premier essai est terminé. Nous allons passer ce programme au simulateur. Mais avant, je vous donne l'ordinogramme du programme que nous avons réalisé. Lorsque vos programmes deviendront complexes, je vous conseille de recourir à l'ordinogramme avant de les écrire.

Ordinogramme 1 (version théoriquement fonctionnelle)



Vous voyez que cette routine d'interruption est on ne peut plus simple, et correspond bien à ce qu'on pourrait s'imaginer de prime abord. Nous allons donc commencer par passer le test du simulateur.

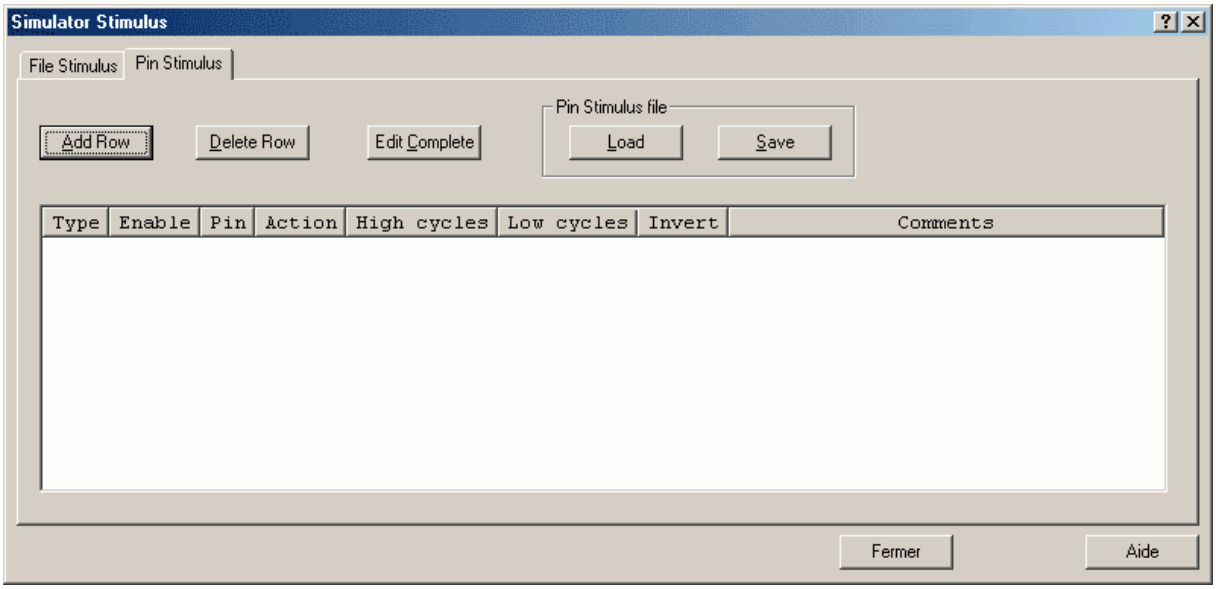
12.14 Passage au simulateur d'une routine d'interruption

N 'oubliez pas que le simulateur doit être en service, si ce n'est pas le cas, faites-le (voir chapitre sur le debugger). Sélectionnez l'affichage de la fenêtre « special functions registers) si ce n'est déjà fait.

Lançons la compilation avec <F10>. Ensuite pressez <F6> pour faire le reset du programme, puis <F7>. Répétez <F7> en suivant l'évolution du programme. N'oubliez pas qu'il va effectuer 68 boucles pour effacer la RAM, soyez patient. Profitez-en pour observer le comportement de FSR pour l'adressage indirect. Vous pouvez également presser <F9>, et ensuite <F5> après quelques instants.

Une fois arrivé dans le programme principal, le programme boucle indéfiniment. En effet, un événement extérieur (bouton-poussoir) est nécessaire pour provoquer le passage dans la routine d'interruption. Je vais maintenant vous expliquer comment simuler un événement extérieur.

Allez dans le menu « debugger -> stimulus ». Si par hasard vous avez des messages d'erreur concernant un fichier, ignorez-les. Sélectionnez l'onglet « pin stimulus »



Cliquez sur « add Row » pour obtenir un bouton d'action. Une nouvelle ligne est créée dans la fenêtre, avec un bouton intitulé « Fire ». Cliquez une fois dans la colonne « pin » juste à côté du bouton. Les cases « pin » et « action » se remplissent. Elargissez les colonnes à la souris pour mieux voir.

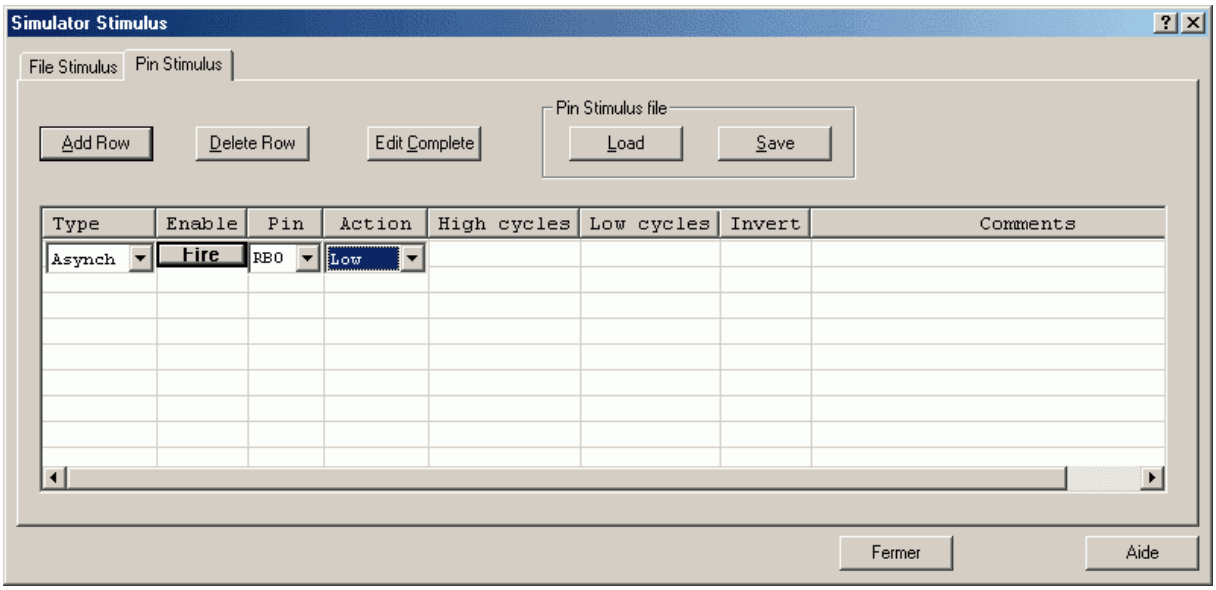
Nous allons maintenant préciser l'action de notre bouton. Dans la case « type », nous sélectionnons « asynch » pour «asynchrone ». En effet, l'événement pourra intervenir à

n'importe quel moment de l'exécution du programme. Ne cliquez pas sur le bouton « fire » pour l'instant.

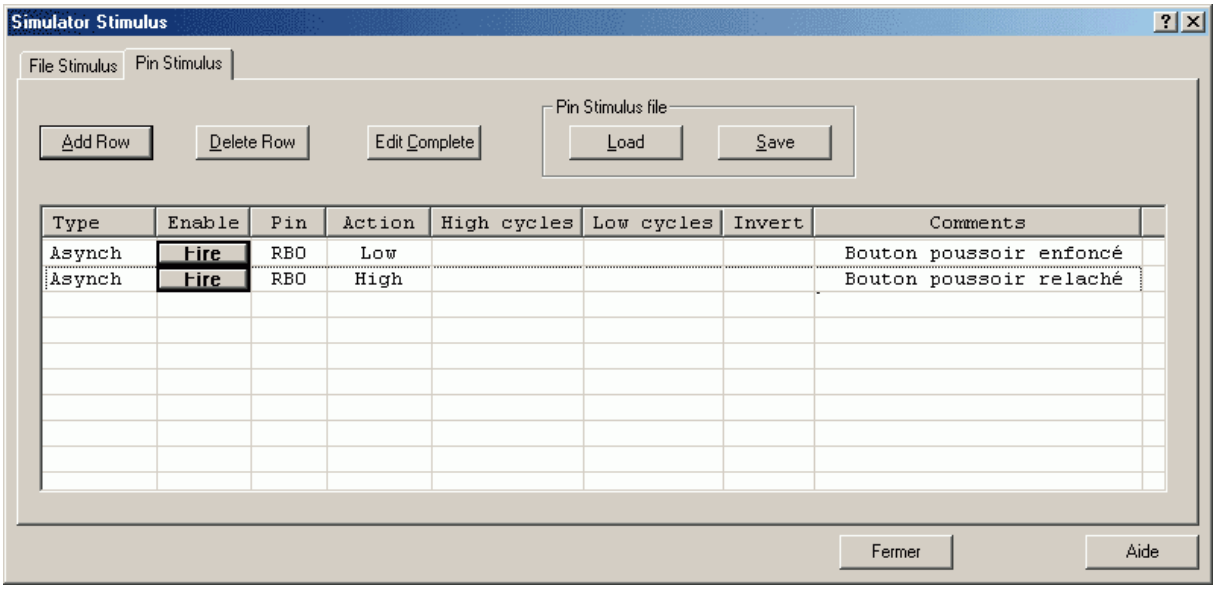
A l'aide du menu déroulant de la case « pin », nous allons déterminer quelle pin sera stimulée par l'action sur le bouton. Comme notre bouton se trouve sur la pin RB0, sélectionnez cette pin.

Si vous regardez dans la case « action », vous voyez que vous avez accès au mode de fonctionnement du bouton. Vous avez le choix entre Pulse (génère une impulsion), Low (place un niveau 0), High (place un niveau 1), ou Toggle (inversion du niveau à chaque pression). Nous choisirons la moins pratique pour cet exemple, mais la plus explicite.

Choisissez donc « Low ». Votre bouton de stimulation est maintenant configuré. Vous devriez obtenir une fenêtre du style suivant :

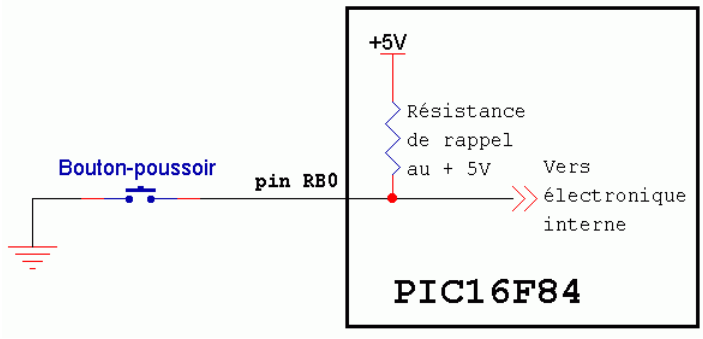


Créez maintenant une nouvelle ligne, avec « add row », et créez un second bouton.



Choisissez « High » dans la case action. Vous pouvez également placer un commentaire dans les cases « comments ». Ne vous génez pas.

Examinons le registre PORTB dans la fenêtre d'affichage des registres spéciaux. Vous voyez que tous les bits sont des 0. En effet, MPLAB® ne peut pas connaître l'électronique que vous avez connectée sur ses pins. C'est donc à vous de lui indiquer le niveau que vous êtes sensé avoir sur les dites pins. Pour ceux qui n'auraient pas compris le fonctionnement de la résistance de rappel, voici une nouvelle fois le schéma équivalant :



Nous voyons donc que, quand le bouton-poussoir n'est pas pressé, nous avons un niveau 1 sur RB0 provoqué par la résistance de rappel que nous avons mise en service.

Pressons donc le second bouton une seule fois, et allons dans l'éditeur. Pressons <F7> pour avancer d'un pas et valider la modification de niveau. Examinez PORTB : RB0 est maintenant passé à 1. Notre bouton-poussoir n'est pas enfoncé. Pressez quelques fois <F7> pour vérifier que rien d'autre ne s'est passé.

Nous allons maintenant simuler la pression du bouton-poussoir :

- Pressez le premier bouton pour envoyer 0 sur RB0.
- Revenez dans l'éditeur et pressez une seule fois sur <F7>. L'instruction qui suit l'événement est alors l'instruction située à l'adresse 0x04, car le passage de 1 à 0 sur RB0 a provoqué notre interruption.
- Avancez lentement par pressions de <F7> dans la routine d'interruption. Examinez l'effet des différentes instructions vues. Une fois la ligne :

```
xorwf    PORTA , f    ; inverser RA2
```

exécutée, vous constatez que la LED s'est allumée (RA2 est passé à 1 sur le registre PORTA). Avancez lentement jusqu'à ce que la ligne :

```
retfie   ; return from interrupt
```

soit sélectionnée et ne pressez plus <F7>. A cet endroit, nous trouvons le retour de la routine d'interruption vers le programme principal. Pressez une nouvelle fois <F7>.

Que se passe-t-il ? Au lieu de revenir au programme principal, nous recommençons une nouvelle interruption.

Pour provoquer une interruption, il faut que le bit Enable ET le bit Flag d'une des sources d'interruptions soient à 1. Or, il n'y a qu'un seul bit Enable à 1, et c'est INTE.

Examinons donc INTF (c'est le bit 1 de INTCON). Ce bit est toujours à 1, donc nouvelle interruption. Nous avons donc commis une erreur classique. Nous avons oublié d'effacer le flag à la fin du traitement de notre interruption. Remarquez que cet effacement est intégré dans la partie « switch » de la routine d'interruption du fichier m16f84.asm. Nous avons effacé cette ligne par mégarde en supprimant les différents tests (en fait je vous l'ai volontairement fait supprimer « par mégarde » : j'ai noté qu'on mémorise mieux suite à une erreur). Somme toutes, vous avez supprimé la ligne « à l'insu de votre plein gré ».

12.15 Première correction : reset du flag

Il nous faut donc ajouter la ligne suivante dans notre sous-routine intrb0

```
bcf INTCON , INTF           ; effacer flag INT/RB0
```

Nous obtenons donc :

```
movlw    B'00000100'        ; bit positionné = bit inversé
BANK0                       ; car on ne sait pas sur quelle banque
                            ; on est dans une interruption (le
                            ; programme principal peut avoir changé
                            ; de banque). Ce n'est pas le cas ici,
                            ; mais c'est une sage précaution
xorwf    PORTA , f          ; inverser RA2
bcf      INTCON , INTF      ; effacer flag INT/RB0
return                      ; fin d'interruption RB0/INT
```

Recompilons notre programme avec <F10>, puis <F6>, et enfin, recommencez toute la procédure que nous venons de voir.

Vérifiez que la routine d'interruption se termine maintenant en rendant la main au programme principal.

La LED 1 est maintenant allumée (RA2 = 1).

Pressez le second bouton pour simuler le relâchement du bouton-poussoir. Pressez quelques fois <F7> et pressez <RB0 (L)> pour simuler une seconde pression de bouton-poussoir. Suivez la routine d'interruption et constatez que cette fois la LED s'éteint.

On obtient donc au simulateur le fonctionnement suivant :

- Une pression sur le B.P. (bouton-poussoir) allume la LED
- Une autre pression éteint la LED
- Et ainsi de suite.

Nous avons donc obtenu le résultat souhaité.

Placez votre PIC® dans le programmateur, et envoyez lui le fichier « Myinter.hex ». Placez votre PIC® sur la platine d'essais (modifiée) et pressez le B.P. à plusieurs reprises. La

LED ne fonctionne pas du tout comme prévu, Elle réagit, mais de manière aléatoire. Que se passe-t-il ?

12.16 Se mettre à l'échelle de temps du PIC®

Et bien, c'est tout simple. Les PIC® sont des composants très rapides. Ils ne travaillent pas à la même échelle de temps que nous. Essayons de nous transformer en PIC®. Nous voyons alors un énorme Bouton poussoir.

Lorsque ce B.P. est pressé, c'est alors une énorme barre qui vient court-circuiter 2 contacts. Cette barre est élastique. Que voit le PIC® ? Il voit une énorme barre flexible qui tombe d'une énorme hauteur sur 2 contacts métalliques. Une fois la barre en contact, elle REBONDIT plusieurs fois. A chaque pression sur le B.P., le PIC® voit donc une série de fermeture et d'ouverture du B.P., au lieu d'une seule à notre échelle de temps.

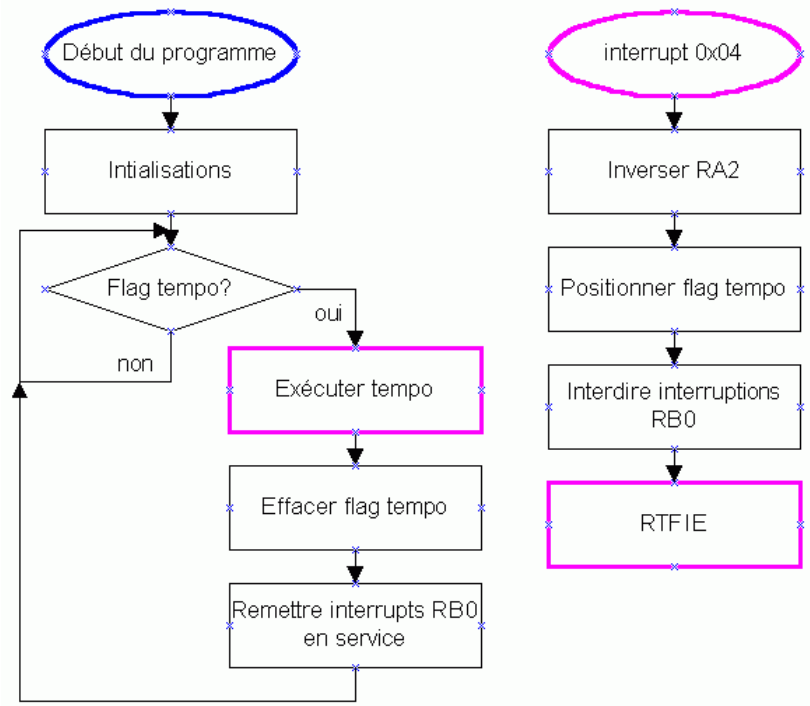
Le PIC® est donc plus rapide que notre B.P.

N'oubliez donc jamais que les PIC® ne travaillent pas à une échelle de temps humaine. Vous devez en tenir compte.

12.17 Le problème de l'anti-rebond

Comment remédier à ce problème ? Et bien, tout simplement en attendant un temps supérieur au temps de rebondissement avant d'autoriser une nouvelle interruption sur RB0.

Nous allons utiliser nos connaissances actuelles pour résoudre ce problème. Il devient utile de dessiner un ordinogramme de ce que nous voulons faire :



Explications

Le programme principal effectue normalement ses initialisations, puis teste si une demande de tempo a été introduite en positionnant le flag « tempo ». Si le flag n'est pas mis, il boucle sans fin.

Si le B.P. est pressé, une interruption est générée, RA2 est inversé, la routine d'interruption positionne le flag tempo à 1 et interdit toute nouvelle interruption de RB0. Toute autre action sur RB0 sera donc sans effet (donc les rebonds ne sont pas pris en compte).

La routine d'interruption prend fin. Retour au programme principal, qui continue alors à tester le flag tempo. Celui-ci vient d'être positionné par la routine d'interruption. Le programme principal appelle alors une routine de tempo (que nous avons déjà vue dans la leçon principale).

Après écoulement du temps nécessaire à la fin des rebonds, le flag tempo est annulé (pour ne pas boucler sans fin), et les interruptions sont à nouveau autorisées, afin de permettre de prendre en compte une nouvelle pression sur le B.P.

Vous voyez donc ici qu'il peut être utile dans un programme d'interrompre et de relancer les interruptions à certains moments spécifiques.

12.18 Finalisation du programme

Tout d'abord, il nous faut une routine de temporisation. Ouvrez le fichier « Led_cli.asm » et effectuez un copier/coller de la routine de temporisation.

```
;*****************************************************************
;                     SOUS-ROUTINE DE TEMPORISATION               *
;*****************************************************************
;-----------------------------------------------------------------
; Cette sous-routine introduit un retard de 500.000 µs.
; Elle ne reçoit aucun paramètre et n'en retourne aucun
;-----------------------------------------------------------------
tempo
   movlw    2                ; pour 2 boucles
   movwf    cmpt3            ; initialiser compteur3
boucle3
   clrf     cmpt2            ; effacer compteur2
boucle2
   clrf     cmpt1            ; effacer compteur1
boucle1
   nop                       ; perdre 1 cycle
   decfsz   cmpt1 , f        ; décrémenter compteur1
   goto     boucle1          ; si pas 0, boucler
   decfsz   cmpt2 , f        ; si 0, décrémenter compteur 2
   goto     boucle 2         ; si cmpt2 pas 0, recommencer boucle1
   decfsz   cmpt3 , f        ; si 0, décrémenter compteur 3
   goto     boucle3          ; si cmpt3 pas 0, recommencer boucle2
   return                    ; retour de la sous-routine
```

Nous allons modifier légèrement cette sous-routine. Nous pouvons enlever la boucle extérieure, car, 500ms c'est beaucoup plus que le temps de rebond du B.P. Enlevons également l'instruction nop. Nous obtenons :

```
;*****************************************************************
;                     SOUS-ROUTINE DE TEMPORISATION               *
;*****************************************************************
;-----------------------------------------------------------------
; Cette sous-routine introduit un retard
; Elle ne reçoit aucun paramètre et n'en retourne aucun
;-----------------------------------------------------------------
tempo
   clrf     cmpt2             ; effacer compteur2
boucle2
   clrf     cmpt1             ; effacer compteur1
boucle1
   decfsz   cmpt1 , f         ; décrémenter compteur1
   goto     boucle1           ; si pas 0, boucler
   decfsz   cmpt2 , f         ; si 0, décrémenter compteur 2
   goto     boucle2           ; si cmpt2 pas 0, recommencer boucle1
   return                     ; retour de la sous-routine
```

Nous ne nous sommes donc pas servi de la variable cmpt3. Nous pouvons donc la supprimer de notre zone des variables. Tant que nous y sommes, nous allons avoir besoin d'un flag, c'est à dire d'un bit. Créons-le dans cette zone.

```
;*****************************************************************
;                   DECLARATIONS DE VARIABLES                     *
;*****************************************************************

   CBLOCK 0x00C               ; début de la zone variables
   w_temp :1                  ; Sauvegarde de W dans interruption
   status_temp : 1            ; Sauvegarde de STATUS dans interrupt
   cmpt1 : 1                  ; compteur de boucles 1 dans tempo
   cmpt2 : 1                  ; compteur de boucles 2 dans tempo
   flags : 1                  ; un octet pour 8 flags
                              ; réservons b0 pour le flag tempo
                              ; b1 : libre
                              ; b2 : libre
                              ; b3 : libre
                              ; b4 : libre
                              ; b5 : libre
                              ; b6 : libre
                              ; b7 : libre
            ENDC              ; Fin de la zone

#DEFINE     tempoF flags, 0   ; Définition du flag tempo
```

Modifions notre programme principal en suivant notre ordinogramme. Nous obtenons :

```
;*****************************************************************
;                       PROGRAMME PRINCIPAL
*
;*****************************************************************
start
   btfss    tempoF              ; tester si tempo flag mis
   goto     start               ; non, attendre qu'il soit mis
   call     tempo               ; oui, exécuter tempo
   bcf      tempoF              ; effacer flag tempo
   bsf      INTCON , INTE       ; remettre interrupts INT en service
   goto     start               ; boucler
   END                          ; directive fin de programme
```

Il ne reste plus qu'à modifier notre routine d'interruption en fonction de notre ordinogramme. Nous obtenons :

```
;*****************************************************************
;                       INTERRUPTION RB0/INT                     *
;*****************************************************************
;-----------------------------------------------------------------
; inverse le niveau de RA2 à chaque passage
; interdit toute nouvelle interruption
; valide le flag tempo
;-----------------------------------------------------------------
intrb0
   movlw    B'00000100'         ; bit positionné = bit inversé
   BANK     0                   ; car on ne sait pas sur quelle banque
                                ; on est dans une interruption (le
                                ; programme principal peut avoir changé
                                ; de banque). Ce n'est pas le cas ici,
                                ; mais c'est une sage précaution
   xorwf    PORTA , f           ; inverser RA2
   bcf      INTCON , INTF       ; effacer flag INT/RB0
   bcf      INTCON , INTE       ; interdire autre inter. RB0
   bsf      tempoF              ; positionner flag tempo
   return                       ; fin d'interruption RB0/INT
```

Compilons notre programme et chargeons le nouveau fichier .hex dans notre PIC®. Lançons l'alimentation. Cela ne fonctionne toujours pas, pourquoi ? Et bien réfléchissons à ce qui se passe.

Une fois la routine d'interruption terminée, les interruptions sont invalidées. L'interrupteur rebondit sans causer d'appel d'interruption. MAIS SON FLAG EST POSITIONNE à cause des rebonds. Souvenez-vous que les bits Enable n'agissent pas sur les flags. Donc, dès que le programme principal remet l'interruption en service, une interruption est générée directement, INTF ayant été positionné avant.

Vous devez donc penser à tout lorsque vous utilisez les interruptions. Si un programme fonctionne au simulateur et pas sur le circuit réel, commencez par soupçonnez des problèmes de timing.

Il nous suffit donc d'ajouter une instruction de reset du flag INTF avant de remettre les interruptions INT en service.

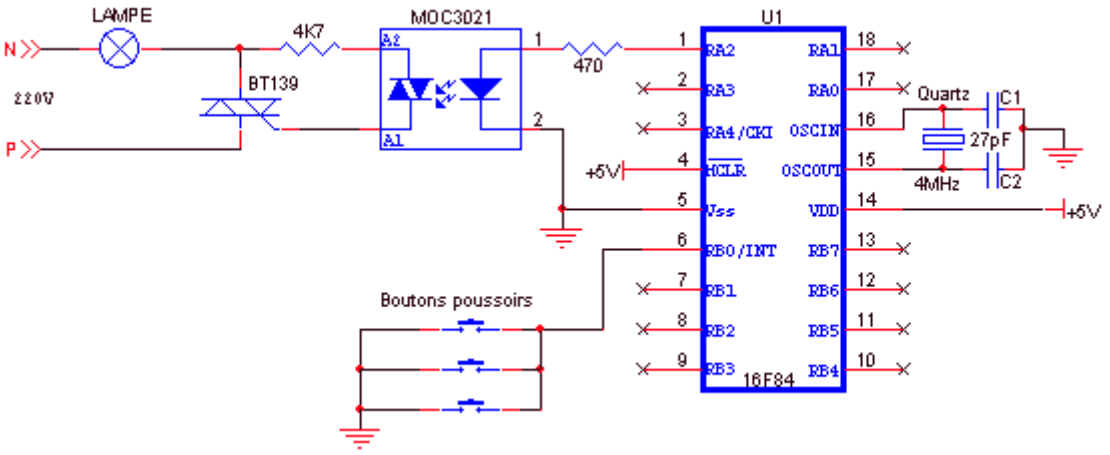
```
   bcf      INTCON , INTF       ; effacer flag INT
```

Bien entendu, vous pouvez vous dire qu'il devient inutile d'effacer ce flag dans la routine d'interruption. C'est logique, mais il vaut mieux prendre de bonnes habitudes. Laissons donc cette instruction.

Recompilez votre programme, et rechargez-le dans votre PIC®. Alimentez votre montage.

Cette fois cela fonctionne parfaitement. Voici un montage très pratique à utiliser. Si vous remplacez la LED par un petit relais ou un triac optocouplé, vous voilà en possession d'un télérupteur. Placez un bouton-poussoir dans chaque endroit d'où vous désirez allumer la lampe, et vous pouvez allumer ou éteindre celle-ci depuis plusieurs endroits, sans utiliser d'interrupteurs spéciaux avec un tas de fils à tirer.

A titre de bonus, voici le schéma d'un télérupteur opérationnel fonctionnant avec votre programme :



Remarque sur notre programme

Nous avons utilisé une temporisation de valeur quelconque. Cette temporisation inhibe toute action sur le B.P. (c'est son but) durant +- 250mS. Donc, vous pourrez presser au grand maximum 4 fois sur le B.P. par seconde.

Mais quelle est la durée réelle des rebonds ? Et bien, elle dépend de l'interrupteur utilisé, technologie et taille. Pour connaître le temps de rebond de votre propre interrupteur, diminuez progressivement la durée de la temporisation. Une fois que votre télérupteur ne fonctionne plus à chaque pression, vous avez atteint la limite. Calculez alors la durée de votre temporisation, vous aurez la durée approximative des rebonds. L'ordre de grandeur est souvent de quelques ms.

12.19 Remarques importantes

Souvenez-vous, qu'une fois que vous utilisez les interruptions dans un programme, vous ne pouvez jamais savoir le temps qui va séparer 2 instructions successives.

En effet, entre les instructions en question peut avoir été traité une ou plusieurs routine(s) d'interruption.

En conséquence, VOUS NE POUVEZ PAS UTILISER de calcul de temps en utilisant le calcul du nombre d'instructions dans toute partie de code dans lequel une interruption risque de survenir.

Notre sous-routine de temporisation de « Led-cli », par exemple, donnerai un délai qui risquerait d'être allongé. Dans ce programme, cependant, les interruptions ne sont plus en service au moment de l'exécution de cette temporisation. Voyez l'ordinogramme.

De plus, pour toute séquence dont le déroulement en temps est critique et ne peut être interrompu, vous devez inhiber les interruptions. A votre charge de les remettre en service en temps utile.

Une fois que vous utilisez les interruptions, votre programme devra affronter des événements asynchrones avec son déroulement. Donc vous ne pourrez JAMAIS tester toutes les éventualités possibles.

VOUS NE POUVEZ DONC JAMAIS ETRE CERTAIN PAR SIMULATION QUE VOTRE PROGRAMME EST COMPLETEMENT DEBUGGE.

Ceci vous explique pourquoi des gestions de processus critiques en temps réel utilisent plusieurs ordinateurs (navette spatiale). Ces ordinateurs étant désynchronisés, un bug de cette nature qui apparaît sur un d'eux a peu de chance de se produire simultanément sur un second. Pour savoir lequel a posé problème, il faut donc un troisième ordinateur, la majorité l'emporte.

Pensez toujours à ceci si vous êtes amenés un jour à réaliser un programme dont dépend la sécurité de personnes ou de biens.

Si vous comprenez bien tout ceci, vous voyez que les bugs éventuels liés à des erreurs de programmation peuvent apparaître à tout moment et de façon qui semble aléatoire. Et vous vous étonnez que votre ordinateur plante ?

12.20 Conclusions

Au terme de ce chapitre, vous pouvez appréhender les mécanismes d'interruption et leur mise en œuvre. Nous utiliserons encore cette possibilité dans la leçon sur le timer.

J'espère avoir démythifié ce concept, trop souvent imaginé comme « la méthode des pros ». En réalité, une fois de plus, pas de magie. Ce n'est qu'une exploitation, certes parfois complexe, de processus simples et faciles à appréhender.

Gardez seulement à l'esprit que vous quittez le monde du synchrone pour entrer dans le monde de l'asynchrone, beaucoup plus difficile à simuler totalement de façon efficace.

L'utilisation des interruptions impose de ce fait la parfaite compréhension des mécanismes mis en place, et vous oblige à envisager toutes les possibilités au niveau de la survenance d'un

événement extérieur. Ceci est surtout vrai pour les programmes avec plusieurs sources d'interruptions.

Je vous conseille donc fortement, lors de la réalisation de programme complexe, de commencer par écrire un algorithme général, soit en français, soit en utilisant des ordinogrammes.

13. Le Timer 0

Dans ce chapitre, nous allons parler temporisations et comptages. Le 16F84 ne comporte qu'un seul timer sur 8 bits, contrairement à d'autres PIC® de la famille (comme le 16F876). Si on examine attentivement le fonctionnement du timer0, on voit qu'il s'agit en fait d'un compteur.

13.1 Les différents modes de fonctionnement

Nous avons vu que le timer0 est en fait un compteur. Mais qu'allez-vous compter avec ce timer? Et bien, vous avez deux possibilités :

- En premier lieu, vous pouvez compter les impulsions reçues sur la pin RA4/TOKI. Nous dirons dans ce cas que nous sommes en mode compteur

- Vous pouvez aussi décider de compter les cycles d'horloge du PIC® lui-même. Dans ce cas, comme l'horloge est fixe, nous compterons donc en réalité du temps. Donc, nous serons en mode « timer ».

La sélection d'un ou l'autre de ces deux modes de fonctionnement s'effectue par le bit 5 du registre OPTION : T0CS pour Tmr0 Clock Source select bit.

T0CS = 1 : Fonctionnement en mode compteur
T0CS = 0 : Fonctionnement en mode timer

Dans le cas où vous décidez de travailler en mode compteur, vous devez aussi préciser lors de quelle transition de niveau le comptage est effectué. Ceci est précisé grâce au bit 4 du registre OPTION : T0SE pour Timer0 Source Edge select bit.

T0SE = 0 : comptage si l'entrée RA4/TOKI passe de 0 à 1
T0SE = 1 : comptage si l'entrée RA4/TOKI passe de 1 à 0

13.2 Le registre tmr0

Ce registre, qui se localise à l'adresse 0x01 en banque 0, contient tout simplement la valeur actuelle du timer0. Vous pouvez écrire ou lire tmr0. Si par exemple vous avez configuré tmr0 en compteur, la lecture du registre tmr0 vous donnera le nombre d'événements survenus sur la pin RA4/TOKI.

13.3 Les méthodes d'utilisation du timer0

Comment utiliser le timer0, et quelles sont les possibilités offertes à ce niveau, voilà de quoi nous allons parler ici.

13.3.1 Le mode de lecture simple

La première méthode qui vient à l'esprit est la suivante : Nous lisons le registre tmr0 pour voir ce qu'il contient. La valeur lue est le reflet du nombre d'événements survenus, en prenant garde au fait que le tmr0 ne peut compter que jusque 255. En cas de dépassement, le tmr0 recommence à 0. C'est donc à vous de gérer cette possibilité.

Petit exemple :

```
    Clrf      tmr0              ; début du comptage 2 cycles plus loin
                                ; (voir remarque encadrée plus bas)
    Xxx                         ; ici un certain nombre d'instructions
    movf      tmr0 , w          ; charger valeur de comptage
    movwf     mavariable        ; sauver pour traitement ultérieur
```

13.3.2 Le mode de scrutation du flag

Nous devons savoir à ce niveau, que tout débordement du timer0 (passage de 0xFF à 0x00) entraîne le positionnement du flag T0IF du registre INTCON. Vous pouvez donc utiliser ce flag pour déterminer si vous avez eu débordement du timer0, ou, en d'autres termes, si le temps programmé est écoulé. Cette méthode à l'inconvénient de vous faire perdre du temps inutilement

Petit exemple :

```
    clrf      tmr0              ; début du comptage dans 2 cycles
                                ; (voir remarque encadrée plus bas)
    bcf       INTCON , T0IF     ; effacement du flag
loop
    btfss     INTCON , T0IF     ; tester si compteur a débordé
    goto      loop              ; non, attendre débordement
    xxx                         ; poursuivre : 256 événements écoulés
```

Mais vous pourriez vous dire que vous ne désirez pas forcément attendre 256 incrémentations de tmr0. Supposons que vous désiriez attendre 100 incrémentations. Il suffit dans ce cas de placer dans tmr0 une valeur telle que 100 incrémentations plus tard, tmr0 déborde.

exemple :

```
    movlw     256-100           ; charger 256 – 100
    movwf     tmr0              ; initialiser tmr0
    bcf       INTCON,T0IF       ; effacement du flag
loop
    btfss     INTCON,T0IF       ; tester si compteur a débordé
    goto      loop              ; non, attendre débordement
    xxx                         ; oui, poursuivre : 100 événements écoulés
```

Ceci pourrait sembler correct, mais en fait toute modification de TMR0 entraîne un arrêt de comptage de la part de celui-ci correspondant à 2 cycles d'instruction multipliés par la valeur du prédiviseur. Autrement dit, un arrêt correspondant toujours à 2 unités TMR0. Il faut donc tenir compte de cette perte, et placer « 256-98 » et non « 256-100 » dans le timer.

13.3.3 Le mode d'interruption

C'est évidemment le mode principal d'utilisation du timer0. En effet, lorsque T0IE est positionné dans le registre INTCON, chaque fois que le flag T0IF passe à 1, une interruption est générée. La procédure à utiliser est celle vue dans la leçon sur les interruptions.

13.3.4 Les méthodes combinées

Supposons que vous vouliez, par exemple, mesurer un temps entre 2 impulsions sur la broche RB0. Supposons également que ce temps soit tel que plusieurs débordements du tmr0 puissent avoir lieu. Une méthode simple de mesure du temps serait la suivante :

1) A la première impulsion sur RB0, on lance le timer 0 en mode interruption.
2) A chaque interruption de tmr0, on incrémente une variable
3) A la seconde interruption de RB0, on lit tmr0 et on arrête les interruptions
4) Le temps total sera donc (256*variable)+tmr0

On a donc utilisé les interruptions pour les multiples de 256, et la lecture directe de tmr0 pour les « unités ».

13.4 Le prédiviseur

Supposons que nous travaillions avec un quartz de 4MHz. Nous avons donc dans ce cas (4000000/4) = 1.000.000 de cycles par seconde. Chaque cycle d'horloge dure donc 1/1000000ème de seconde, soit 1µs.

Si nous décidons d'utiliser le timer0 dans sa fonction timer et en mode interruption. Nous aurons donc une interruption toutes les 256µs, soit à peu près tous les quarts de millième de seconde.

Si nous désirons réaliser une LED clignotante à une fréquence de +- 1Hz, nous aurons besoin d'une temporisation de 500ms, soit 2000 fois plus. Ce n'est donc pas pratique.
Nous disposons pour améliorer ceci d'un PREDIVISEUR .

Qu'est-ce donc ? Et bien, tout simplement un diviseur d'événements situé AVANT l'entrée de comptage du timer0. Nous pourrons donc décider d'avoir incrémentation de tmr0 tous les 2 événements par exemple, ou encore tous les 64 événements.

Regardez le tableau de la page 16 du datasheet. Vous voyez en bas le tableau des bits PS0 à PS2 du registre OPTION qui déterminent la valeur du prédiviseur.

Ces valeurs varient, pour le timer0, entre 2 et 256. Le bit PSA, quant à lui, détermine si le prédiviseur est affecté au timer0 ou au watchdog. Voici un tableau exprimant toutes les possibilités de ces bits :

PSA	PS2	PS1	PS0	/tmr0	/WD	Temps tmr0	Temps typique Watchdog (minimal)
0	0	0	0	2	1	512 µs	18 ms (7ms)
0	0	0	1	4	1	1024 µs	18 ms (7ms)
0	0	1	0	8	1	2048 µs	18 ms (7ms)
0	0	1	1	16	1	4096 µs	18 ms (7ms)
0	1	0	0	32	1	8192 µs	18 ms (7ms)
0	1	0	1	64	1	16384 µs	18 ms (7ms)
0	1	1	0	128	1	32768 µs	18 ms (7ms)
0	1	1	1	256	1	65536 µs	18 ms (7ms)
1	0	0	0	1	1	256 µs	18 ms (7ms)
1	0	0	1	1	2	256 µs	36 ms (14 ms)
1	0	1	0	1	4	256 µs	72 ms (28 ms)
1	0	1	1	1	8	256 µs	144 ms (56 ms)
1	1	0	0	1	16	256 µs	288 ms (112 ms)
1	1	0	1	1	32	256 µs	576 ms (224 ms)
1	1	1	0	1	64	256 µs	1,152 Sec (448 ms)
1	1	1	1	1	128	256 µs	2,304 Sec (996 ms)

- PSA à PS0 sont les bits de configuration du prédiviseur
- /tmr0 indique la valeur du prédiviseur résultante sur le timer0
- /WD indique la valeur du prédiviseur résultante sur le Watchdog
- temps tmr0 indique le temps max entre 2 interruptions tmr0 avec quartz de 4MHz
- Temps watchdog indique le temps typique disponible entre 2 reset watchdog (indépendant du quartz utilisé). La valeur entre parenthèses indique le temps minimal, qui est celui à utiliser pour faire face à toutes les circonstances.

Remarques importantes :

- Il n'y a qu'un prédiviseur, qui peut être affecté au choix au timer du watchdog (que nous verrons plus tard) ou au timer0. Il ne peut être affecté aux deux en même temps.

- Il n'existe pas de prédiviseur = 1 pour le timer0. Si vous ne voulez pas utiliser le prédiviseur, vous devez donc impérativement le sélectionner sur le watchdog avec une valeur de 1 (ligne verte du tableau).

- La valeur contenue dans le prédiviseur n'est pas accessible. Par exemple, si vous décidez d'utiliser un prédiviseur de 64, et qu'il y a un moment donné 30 événements déjà survenus, vous n'avez aucun moyen de le savoir. Le prédiviseur limite donc la précision en cas de lecture directe.

- L'écriture dans le registre tmr0 efface le contenu du prédiviseur. Les événements survenus au niveau du prédiviseur sont donc perdus.

- Toute modification de TMR0 (effacement, incrémentation, opération...) entrainera un arrêt du timer correspondant aux deux prochaines incrémentations, ce qui donne deux unités comptées en moins que ce qui était prévu.

13.5 Application pratique du timer0

Nous allons mettre en œuvre notre tmr0 dans une première application pratique. Reprenons donc notre premier exercice, à savoir, faire clignoter une LED à la fréquence approximative de 1Hz.

13.5.1 Préparations

Faites un copier/coller de votre nouveau fichier m16f84.asm et renommez cette copie « Led_tmr.asm ». Relancez MPLAB® et créez un nouveau projet intitulé « Led_tmr.pjt ». Ajoutez-y votre nœud « Led_tmr.asm ».

Créez votre en-tête (je continue d'insister)

```
;*****************************************************************
;                                                                *
; Fait clignoter une LED à une fréquence approximative de 1Hz    *
;                                                                *
;*****************************************************************
;                                                                *
;    NOM: LED CLIGNOTANTE AVEC TIMER0                            *
;    Date: 17/02/2001                                            *
;    Version: 1.0                                                *
;    Circuit: Platine d'essai                                    *
;    Auteur: Bigonoff                                            *
;                                                                *
;*****************************************************************
;                                                                *
;    Fichier requis: P16F84.inc                                  *
;                                                                *
;*****************************************************************
;                                                                *
;    Notes: Utilisation didactique du tmr0 en mode interruption  *
;*****************************************************************
```

Définissez ensuite les _CONFIG en plaçant le watch-dog hors service.

Calculons ensuite le nombre de débordements de tmr0 nécessaires. Nous avons besoin d'une temporisation de 500 ms, soit 500.000µs.

Le timer0 génère, sans prédiviseur, une interruption toutes les 256µs . Nous allons donc utiliser le prédiviseur. Si nous prenons la plus grande valeur disponible, soit 256, nous aurons donc une interruption toutes les (256*256) = 65536µs.

Nous devrons donc passer (500.000/65536) = 7,63 fois dans notre routine d'interruption. Comme nous ne pouvons pas passer un nombre décimal de fois, nous choisirons 7 ou 8 fois, suivant que nous acceptons une erreur dans un sens ou dans l'autre.

Notez que si vous passez 7 fois, vous aurez compté trop peu de temps, il sera toujours possible d'allonger ce temps. Dans le cas contraire, vous aurez trop attendu , donc plus de correction possible.

Il est évident que l'acceptation d'une erreur est fonction de l'application. Si vous désirez faire clignoter une guirlande de Noël, l'erreur de timing sera dérisoire. Si par contre vous désirez construire un chronomètre, une telle erreur sera inacceptable. Commençons donc par ignorer l'erreur.

Nous allons décider d'utiliser une prédivision de 256 avec 7 passages dans la routine d'interruption. Le temps obtenu sera donc en réalité de (256*256*7) = 458752 µs au lieu de nos 500.000µs théoriques.

En reprenant notre tableau page16 sur le contenu du registre OPTION, nous devrons donc initialiser celui-ci avec : B'10000111', soit 0x87. En effet, résistances de rappel hors-service (on n'en n'a pas besoin), source timer0 en interne et prédiviseur sur timer0 avec valeur 256. Nous obtenons donc :

```
OPTIONVAL EQU H'87'             ; Valeur registre option
                                ; Résistance pull-up OFF
                                ; Préscaler timer à 256
```

Ensuite nous devons déterminer la valeur à placer dans le registre INTCON pour obtenir les interruptions sur le timer0. Ce sera B'10100000', soit 0xA0

```
INTERMASK EQU H'A0'             ; Interruptions sur tmr0
```

Ensuite, nos définitions :

```
;*****************************************************************
;                               DEFINE                           *
;*****************************************************************
#DEFINE LED  PORTA,2            ; LED
```

Ne touchons pas à notre routine d'interruption principale, car nous avons suffisamment de place pour conserver nos tests.

Ecrivons donc notre routine d'interruption timer. Nous voyons tout d'abord que nous allons devoir compter les passages dans tmr0, nous allons donc avoir besoin d'une variable. Déclarons-la dans la zone 0X0C.

```
   cmpt : 1        ; compteur de passage
```

13.5.2 L'initialisation

Comme il est plus facile de détecter une valeur égale à 0 qu'à 7, nous décrémenterons donc notre variable de 7 à 0. Nous inverserons la LED une fois la valeur 0 atteinte. Nous devons donc initialiser notre variable à 7 pour le premier passage.

Nous effectuerons ceci dans la routine d'initialisation, avant le goto start. Profitons-en également pour placer notre port LED en sortie.

Nous obtenons donc :

```
;*****************************************************************
;                        INITIALISATIONS                         *
;*****************************************************************
init
   clrf     PORTA              ; Sorties portA à 0
   clrf     PORTB              ; sorties portB à 0
   clrf     EEADR              ; permet de diminuer la consommation
   BANK1                       ; passer banque1
   movlw    OPTIONVAL          ; charger masque
   movwf    OPTION_REG         ; initialiser registre option

      ; Effacer RAM
      ; ------------
   movlw    0x0c               ; initialisation pointeur
   movwf    FSR                ; pointeur d'adressage indirect
init1
   clrf     INDF               ; effacer ram
   incf     FSR,f              ; pointer sur suivant
   btfss    FSR,6              ; tester si fin zone atteinte (>=0x40)
   goto     init1              ; non, boucler
   btfss    FSR,4              ; tester si fin zone atteinte (>=0x50)
   goto     init1              ; non, boucler

                ; initialiser ports
                ; -----------------
   bcf      LED                ; passer LED en sortie
   BANK0                       ; passer banque 0

   movlw    INTERMASK          ; masque interruption
   movwf    INTCON             ; charger interrupt control

                ; initialisations variables
                ; -------------------------
   movlw    7                  ; charger 7
   movwf    cmpt               ; initialiser compteur de passages

   goto     start              ; sauter programme principal
```

13.5.3 La routine d'interruption

Réalisons donc maintenant notre routine d'interruption :

Tout d'abord, on décrémente notre compteur de passage, s'il n'est pas nul, on n'a rien à faire cette fois.

```
   decfsz          cmpt , f    ; décrémenter compteur de passages
   return                      ; pas 0, on ne fait rien
```

Ensuite, si le résultat est nul , nous devons inverser la LED et recharger 7 dans le compteur de passages. Voici le résultat final :

```
;*****************************************************************
;                      INTERRUPTION TIMER 0                      *
;*****************************************************************
inttimer
   decfsz   cmpt , f          ; décrémenter compteur de passages
   return                     ; pas 0, on ne fait rien
   BANK0                      ; par précaution
   movlw    b'00000100'       ; sélectionner bit à inverser
   xorwf    PORTA , f         ; inverser LED
   movlw    7                 ; pour 7 nouveaux passages
   movwf    cmpt              ; dans compteur de passages
   return                     ; fin d'interruption timer
```

Il ne nous reste plus qu'à effacer la ligne

```
   clrwdt                     ; effacer watch dog
```

du programme principal, puisque le watchdog n'est pas en service.

Compilez votre programme. Nous allons maintenant le passer au simulateur.

N'oubliez pas de mettre le simulateur en service, et ouvrez la fenêtre des registres spéciaux. Avancez ensuite votre programme en pas à pas jusqu'à ce qu'il arrive dans le programme principal.

Remarques

- Le dernier registre est dans la fenêtre des registres spéciaux « T0pre » est un registre qui n'existe pas physiquement dans le PIC®. C'est MPLAB® qui compte les prédivisions pour les besoins de la simulation.

- Chaque fois que « T0pre » atteint la valeur de prédivision, tmr0 est incrémenté de 1. Ce n'est que lorsqu'il débordera que nous aurons une interruption.

- Dans le programme principal, « T0pre » est incrémenté de 2 unités à chaque pression sur <F7>. C'est normal, car ce programme ne comporte qu'un saut (goto), et chaque saut prend 2 cycles.

13.6 Modification des registres dans le simulateur

Comme nous n'allons pas passer des heures à simuler ce programme, nous allons modifier les registres en cours de simulation. Ceci est très simple à effectuer dans MPLAB®6, puisqu'il suffit de double-cliquer sur la valeur à modifier, dans la fenêtre « special function registers ». De plus, vous pouvez cliquer dans la colonne de votre choix, ce qui vous permet d'entrer la valeur en décimal, hexadécimal, ou binaire.

Nous allons nous servir de cette possibilité. Premièrement, supprimons le prédiviseur. Pour ce faire, nous allons écrire B'10001000', soit 0x88. Double-cliquez sur la case « hex » de la ligne « OPTION_REG »

Maintenant, chaque pression de <F7> incrémente tmr0 (pas de prédiviseur)

Ouvrez ensuite une fenêtre de visualisation des variables, avec « view >watch». Affichez ensuite la variable « cmpt » comme expliqué dans les chapitres précédents.

Continuez de presser <F7> et constatez que le débordement de tmr0 provoque une interruption et que cette interruption provoque la décrémentation de cmpt. Pour ne pas attendre trop longtemps, servez-vous de la fenêtre « modify » pour positionner cmpt à 1.

Ensuite, poursuivez la simulation. Vous constaterez que la prochaine interruption provoque la modification de RA2.

13.7 Mise en place sur la platine d'essais

Chargez le fichier .hex obtenu dans votre PIC® et alimentez votre platine d'essais. Comptez les allumages de la LED obtenus en 1 minute. Vous devriez trouver aux alentours de 65/66 pulses par minute. Ceci vous montre la précision obtenue.

En réalité, vous aurez un allumage toutes les (256*256*7*2) = 917504µS.

En 1 minute, on devrait obtenir : 60.000.000/917504 = 65,3 allumages. La théorie rejoint la pratique.

Le fichier est fourni sous la dénomination « led_tmr1.asm ».

13.8 Première amélioration de la précision

Nous allons chercher à améliorer la précision de notre programme. Nous pouvons commencer par modifier notre prédiviseur. Essayons plusieurs valeurs successives :

/1 : donne 500000/256 = 1953,125 passages. Pas pratique
/2 : donne 500000/512 = 976,5625 passages. Pas plus pratique
/4 : donne 500000/1024 = 488,28125 passages . Idem
/8 : donne 500000/2048 = 244,140625 passages. Dans ce cas, un seul compteur est également nécessaire, car le nombre de passages est inférieur à 256.

Quelle va être la précision obtenue ? Et bien, nous initialiserons cmpt à 244, avec prédiviseur à 8. Dans ce cas, la durée obtenue sera de :

256*8*244 = 499712 µs, donc 499712*2 = 999424µs par allumage.

En une minute, nous aurons donc 60000000/999424 = 60,034 allumages. Voici donc une précision nettement meilleure.

Vous pouvez maintenant modifier vous-même votre programme selon ces indications.

Vous voyez que vous devez modifier la valeur 07 en 244 à 2 endroits. Ce n'est pas pratique. Ajoutez donc une assignation, par exemple :

```
TIMEBASE     EQU    D'244'; base de temps = 244 décimal
```

Si vous avez un problème, le fichier fonctionnel de cet exercice est disponible sous la dénomination « Led_tmr.asm ».

- Avantages obtenus : Une plus grande précision

- Inconvénient : plus d'interruptions générées, donc plus de temps perdu pour le programme principal. Dans notre cas, cela n'a pas d'importance, le programme ne fait rien d'autre, mais ce ne sera pas toujours le cas.

13.9 Seconde amélioration de la précision

Vous pouvez encore améliorer la précision de votre programme. En effet, vous pouvez ne pas utiliser de prédiviseur, donc utiliser plusieurs compteurs pour 1953,125 passages. Au 1953ème passage, vous pourrez même générer une dernière tempo en ajoutant une valeur au tmr0. Par exemple :

- On détecte 1953 passages à l'aide de plusieurs compteurs

- Lors du 1953ème passage, on en est à 1953*256 = 499968 µs, il nous manque donc : 500.000 - 499.968 = 32µs.

- On devrait donc ajouter 256-32 = 224 à tmr0, mais vu les 2 incréments perdus systématiquement lors de toute modification de TMR0, on ajoutera 256-30 :

```
movlw    226          ; débordement dans 30 + 2 cycles
addwf    tmr0
```

Bien entendu, 32µs pour tout réaliser, c'est très court, aussi nous devrons optimiser les routines d'interruption au maximum. Suppression des tests et des sous-programmes etc. Mais cela reste en général à la limite du possible. Nous ne traiterons pas ce procédé ici, car cela ne présente pas d'intérêt, d'autant que les interruptions vont finir par occuper la majorité du temps CPU.

13.10 La bonne méthode - Adaptation de l'horloge

Supposons que vous vouliez construire un chronomètre. La précision est la donnée la plus importante dans ce cas, et passe bien avant la vitesse. Nous allons donc nous arranger pour que les démultiplicateurs tombent sur des multiples entiers. Comment ? Et bien simplement en changeant le temps d'une instruction, donc, en changeant le quartz du PIC®.

Exemple :

Comme nous ne pouvons pas accélérer un PIC® au dessus de sa vitesse maximale (nous utilisons un PIC® 4MHz), nous pouvons seulement le ralentir. Nous partons donc d'une base de temps trop rapide.

Par exemple : reprenons notre cas de départ : prédiviseur à 256, compteur de passages à 7. Durée avec un quartz de 4MHz : 256*256*7 par tempo, donc 256*256*7*2 par allumage, soit 917504 µs. Or, nous désirons 1000000µs.

Il suffit donc de recalculer à l'envers :
Que doit durer une instruction ? 1000000/(256*256*7*2) = 1,089913504µs.

Cela nous donne donc une fréquence d'instructions de 1/1,089913504µs = 0,917504 MHz.

Comme la fréquence des cycles internes est égale à la fréquence du quartz/4, nous aurons donc besoin d'un quartz de 0,917504 * 4 = 3,670016 MHz. (MHz car nous avons divisé par des µs : or, diviser par un millionième revient à multiplier par un million).

La seule contrainte est donc de savoir s'il existe des quartz de cette fréquence disponibles dans le commerce. Dans le cas contraire, vous recommencez vos calculs avec d'autres valeurs de prédiviseurs et de compteur.

Si vous trouvez donc un quartz de fréquence appropriée, vous obtenez une horloge de la précision de votre quartz. Vous ajoutez un affichage, et voilà une horloge à quartz.

13.11 La méthode de luxe : La double horloge

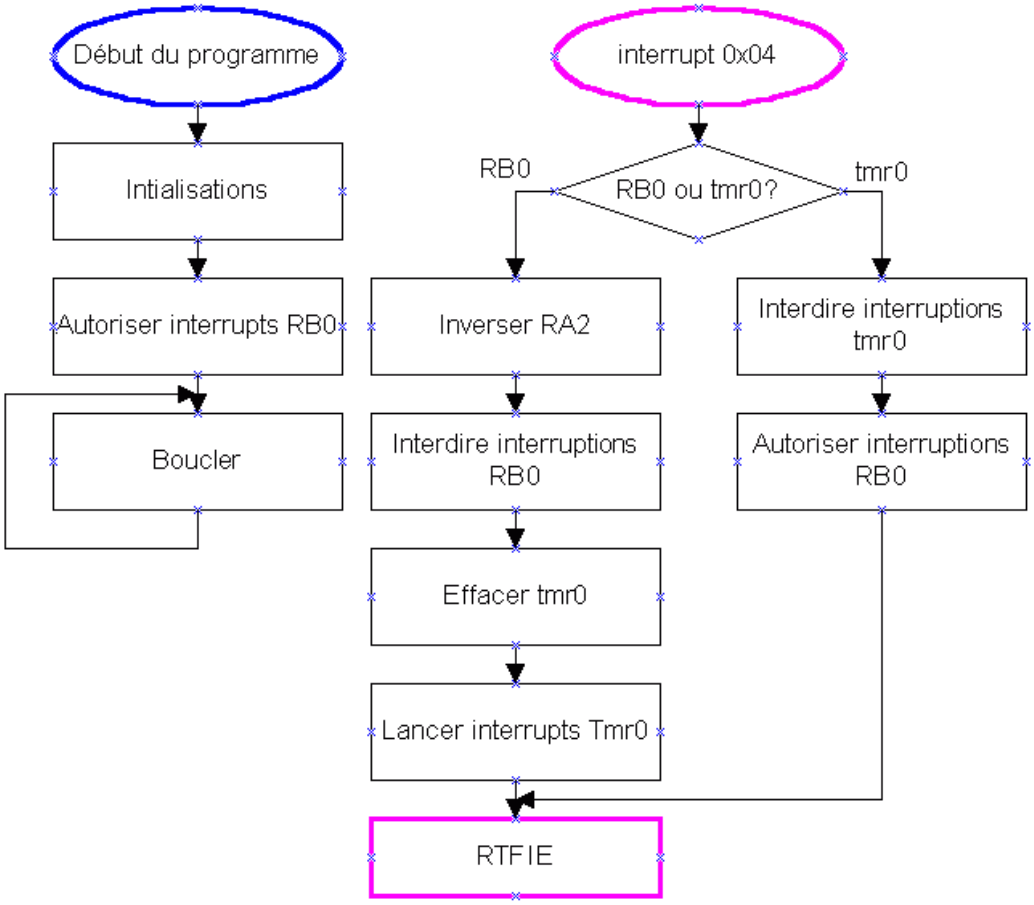
La méthode précédente présente l'inconvénient de ralentir le PIC®. Que faire si vous voulez à la fois une vitesse maximale et une précision également maximale ? Et bien, aucun problème.

Vous alimentez votre PIC® avec votre quartz et vous créez un autre oscillateur externe avec votre quartz spécial timing. Vous appliquez le signal obtenu sur la pin RA4/TOKI et vous configurez votre timer0 en mode compteur.

Donc, votre PIC® tourne à vitesse maximale, et les interruptions timer0 sont générées par une autre base de temps, plus adaptée à la mesure de vos événements.

13.12 Exemple d'utilisation de 2 interruptions

Dans ce petit exemple nous allons utiliser 2 sources d'interruption différentes, afin de vous montrer un exemple concret de ce type d'utilisation. Nous allons recréer notre programme de télérupteur, mais en remplaçant la temporisation par une interruption sur le timer0.



Remarquez que notre programme principal ne fait plus rien. Vous pouvez donc utiliser d'autres possibilités sur cette carte sans perturber le fonctionnement du télérupteur.

Nous aurons donc une interruption pour RB0, et une autre pour tmr0. Vous voyez ci-dessous l'ordinogramme qui va nous servir.

Effectuez une copie de votre fichier m16f84.asm et renommez-le « telerupt.asm ».

Créez un nouveau projet « telerupt.pjt ». Editez votre fichier comme précédemment : coupure du watchdog, positionnement de la LED en sortie, mise en service initiale des interruptions RB0/INT.

Créez votre routine d'interruption timer0 toutes les 260ms, soit prédiviseur à 256, et 4 passages .

Essayez de réaliser vous-même ce programme. Chargez-le dans votre PIC® et lancez-le. Notez que l'ordinogramme ne contient pas le compteur de passages dans tmr0. Je vous laisse le soin de réfléchir.

Une pression sur le B .P. allume la LED, une autre l'éteint. Si cela ne fonctionne pas, cherchez l'erreur ou servez-vous du simulateur. Je vous fourni le programme fonctionnel dans le cas où vous seriez bloqués.

Remarque

Il est très important de bien comprendre qu'il faut effacer tmr0 AVANT d'effacer le flag T0IF et de relancer les interruptions tmr0.

En effet, si vous faites le contraire, vous risquez que tmr0 déborde entre le moment de l'effacement du flag et le moment de l'effacement de tmr0. Dans ce cas le flag serait remis immédiatement après l'avoir effacé. Votre programme pourrait donc avoir des ratés par intermittence.

Il ne faut pas non plus oublier que toute modification de TMR0 entraine le nom comptage des deux prochaines incrémentations prévues.

13.13 Conclusion

Vous savez maintenant exploiter le timer0. Les méthodes évoquées ici sont une base de travail pour des applications plus sérieuses. Je vous conseille vraiment d'effectuer toutes les manipulations évoquées. Même les erreurs vous seront profitables.

Notes :

14. Les accès en mémoire « eeprom »

Je vais vous parler dans ce chapitre des procédures d'accès dans l'eeprom interne du PIC®. Il ne faut pas confondre ceci avec l'écriture dans une eeprom externe type 2416. Pour ce type d'eeprom, il « suffit de » suivre les directives du datasheet du composant concerné, j'en parlerai d'ailleurs dans la seconde partie du cours.

14.1 Taille et localisation de la mémoire « eeprom »

L'adresse physique de la zone eeprom commence, pour les PIC® mid-range, à l'adresse 0x2100.

Cette adresse se situe hors de l'espace d'adressage normal des PIC® (rappelez-vous, maximum 8Kmots, donc adresse maxi : 0x1FFF), donc nous pouvons déjà en déduire qu'il nous faudra utiliser une procédure spéciale pour y accéder.

Notez déjà que si ces emplacements ne sont pas accessibles directement par le programme, par contre ils le sont au moment de la programmation. Vous pourrez donc initialiser votre zone eeprom au moment de programmer votre composant.

Ceci est également vrai pour des registres spéciaux des PIC®. Par exemple, l'adresse 0x2007 contient les paramètres que vous écrivez dans _CONFIG. Vous pourriez donc remplacer cette directive par une initialisation directe à l'adresse 0x2007. Je vous le déconseille cependant pour des raisons de portabilité et d'évolution rapide vers une autre famille de PIC®. De plus, pourquoi faire compliqué quand on peut faire simple ?

De même, l'adresse 0x2006 contient l'identification du composant. C'est ainsi qu'un programmateur évolué peut faire la distinction entre un 16F84 et un 16F84A, car leur identification constructeur diffère.

Le 16F84 dispose de 64 emplacements eeprom disponibles pour votre libre usage. Nous allons voir comment les utiliser.

14.2 Préparation du programme

Commencez par effectuer un copier/coller de votre fichier « Led_tmr1.asm » et renommez cette copie en « eep_test.asm ». Construisez votre nouveau projet dans MPLAB® avec le même nom et ajoutez-y ce nœud.

Editez la zone d'en-tête du programme

```
;*****************************************************************
;                                                                *
; Fait clignoter une LED à une fréquence dépendant d'une valeur en    *
; eeprom                                                         *
;                                                                *
;*****************************************************************
;                                                                *
;    NOM: LED CLIGNOTANTE AVEC TIMER0 et utilisation de l'eeprom      *
;    Date: 18/02/2001                                            *
;    Version: 1.0                                                *
;    Circuit: Platine d'essai                                    *
;    Auteur: Bigonoff                                            *
;                                                                *
;*****************************************************************
;                                                                *
;    Fichier requis: P16F84.inc                                  *
;                                                                *
;*****************************************************************
;                                                                *
;    Notes: Démonstration de l'utilisation des données en eeprom      *
;           La base de temps de clignotement est contenue dans        *
;           l'eeprom.                                            *
;
*
;*****************************************************************
```

Ajoutons ensuite une variable dans la zone des variables. Elle contiendra la valeur à recharger dans le compteur.

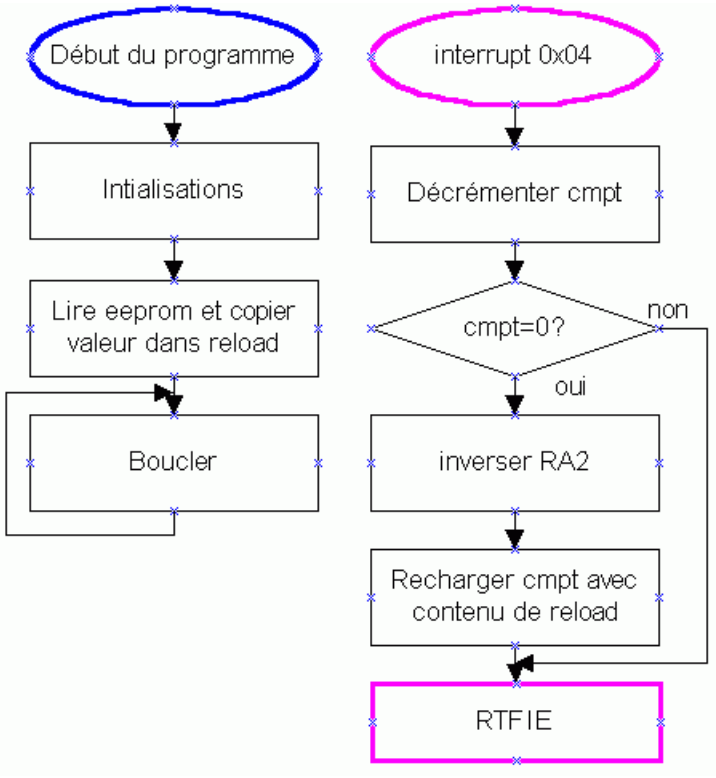
```
reload : 1            ; valeur à recharger dans compteur
```

Dans notre programme initial, à chaque fois que le compteur de passages dans le timer arrivait à 0, on le rechargeait avec la valeur 0x07. Maintenant, nous le rechargerons avec la valeur contenue dans la variable « reload ». La procédure utilisée est la suivante :

- On initialise un emplacement eeprom « eereload »avec la valeur 0x07 lors de la programmation
- Au démarrage, on lit l'eeprom « eereload » et on place son contenu dans « reload »
- Le contenu de reload est utilisé pour recharger le cmpt une fois celui-ci arrivé à 0.

Avantage de la procédure : si on modifie la valeur de la base de temps dans l'eeprom, cette modification ne sera pas perdue au moment de la remise sous tension de l'eeprom.

Je vous donne l'ordinogramme de ce que nous allons réaliser dans un premier temps.



Vous pourriez remarquer qu'il peut sembler inutile de lire l'eeprom et de recopier son contenu dans reload. Pourquoi donc ne pas utiliser la valeur eeprom directement dans le reste de notre programme ? La réponse est simple. La procédure de lecture en eeprom est plus complexe qu'une simple lecture en RAM. Il faut donc limiter les accès eeprom au maximum.

Commençons par modifier notre routine d'interruption. La seule ligne à modifier c'est celle qui chargeait d'office la valeur 7 dans w. Maintenant, nous y mettons le contenu de reload. Nous aurons donc :

```
inttimer
   decfsz     cmpt , f          ; décrémenter compteur de passages
   return                       ; pas 0, on ne fait rien
   BANK0                        ; par précaution
   movlw      b'00000100'       ; sélectionner bit à inverser
   xorwf      PORTA , f         ; inverser LED
   movf       reload , w        ; charger valeur contenue dans reload
   movwf      cmpt              ; dans compteur de passages
   return                       ; fin d'interruption timer
```

14.3 Initialisation de la zone eeprom

Nous voyons sur notre ordinogramme que nous lisons notre eeprom afin de placer le contenu dans notre variable. Mais nous devons bien quant même initialiser cette eeprom au moment de la programmation de notre PIC®.

Vous vous doutez bien qu'il ne sert à rien d'initialiser l'eeprom à chaque démarrage du PIC®, sinon, quel est l'intérêt d'utiliser une zone mémoire qui résiste au reset et à la mise hors tension ?

Nous initialiserons donc cette zone directement au moment de la programmation. Ceci s'effectue à l'aide de la directive « DE » pour Data Eeprom, placée dans la zone de données eeprom, c'est à dire en 0x2100.

Créons donc une zone eeprom, tout de suite après celle des variables.

```
;*****************************************************************
;                    DECLARATIONS DE LA ZONE EEPROM                *
;*****************************************************************
   org 0x2100            ; adresse début zone eeprom
   DE 0x07               ; valeur de recharge du compteur
```

Lancez la compilation de votre programme. Vous voulez sans doute vérifier que votre eeprom contient bien la valeur 0x07 ? Rien de plus simple : lancez « EEPROM memory » dans le menu « Windows » et vous voyez votre valeur.

Mais, allez-vous me répondre, la valeur est à l'adresse 0x00 et pas à l'adresse 0x2100 ?

En effet, il faut distinguer 2 adresses. L'adresse physique de cet emplacement mémoire est bien 0x2100. Cette adresse est uniquement accessible en mode programmation.

Par contre, votre programme accédera à ces emplacements à partir d'une procédure spéciale et avec une adresse dite relative. Cette adresse d'accès commence donc à 0x00.

Donc, pour résumer, pour accéder à l'adresse 0x2100, vous utiliserez la procédure d'accès EEPROM avec l'adresse 0x00. Et ainsi de suite : 0x2101 correspondra à 0x01…

Bien entendu, vous pouvez également donner un nom à ces adresses, tout comme pour les variables. Utilisons donc le nom eereload pour désigner la valeur de reload contenue en eeprom à l'adresse 0x00. Ajoutons simplement un define dans la zone d'initialisation eeprom

```
#DEFINE      eereload    0x00  ; adresse eeprom de eereload
```

ou encore

```
eereload     EQU         0x00
```

Dans notre ordinogramme nous allons avoir besoin de lire l'eeprom. 4 registres sont utilisés pour accéder à l'eeprom. Nous allons maintenant les examiner.

14.4 Le registre EEDATA

C'est dans ce registre que va transiter la donnée à écrire vers ou la donnée lue en provenance de l'eeprom. Ce registre est situé à l'adresse 0x08 banque 0.

14.5 Le registre EEADR

Dans ce registre, situé à l'adresse 0x09 banque 0, nous allons préciser sur 8 bits l'adresse concernée par l'opération de lecture ou d'écriture en eeprom. Nous voyons déjà que pour cette famille de PIC®, nous ne pourrons pas dépasser 256 emplacements d'eeprom. Pour le 16F84, la zone admissible va de 0x00 à 0x3F, soit 64 emplacements.

14.6 Le registre EECON1

Ce registre, situé à l'adresse 0x88 en banque1, contient 5 bits qui définissent ou indiquent le fonctionnement des cycles de lecture/écriture en eeprom. Voici son contenu :

bits 7/6/5

non utilisés

bit 4 : EEIF

Pour EEprom write operation Interrupt Flag bit. C'est le flag qui est en liaison avec l'interruption EEPROM. Il passe à 1 une fois l'écriture en eeprom terminée. Si le bit EEIE du registre INTCON est à 1, une interruption sera alors générée

bit 3 : WRERR

WRite ERRor. C'est un bit d'erreur. Il passe à 1 si une opération d'écriture en eeprom a été interrompue, par exemple par un reset.

bit 2 : WREN

WRite ENable. Autorisation de démarrage du cycle d'écriture

bit 1 : WR

WRite. Démarrage du cycle d'écriture. Est remis à 0 automatiquement une fois l'écriture terminée.

bit 0 : RD

ReaD. Démarrage d'un cycle de lecture. Reste à 1 durant un cycle, puis est remis à 0 automatiquement

Remarque : Dans le cas où le cycle d'écriture serait interrompu suite au dépassement du watchdog ou à un reset, vous pouvez lire le bit WRERR qui vous le signalera. Les registres EEDATA et EEADR demeurent inchangés et vous pouvez relancer le cycle d'écriture. Ceci ne fonctionne évidemment pas pour une coupure de tension. Dans ce cas, je vous expliquerai ma méthode personnelle de vérification à la fin de ce chapitre.

14.7 Le registre EECON2

Nous revoici en présence d'un registre « fantôme », puisque ce registre n'existe pas. Il s'agit tout simplement d'une adresse 0x89 banque1, qui sert à envoyer des commandes au PIC® concernant les procédures eeprom. Vous ne pouvez l'utiliser qu'en vous servant des instructions expliquées plus bas.

14.8 Accès en lecture dans la mémoire « eeprom »

Pour lire une donnée en eeprom, il suffit de placer l'adresse concernée dans le registre EEADR. Ensuite, vous positionnez le bit RD à 1. Vous pouvez ensuite récupérer la donnée lue dans le registre EEDATA. Il ne faut pas bien sûr oublier les différents changements de banques.

Comme cette procédure est courte et toujours la même, nous allons créer une macro à cette intention. Comme la macro doit contenir l'adresse de lecture, nous réaliserons une macro avec passage de paramètre. Voici la macro à ajouter. Vous devrez être dans la banque 0 pour appeler cette macro, et elle vous retourne la valeur lue dans le registre W.

```
READEE      macro adeeprom          ; macro avec 1 paramètre (argument)
   movlw    adeeprom                ; charger adresse eeprom (argument reçu)
   movwf    EEADR                   ; adresse à lire dans registre EEADR
   bsf      STATUS , RP0            ; passer en banque1
   bsf      EECON1 , RD             ; lancer la lecture EEPROM
   bcf      STATUS , RP0            ; repasser en banque 0
   movf     EEDATA , w              ; charger valeur lue dans W
   endm                             ; fin de la macro
```

Vous remarquez que vous passez un argument à la macro. Vous désignez cet ou ces arguments après la directive macro.

Nous avons utilisé ici l'argument adeeprom pour indiquez l'adresse eeprom. Chaque utilisation de adeeprom dans notre macro sera remplacée par l'argument reçu au moment de l'appel de la macro.

Pour utiliser cette macro, nous devons donc lui passer un argument. Par exemple :

```
READEE  eereload            ; lecture de l'adresse eereload de l'eeprom
```

Lira l'eeprom à l'adresse eereload, c'est à dire l'adresse 0x00.

Cette macro, comme toutes les modifications principales, seront ajoutées à votre fichier m16f84.asm. Je vous le fournis modifié sous la dénomination « m16f84_n2.asm ».

A partir de la prochaine leçon, il remplacera votre fichier « m16F84.asm » actuel.

Revenons à notre ordinogramme. Nous devons donc ajouter la lecture de l'eeprom dans l'initialisation, et placer cette valeur lue dans reload ET dans cmpt. Voici la routine modifiée :

```
  ; initialisations variables
  ; --------------------------
READEE    eereload     ; lire emplacement eeprom 0x00
movwf     reload       ; placer dans reload
movwf     cmpt         ; et initialiser compteur de passages
goto      start        ; sauter programme principal
```

Compilez votre programme et placez-le dans le PIC®. La LED doit maintenant clignoter à une fréquence de 1Hz. Si cela ne fonctionne pas, vérifiez ou consultez le fichier « eep_test1.asm » fourni avec cette leçon.

14.9 L'accès en écriture à la zone eeprom

Maintenant vous allez me dire avec raison que cela ne sert à rien de lire en eeprom si on n'arrive pas à y écrire. Notre programme ne présente donc rien de plus que ce que nous avions auparavant. C'est tout à fait justifié. Aussi allons-nous étudier la méthode d'écriture. Comme vous vous en doutez, cette méthode utilise les mêmes registres.

La procédure à suivre consiste d'abord à placer la donnée dans le registre EEDATA et l'adresse dans EEADR. Ensuite une séquence spécifique (il n'y a rien à comprendre, c'est imposé par le constructeur) doit être envoyée au PIC®.

Remarques

- Microchip® recommande que cette procédure spécifique ne soit pas interrompue par une interruption, donc nous couperons les interruptions durant cette phase.

- A la fin de la procédure d'écriture, la donnée n'est pas encore enregistrée dans l'eeprom. Elle le sera approximativement 10ms plus tard (ce qui représente tout de même pas loin de 10.000 instructions). Vous ne pouvez donc pas écrire une nouvelle valeur en eeprom, ou lire cette valeur avant d'avoir vérifié la fin de l'écriture précédente.

- La fin de l'écriture peut être constatée par la génération d'une interruption (si le bit EEIE est positionné), ou par la lecture du flag EEIF (s'il avait été remis à 0 avant l'écriture), ou encore par consultation du bit WR qui est à 1 durant tout le cycle d'écriture.

Nous allons écrire une macro d'écriture en eeprom. Cette fois, nous devrons passer 2 paramètres, à savoir la donnée à écrire (mais nous supposerons que nous l'avons placée dans W), et l'adresse d'écriture.

```
WRITEE      macro addwrite          ; la donnée se trouve dans W
   LOCAL    loop                    ; étiquette locale
   movwf    EEDATA                  ; placer data dans registre
   movlw    addwrite                ; charger adresse d'écriture
   movwf    EEADR                   ; placer dans registre
loop
   bcf      INTCON , GIE            ; interdire interruptions
   btfsc    INTCON , GIE            ; tester si GIE bien à 0
   goto     loop                    ; non, recommencer
   bsf      STATUS , RP0            ; passer en banque1
   bcf      EECON1 , EEIF           ; effacer flag de fin d'écriture
   bsf      EECON1 , WREN           ; autoriser accès écriture
   movlw    0x55                    ; charger 0x55
   movwf    EECON2                  ; envoyer commande
   movlw    0xAA                    ; charger 0xAA
   movwf    EECON2                  ; envoyer commande
   bsf      EECON1 , WR             ; lancer cycle d'écriture
   bcf      EECON1 , WREN           ; verrouiller prochaine écriture
   bsf      INTCON , GIE            ; réautoriser interruptions
   bcf      STATUS , RP0            ; repasser en banque 0
   endm
```

Remarques

- J'ai utilisé ici 3 instructions pour mettre GIE à 0. Ceci était nécessaire à cause d'un bug dans le 16C84, mais a été corrigé dans le 16F84. Le test de vérification n'est plus nécessaire à partir de cette version. Mais ceci me donnait l'occasion de vous montrer les étiquettes locales. Laissez-le donc pour exemple.

- La directive LOCAL précise que le symbole utilisé dans cette macro n'existera qu'à l'intérieur de celle-ci. Sans cette directive, si vous utilisiez un seul appel de la macro WRITEE dans votre programme, aucun problème. Si maintenant vous utilisiez 2 fois cette macro, pour chaque appel MPASM® remplacerait WRITEE par toute la liste d'instructions contenues jusque la directive endm. Vous auriez donc dans votre programme réel 2 fois l'étiquette « loop » à 2 emplacements différents. Ceci générerait une erreur. La directive LOCAL informe MPASM® que chaque utilisation de la macro travaille avec une étiquette « loop » différente. Il n'y a donc plus double emploi.

- Toute la partie surlignée de jaune contient la séquence imposée par Microchip® pour l'écriture en eeprom.

- La procédure d'écriture en eeprom est relativement longue et prend énormément de temps. Pensez que pour remplir la zone complète de 64 octets, il faudra plus de 6/10ème de seconde.

- Le nombre de cycle d'écritures en eeprom est limité. La durée de vie de l'eeprom est d'environ 10 millions de cycles. Si votre programme comporte une erreur et qu'il écrit sans arrêt des données dans l'eeprom, votre PIC® sera hors service en un peu moins de 28 heures. Soyez donc vigilants et vérifiez votre programme.

- Avant d'écrire dans l'eeprom, vous devez vérifier qu'une autre écriture n'est en cours. Utilisez le bit WR du registre EECON1. S'il vaut 0, il n'y a pas d'écriture en cours.

14.10 Utilisation pratique de la mémoire « eeprom »

Maintenant, nous allons de nouveau modifier notre programme pour qu'il écrive dans l'eeprom. Nous allons incrémenter la durée de temporisation tous les 16 clignotements de la LED (donc tous les 32 passages dans la routine d'interruption), et sauvegarder cette nouvelle valeur dans l'eeprom.

Nous n'aurons pas besoin des interruptions eeprom ici, mais vous devez avoir compris le principe des interruptions suffisamment bien pour pouvoir vous en servir en cas de besoin.

Nous allons donc procéder à la modification de notre routine d'interruption. Nous devons ajouter un second compteur (cmpt2) en zone RAM. Déclarons donc cette variable.

```
cmpt2 : 1                ; compteur de passages 2
```

Oui, je sais, il y avait moyen d'optimiser tout ça pour n'utiliser qu'un compteur. Ce n'est pas le but de cet exercice. Autant rester clair et centrer notre attention sur le sujet de ce chapitre.

La modification est très simple, en réalité. Dans la routine d'interruption timer, nous incrémenterons le compteur 2 qui sera utilisé dans le programme principal.

Un petit mot sur le bug des interdictions d'interruption sur le 16C84

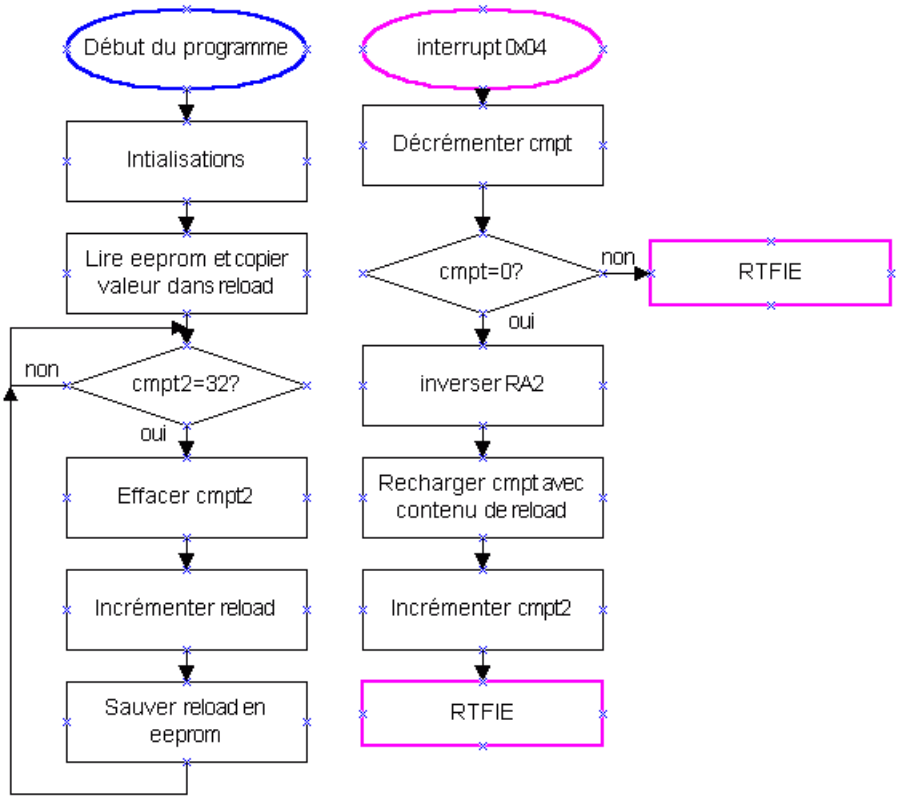
Sur le 16C84, la fin de l'autorisation des interruptions n'était prise en compte qu'après l'instruction suivante. Donc, entre « bcf INTCON, GIE » et l'instruction suivante, une interruption pouvait avoir lieu.

Comme « retfie » remettait automatiquement le bit GIE à 1, l'interruption était remise en service à l'insu du programmeur.

Il était donc nécessaire, après la mise de à 0 de GIE, de vérifier si cette mise à 0 s'était effectivement réalisée. Dans le cas contraire, c'est qu'une interruption avait eu lieu à cet endroit, et il fallait donc recommencer l'opération.

A partir du 16F84, les interruptions sont interdites dès l'instruction exécutée, et donc avant l'exécution de l'instruction suivante . Le bug n'existe donc plus.

Voici le nouvel ordinogramme obtenu.



Et la routine d'interruption timer :

```
;*****************************************************************
;                     INTERRUPTION TIMER 0                      *
;*****************************************************************
inttimer
            ; tester compteur de passages
            ; --------------------------
   decfsz   cmpt , f            ; décrémenter compteur de passages
   return                       ; pas 0, on ne fait rien
            ; inverser LED
            ; ------------
   BANK0                        ; par précaution
   movlw    b'00000100'         ; sélectionner bit à inverser
   xorwf    PORTA , f           ; inverser LED

      ; recharger compteur de passages
      ; ------------------------------
   movf     reload , w          ; charger valeur contenue dans reload
   movwf    cmpt                ; dans compteur de passages

      ; incrémenter compteur de passages 2
      ; ----------------------------------
   incf     cmpt2 , f           ; incrémenter compteur de passages2
   return                       ; fin d'interruption timer
```

Nous allons écrire dans l'eeprom depuis notre programme principal. Il y a deux raisons à ne pas écrire dans l'eeprom depuis notre routine d'interruption timer.

- Il faut, dans la mesure du possible, quitter le plus rapidement possible une routine d'interruption, celles-ci étant en général dans un programme, réservées aux traitements urgents des informations.

- Tout ce qui peut être facilement placé ailleurs doit l'être. Nous ne pouvons par remettre les interruptions en service (GIE) depuis une routine d'interruption. Or, notre macro d'écriture eeprom remet GIE à 1. Si vous vouliez utiliser cette macro dans une routine d'interruption, vous devriez au préalable supprimer la remise en service de GIE.

Voyons donc notre programme principal :

```
;*****************************************************************
;                       PROGRAMME PRINCIPAL                      *
;*****************************************************************
start
      ; tester si 16 inversions de LED
      ; ------------------------------
   btfss    cmpt2 , 5          ; tester si 32 passages
   goto     start              ; non, attendre
   clrf     cmpt2              ; oui, effacer compteur2

      ; incrémenter reload
      ; ----------------------
   incf     reload , f         ; incrémenter reload
   incf     reload , f         ; incrémenter 2 fois c'est plus visible

     ; Tester si écriture précédente eeprom terminée
     ; ---------------------------------------------
                              ; facultatif ici, car le temps écoulé
                              ; est largement suffisant
   BANK1                      ; passer banque1
wait
   btfsc    EECON1 , WR       ; tester si écriture en cours
   goto     wait              ; oui, attendre
   BANK0                      ; repasser en banque 0

     ; écrire contenu de reload en eeprom
     ; ----------------------------------
   movf     reload , w   ; charger reload
   WRITEE   eereload           ; écrire à l'adresse 0x00
   goto     start              ; boucler
   END                         ; directive fin de programme
```

Compilez le programme et chargez-le dans le PIC®. Placez le montage sur votre platine d'essais. Remarquez que la LED clignote à une fréquence approximative de 1Hz. Au 17ème allumage, la fréquence de clignotement diminue légèrement, et ainsi de suite tous les 16 passages.

Estimez la fréquence de clignotement en cours, et coupez l'alimentation du PIC®. Attendez quelques secondes, et remettez le PIC® sous tension. La LED clignote à la fréquence précédente, car le paramètre a été sauvegardé en EEPROM .

Le fichier tel qu'il devrait être à la fin de cette leçon est disponible sous la dénomination « eep_test.asm»

14.11 Sécurisation des accès en mémoire « eeprom »

Lorsque vous écrivez dans la zone eeprom d'un PIC®, vous ne pouvez jamais être sûr qu'une coupure de courant ne va pas interrompre la procédure. En conséquence, vous devez vous assurer que les données dans l'eeprom sont bien valides avant de les utiliser au redémarrage suivant.

Voici une procédure qui permet d'en être certain.

- Placez un en-tête dans le premier ou les premiers octets de votre eeprom. Par exemple, nous placerons 0x55 dans l'octet 0.

- Lors de l'écriture de nouvelles valeurs dans l'eeprom, nous commençons par effacer notre en-tête.

- Puis nous écrivons notre ou nos donnée(s) dans l'eeprom

- Pour finir, nous réinscrivons notre en-tête dans l'eeprom.

- Lors d'un démarrage du PIC®, si l'en-tête est présent, cela signifie que les données eeprom sont valides. Dans le cas contraire, le cycle d'écriture a été interrompu. A vous alors de réagir en conséquence, par exemple en réinscrivant dans l'eeprom les valeurs par défaut.

- Pour augmenter la sécurité du système pour les données critiques, vous pouvez placer plusieurs octets d'en-tête.

14.12 Conclusion

Voici encore une nouvelle étape franchie dans votre connaissance du 16F84. Vous pouvez maintenant utiliser la zone eeprom de votre PIC® pour y stocker vos données rémanentes.

N'hésitez pas à expérimenter, mais n'oubliez pas de vérifier votre programme pour éviter les écritures inutiles et trop fréquentes dans votre eeprom.

Pas de panique cependant, si votre programme est correctement écrit, vous n'êtes pas prêt d'atteindre les 1.000.000 de cycles d'écritures garantis.

15. Le watchdog

Le watchdog, ou chien de garde est un mécanisme de protection de votre programme. Il sert à surveiller si celui-ci s'exécute toujours dans l'espace et dans le temps que vous lui avez attribués.

15.1 Le principe de fonctionnement

La mise en service ou l'arrêt du watchdog se décide au moment de la programmation de votre PIC®, à l'aide de la directive _CONFIG. Si « _WDT_OFF » est précisé, le watchdog ne sera pas en service. Si au contraire vous précisez « _WDT_ON », le watchdog sera actif.

IL N'EST DONC PAS POSSIBLE DE METTRE EN OU HORS SERVICE LE WATCHDOG DURANT L'EXECUTION DE VOTRE PROGRAMME.

Le fonctionnement du watchdog est lié à un timer interne spécifique, qui n'est pas synchronisé au programme, ni à un événement extérieur.

La durée spécifique de débordement de ce timer est approximativement de 18ms. Cette valeur est à prendre avec précaution, car elle varie en fonction de différents paramètres comme la tension d'alimentation ou la température.

La valeur minimale de 7ms est celle que vous devrez utiliser dans la pratique.

En effet, Microchip® vous garanti qu'aucun PIC® ne provoquera un reset avant ces 7ms. Il vous indique que le temps moyen de reset de ses PIC® sera de 18ms, mais il ne vous garantit pas ce temps, c'est juste un temps « généralement constaté ».

Chaque fois que l'instruction clrwdt est envoyé au PIC®, le timer du watchdog est remis à 0, ainsi que la valeur contenue dans son prédiviseur. Si par accident cette instruction n'est pas reçue dans le délai prévu, le PIC® est redémarré à l'adresse 0x00 et le bit *TO* du registre STATUS est mis à 0.

En lisant ce bit au démarrage, vous avez donc la possibilité de détecter si le PIC® vient d'être mis sous tension, ou si ce démarrage est du à un « plantage » de votre programme.

15.2 Le prédiviseur et le watchdog

Nous avons vu dans les leçons précédentes que le prédiviseur pouvait être affecté au tmr0 ou au watchdog, via le bit PSA du registre OPTION. Si nous décidons de mettre le prédiviseur sur le watchdog (PSA = 1) , le tableau de la page 16 du datasheet nous donnera les valeurs du prédiviseur obtenues suivant les bits PS0/PS2.

En réalité, pour le watchdog, il s'agit d'un postdiviseur, mais cela ne concerne que l'électronique interne du PIC®. Vous n'avez pas à vous tracasser pour cette différence.

Ce postdiviseur multiplie le temps de débordement du timer du watchdog. Par exemple, avec un diviseur de 2, vous obtenez un temps minimal de 7ms*2 = 14ms. En sachant que le reset s'effectuera en réalité en général après une durée de 18ms*2= 36ms.

Donc, avec un quartz de 4MHz, cela vous oblige à envoyer l'instruction clrwdt au moins une fois tous les 14.000 cycles d'instructions. Dans la plupart des cas, le reset s'effectuera en réalité après 18ms*2 = 36ms, soit 36.000 cycles d'instructions.

15.3 Les rôles du watchdog

Le watchdog est destiné à vérifier que votre programme ne s'est pas « égaré » dans une zone non valide de votre programme (parasite sur l'alimentation par exemple), ou s'il n'est pas bloqué dans une boucle sans fin (bug du programme). Il sert également à réveiller un PIC® placé en mode « sleep », ce que nous verrons plus tard.

Notez que nous parlons du module watchdog intégré au PIC®. Cependant, tout microcontroleur (et donc un PIC®) peut être équipé d'un système de watchdog externe (circuit resetté à intervalles périodiques via une pin du microcontroleur), qui présente également des avantages particuliers, comme celui de pouvoir agir directement sur l'électronique externe (coupure de l'alimentation générale par exemple).

Je vais lister ici quelques notions liées au watchdog, qu'il soit interne au PIC®, ou éventuellement externe :

1) Le watchdog est un mécanisme interne ou externe destiné à assurer une sécurité relative à l'application placée sur site. Il s'active lorsqu'il n'est pas sollicité à intervalles réguliers. Il s'assure que le logiciel embarqué semble continuer à fonctionner selon la séquence prévue.

2) Le watchdog peut servir selon les cas à redémarrer une application depuis le début, à la stopper, à la mettre dans un état sécurisé, à signaler un défaut, ou à toute autre action voulue par le concepteur de l'application

3) Le watchdog intégré au PIC® est tributaire de son fonctionnement électronique correct. Son rôle consiste à provoquer un reset du PIC®, à charge du concepteur du logiciel embarqué de prendre les mesures nécessaires éventuelles si le reset est dû au watchdog (arrêt, redémarrage ordinaire ou limité, signalisation etc.). Des bits spécifiques sont prévus pour tester la cause d'un reset au sein du programme.

4) Le watchdog peut être externe, ce qui, au prix d'une plus grande complexité de la carte, peut présenter l'avantage de pouvoir agir même en cas de défaillances électroniques du PIC® et de ses commandes, le mécanisme n'étant plus tributaire du fonctionnement de celui-ci. Il peut ainsi par exemple couper l'alimentation générale d'un système, stopper des moteurs, etc. même si les circuits de commande sont défectueux.

5) Le watchdog intégré est une sécurité très efficace au vu de sa simplicité et relativement fiable pour se sortir de situations non prévues plaçant le programme du PIC® dans un état indéterminé. Il devrait d'office être utilisé pour toute application finalisée.

6) Le watchdog peut remédier à des situations anormales provoquées aussi bien par un bug non détecté en phase de debuggage que par une anomalie de déroulement du programme provoqués soit par un phénomène externe (perturbations, problèmes d'alimentation etc.), par un état non prévu d'un périphérique (blocage en cours de transmission), ou toute autre situation anormale.

7) Le watchdog n'est pas une méthode infaillible et absolue permettant de se sortir de toutes les situations délicates. Pour les applications à risque critique ou létal (ascenseurs, machines-outils, etc.), des mécanismes redondants de sécurités supplémentaires hardwares (sans "intelligence embarquée") doivent toujours être prévus. Le port de la ceinture de sécurité augmente votre sécurité, mais ne garantira jamais que vous vous sortirez indemne d'un accident.

8) Le watchdog n'est pas destiné à masquer des erreurs de conception des logiciels, un logiciel correctement conçu doit pouvoir tourner sans problème sans activation du watchdog, qui ne sera donc validé qu'après la phase de debuggage.

9) Il ne faut pas insérer d'instruction « clrwdt » dans les routines d'interruption, sous peine de cacher une anomalie au niveau du programme principal.

10) Dans les PIC®, le watchdog intégré peut permettre la sortie du mode veille du PIC® à intervalles de temps réguliers et plus ou moins précis.

11) Les PIC® 16F (au contraire d'autres modèles de PIC®) sont dépourvus de l'instruction « reset ». Une façon simple de provoquer un reset software est d'utiliser une boucle sans fin ne contenant pas d'instruction « clrwdt », ce qui provoquera un reset du PIC® par débordement du watchdog.

15.4 Utilisation correcte du watchdog

La première chose à faire, si vous désirez profiter de cette protection intégrée, est de paramètrer la _CONFIG pour la mise en service du watchdog. La première chose à constater, c'est que :

Si vous indiquez « _WDT_ON » pour un programme qui ne gère pas le watchdog, celui-ci redémarrera sans arrêt, et donc ne fonctionnera pas, car il ne contiendra aucune instruction « clrwdt ».

C'est une erreur fréquente pour ceux qui ne maîtrisent pas les bits de configuration de leur programmateur. Les bits de configurations indiqués dans le fichier sont en effet modifiables par la plupart des logiciels de programmation, qui sont capables de forcer une valeur de _CONFIG différente de celle prévue par le concepteur du programme PIC®.

Ceci est d'autant plus fréquent que certains programmeurs oublient d'inclure la directive «_CONFIG » dans leur programme, ce qui ne permet pas la configuration automatique des flags de configuration. C'est une très mauvaise pratique, car l'utilisateur final devra savoir (ou deviner) quel est l'état des flags qu'il lui faudra configurer au moment de la programmation.

Ensuite, vous devez placer une ou plusieurs instructions « clrwdt » dans votre programme en vous arrangeant pour qu'une instruction « clrwdt » soit reçue dans les délais requis par votre PIC®. Pour rappel, ne tenez pas compte du temps nominal de 18ms, mais plutôt du temps de la situation la plus défavorable. Ce temps est de minimum 7ms. En prenant ce temps comme temps maximum autorisé, vous êtes certain que vote programme fonctionnera dans toutes les conditions.

15.5 Ce qu'il ne faut pas faire

Souvenez-vous qu'une interruption interrompt le programme et branche à l'adresse 0x04. Une fois l'interruption terminée, le programme est reconnecté à l'endroit où il se trouvait, même si c'est hors de votre zone normale de programme.

Donc, si vous placez une instruction « clrwdt » dans une routine d'interruption, cette instruction risque de s'exécuter même si votre programme est planté. C'est donc le contraire du but recherché. En conséquence :

IL NE FAUT JAMAIS UTILISER L'INSTRUCTION CLRWDT DANS UNE ROUTINE D'INTERRUPTION.

15.6 Mesure du temps réel du watchdog

La valeur de 7ms est la valeur minimale, la valeur de 18ms est la valeur généralement constatée. Mais quelle est la valeur effective de votre propre PIC® ?

Nous allons tenter de répondre à cette question.

Effectuez un copier/coller de votre fichier « Led_cli.asm ». Renommez ce fichier en « wdt.asm » et créez un nouveau projet.

Dans un premier temps, ne touchez pas à la _CONFIG

Modifiez la valeur du registres OPTION

```
OPTIONVAL    EQU    B'10001111'  ; Valeur registre option
                                 ; Résistance pull-up OFF
                                 ; préscaler Wdt = 128
```

Ensuite, supprimez la variable cmpt3 et modifiez la routine tempo pour enlever la boucle extérieure.

```
;*****************************************************************
;                      SOUS-ROUTINE DE TEMPORISATION              *
;*****************************************************************
;-----------------------------------------------------------------
; Cette sous-routine introduit un retard de 250ms.
; Elle ne reçoit aucun paramètre et n'en retourne aucun
;-----------------------------------------------------------------
tempo
     clrf    cmpt2              ; effacer compteur2
boucle2
     clrf    cmpt1              ; effacer compteur1
```

```
boucle1
     nop                        ; perdre 1 cycle
     decfsz  cmpt1 , f          ; décrémenter compteur1
     goto    boucle1            ; si pas 0, boucler
     decfsz  cmpt2 , f          ; si 0, décrémenter compteur 2
     goto    boucle2            ; si cmpt2 pas 0, recommencer boucle1
     return                     ; retour de la sous-routine
```

Ajoutons la ligne

```
   clrwdt                       ; effacer watchdog
```

Juste après l'étiquette init pour être bien sûr de remettre le watchdog à 0 (en réalité, inutile).

Modifions pour finir le programme principal pour qu'il allume la LED après 250ms :

```
;*****************************************************************
;                       PROGRAMME PRINCIPAL                      *
;*****************************************************************
start
   call     tempo          ; on attends 250ms
   bsf      LED            ; allumer la LED
loop
   goto     loop           ; on reste ici
   END                     ; directive fin de programme
```

Compilez le programme, chargez-le dans votre PIC®, et alimentez votre platine.

Que se passe-t-il ? Et bien, après approximativement ¼ seconde, la LED s'allume, et c'est tout. Il ne se passe plus rien. C'est bien ce que nous avions prévu.

Modifiez maintenant la ligne _CONFIG pour mettre le watchdog en service :

```
__CONFIG   _CP_OFF & _WDT_ON & _PWRTE_ON & _HS_OSC
```

Recompilez votre programme, et rechargez-le dans votre PIC®. Alimentez votre montage.

Que se passe-t-il ? Et bien maintenant la LED clignote. Vous constaterez probablement que le temps séparant deux extinctions (ou deux allumages) est approximativement 2secondes.

Explication

Et bien, notre watchdog n'a pas été remis à zéro depuis le démarrage du PIC®. Donc, une fois le temps : « durée de base * prédiviseur » atteint, un reset est provoqué qui entraîne une extinction de la LED et un redémarrage sans fin du cycle.

Pour connaître la base de temps de notre watchdog personnel dans les circonstances actuelles de tension et de température, nous devons donc diviser le temps entre 2 extinctions (ou allumage) par le prédiviseur, c'est à dire 128.

Personnellement, j'ai chronométré 2,2 secondes. Donc, mon watchdog travaille avec un temps de base de : 2200ms / 128 = 17,2 ms. On est donc bien dans les environs des 18ms

typiques annoncées. Remarquez que c'est moins que les 18ms typiques. Donc, si j'avais utilisé un clrwdt toutes les 18ms, mon programme aurait planté sur ce PIC®. Preuve que c'est bien la valeur de 7ms qu'il faut prendre en compte.

15.7 Simulation du plantage d'un programme

Effectuez de nouveau une copie de votre fichier « Led_cli.asm» et renommez cette copie « secur.asm ». Créez un nouveau projet.

Changez la ligne de configuration :

```
OPTIONVAL EQU H'0F'             ; Valeur registre option
                                ; Résistance pull-up ON
                                ; Préscaler wdt à 128
```

et le DEFINE (vu que nous avons modifié notre platine)

```
#DEFINE BOUTON     PORTB,0      ; bouton-poussoir
```

Modifiez ensuite votre programme principal :

```
start
   bsf      LED          ; allumer la LED
   call     tempo        ; appeler la tempo de 0.5s
   bcf      LED          ; éteindre LED (LEDOFF)
   call     tempo        ; appeler la tempo de 0.5s
   btfsc    BOUTON       ; tester bouton-poussoir
   goto     start        ; pas pressé, boucler
plante
   goto     plante       ; le programme n'est pas sensé
                         ; arriver ici
                         ; simulation de plantage
   END                   ; directive fin de programme
```

De cette manière, une pression du bouton-poussoir envoie le programme dans une zone que nous avons créée et qui simule un plantage du programme sous forme d'une boucle sans fin.

Compilez le programme et chargez-le dans votre PIC®. Alimentez le montage : la LED clignote.

Maintenez un instant le bouton-poussoir enfoncé, la LED s'arrête de clignoter, le programme est dans une boucle sans fin qui simule dans notre cas le plantage suite, par exemple, à un parasite qui a éjecté notre programme hors de sa zone de fonctionnement normal.

15.7.1 Correction avec utilisation du watchdog

Modifions maintenant notre programme. Tout d'abord, nous mettons le watchdog en service, comme vu précédemment dans la _CONFIG. Nous allons ensuite nous arranger pour remettre notre watchdog à 0 à intervalles réguliers.

Voici donc notre programme principal modifié :

```
start
   bsf      LED          ; allumer la LED
   clrwdt                ; effacer watchdog
   call     tempo        ; appeler la tempo de 0.5s
   bcfLED                ; éteindre LED (LEDOFF)
   clrwdt                ; effacer watchdog
   call     tempo        ; appeler la tempo de 0.5s
   btfsc    BOUTON       ; tester bouton-poussoir
   goto     start        ; pas pressé, boucler
plante
   goto     plante       ; le programme n'est pas sensé
                         ; arriver ici
                         ; simulation de plantage
   END                   ; directive fin de programme
```

Nous avions programmé notre watchdog avec un prédiviseur de 128, ce qui nous impose d'envoyer une commande clrwdt toutes les 7*128 ms, soit toutes les 896ms.

Comme l'appel à notre tempo prend 500ms, nous devrons donc envoyer clrwdt avant ou après chaque appel de tempo, pour ne pas dépasser ce temps.

Nous aurions pu, au lieu de ces 2 instructions, utiliser un seul « clrwdt » dans la sous-routine « tempo ».

Recompilons notre programme, et rechargeons le PIC®. Lançons l'alimentation. La LED clignote toujours.

Pressez le bouton quelques instants, la LED s'arrête de clignoter un instant, puis recommence de nouveau. La watchdog a récupéré le plantage de votre programme.

15.8 Choix de la valeur du prédiviseur

En général, il faut essayer de calculer le prédiviseur de façon à ne pas devoir placer de commande clrwdt en de trop nombreux endroits.

Il faut également tenir compte du temps de réaction obtenu en augmentant le prédiviseur. Si une récupération de plantage en 2 secondes vous convient ou si votre programme nécessite une récupération en 18ms max conditionnera la valeur du prédiviseur à utiliser. Tout est donc histoire de compromis.

Vous n'aurez cependant pas toujours le choix. Si votre prédiviseur est déjà occupé pour le timer0, par exemple, il ne vous restera pas d'autres choix que d'envoyer une commande clrwdt toutes les 7ms.

Dans le cas présent, vous auriez du placer une instruction de ce type au cœur de la routine de temporisation, car celle-ci dure plus de 7ms.

Rappelez-vous de ne pas utiliser cette instruction dans une routine d'interruption, car cela est contraire au principe même du watchdog (sauf cas très spéciaux). Vous pouvez par contre utiliser clrwdt dans une sous-routine sans aucun problème.

15.9 Temps typique, minimal, et maximum

Nous avons vu apparaître plusieurs notions de temps de watchdog. Il est important de bien effectuer la distinction entre les différentes valeurs. Je vais donc récapituler ici :

- Le temps typique (18ms) est le temps que met EN GENERAL le watchdog pour provoquer le reset de votre programme en cas de plantage. C'est donc le « temps de réaction » normal (ou typique) du watchdog

- Le temps minimal (7ms), c'est le délai maximum dont vous disposez entre 2 instructions « clrwdt » pour éviter un reset de votre programme non désiré.

- Le temps maximum (33ms), c'est le temps de réaction du watchdog dans le cas le plus défavorable en fonction du composant et des conditions d'utilisation. Microchip® vous garantit ici que le reset s'effectuera au maximum en 33 ms.

15.10 Conclusion

Vous êtes maintenant en mesure de créer des programmes résistants aux plantages classiques, pour peu que vous utilisiez judicieusement le watchdog. En général, il est préférable de faire l'effort de l'utiliser, car le surplus de travail est négligeable en contrepartie de la sécurité de fonctionnement obtenue.

ATTENTION : Le watchdog utilisé en protection dans le cas d'une programmation correcte et d'une carte bien conçue ne devrait jamais entrer en fonctionnement. Il n'agira donc en général que dans de très rares occasions (parasites violents, orage). Il ne doit pas servir à masquer une erreur de conception de votre programme. Celui-ci doit pouvoir fonctionner sans le secours du watchdog.

Je vous conseille donc de réaliser vos programmes de la façon suivante :

- Vous écrivez votre programme en y plaçant comme prévu les instructions clrwdt, mais VOUS NE METTEZ PAS LE WATCH-DOG EN SERVICE avec la directive _CONFIG
- Vous mettez votre programme en PIC®, et vous le debuggez en le laissant tourner suffisamment de temps pour être certain de l'absence de bug.
- Une fois le programme fiabilisé, vous reprogrammez votre PIC® avec le watchdog en service.

De cette façon, vous êtes certain que votre watchdog ne va pas servir à récupérer une erreur de programmation de votre part (en relançant le PIC® lors d'un blocage dans une boucle sans fin, par exemple).

16. Le mode Sleep

Nous allons étudier dans cette leçon un mode très particulier des PIC®, qui leur permet de se mettre en sommeil afin de limiter leur consommation.

16.1 Principe de fonctionnement

Le mode « sleep » ou « power down » est un mode particulier dans lequel vous pouvez placer votre PIC® grâce à l'instruction « sleep ». Une fois dans ce mode, le PIC® est placé en sommeil et cesse d'exécuter son programme. Dès réception de cette instruction, la séquence suivante est exécutée :

- Le watchdog est remis à 0, exactement comme le ferait une instruction « clrwdt ».
- Le bit *TO* du registre STATUS est mis à 1.
- Le bit *PD* du registre STATUS est mis à 0.
- L'oscillateur est mis à l'arrêt, le PIC® n'exécute plus aucune instruction.

Une fois dans cet état, le PIC® est à l'arrêt. La consommation du circuit est réduite au minimum. Si le tmr0 est synchronisé à l'horloge interne, il est également mis dans l'incapacité de compter.

Par contre, il est très important de se rappeler que le timer du watchdog possède son propre circuit d'horloge. Ce dernier continue de compter comme si de rien n'était.

Pour profiter au maximum de la chute de la consommation (montage sur piles par exemple), Microchip® recommande de veiller à ce que les pins d'entrées/sorties et l'électronique connectée soient à des niveaux 0 ou 1 tels qu'il n'y ait aucun passage de courant qui résulte du choix de ces niveaux. Ceci vous montre que les niveaux présents sur les pins au moment de la mise en mode sleep restent d'application.

16.2 La sortie du mode « sleep »

Le passage en mode « sleep » n'a réellement d'intérêt que s'il est possible d'en sortir. Le 16F84 ne réagit dans ce mode qu'aux événements suivants, qui sont seuls susceptibles de replacer le 16F84 en mode de fonctionnement normal. Ces événements sont les suivants :

- Application d'un niveau 0 sur la pin *MCLR*. Ceci provoquera un reset du 16F84. Le PIC® effectuera un reset classique à l'adresse 0x00. L'utilisateur pourra tester les bits *TO* et *PD* lors du démarrage pour vérifier l'événement concerné (reset, watch-dog, ou mise sous tension).
- Ecoulement du temps du timer du watchdog. Notez que pour que cet événement réveille le PIC®, il faut que le watchdog ait été mis en service dans les bits de configuration.

Dans ce cas particulier, le débordement du watchdog ne provoque pas un reset du PIC®, il se contente de le réveiller. L'instruction qui suit est alors exécutée au réveil.

- Apparition d'une interruption RB0/INT, RB ou EEPROM.

Notez dans ce dernier cas, que pour qu'une telle interruption puisse réveiller le processeur, il faut que les bits de mise en service de l'interruption aient été positionnés. Par contre le bit GIE n'a pas besoin d'être mis en service pour générer le réveil du PIC®.

Vous pouvez donc décider par exemple de réveiller le PIC® à la fin du cycle d'écriture EEPROM. Pour ce faire, vous devez mettre le bit EEIE de INTCON à 1 et lancer le cycle d'écriture, suivi par l'instruction « sleep » Une fois l'écriture terminée, le PIC® est réveillé et poursuit son programme.

16.3 Réveil avec GIE hors service.

Si votre PIC® est réveillé par une interruption alors que le BIT GIE de INTCON est mis à 0, le programme se poursuivra tout simplement à l'instruction qui suit l'instruction « sleep ».

16.4 Réveil avec GIE en service

Dans le cas où votre bit GIE est positionné, un réveil suite à une interruption entraînera la séquence suivante.

- L'instruction qui suit l'instruction « sleep » est exécutée.

- Le programme se branche ensuite à l'adresse 0x04 comme une interruption ordinaire.

Notez que si vous ne voulez pas exécuter l'instruction qui suit l'instruction « sleep », il vous suffit de placer à cet endroit une instruction « nop ».

16.5 Mise en sommeil impossible

Si le bit GIE est positionné à 0, et que les bits de mise en service et le flag d'une interruption sont tous deux à 1 au moment de l'instruction « sleep » (par exemple INTE=INTF =1), l'instruction « sleep » est tout simplement ignorée par le processeur.

Ceci est logique, car les conditions de réveil sont déjà présentes avant la mise en sommeil. C'est donc à vous de remettre éventuellement ces bits à 0. Le bit *PD* vous permettra de savoir si votre instruction « sleep » s'est exécutée (*PD* = 0).

Si le bit GIE est positionné à 1, il va sans dire, mais je l'écris tout de même, que le cas précédent ne pourra survenir, vu qu'une interruption serait alors générée, interruption qui provoquerait l'effacement du flag de INTE ou INTF. Excepté bien entendu si vous avez placé votre instruction « sleep » à l'intérieur de la routine d'interruption.

Notez également que dans le cas où l'instruction « sleep » n'est pas exécutée, votre watchdog n'est pas non plus remis à 0. Si vous deviez le faire à ce moment, et si vous n'êtes pas sûr de la bonne exécution de « sleep », ajoutez l'instruction « clrwdt » avant cette instruction.

16.6 Utilisation du mode « sleep »

Nous allons réaliser un petit exercice sur la mise en veille de notre PIC®.

Effectuez un copier/coller de votre nouveau fichier « m16F84.asm » et renommez-le en « sleep.asm » Créez un nouveau projet.

Placez le prédiviseur sur le watchdog avec une valeur de 32. Ceci nous donne une valeur typique de débordement de 18ms*32 = 576 ms.

```
OPTIONVAL EQU H'8D'             ; Valeur registre option
                                ; Préscaler wdt à 32
```

Définissez votre LED sur la pin RA2

```
#DEFINE LED  PORTA,2            ; LED de sortie
```

Placez ensuite votre LED en sortie dans la routine d'initialisation (attention, en banque1)

```
   bcf      LED                 ; LED en sortie
```

Ecrivons ensuite notre programme principal :

```
;*****************************************************************
;                       PROGRAMME PRINCIPAL                      *
;*****************************************************************
start
   bsf      LED          ; Allumage de la LED
   sleep                 ; mise en sommeil
   bcf      LED          ; extinction de la LED
   sleep                 ; mise en sommeil
   goto     start        ; boucler
```

Le fonctionnement est très simple. Après avoir allumé ou éteint la LED, le PIC® est mis en sommeil. Une fois le temps du watchgog écoulé, le PIC® se réveille et exécute l'instruction suivante. Voilà donc un programme de clignotement de votre LED ultracourt et avec une consommation minimale.

De plus, voici une autre façon simple de mesurer le temps de watchdog de votre propre PIC®.

Remarque

Notez que le temps de réveil de votre PIC® n'est pas instantané. En effet, si vous utilisez un quartz, le PIC® attendra 1024 cycles d'horloge avant de relancer le programme. Il vous faudra en tenir compte.

Ce « temps mort » est nécessaire pour que l'oscillateur de précision à quartz atteigne une certaine stabilité.

Attention donc, soyez conscients que le réveil de votre PIC® prendra un certain temps.

16.7 Cas typiques d'utilisation

Ce mode de fonctionnement est principalement utilisé dans les applications dans lesquelles la consommation en énergie doit être économisée (piles). On placera donc dans ce cas le PIC® en mode « Power Down » ou « sleep » aussi souvent que possible.

Une autre application typique est un programme dans lequel le PIC® n'a rien à faire dans l'attente d'un événement extérieur particulier. Dans ce cas, plutôt que d'utiliser des boucles sans fin, une instruction « sleep » pourra faire efficacement l'affaire.

16.7 Pour une consommation minimale

Le mode sleep assure une mise en veille du PIC®. Son rôle premier étant l'économie d'énergie, il incombe de baisser la consommation globale du composant. Pour ce faire, il ne suffit pas de placer le PIC® en mode sleep, il faut encore veiller à tout son environnement.

Tout d'abord, il faut savoir que le passage en mode sleep ne modifie pas l'état des pins configurées en sortie. Si au moment de passer en mode sleep votre PIC® allumait une led, alors elle restera allumée (et consommera du courant). Autrement dit, placez, avant de passer en sleep, toutes les pins configurées en sortie, soit à 1, soit à 0, de façon à ce qu'aucune ne provoque une consommation de courant.

Il ne faut pas évidemment oublier la consommation des résistances de pull-up sur le PORTB, si vous les avez mises en service. Toute entrée reliée à la masse consommera alors du courant.

Toutes les entrées non utilisées ainsi que l'entrée Tocki devront être forcées soit à la masse, soit à Vdd, car si vous les laissez flottantes, chaque transition accidentelle de niveau consommera un petit courant. Notez que vous arrivez au même résultat en configurant les pins inutilisées comme sorties.

Une éventuelle horloge externe devra être stoppée.

Pour d'autres PIC®, il y a encore d'autres mesures à prendre, comme l'arrêt du convertisseur analogique/numérique. Vous l'aurez compris, il faut considérer le problème de la consommation sous son ensemble, et ne pas se borner à écrire l'instruction « sleep ».

16.8 Conclusion

Vous êtes maintenant capable de placer un PIC® en mode de sommeil afin d'économiser au maximum l'énergie. Je ne doute pas que vous trouverez des tas d'applications pour ce mode très pratique et très simple à gérer.

17. Le reste du datasheet

Et voilà, nous arrivons au bout de l'étude de notre 16F84. Dans ce chapitre, je vais parcourir avec vous le datasheet du 16F84 pour voir tout ce dont nous n'avons pas parlé jusqu'à présent.

C'est donc une sorte de fourre-tout que je vous propose ici. Je n'entrerai cependant pas dans les descriptions techniques détaillées, car ceci n'intéressera que les électroniciens. Ceux-ci étant parfaitement aptes à comprendre une figure telle que la figure 3-1. Pour les autres, ceci n'apportera rien de plus à l'utilisation des PIC®.

Cette leçon vous montrera du même coup comment comprendre un datasheet.

Comme les datasheet sont sans cesse mis à jour chez Microchip®, je joins à la leçon celui qui m'a servi jusqu'à présent pour faire ces cours, et qui s'appelle «16F84.pdf »

17.1 La structure interne

Figure 3-1, justement, vous voyez comment est construit un 16F84. Comme je le disais plus haut, ceci ne présente qu'un intérêt limité pour l'utilisation pratique du processeur.

Cependant, vous pouvez remarquer les largeurs de bus internes qui vous rappellent les limitations des modes d'adressage. Vous voyez par exemple que le PC (Program Counter) n'a qu'une largeur de 13 bits.

Le cœur du 16F84, comme dans tout processeur, est l'ALU. C'est dans cette Unité Arithmétique et Logique que s'effectuent tous les calculs. Notez la liaison entre le registre W et l'unité ALU.

17.2 La séquence de décodage

Figure 3-2, vous voyez clairement la division d'un cycle d'horloge en 4. Chacun des 4 clocks nécessaires à la réalisation d'une instruction est détaillée. Vous voyez l'exemple 3-1 qui montre l'exécution d'un bout de programme.

Notez que durant qu'une instruction est exécutée, la suivante est déjà chargée (« fetch »). Ceci explique que lors d'un saut, l'instruction suivante chargée n'étant pas celle qui doit être exécutée, il faut alors un cycle supplémentaire pour charger la bonne instruction.

17.3 Organisation de la mémoire

Sur la figure 4-1 vous voyez l'organisation mémoire du PIC®. Notez que la pile et le PC sont situés hors de l'espace d'adressage, et donc sont inaccessibles par le programme.

17.4 Les registres spéciaux

Le tableau 4-1 vous donne une vue globale de tous les registres spéciaux. La première colonne vous donne l'adresse du registre , la seconde le nom symbolique du registre, ensuite le nom de chaque bit.

L'avant-dernière colonne vous donne la valeur de chaque bit après une mise sous tension, tandis que la dernière colonne fait de même pour les autres types de reset (watchdog et *MCLR*). Les bits notés 0 ou 1 sont ceux dont le niveau est celui indiqué. Les bits notés « u » sont les bits non affectés par un reset (unchanged = inchangé). Les bits notés « x » sont les bits dont l'état ne peut être connu à ce moment.

17.5 L'électronique des ports

Les figures 5-1 à 5-4 permettront aux électroniciens de comprendre les spécificités des ports IO au niveau de leurs caractéristiques électriques. Vous verrez alors, comme je vous l'ai indiqué, que la pin RA4, par exemple, est une sortie à « collecteur ouvert », ou plus précisément à « drain ouvert », configuration qui ne permet pas d'imposer un niveau haut sur la sortie.

17.6 Le registre de configuration

Le registre de configuration est situé à l'adresse 0x2007, hors de l'espace d'adressage normal du PIC®. Il n'est accessible qu'au moment de la programmation.

Vous accédez à ce registre à l'aide de la directive « _CONFIG », ou à l'aide de la directive « DA » précédée de la directive « ORG 0x2007 ». A ce moment, la donnée que vous indiquerez suite à la directive « DA » précisera le niveau des 14 bits utiles pour le 16F84 (5 bits pour le 16C84).

L'explication et la position de ces 14 bits sont donnés figure 8-1. N'oubliez pas qu'il s'agit ici d'un mot de 14 bits (tout comme les instructions).

A titre d'exemple, la ligne suivante :

```
__CONFIG   _CP_OFF & _WDT_ON & _PWRTE_ON & _HS_OSC
```

Correspondra donc pour un 16F84 à ceci :

```
ORG   0x2007
DA    B'111111111111110'
```

JE VOUS DECONSEILLE CE GENRE DE PRATIQUE, nous verrons plus bas pourquoi.

17.7 Les différents types d'oscillateurs

La figure 8-3 nous montre la configuration que nous avons utilisée pour l'horloge de notre PIC®. C'est l'utilisation avec quartz externe. La résistance RS est inutile pour un quartz classique. Si vous utilisez un résonateur céramique avec condensateurs intégrés, au lieu d'un quartz, vous pourrez supprimer les condensateurs C1 et C2.

La table 8-1 montre les différentes valeurs à utiliser en fonction des fréquences, ainsi que les modes correspondants sélectionnés par les bits de configuration FOSC1 et FOSC0 si vous utilisez un résonateur céramique.

La table 8-2 fait de même pour les oscillateurs à quartz.
Notez que les bits cités sont intégrés dans les _HS_ OSC et autres configurations d'oscillateur. C'est ce qui explique que vous ne les ayez jamais rencontrés directement.

La figure 8-4 indique comment utiliser une horloge externe au lieu de l'oscillateur interne. Rappelez-vous dans ce cas de ne jamais paramètrer votre _CONFIG sur RC sous peine de destruction de votre PIC®.

Les figures 8-5 et 8-6 vous montrent comment construire un oscillateur externe. Je vous conseille à titre personnel d'utiliser dans ce cas le 74HCU04 qui fonctionne bien pour des fréquences supérieures ou égales à 4MHz.

EVITEZ LES MODELES DE TYPE LS (74LS04), sous peine de déboires.

J'utilise pour ma part dans ce cas un schéma dérivé du montage parallèle indiqué. Mon montage, sur 4MHz, utilise une R de 3,3Mohms à la place des 47Kohms, je remplace la R ajustable par une fixe de 2Kohms, et je supprime la R ajustable à l'extrême gauche du schéma. Je remplace les deux condensateurs par des 27pF.

Reste le mode d'oscillateur avec réseau Résistance Condensateur. Dans ce mode, l'oscillation est assurée par deux composants passifs, sans nécessiter de quartz (pour raisons d'économie).

N'utilisez ce mode que si vos constantes de temps dans votre programme ne sont pas critiques. Connectez suivant la figure 8-7 et utilisez une Résistance de l'ordre de 47Kohms et un condensateur de l'ordre de 27 picofarads (27pF).

N'oubliez pas dans tous les cas que la vitesses d'exécution d'une instruction est le quart de la vitesse d'horloge. Utilisez la formule suivante pour calculer le temps d'un cycle de votre programme. Soit F la fréquence de l'horloge fournie et T le temps d'un cycle :

T = 4 / F.

Par exemple, pour un quartz de 4MHz : T = 4 / 4000000 = 1 microseconde.

Et réciproquement : F = 4 / T pour calculer la fréquence qui correspond à un temps de cycle calculé.

Donc, si je veux un temps de cycle de 1,5 µs (microseconde), il me faudra une horloge cadencée à :

$F = 4 / (1{,}5*10^{-6)} = 2{,}667 * 10^{6} = 2{,}667$ MHz (Mega Hertz)

17.7.1 La précision de l'oscillateur

N'oubliez pas dans vos calculs de tenir compte de l'erreur toujours existante. La tolérance de votre horloge est directement tributaire de la méthode utilisée. Une horloge à quartz donnera une bien meilleure précision qu'un simple réseau RC.

A titre d'exemple, supposons que vous désiriez construire une horloge avec une carte à PIC®. Nous allons évaluer l'ordre de grandeur des erreurs obtenues en fonction du type d'oscillateur retenu.

Nous supposerons que votre logiciel est correctement réalisé, et que l'erreur de mesure de temps, à ce niveau, est nulle.

Commençons par le réseau RC.

Si vous choisissez ce mode pour une horloge, alors vous avez vraiment fait le mauvais choix. En effet, la fréquence de l'oscillateur varie alors en fonction de la température, de la précision des composants, varie dans le temps, et en plus est différente d'un composant à l'autre. Ne vous étonnez pas, alors, si vous obtenez une erreur de 12 heures toutes les 24 heures. A rejeter.

Si maintenant vous avez décidé d'utiliser un résonateur céramique, la table 12-1 vous donne les précisions obtenues en fonction de quelques marques et modèles testés par Microchip®. Ces valeurs nous donnent une précision de l'ordre de 0.5%.

Sachant que dans une journée, il y a 24 heures, et que chaque heure contient 3600 secondes, nous pouvons dire qu'une journée compte : 24 * 3600s = 86400 s.

Une erreur de 0.5% nous donne donc une erreur estimée de $86400 * 5 * 10^{-3} = 432$ secondes. Votre horloge risque donc de « dériver » de plus de 6 minutes par jour.

Passons au quartz. Celui-ci va nous donner une erreur typique, et comme indiqué sur le tableau 12-2, de l'ordre de 50 PPM (Part Par Million), soit 0,000005%. Calculons donc la dérive de notre horloge :

$86400 * 50 * 10^{-6} = 4{,}32$ s.

Voici déjà un résultat beaucoup plus acceptable.

Ce petit aparté avait pour but de vous faire sentir par un exemple concret l'ordre de grandeur des précisions pouvant être obtenues par des méthodes courantes.

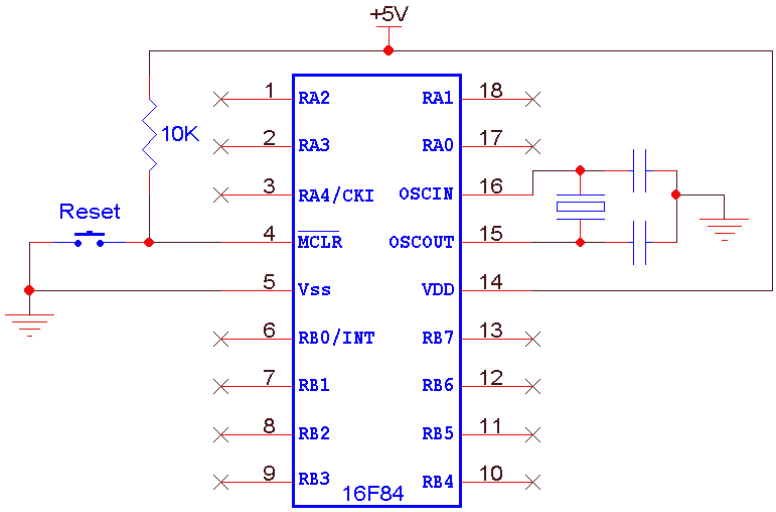
Notez, que pour notre exemple d'horloge, nous pouvions obtenir des précisions encore plus grandes. Nous pouvions par exemple utiliser un circuit d'horloge externe spécifiquement conçu pour de telles applications, ou encore compter les sinusoïdes du réseau.

En effet, cette dernière méthode est très fiable, car les sociétés de production d'électricité corrigent la fréquence en permanence, avec obligation de respecter un nombre exact de cycles au bout de 24 heures (50Hz = 50 cycles par seconde).

17.8 Le reset

Figure 8-8 vous avez le schéma du circuit de reset interne. Ne vous inquiétez pas, tout le travail a été fait pour vous. Retenez ceci : pour faire fonctionner le PIC® normalement, reliez la pin *MCLR* au +5V. La mise à la masse de cette broche provoque un reset du PIC®.

SI vous voulez par exemple un bouton reset sur votre montage, reliez la pin *MCLR* au +5 via une résistance de 10Kohms. Placez votre bouton reset entre la pin *MCLR* et la masse, comme dans ce petit schéma.



Le tableau 8-3 vous montre tous les événements liés au reset, et leurs effets sur le PC (où le programme se branchera) et le registre STATUS. Je vous traduis le tableau ci-dessous.

Evénement	Branchement (PC)	STATUS
Mise sous tension	0x00	00011xxx
MCLR à la masse durant fonctionnement normal	0x00	000uuuuu
MCLR à la masse durant mode sleep	0x00	0001uuuu
Reset par dépassement timer watchdog	0x00	00001uuu
Sortie du mode sleep par dépassement watchdog	Suite programme (PC+1)	uuu00uuu
Sortie du mode sleep par interrupt avec GIE = 0	PC + 1	uuu10uuu
Sortie du mode sleep par interrupt avec GIE = 1	PC + 1 puis 0x04	uuu10uuu

Le tableau 8-4 est très intéressant, car il vous montre le contenu de chaque registre après un reset et après un réveil. Rappelez-vous que les valeurs « x » renseignent un état inconnu, « u » signifie inchangé par rapport à la valeur précédente, et 'q' signifie que l'état dépend de la cause de l'événement (voir tableau ci-dessus).

17.9 La mise sous tension

Dès que vous placez votre PIC® sous tension, un circuit interne analyse la tension d'alimentation. Un reset automatique est généré dès que cette tension monte dans la zone 1,2 à 1,7 volts. Ceci est le reset de mise sous tension (Power On Reset). Notez déjà que ce circuit ne provoque pas un reset si la tension baisse. C'est à vous de gérer ce cas particulier s'il vous pose problème.

Notez par contre que d'autres PIC® gèrent ce problème, comme le 16F876. Je l'aborderai donc dans la seconde partie.

Ce reset déclenche un timer interne et indépendant de la vitesse du PIC®. Ce timer maintient le PIC® à l'arrêt durant un temps typique de 72ms dès détection de la condition de reset . Ce timer est appelé « PoWeR-up Timer » ou PWRT. Il peut être mis en ou hors service via le bit de configuration PWRTE. Je vous conseille de le mettre toujours en service, sauf si le temps de démarrage sous tension était critique pour votre application.

Ensuite, après écoulement du temps précédent, nous avons un temps supplémentaire sous forme d'un comptage de 1024 oscillations de l'horloge principale (Oscillator Start-up Timer).

Ceci permet de s'assurer d'un fonctionnement stable de cette horloge. Cet OST n'est pas utilisé pour le mode d'oscillateur RC, et est en service pour les mises sous tension et les réveils (pas pour les autres formes de reset, pour lequel l'oscillateur est sensé être déjà dans un état stable).

La figure 8-10 vous montre le chronogramme typique d'une mise sous tension. J'explique rapidement. Sachez que le déplacement horizontal de gauche à droite représente le temps.

La ligne 1 montre l'arrivée de l'alimentation (VDD). Une fois cette alimentation arrivée à 1,2/1,7V, le processus de mise sous tension est amorcé.

La seconde ligne vous montre que le démarrage du processus ne dépend pas du niveau de la ligne *MCLR*. Dans cet exemple, cette ligne passe à l'état haut un peu plus tard. Si vous l'avez reliée au +5V, elle passera à 1 en même temps que la ligne VDD sans aucune sorte d'influence.

Ensuite, vous voyez que le reset interne (POR) est validé dès que l'alimentation est passée au point précité. S'écoule ensuite éventuellement le temps de 72ms du PWRT s'il est mis en service.

A la fin de ce temps, l'OST démarre ses 1024 cycles d'horloge pour les modes concernés. La procédure est alors terminée.

Si à ce moment *MCLR* est à 1, le PIC® démarre directement (tableau 8-10). Si *MCLR* est toujours à 0, le PIC® démarrera instantanément dès le passage de *MCLR* à 1 (tableau 8-11).

Le tableau 8-13 vous montre ce qui se passerait si l'alimentation montait trop lentement en tension, alors que *MCLR* est relié à l'alimentation.

Dans ce cas, le reset interne serait terminé avant l'établissement de la tension normale, ce qui n'est pas conseillé. Dans ce cas, utilisez le schéma de la figure 8-9 pour ralentir la montée en tension sur *MCLR* et allonger le temps du reset.

Les figures 8-14 et 8-15 vous donnent les méthodes de protection à utiliser en cas de baisse de la tension sans arrêt complet.

17.10 Caractéristiques électriques

A partir du chapitre 11, vous trouvez toutes les spécifications électriques de votre composant. Ceci intéresse les électroniciens concepteurs de montages particuliers. Pour un usage courant, les explications des leçons précédentes suffisent largement. Pour des applications pointues au niveau des spécificités électriques, il vous faudra étudier ces tableaux plus en détail. Je pense donc inutile de détailler ces caractéristiques détaillées.

17.11 Portabilité des programmes

Voilà un point crucial. Que deviendra votre programme si vous changez de modèle de PIC® ? Voici ce que vous devez faire pour assurer la portabilité de votre programme

- Vous devez impérativement utiliser les directives prévues (_CONFIG) au détriment des accès directs style (ORG 0x2007). En effet, ces emplacements et leur contenu sont susceptibles de modifications d'un modèle à l'autre.

- Vous devez utiliser les fichiers « .inc » de Microchip® correspondant au PIC® sur lequel va tourner votre programme. Par exemple « P16f84.inc ».

- Vous devez lire les nouveaux datasheets pour analyser les différences entre le composant initial et le nouveau. Au besoin effectuez des modifications dans votre source.

- Vous pouvez ensuite recompiler votre programme, après avoir remplacé la ligne « include » par celle contenant le nouveau fichier « include »

Notez donc que votre programme sera d'autant plus facilement portable que vous aurez suivi les consignes données dans ce cours, que vous aurez utilisé au maximum les déclarations, define et macros, et que vous aurez commenté votre programme.

Notez également que si vous avez compris ce qui précède, UN FICHIER .HEX CONCU POUR UN COMPOSANT NE PEUT JAMAIS ETRE UTILISE TEL QUEL POUR UN AUTRE COMPOSANT, sauf cas très rares.

N’espérez donc pas placer tel quel votre fichier 16F84 dans un 16F876. Dans la seconde partie du cours, j’expliquerai comme procéder pour effectuer la migration de votre programme.

17.12 Les mises à jour des composants

A l’appendice E, vous trouverez un tableau donnant les différences entre le 16C84 et le 16F84. Nous allons y jeter un œil.

La première ligne montre que PWRTE a changé de niveau. En effet, PWRTE à 1 mettait en service le timer sur reset de 72ms pour le 16C84. Pour le 16F84, il faut mettre ce bit à 0.

Notez déjà que si vous avez utilisé la directive _CONFIG, il vous suffira de recompiler votre programme, car le nouveau fichier « p16F84.inc » intègre automatiquement la modification. Si par contre vous avez voulu utiliser une écriture directe en 0x2007, en dépit des avertissements de ces leçons, il vous faudra modifier votre programme.

Ensuite vous voyez que la capacité en RAM utilisateur (variables) est passée de 36 octets à 68 octets.

Puis vous constatez que Microchip® a ajouté un filtre sur la pin *MCLR* afin d’éviter que des parasites ne provoquent un reset inopinément. Ceci allonge cependant la longueur de l’impulsion nécessaire pour provoquer un reset souhaité.

Viennent ensuite un avertissement sur quelques caractéristiques électriques du PIC®. Cet avertissement vous renvoie aux différents tableaux des caractéristiques électriques.

Directement après, encore une correction au niveau du fonctionnement du PORTA, lorsque le PIC® était utilisé avec une fréquence inférieure à 500 KHz.

Puis Microchip® a ajouté un « trigger de Schmitt » au niveau de l’entrée RB0 lorsqu’elle est utilisée en interruption. Pour les non-électroniciens, sachez simplement que cela limite le risque de fausse interruption suite à un niveau incertain sur la pin.

Vous voyez également que la mise à 0 des bits 7 et 6 de EEADR ne provoque plus une modification du courant consommé par le PIC®. C’est une correction de bug. En effet, sur le 16C84, le fait de laisser ces pins à l’état haut augmentait le courant consommé. Je vous en ai déjà parlé.

Notez ensuite que le fameux CP (code protect) qui contenait un seul bit sur le 16C84, passe maintenant à 9 bits. Ceci permet de créer différents types de code protect différents sur les autres PIC® de la famille (protection data, protection eeprom etc.).

Une fois de plus, si vous aviez utilisé les directives _CONFIG, il vous suffit de recompiler votre programme après avoir simplement remplacé le type de composant dans la directive « include ». Pour les autres, il vous faudra relire le datasheet en entier pour chaque nouvelle version du même composant, et de modifier le programme en conséquence.

Constatez enfin la correction d'un bug interne lorsque vous placiez le bit GIE à 0. Désormais, une interruption ne pourra plus être générée au cycle suivant. La boucle « loop » qui était nécessaire pour vérifier la bonne mise hors service des interruptions n'est plus nécessaire sur le 16F84. Cette boucle était de la forme :

```
loop
   bcfINTCON , GIE             ; Arrêter les interruptions
   btfsc    INTCON , GIE       ; Tester si une interruption n'a pas remis GIE à
1
   goto     loop                    ; si, alors on recommence
```

Je vous en ai également parlé dans cet ouvrage.

17.13 Conclusion

Vous voilà maintenant en possession de la totalité des informations nécessaires pour développer vos propres programmes en 16F84.

Attention, il vous faudra cependant pratiquer avant de devenir un « crack » de la programmation. Vous vous direz souvent : « c'est pas possible, ça devrait marcher, mon 16F84 est défectueux ». Dans 99% des cas, ce sera un bug de votre programme, que vous trouverez peut-être après plusieurs jours de recherche. Ne vous découragez pas et relisez les documentations, même si vous êtes certains de les connaître par cœur.

Je vous ai souvent volontairement fait faire des erreurs, que j'ai corrigées par la suite. J'ai fait cette démarche dans le but de vous faciliter la vie, en vous montrant ce qui étaient parfois de fausses évidences, et en vous expliquant la bonne démarche pour obtenir un résultat fiable.

Je ne vous laisse pas encore tomber, car je vais vous expliquer quelques astuces bien pratiques pour les programmes les plus courants. Mais déjà maintenant, vous êtes capables de travailler seul.

Je vous ai fourni le dictionnaire de rimes, à vous d'écrire les poèmes…

Notes :…

18. Astuces de programmation

Dans ce chapitre, nous allons examiner quelques méthodes simples pour se tirer de situations classiques.

18.1 Les comparaisons

Quoi de plus simple que d'effectuer une comparaison entre 2 nombres. Il suffit d'effectuer une soustraction. Soit par exemple comparer mem1 avec mem2 :

```
movf    mem1 , w    ; charger mem1
subwf   mem2 , w    ; soustraire mem2 – mem1
```

Il vous suffit ensuite de tester les bits C et Z du registre STATUS pour connaître le résultat :

- Si Z = 1 , les 2 emplacements mémoires contiennent la même valeur
- Si Z = 0 et C = 1, le résultat est positif, donc mem2 est supérieur à mem1
- Si Z = 0 et C = 0, le résultat est négatif, donc mem2 est inférieur à mem1

Si vous désirez simplement comparer l'identité entre 2 valeurs, sans modifier C, vous pouvez également utiliser l'instruction « xor »

```
movf    mem1 , w    ; charger mem1
xorwf   mem2 , w    ; Si égalité, tous les bits sont à 0 et  Z est à 1
```

18.2 Soustraire une valeur de w

Supposons que vous avez une valeur dans W et que vous désirez soustraire 5 de cette valeur. Le premier réflexe est le suivant :

```
sublw   5
```

C'est une erreur classique, car vous avez en réalité effectué (5-w) au lieu de (w-5). Il vous faudra donc effectuer le complément à 2 de cette valeur pour obtenir le bon résultat. Effectuez plutôt ceci :

```
addlw  -5
```

Et oui, ajouter –5 correspond à soustraire 5. Faites-le par écrit si vous n'êtes pas convaincu. Par contre, la gestion du bit C nécessitera un peu plus de gymnastique, je vous laisse y réfléchir.

18.3 Les multiplications

Comment effectuer une multiplication ? Et bien tout simplement de la même manière que nous l'effectuons manuellement.

Nous allons créer une routine qui multiplie ensemble 2 nombres de 8 bits. Le résultat nécessitera donc 16 bits, donc 2 octets. En effet, pour obtenir le nombre de digits maximal du résultat d'une multiplication, il suffit d'additionner le nombre de digits des différentes opérandes.

Réalisons donc une multiplication manuelle. Nous allons multiplier 12 par 13. Nous allons travailler avec 4 bits multipliés par 4 bits, avec résultat sur 8 bits. Ceci afin de réduire notre explication. Exécutons donc notre multiplication manuellement.

				1	1	0	0	12
			X	1	1	0	1	13
				1	1	0	0	12
			0	0	0	0	0	0
		1	1	0	0	0	0	48
	1	1	0	0	0	0	0	96
1	0	0	1	1	1	0	0	156

Qu'avons-nous effectué ? Et bien nous avons multiplié 12 par chaque « chiffre » de 13 en commençant par la droite . Nous avons décalé d'une rangée vers la gauche chaque résultat intermédiaire avant sont addition finale.

La particularité en binaire, c'est qu'il n'y a que 2 chiffres : 0 et 1. Donc on multiplie soit par 0 (ce qui revient à ne rien faire) soit par 1 (ce qui revient à simplement recopier le chiffre).

Voici donc ce que nous obtenons en binaire :

- On multiplie 1100 par 1 . On obtient donc 1100 (D'12')
- On multiplie 1100 par 0 et on décale le résultat vers la gauche. On obtient donc 0000
- On multiplie 1100 par 1 et on décale le résultat vers la gauche. On obtient 1100 complété par 00, soit 110000, donc D '48'
- On multiplie 1100 par 1 et on décale le résultat vers la gauche. On obtient 1100 complété par 000, soit 1100000, donc D'96'.
- On additionne le tout et on obtient 10011100, soit D'156'.

Si on réalisait ce programme, il nous faudrait 4 variables supplémentaires pour stocker les résultats intermédiaires. 8 dans le cas d'une multiplication de 8 bits par 8 bits. On peut alors imaginer de se passer de ces résultats intermédiaires en procédant à l'addition au fur et à mesure du résultat intermédiaire avec le résultat final.

On obtient alors l'algorithme suivant :

- On multiplie 1100 par 1. On place le résultat dans résultat final
- On multiplie 1100 par 0. On décale une fois. On ajoute au résultat final
- On multiplie 1100 par 1. On décale 2 fois. On ajoute au résultat final
- On multiplie 1100 par 1. On décale 3 fois. On ajoute au résultat final.

Vous pouvez tenter d'écrire ce programme. Ceci reste pratique pour des multiplications de 4 par 4 bits. Pour des multiplications de 8 bits par 8 bits, vous allez devoir réaliser des tas d'additions de nombres de 16 bits avec des nombres de 16 bits. De plus vous devrez décaler le multiplicateur, ce qui impliquera, soit de le modifier (ce qui n'est peut-être pas toujours souhaitable), soit de le sauvegarder, ce qui consomme 1 octet supplémentaire.

En réfléchissant un peu, on peut se dire que plutôt que de décaler le multiplicateur vers la gauche, nous pouvons décaler le résultat vers la droite. Ceci revient au même, mais vous n'avez plus que des additions de 8 bits, toujours dans le poids fort du résultat
Voici comment cela fonctionne avec notre exemple, pour ne pas faire trop long :

- On multiplie 1100 par 1. On place le résultat à fond à gauche du résultat. Dans résultat on a : 11000000
- On décale le résultat vers la droite : résultat = 01100000
- On multiplie 1100 par 0 . On ajoute au résultat : résultat = 01100000
- On décale le résultat vers la droite : résultat = 00110000
- On multiplie 1100 par 1. On ajoute au résultat à gauche : résultat :11000000 + 00110000 = 11110000
- On décale le résultat vers la droite : résultat = 01111000
- On multiplie 1100 par 1. On ajoute au résultat à gauche : résultat : 11000000 + 01111000 = 100111000 (9 bits). Le bit8 (en vert) est dans le carry.
- On décale le résultat vers la droite : résultat = 10011100 = D'156'

Notez qu'on ajoute toujours au quartet de poids fort (centrage à gauche). Dans le cas d'une multiplication de 8 bits par 8 bits, on ajoutera donc à l'octet de poids fort. Par moment, nous aurons débordement de l'addition sur 9 bits. Souvenez-vous que le report se trouve dans le carry. Or, dans un décalage, le carry est amené dans le résultat, donc on récupère automatiquement notre 9ème bit, qui deviendra le 8ème après décalage vers la droite.

Cette procédure est très simple à mettre en programme : la preuve, voici le pseudo-code pour une multiplication 8 bits par 8 bits.

```
Effacer le résultat
Pour chacun des 8 bits du multiplicateur
        Si bit de droite du multiplicateur = 1
                Ajouter multiplicande au poids fort du résultat
        Décaler 16 bits du résultat vers la droite
        Décaler multiplicateur vers la droite
Bit suivant
```

Si nous plaçons de plus le décalage du multiplicateur en-tête, avant le test de bit, nous récupérons le bit à tester dans le carry. Le programme devient :

```
Effacer le résultat
Pour chacun des 8 bits du multiplicateur
        Décaler multiplicateur vers la droite
        Si carry = 1
                Ajouter multiplicande au poids fort du résultat
        Décaler 16 bits du résultat vers la droite
```

Bit suivant

Remarquez que ceci est très simple. Nous allons mettre ce pseudo-code sous forme de programme. Nous utiliserons la variable multi comme multiplicateur, multan comme multiplicande, resulH comme résultat poids fort et resulL comme poids faible. multemp est le multiplicateur temporaire qui sera modifié. cmpt est le compteur de boucles Voici donc le programme :

```
    clrf     resulH            ; effacer résultat poids fort
    clrf     resulL            ; idem poids faible
    movlw    0x08              ; pour 8 bits
    movwf    cmpt              ; initialiser compteur de boucles
    movf     multi , w         ; charger multiplicateur
    movwf    multemp           ; sauver dans multemp
    movf     multan , w        ; multiplicande dans w
loop
    rrf      multemp , f       ; décaler multiplicateur vers  la droite
    btfsc    STATUS , C        ; tester si bit sorti = 1
    addwf    resulH , f        ; oui, ajouter au résultat poids fort
    rrf      resulH , f        ; décaler résultat poids fort
    rrf      resulL , f        ; décaler résultat poids faible
    decfsz   cmpt , f          ; décrémenter compteur de boucles
    goto     loop              ; pas fini, bit suivant
```

Vous pouvez créer un petit projet et tester ce programme dans le simulateur. Vous verrez qu'il fonctionne parfaitement. Vous pouvez en faire une sous-routine ou une macro à ajouter dans votre propre fichier include.

Les 2 lignes en gris effectuent un décalage du résultat sur 16 bits. Le bit perdu dans resulH est récupéré dans le carry et remis comme bit7 dans resulL. Astuce de programmation : si l'addition précédente à eu lieu (ligne bleue), le 9ème bit résultant de l'addition est dans le carry, et se retrouve donc comme b7 dans resulH. Par contre, s'il n'y a pas eu l'addition, le carry est forcément à 0 du fait du test (en jaune). Vous voyez maintenant l'intérêt d'avoir testé le bit faible du multiplicateur en se servant du carry.

Rassurez-vous, c'est avec l'expérience que vous arriverez à ce genre de résultat. Ce petit programme est disponible sous la dénominations : « multi.asm »

18.4 Multiplication par une constante

Dans le cas précédent, nous avons effectuer une multiplication entre 2 variables qui peuvent prendre n'importe quelle valeur. Dans le cas où vous utilisez une constante, c'est à dire que vous connaissez le multiplicateur au moment de la conception de votre programme, vous devez essayer, pour des raisons d'optimisation de raisonner en multiplications par 2.

En effet, pour effectuer une multiplication par 2 il suffit de décaler la valeur vers la gauche. C'est très rapide à mettre en œuvre. Ne pas oublier cependant de mette C à 0 avant le décalage, pour ne pas entrer un bit non désiré dans le mot décalé.

Supposons donc que vous deviez, dans votre programme, effectuer des multiplications par 10 (décimal). Et bien nous allons essayer de ramener ceci en multiplications par 2. C'est tout simple :

Pour multiplier par 10, il faut :

- multiplier par 2 (donc décaler l'opérande vers la gauche)
- multiplier encore par 2 (= multiplier par 4, donc décaler encore une fois).
- Ajouter l'opérande originale (donc opérande + opérande*4 = opérande *5)
- Multiplier le résultat par 2 (donc décalage vers la gauche).

Voici donc une multiplication par 10 très rapide. On a fait « *2, *2, +1, *2). Donc 3 décalages et une simple addition.

En général, on peut s'arranger pour obtenir les résultats des multiplications par des constantes en utilisant cette méthode.

Ceci vous fera gagner un précieux temps programme.

Voici un petit programme qui multiplie un nombre de 4 bits contenu dans mem1 par 10. Résultat dans resul. Vous pouvez adapter pour un nombre de 8 bits très facilement :

```
    movf     mem1 , w          ; charger opérande sur 4 bits
    movwf    resul             ; sauver dans résultat
    bcf      STATUS , C        ; effacer carry
    rlf      resul , f         ; multiplier par 2
    rlf      resul , f         ; multiplier par 4
    addwf    resul , f         ; ajouter mem1, donc multiplier par 5
    rlf      resul , f         ; multiplier par 10
```

18.5 Adressage indirect pointant sur 2 zones différentes

Voici encore une astuce. Imaginez que vous deviez copier 15 variables d'un emplacement mémoire vers un autre (ou encore comparer 2 zones mémoires différentes etc.). En fait vous allez rapidement vous heurter à un problème. Vous ne disposez que d'un seul pointeur FSR pour pointer sur vos variables.

Réalisons donc ce programme : mem1 est l'adresse de départ de la première zone, mem2 l'adresse de départ de la seconde zone

```
    movlw    mem               ; on pointe sur la première zone
    movwf    FSR               ; on initialise le pointeur sur la zone source
    movlw    15                ; 15 variables à transférer
    movwf    cmpt              ; sauver dans compteur de boucles
loop
    movf     INDF , w          ; charger source
    movwf    tampon            ; on doit sauver dans un emplacement tampon
    movlw    mem2-mem1         ; écart d'adresse entre les 2 emplacements
    addwf    FSR , f           ; pointer sur destination
    movf     tampon , w        ; recharger valeur source
    movwf    INDF              ; sauver dans destination
    movlw    (mem1-mem2)+1     ; Ecart entre adresse de destination et adresse
                               ; de la source suivante, d'où le +1
    addwf    FSR , f           ;  ajouter au pointeur
    decfsz   cmpt , f          ; décrémenter compteur de boucles
    goto     loop              ; pas dernier emplacement, suivant
```

Ce programme est valable lorsque vous n'avez pas le choix des zones de départ et de destination. Maintenant, qui vous empêche de placer vos variables dans des endroits qui vous arrangent ? Plaçons mem1 et mem2 dans la zone de variables :

```
CBLOCK   0x10  ; on choisit une zone mémoire dans la RAM
mem1 : 15            ; 15 emplacements pour la source
ENDC

CBLOCK     0x30        ; on choisit une autre zone, mais pas au hasard
mem2 : 15              ; 15 emplacements destination.
ENDC
```

Ou encore, mais on ne voit pas bien l'organisation des zones utilisées :

```
mem1 EQU 0x10
mem2 EQU 0x30
```

Maintenant, la source va de B'00010000' à B'00011111 ', et la destination de B'00110000' à B'00111111'

Il n'y a donc que le bit 5 de FSR qui change entre source et destination. Pour modifier des bits, pas besoin de l'accumulateur, donc pas besoin de sauver et recharger la valeur à transférer.

Nous pouvons donc modifier notre programme de la manière suivante :

```
   movlw    mem1              ; on pointe sur la première zone
   movwf    FSR               ; on initialise le pointeur sur la zone source
   movlw    15                ; 15 variables à transférer
   movwf    cmpt              ; sauver dans compteur de boucles
loop
   movf     INDF , w          ; charger source
   bsf      FSR , 5           ; on pointe sur destination
   movwf    INDF              ; sauver dans destination
   bcf      FSR , 5           ; on pointe sur source
   incf     FSR , f           ; pointer sur suivant
   decfsz   cmpt , f          ; décrémenter compteur de boucles
   goto     loop              ; pas dernier emplacement, suivant
```

Avec les PIC® qui utilisent plusieurs banques, comme le 16F876, vous pouvez également utiliser la même adresse dans 2 banques différentes, et passer d'une à l'autre en modifiant le bit IRP du registre STATUS.

.

18.6 Les tableaux en mémoire programme

Supposons que nous avons besoin d'un tableau de taille importante. Par exemple un tableau de 200 éléments. Où placer ce tableau ? Dans la RAM il n'y a pas assez de place, dans l'eeprom non plus. Ne reste donc plus que la mémoire de programme.

Le problème, c'est que le 16F84, contrairement au 16F876, par exemple, ne dispose d'aucune méthode pour aller lire les données dans la mémoire programme. La seule méthode pour y avoir accès est d'utiliser des instructions.

Supposons un cas simple : nous voulons créer un tableau contenant le carré des nombres de 0 à 15. Nous pourrions utiliser un petit sous-programme de la forme suivante :

- On teste si le nombre passé en argument = 0. Si oui, on retourne 0
- On teste si le nombre passé en argument = 1. Si oui, on retourne 1
- On teste si le nombre passé en argument = 2. Si oui, on retourne 4
- Et ainsi de suite…

Vous voyez tout de suite qu'il faudra plusieurs instructions par valeur du tableau. En cas d'un tableau de 200 éléments, on n'aura pas assez de mémoire programme pour tout écrire. Nous allons donc utiliser une autre astuce.

Le cœur de l'astuce est d'utiliser l'instruction retlw qui permet de retourner une valeur passée en argument. Ecrivons donc notre tableau sous forme de retlw : cela donne :

```
    retlw    0          ; carré de 0 = 0
    retlw    1          ; carré de 1 = 1
    retlw    4          ; carré de 2 = 4
    retlw    9          ; carré de 3 = 9
    retlw    16         ; carré de 4 = 16
    retlw    25         ; carré de 5 = 25
    retlw    36         ; carré de 6 = 36
    retlw    49         ; carré de 7 = 49
    retlw    64         ; carré de 8 = 64
    retlw    81         ; carré de 9 = 81
    retlw    100        ; carré de 10 = 100
    retlw    121        ; carré de 11 = 121
    retlw    144        ; carré de 12 = 144
    retlw    169        ; carré de 13 = 169
    retlw    196        ; carré de 14 = 196
    retlw    225        ; carré de 15 = 225
```

Il ne nous reste plus qu'à trouver une astuce pour nous brancher sur la bonne ligne de notre tableau/programme. Si nous nous souvenons que nous pouvons effectuer des opérations sur le PCL, nous pouvons utiliser cette propriété.

Supposons donc que le nombre à élever au carré soit contenu dans le registre w. Si, une fois sur la première ligne du tableau, nous ajoutons w à PCL, nous sauterons directement sur la bonne ligne. Complétons donc notre sous-routine. Il nous suffit donc d'ajouter avant notre tableau la ligne :

```
carre
    addwf    PCL , f     ; ajouter w à PCL
```

Donc, si on charge 4 dans w et qu'on effectue un appel « call carre ». Le programme se branche sur cette ligne, le PCL est incrémenté de 4 et le programme exécute donc alors la ligne « retlw 16 ».

Rappelez-vous, en effet, que le PC pointe toujours sur l'instruction suivante, donc au moment de l'exécution de « addwf PCL , f », celui-ci pointait sur la ligne « retlw 0 ».

A ce stade, vous devez vous rappeler que l'adresse pour une opération sur le registre PCL est complétée par le contenu de PCLATH. Vous devez donc initialiser correctement celui-ci avec le numéro de la « page de 8 bits » de l'adresse de votre tableau.

Par exemple, si votre tableau est à l'adresse 0x200, vous devrez mettre 0x02 dans PCLATH.

De plus, l'opération sur PCL ne modifiera pas automatiquement PCLATH, donc votre tableau ne devra pas dépasser 256 éléments, et, de plus, ne devra pas déborder sur la « page de 8 bits » suivante. Une page étant la plage adressée par les 256 valeurs possibles de PCL, sans toucher à PCLATH.

Comment éviter ceci ? Tout simplement en imposant au tableau une adresse de départ telle que le tableau tienne tout entier dans la même « page » de 256 emplacements.

Ceci permet alors d'utiliser un tableau de 256 éléments. Voici donc les emplacements disponibles pour un tableau d'une telle taille :

Adresse B'00000 00000000', soit adresse 0x00. Pas utilisable, car adresse de reset
Adresse B'00001 00000000', soit adresse 0x100. Premier emplacement utilisable
Adresse B'00010 00000000', soit adresse 0x200. Second emplacement utilisable
Adresse B'00011 00000000', soit adresse 0x300. Dernier emplacement utilisable.

Il va de soi que si vous avez besoin d'un plus petit tableau, vous pouvez le commencer ailleurs, en veillant bien à ne pas occasionner de débordement. Vous pouvez donc ajouter la ligne : org 0x300 avant l'étiquette de début de sous-routine.

Notez que ce que j'explique ici est valable pour les tableaux dont la fin est dans la même page que le début. Or, comme la ligne « addwf » est incluse dans la page, il faut soustraire cette ligne des 256 emplacements disponibles. Nous obtenons donc :

```
;*****************************************************************
;                       TABLEAU DES CARRES                       *
;*****************************************************************
;-----------------------------------------------------------------
; N'oubliez pas que la directive "END" doit se
; trouver après la dernière ligne de votre programme. Ne laissez donc
; pas cette directive dans le programme principal.
;-----------------------------------------------------------------
   org 0x300       ; adresse du tableau
carre
   addwf    PCL , f      ; ajouter w à PCL
   retlw    .0           ; carré de 0 = 0
   retlw    .1           ; carré de 1 = 1
   retlw    .4           ; carré de 2 = 4
   retlw    .9           ; carré de 3 = 9
   retlw    .16          ; carré de 4 = 16
   retlw    .25          ; carré de 5 = 25
   retlw    .36          ; carré de 6 = 36
   retlw    .49          ; carré de 7 = 49
   retlw    .64          ; carré de 8 = 64
   retlw    .81          ; carré de 9 = 81
   retlw    .100         ; carré de 10 = 100
   retlw    .121         ; carré de 11 = 121
   retlw    .144         ; carré de 12 = 144
```

```
    retlw    .169         ; carré de 13 = 169
    retlw    .196         ; carré de 14 = 196
    retlw    .225         ; carré de 15 = 225
    END                   ; directive fin de programme
```

Il ne nous reste plus qu'à construire un petit programme principal qui fasse appel à cette sous-routine.

```
;*****************************************************************
;                       DEMARRAGE SUR RESET                      *
;*****************************************************************
    org 0x000             ; Adresse de départ après reset

;*****************************************************************
;                       PROGRAMME PRINCIPAL                      *
;*****************************************************************
start
    clrf     nombre             ; effacer nombre
loop
    movf     nombre , w         ; charger nombre
    call     carre              ; prendre le carré du nombre
    incf     nombre , f         ; incrémenter nombre
    btfss    nombre , 4         ; tester si nombre >15
    goto     loop               ; non, nombre suivant
    goto     start              ; oui, on recommence à 0
```

Compilez votre programme puis passez-le au simulateur. Vous constaterez…. Que cela ne fonctionne pas. Pourquoi ?

Et bien, en réalité, toute opération dont le PC est la destination fait intervenir PCLATH. Or nous avons laissé PCLATH à 0. L'adresse de calcul n'est donc pas la bonne. Avant le saut sous aurions du ajouter :

```
    movlw    03
    movwf    PCLATH
```

Afin de permettre à PCLATH de pointer sur la page 0x300 lors du calcul d'incrémentation de PCL.

Vous pouvez avantageusement remplacer la ligne org 0x300 par :

```
repere
     ORG (repere+31)& 0x3E0    ; adresse du tableau
```

Le but de cette manipulation est d'obtenir automatiquement la première adresse disponible pour laquelle il n'y aura pas de débordement de notre PCL. Pour un tableau de 256 éléments, utilisez « ORG (repere+255) & 0x300. Faites des essais à la main sur un bout de papier pour vous en convaincre. Par exemple, on calcule la fin du tableau (repere+255). Org pointe alors sur la bonne page. On élimine ensuite les bits correspondants (255 = 8 bits) de la taille mémoire totale du programme (0x3FF). Reste donc & 0x300.

Le fichier obtenu est disponible sous la dénomination « tableau.asm »

Il est important de comprendre que c'est l'ensemble des instructions « retlw » qui doivent se trouver dans la même page. Dans le cas précédent (petit tableau), vous notez qu'il y a une instruction qui se trouve déjà dans cette page, ce qui fait 1 emplacement de moins disponible.

Nous aurions pu modifier le tableau en conséquence, de façon que le premier élément du tableau pointe à l'offset « 0 » de la « page de 8 bits ».

```
   org 0x2FF       ; adresse du tableau (première ligne « retlw » = 0x300)
carre
   addwf    PCL , f      ; ajouter w à PCL
   retlw    8            ; premier élément du tableau : adresse 0x300
   retlw    4            ; second élément du tableau : adresse 0x301
   …
```

Toujours pour le cas d'un tableau de 256 éléments, nous voyons que nous devons impérativement commencer le tableau à un offset de 0 (PCL = 0). Inutile donc de faire une addition de « 0 », il suffit de placer directement « w » dans PCL, quelque soit l'adresse de cette imputation, du moment que PCLATH soit correctement configuré. Nous obtenons donc :

```
carre
   movwf    PCL , f      ; ajouter w à PCL (adresse = suite du programme
principal)
   org 0x300             ; adresse effective du tableau(256 emplacements
disponibles)
   retlw    8            ; premier élément du tableau : adresse 0x300
   retlw    4            ; second élément du tableau : adresse 0x301
   …
```

18.7 Les variables locales

Un des reproches qu'on entend souvent au sujet des PIC® est le nombre restreint des variables disponibles. Il ne faudrait cependant pas croire que, sous prétexte que nous programmions en assembleur, nous ne disposions pas de variables locales.

En fait, l'utilisation de ce type de variable est très simple et réduit considérablement le nombre de variables nécessaires dans un programme. Voici ma méthode personnelle d'utilisation de ce type de variables.

Pour rappel, ce sont des variables qui ne seront utilisées qu'à l'intérieur d'une sous-routine, et qui ne servent plus une fois la sous-routine terminée. Partons donc d'un programme imaginaire qui utilise deux sous-routines.

- La première sous-routine s'appelle « tempo », elle utilise 2 compteurs pour effectuer un comptage de temps.

- La seconde est la sous-routine « fonction ». Elle reçoit un paramètre dans W, doit sauver le résultat dans une variable, et utilise 2 autres variables pour ses calculs intermédiaires.

- La condition pour l'utilisation de variables locales identiques, est qu'une des routines n'appelle pas l'autre.

18.7.1 Détermination des variables locales

Si nous examinons les routines, nous constatons que les 2 variables de la sous-routine « tempo » ne sont utiles qu'à l'intérieur de celle-ci. Nous pouvons donc utiliser des variables locales.

La variable résultat de la sous-routine « fonction » devra être conservé après exécution de cette sous-routine, ce n'est donc pas une variable locale. Nous dirons que c'est une variable globale. Par contre, les 2 variables utilisées pour les calculs intermédiaires ne seront plus d'aucune utilité. Ce sont des variables locales.

18.7.2 Construction sans variables locales

Que serait notre zone de variables sans l'utilisation des variables locales ? Et bien créons notre zone de variables :

```
CBLOCK 0X0C
Cmpt1 : 1             ; compteur 1 pour routine tempo
Cmpt2 : 1             ; compteur 2 pour routine tempo
Resultat : 1          ; résultat pour routine fonction
Interm1 : 1           ; résultat intermédiaire1 pour fonction
Interm2 : 1           ; résultat intermédiaire2 pour fonction
ENDC
```

Nous voyons que nous aurons besoin de 5 variables. Utilisons maintenant les variables locales.

18.7.3 Construction avec variables locales

Premièrement, nous allons réserver les emplacements mémoires nécessaires pour les variables locales. Le plus grand nombre de variables locales utilisées par une fonction, est de 2. Nous aurons donc 2 variables locales. Déclarons-les.
Ensuite, il nous reste une variable globale à ajouter. Créons notre zone data :

```
CBLOCK 0X0C
Local1 : 1                ; variable locale 1
Local2 : 1                ; variable locale 2
Resultat : 1              ; résultat pour fonction
ENDC
```

Il ne nous faudra donc plus que 3 variables au lieu des 5 initiales. Pour mieux nous y retrouver, nous pouvons attribuer les mêmes noms que précédemment pour nos variables locales, en ajoutant simplement des DEFINE ou des EQU.

```
#DEFINE     cmpt1 Local1      ; compteur1 = variable locale1
#DEFINE     cmpt2 Local2      ; compteur2 = variable locale2
#DEFINE     interm1 Local1    ; résultat intermédiaire1 = variable locale1
#DEFINE     interm2 Local2    ; résultat intermédiaire2 = variable locale2
```

Et voilà, pas besoin du C pour utiliser les variables locales très simplement.

18.8 Division par une constante

Voici une astuce qui m'a été envoyée par un internaute, Richard L. :

Pour diviser un nombre par une constante, il suffit de multiplier ce nombre par une autre constante qui vaut : 256 divisé par la constante initiale.

On obtient un résultat sur 16 bits, le résultat de la division se trouve dans l'octet de poids fort.

Exemple :

Pour diviser un nombre par 10, il suffit de le multiplier par (256/10). En hexadécimal, 256/10 donne 0x19 ou 0x1A (il faut bien arrondir)

Imaginons que notre variable contienne la valeur 120 décimal, soit 0x78.

Si on multiplie 0x78 par 0x1A, ça donne 0x0C30. Le poids fort est donc 0x0C, ce qui donne 12 en décimal.

Si on avait choisi 0x19, on aurait trouvé 11 comme réponse.

18.9 Conclusion

Vous voici en possession de quelques astuces de programmation. Le but était de vous montrer les méthodes de raisonnement afin de vous permettre de développer vos propres trucs et astuces.

19. La norme ISO 7816

Le but de cette annexe est de vous montrer comment réaliser une application pratique en dépassant les limites apparentes du PIC®. J'ai choisi de créer l'ossature d'un programme de gestion d'une carte ISO7816, car c'est un sujet d'actualité (ne me demandez pas pourquoi, je ferai semblant de ne pas le savoir).

De plus, c'est un excellent prétexte pour montrer qu'il est possible d'utiliser un 16F84, pourtant dépourvu de la gestion série, pour communiquer de cette manière avec le monde extérieur.

Cette annexe se limitera à la norme en général, sans entrer dans une application spécifique de cette norme. Le but n'est pas en effet de construire votre propre application (par exemple une serrure codée), mais plutôt de vous montrer comment la concevoir vous-même.

Inutile donc de m'envoyer du courrier pour savoir comment cette norme est utilisée en pratique pour telle ou telle application non publique, je n'y répondrai pas.

19.1 Spécificités utiles de la norme ISO 7816

Voyons tout d'abord quelques spécificités de la norme 7816 qui va nous servir à créer notre application.

Vous trouverez ces cartes partout dans le commerce de composants électroniques, sous la dénomination « carte pour serrure codée ». Elles permettent de réaliser des tas d'applications, et les lecteurs sont disponibles partout. Vous aurez besoin pour communiquer avec cette carte d'une interface dont vous trouverez le schéma sur Internet sous la dénomination « phœnix », et d'un logiciel de communication ou du logiciel fourni avec l'interface.

- La carte ne dispose pas de sa propre horloge, elle est fournie par l'interface, ou maître, et est généralement sous la barre des 4MHz.

- Le temps séparant chaque bit est défini par la norme comme étant de 1 bit émis toutes les 372 impulsions de l'horloge du maître. Ceci donne un débit de l'ordre de 9600 bauds avec un quartz de 3,57 MHz (3570000/372), ou encore 10752 bauds avec notre quartz de 4Mhz

- La transmission est du type asynchrone, avec 1 start-bit, 8 bits de data, 1 bit de parité paire, et 2 stop-bits.

- La communication est du type half-duplex, c'est à dire que les entrées et les sorties s'effectuent par alternance et en se servant de la même ligne physique.

19.1.1 Les commandes ISO 7816

Nous allons dans cet exercice utiliser des commandes « bidon » de la norme ISO7816. Il faut savoir que notre carte devra répondre à des commandes organisées suivant le protocole standard de cette norme.

La carte ne prend jamais l'initiative de l'échange d'informations. Elle ne fait que répondre à des commandes de la forme :

CLASSE INSTRUCTION PARAMETRE1 PARAMETRE2 LONGUEUR

Ou encore, sous forme abrégée :

C LASS INS P1 P2 LEN

Voici les significations de chacun des octets de la commande reçue :

- **CLASS** : détermine le groupe d'instructions concernées. Si le nombre d'instructions utilisé n'est pas élevé, rien n'interdit de n'utiliser qu'une seule classe. C'est ce que nous ferons dans cet exercice simplifié.

- **INS** : c'est l'instruction proprement dite, ou encore la commande. C'est cet octet qui détermine l'opération à effectuer par la carte.

- **P1** et **P2** : sont 2 paramètres que la carte reçoit avec la commande. Leur signification dépend de la commande envoyée. Ils peuvent être inutilisés mais doivent tout de même être présents.

- **LEN** : peut représenter 2 choses. Soit c'est le nombre d'octets qui suit la commande, dans ce cas la carte s'attendra à recevoir LEN octets supplémentaires avant de traiter la commande dans sa totalité. Soit ce sera la longueur de la chaîne de réponse que le maître s'attend à recevoir en provenance de la carte. C'est l'instruction qui permet d'établir le rôle de LEN.

Notez qu'il existe des modes étendus pour la norme ISO7816, qui permet à la fois d'envoyer et de recevoir plusieurs octets en une seule opération. Ce mode est défini comme mode T=1. Dans notre cas, nous supposons que l'information ne circule que dans un sens pour une instruction donnée (Mode T=0).

19.1.2 Le protocole d'échange d'informations

Voici comment se déroule un échange standard d'informations entre le maître et la carte ISO 7816.

- A la mise en service, le maître génère un RESET sur la pin *MCLR*, la carte répond avec un certain nombres d'octets. Cette réponse s'appelle ATR, pour Answer To Reset (réponse au reset).

- Le maître envoie la commande Class INS P1 P2 LEN
- La carte renvoie l'instruction comme accusé de réception INS
- Le maître envoie éventuellement les données complémentaires
- La carte envoie éventuellement la réponse
- La carte envoie le status et repasse en mode d'attente de commande

Donc, si nous supposons par exemple l'instruction imaginaire 0x14 de la classe 0xD5 qui nécessite une réponse de la part de la carte (instruction carte vers maître). Le maître précise P1 =0x01 et P2 = 0x02. Nous aurons donc (en imaginant les octets de data) :

- Le maître envoie : D5 14 01 02 10
- Nous en déduisons que le maître attend 16 octets en réponse (0x10 = D'16')
- La carte répond : 14 (accusé de réception)
- La carte envoit : 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F (par exemple), soit 16 octets de réponse, exclus l'accusé de réception et le status.
- La carte envoit : 90 00 : status employé en général pour indiquer que l'opération s'est bien passée

Si, par contre, dans les mêmes conditions, nous imaginons l'instruction 0x15 accompagnée d'octets à destination de la carte (instruction maître vers carte), nous aurons :

- Le maître envoit : D5 15 01 02 10
- La carte répond : 15 (accusé de réception)
- Le maître envoit : 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F (par exemple)
- La carte répond : 90 00

Vous voyez que seul le sens de circulation des 16 octets de data change de direction. C'est donc toujours le maître qui envoie la trame de départ, et la carte qui envoie l'accusé de réception et le status.

19.2 Les liaisons série asynchrones

Les échanges d'information entre le maître et la carte se font sur une liaison série asynchrone. Pour comprendre la suite de ce chapitre, il faut donc comprendre ce que cela signifie.

Le mode série est défini pour signaler que tous les bits d'un octet vont être envoyés en série les uns derrière les autres. Ceci par opposition au mode parallèle pour lequel tous les bits seraient envoyés en même temps, chacun, de ce fait, sur un fil séparé.

Asynchrone est l'opposé de synchrone, c'est-à-dire qu'il s'agit d'une liaison qui ne fournit pas une horloge destinée à indiquer le début et la fin de chaque bit envoyé.

Nous aurons donc besoin d'un mécanisme destiné à repérer la position de chaque bit. Notez qu'il s'agit ici d'un mode asynchrone particulier, étant donné que la carte utilise la même horloge que le maître. La vitesse ne sera donc exceptionnellement pas donnée en bauds, mais en nombre d'impulsions d'horloge. Un bit toutes les 372 impulsions d'horloge.

Pour recevoir correctement les bits envoyés, il faut convenir d'un protocole de communication. Ce dernier doit comprendre les informations suivantes :

- La vitesse de transmission en bauds
- Le format, c'est à dire le nombre de start-bits, de stop-bits, de bits de données et le type de parité

Nous allons maintenant expliquer ces concepts et indiquer quelles sont les valeurs employées dans la norme ISO 7816.

19.2.1 Le start-bit

Au repos, la ligne se trouve à l'état haut. L'émetteur fait alors passer la ligne à l'état bas : c'est le start-bit. C'est ce changement de niveau qui va permettre de détecter le début de la réception des bits en série.

Les valeurs admissibles sont 1 ou 2 start-bit(s). La norme ISO 7816 nécessite un start-bit.

19.2.2 Les bits de données

Après avoir reçu le start-bit, on trouve les bits de données, en commençant par le bit 0. Les normes usuelles utilisent 7 ou 8 bits de data. Pour la norme ISO 7816, nous aurons 8 bits de données, ce qui nous donne des valeurs admissibles de 0 à 0xFF pour chaque octet reçu.

19.2.3 Le bit de parité

Le bit de parité est une vérification du bon déroulement du transfert. Lors de l'émission, on comptabilise chaque bit de donnée qui vaut 1. A la fin du comptage, on ajoute un bit à 1 ou à 0 de façon à obtenir un nombre de bits total impair ou pair.

On dira qu'on utilise une parité paire si le nombre de bits à 1 dans les bits de données y compris le bit de parité est un nombre pair.

De même, une parité impaire donnera un nombre impair de bits à 1.

Notez que le bit de parité n'est pas indispensable dans une liaison série asynchrone. Nous avons donc 3 possibilités, à savoir pas de parité, parité paire, ou parité impaire.

Dans le cas de la norme ISO 7816, nous devrons utiliser une parité paire.

19.2.4 Le stop-bit

Après la réception des bits précédents, il est impératif de remettre la ligne à l'état haut pour pouvoir détecter le start-bit de l'octet suivant. C'est le rôle du stop-bit. Les valeurs admissibles sont de 1 ou 2 stop-bits.

Dans la norme ISO 7816, nous utiliserons 2 stop-bits, c'est à dire tout simplement un stop-bit d'une durée équivalente à la durée de 2 bits.

19.2.5 Vitesse et débit

La durée de chaque bit est une constante et dépend de la vitesse de transmission. Par exemple, pour une vitesse de 9600 bauds, c'est à dire 9600 bits par seconde, chaque bit durera 1s/9600 = 104,17 µS.

Le temps nécessaire pour recevoir un octet entier est la somme du temps nécessaire pour recevoir chacun des bits de cet octet. Dans le cas de la norme ISO 7816, nous utiliserons 1 start-bit + 8 bits de données + 1 bit de parité + 2 stop-bits = 12 bits. Le temps total pour recevoir un octet est donc de 1250 µS.

Le débit maximum en octets par seconde est donc égal à :

Débit maximum = 1 / (nbre de bits * durée d'un bit), soit pour la norme ISO 7816 :

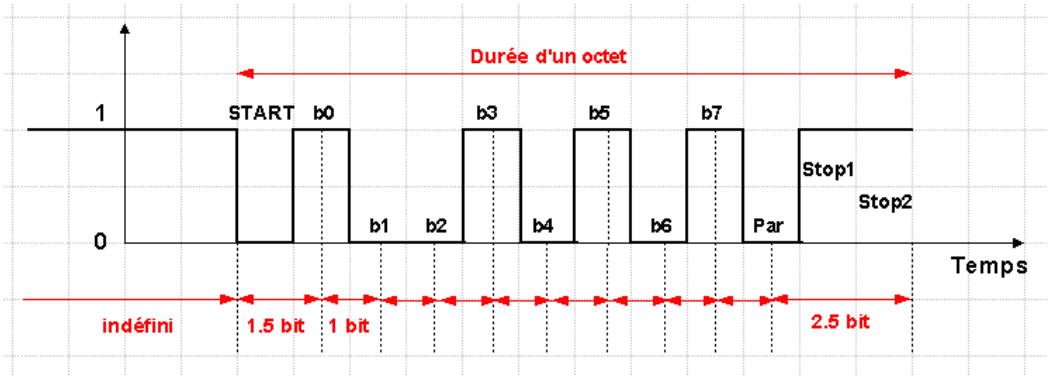
$1 / 1250 * 10^{-6} =$ 800 octets par seconde.

CECI POUR UNE VITESSE DE TRANSFERT DE 9600 BAUDS. Notez que cette valeur n'est pas imposée par la norme, il est donc possible d'augmenter ce débit.

Nous parlons ici de débit maximum car ceci est le cas dans lequel un octet commence dès la fin du précédent, ce qui n'est pas obligatoire pour une liaison asynchrone.

19.3 Acquisition des bits

Nous allons d'abord représenter ce que nous venons de voir sous forme d'un graphique. Sur l'axe vertical nous avons les niveaux, sur l'axe horizontal, le temps. Chaque carré représente la durée d'un bit.



Examinons ce graphique. La ligne est à l'état haut depuis un temps indéterminé. Survient le start-bit, qui commence 2 carrés après notre axe vertical. Nous voyons que le meilleur moment pour lire le bit 0 est de le mesurer au milieu de sa largeur afin d'obtenir le moins de risque d'erreur possible.

Souvenez-vous que pour la norme ISO 7816, la largeur d'un bit vaut 372 impulsions d'horloge. Comme notre PIC® divise déjà la fréquence d'horloge par 4, cela nous fera une largeur de bit équivalent à 372/4 = 93 impulsions d'horloge.

Le bit 0 sera donc lu 93 + 46 cycles d'horloge après le début du start-bit. Ensuite nous pourrons lire tous les bits à un intervalle de temps équivalent à 93 instructions.

Une fois le bit de parité lu, nous avons 2 possibilités :

- Si nous souhaitons faire suivre cette lecture par la lecture d'un autre octet, nous devons nous positionner quelque part dans les 2 stop-bits afin d'être prêt à détecter le start-bit suivant. Nous devons donc attendre entre 0.5 et 2.5 bits.

- Si nous souhaitons envoyer un octet après cette lecture, nous ne pourrons pas le faire avant la fin du second stop-bit, soit au minimum après 2.5 cycles.

Par curiosité, quel est donc l'octet que nous avons reçu dans cet exemple ? Et bien tout simplement :

B0 = 1
B1 = 0
B2 = 0
B3 = 1
B4 = 0
B5 = 1
B6 = 0
B7 = 1
Parité = 0

L'octet reçu est donc B'10101001' , soit 0XA9. Profitons-en pour vérifier la parité. Nombre de bits à 1 reçus = 4, donc parité paire, c'est donc conforme à la norme ISO 7816.

19.4 Caractéristique des cartes « standard »

Sur les cartes du commerce, de type « carte pour serrure codée », les connexions utilisées sont généralement les suivantes :

Nom pin	Pin	Signal	Rôle du signal	Type
MCLR	4	**RST**	Commande de reset de la carte	Entrée
VSS	5	**GND**	Masse	Alimentation
RB4	10	**SDA**	data pour eeprom externe (24C16)	Bi-directionnel
RB5	11	**SCL**	horloge pour eeprom externe	Sortie
RB7	13	**DATA**	données mode série half-duplex	Bi-directionnel
VDD	14	**+5V**	Alimentation +5V	Alimentation
CLKIN	16	**CLK**	Entrée de l'horloge externe	Oscillateur

19.5 Création et initialisation du projet

Suivant note méthode habituelle, copiez / collez votre fichier « m16f84.asm » et renommez la copie « iso7816.asm ». Créez votre projet dans MPLAB®.

Ensuite, créez votre en-tête de programme :

```
;*****************************************************************
;   Ce fichier est la base de départ pour la gestion d'une carte       *
;   répondant à la norme ISO7816.                                      *
;                                                                      *
;*****************************************************************
;                                                                      *
;    NOM:        ISO7816                                               *
;    Date:       10/03/2001                                            *
;    Version:    1.1                                                   *
;    Circuit:    Carte pour serrure codée                              *
;    Auteur:     Bigonoff                                              *
;                                                                      *
;*****************************************************************
;                                                                      *
;    Fichier requis: P16F84.inc                                        *
;                                                                      *
;*****************************************************************
;                                                                      *
;    - MCLR     Commande de reset de la carte       Entrée             *
;    - RB4      SDA - data pour eeprom externe      Bi-directionnel    *
;    - RB5      SCL - horloge pour eeprom externe   Sortie             *
;    - RB7      DATA - données mode série           Bi-directionnel    *
;                                                                      *
;*****************************************************************
```

Définissons la configuration : Nous n'allons pas utiliser le watch-dog pour cette application, et nous utiliserons une horloge externe. Nous aurons donc

```
   LIST      p=16F84           ; Définition de processeur
   #include <p16F84.inc>       ; Définitions des constantes
   radix dec                   ; travailler en décimal par défaut

   __CONFIG   _CP_OFF & _WDT_OFF & _PWRTE_ON & _HS_OSC

                  ; Code protection OFF
                  ; Timer reset sur power on en service
                  ; Watch-dog hors service
                  ; Oscillateur quartz grande vitesse
```

Calculons maintenant la valeur à encoder dans notre registre OPTION :

Nous devons utiliser, pour des raisons de compatibilité électronique, la résistance de rappel au +5V de la pin RB7. Nous aurons donc b7 du registre OPTION = 0. Ensuite, nous allons travailler avec le timer0.

Comme l'écart entre 2 bits est de 93 instructions, nous travaillerons sans prédiviseur, ce qui nous impose de rediriger le prédiviseur vers le watchdog.

Définissons donc la valeur à placer dans OPTION :

```
OPTIONVAL    EQU    0x08  ; Valeur registre option
                          ; Résistance pull-up ON
                          ; Préscaler timer à 1
```

19.6 La base de temps

Nous voyons déjà que nous allons avoir besoin de 2 temporisations : une de 93 instructions et une de 93+46 instructions, correspondant à 1 et 1,5 bits. Nous allons donc construire une sous-routine.

```
;*****************************************************************
;                       TEMPORISATION                            *
;*****************************************************************
;---------------------------------------------------------------------
; temp_1bd: initialise tmr0 pour que la temporisation soit égale à
;           l'équivalent d'un bit et demi, soit 46+93 incrémentations
;           de tmr0 – le temps nécessaire pour arriver dans la routine
; temp_1b : Attend que l'écart entre la précédente tempo et la tempo
;           actuelle soit de 1 bit, soit 93 instructions
;---------------------------------------------------------------------
temp_1bd
   movlw    -38               ; en tenant compte des 2 cycles d'arrêt de Tmr0
   movwf    TMR0              ; initialiser tmr0
   call     temp_suite        ; et attendre 1/2 bit

temp_1b
   movlw    -91               ; écart entre 2 bits + 2 cycles d'arrêt
   addwf    TMR0 , f          ; ajouter à la valeur actuelle

temp_suite
   bcf      INTCON , T0IF     ; effacer flag

temp_wait
   btfss    INTCON , T0IF     ; attendre débordement timer
   goto     temp_wait         ; pas fini, attendre
   return                     ; et sortir
```

Comment cette routine fonctionne-t-elle ? Et bien vous remarquez que vous avez 2 points d'entrée, temp_1bd et temp_1b. Commençons par examiner ce dernier point.

Nous commençons par charger la durée correspondant à 91 instructions avant le débordement du timer, soit 0 – 91, ou encore plus simplement –91. Il ne faut cependant pas oublier qu'un certain nombre d'instructions ont déjà été exécutées depuis le dernier passage dans la routine timer, lors de la réception ou l'émission du bit précédent (et qu'on perd 2 incrémentations du timer du fait de sa modification).

Quel est le nombre de ces instructions ? Et bien tout simplement le nombre contenu dans TMR0, puisque le précédent passage avait attendu sa remise à 0. Il ne faut donc pas placer –91 dans tmr0, mais bien AJOUTER –91 au contenu actuel de TMR0.

De cette façon, nous sommes assurés que le débordement du TMR0 se fera exactement 93 cycles après le précédent débordement, quelque soit le nombre d'instructions exécuté entre les 2 appels. Ceci, bien entendu à condition que le timer n'ai pas débordé avant, mais ceci impliquerait que nous n'avons pas le temps de tout exécuter entre 2 bits, donc que le PIC® n'est pas suffisamment rapide, ou notre application pas suffisamment optimisée en temps d'exécution.

Tout de suite après, nous replaçons le flag T0IF à 0, et nous attendons simplement que le débordement du timer entraîne le repositionnement du flag.

Examinons maintenant le point d'entrée temp_1bd : Nous commençons par initialiser TMR0 avec –38, pour démarrer la temporisation de 40 instructions à partir de ce point.

Comme c'est au début de l'acquisition que nous avons besoin d'un demi-bit, il n'y a pas eu de bit précédent, donc nous devons placer cette valeur et non faire une addition. De plus, entre la détection du start-bit et l'initialisation de TMR0, il s'est passé quelques instructions. On peut grossièrement considérer qu'il s'est passé 6 instructions, ce qui nous donne une valeur à charger de –(46 –6) = -40, et en tenant compte des 2 cycles perdus : -38.

Quelle est en réalité notre marge d'erreur ? Et bien tout simplement la moitié de la durée d'un bit, c'est à dire 46 cycles, afin de ne pas tomber sur le bit contigu. Nous ne sommes donc pas à 1 cycle près, mais il faut rester le plus précis possible pour le cas où une dérive de l'appareil connecté augmenterait notre imprécision. Il faut penser que le concepteur du logiciel du maître a peut-être lui aussi profité de cette tolérance.

Ensuite nous appelons la sous-routine elle-même de façon à attendre la durée prévue, puis la sous-routine poursuit en exécutant la temporisation de 93 cycles. Nous avons donc attendu en tout 93 + 40 cycles, soit la durée de 1,5 bit. De plus cette sous-routine n'utilise aucune variable.

19.7 Réception d'un octet

Maintenant que vous avez compris comment recevoir un octet en mode série asynchrone, il est temps d'écrire notre sous-programme de réception d'un octet. Dans cet exercice, nous ne vérifierons pas si le bit de parité reçu est correct. Je vous laisse de soin d'intégrer ce test vous-même si cela vous intéresse. Dans ce cas, vous devrez lire 9 bits et vérifier si le nombre de « 1 » reçus est pair. Il y a plusieurs méthodes possibles.

Notre sous-programme devra donc effectuer les opérations suivantes :

- Attendre le début du start-bit
- Attendre une durée de 93 + 46 instructions
- Pour chacun des 8 bits, lire le bit et le placer dans le caractère reçu en bonne position
- Se positionner quelque part dans les stop-bits

Voici donc notre sous-programme :

```
;*****************************************************************
;              Réception d'un octet provenant du maître          *
;*****************************************************************
;-----------------------------------------------------------------
; Caractère lu dans W. La parité pas n'est pas vérifiée
;-----------------------------------------------------------------
Receive
                  ; attendre début start-bit
                  ; ------------------------

   btfsc    SERIAL            ; Tester si start bit arrivé
   goto     Receive           ; non, attendre

            ; se positionner sur le milieu du 1er bit utile
            ; ---------------------------------------------
```

```
   call     temp_1bd          ; attendre 1bit et demi

                  ; réception du caractère
                  ; ----------------------

   movlw    0x8               ; pour 8 bits
   movwf    cmptbts           ; dans compteur de bits

Recloop
   bcf      STATUS , C        ; Carry = 0
   btfsc    SERIAL            ; tester si bit = 0
   bsf      STATUS , C        ; Carry = bit reçu
   rrf      caract , f        ; faire entrer le bit par la gauche
   call     temp_1b           ; attendre milieu caractère suivant
   decfsz cmptbts , f         ; décrémenter compteur de bits
   goto     Recloop           ; pas dernier, suivant

   ; on pointe actuellement sur le centre du bit de parité
   ; reste donc à attendre +- 1.5 bits pour être dans le second stop-bit
   ; ------------------------------------------------------------------

   call     temp_1bd          ; Attendre 1,5 bit
   movf     caract , w        ; charger caractère lu
   return                     ; et retour
```

Ce sous-programme ne contient pas de difficultés particulières. Remarquez que plutôt que de positionner chaque bit reçu directement à la bonne position, on le fait entrer dans b7 en se servant d'une instruction « rrf ». Le bit précédent est de fait reculé en b6 et ainsi de suite.

Au final, le premier bit lu se retrouve donc en b0, le dernier restant en b7, c'était bien le but recherché.

Concernant la dernière temporisation, nous pointons à ce moment sur le centre du 9ème bit, c'est à dire le bit de parité. Comme nous ne le traitons pas, inutile de le lire. Nous devons alors nous positionner sur un endroit où la ligne est repassée au niveau « 1 », c'est à dire un des stop-bits, afin de pouvoir éventuellement commencer une attente d'un nouveau caractère.

Nous devons donc attendre entre 0.5 et 2.5 bits. La sous-routine 1.5 bit est en plein dans cette zone et est directement utilisable.

Le présent sous-programme utilise des variables que nous déclarerons plus loin.

19.8 L'émission d'un caractère

N'oublions pas que notre carte n'émet qu'en réponse à une interrogation du « maître ». Donc, notre sous-programme d'émission sera appelé après le sous-programme de réception d'un octet.

Il est important également de se souvenir que nous travaillons en mode half-duplex, c'est à dire que la même ligne sert pour les entrées et les sorties. Comme chacun des interlocuteurs parle à tour de rôle, il faut laisser à chacun le temps de repasser en lecture après l'envoi de son dernier message. Ceci s'appelle « temps de retournement ». Il faut également

se rappeler que notre routine de réception s'achèvera quelque part au milieu des stop-bits, il faudra également laisser à l'émetteur le temps de finir d'envoyer la fin de ses stop-bits.

Un bon compromis et la facilité d'écriture du programme nous permet de choisir une attente de 1.5 bit avant de commencer à émettre. J'ai choisi cette valeur car cette temporisation permet d'initialiser le timer. Cette valeur n'est cependant pas critique. Il faut simplement répondre après que le « maître » soit placé en mode de réception, et avant qu'il ne considère que vote carte n'a pas répondu.

Notre programme va donc effectuer les opérations suivantes :

- Attendre temps choisi avant émission
- Passer en émission et envoyer le start-bit
- Envoyer les 8 bits en commençant par b0
- Pour chaque bit « 1 » envoyé, inverser la parité, pour obtenir une parité paire
- Envoyer la parité
- Envoyer les 2 stop-bits
- Repasser en réception

Voici donc notre sous-programme :

```
;*****************************************************************
;                 Envoi d'un octet vers le lecteur de carte          *
;*****************************************************************
;-----------------------------------------------------------------
; envoie l'octet contenu dans le registre w vers le lecteur de carte
;-----------------------------------------------------------------
Send
   movwf    caract            ; Sauver caractère à envoyer
   call     temp_1bd          ; attendre 1 bit et demi
   BANK1                      ; passer banque1
   bcf      SERIAL            ; port série en sortie
   BANK0                      ; repasser banque 0

                         ; envoyer start-bit
                         ; -------------------
   bcf      SERIAL            ; envoyer 0 : start-bit
   clrf     parite            ; effacer bit de parité

   movlw    8                 ; pour 8 bits à envoyer
   movwf    cmptbts           ; dans compteur de bits
   call     temp_1b           ; attente entre 2 bits

                ; envoyer 8 bits de data
                ; --------------------------
Send_loop
   rrf      caract , f        ; décaler caractère, b0 dans carry
   rrf      caract , w        ; carry dans b7 de w
   xorwf    parite , f        ; positionner parité
   xorwf    PORTB , w         ; Garder 1 si changement sur SERIAL
   andlw    0x80              ; on ne modifie que RB7
   xorwf    PORTB , f         ; si oui, inverser RB7
   call     temp_1b           ; attente entre 2 bits
   decfsz   cmptbts , f       ; décrémenter compteur de bits
   goto     Send_loop         ; pas dernier, suivant
```

```
                ; envoyer parité
                ; --------------
movf     parite , w         ; charger parité paire calculée
xorwf    PORTB , w          ; Si serial différent de bit à envoyer
andlw    0x80               ; on ne modifie que RB7
xorwf    PORTB , f          ; alors inverser RB7
call     temp_1b            ; attendre fin de parité

                ; envoyer 2 stop-bits
                ; -------------------
BANK1                       ; passer banque1
bsf      SERIAL             ; repasser en entrée (et niveau haut)
BANK0                       ; passer banque 0
call     temp_1b            ; attendre temps entre 2 bits
call     temp_1b            ; attendre temps entre 2 bits
return                      ; et retour
```

Si vous êtes attentifs, vous avez remarquer une légère inversion en ce qui concerne la fin du protocole. En effet, plutôt que d'envoyer un niveau 1 (stop-bit) puis d'attendre 2 bits et enfin de repasser en entrée, nous sommes passé en entrée puis avons attendu 2bits.

Ceci est strictement identique, car la résistance de rappel au +5V du PORTB, que nous avons activée se charge d'imposer un niveau haut sur RB7 dès que cette pin est remise en entrée, un niveau haut correspondant à un stop-bit.

La routine servant à envoyer les 8 bits utilise l'instruction « xorwf » au lieu de tester si le bit à émettre vaut 1 ou 0. La routine commence par placer le bit à émettre en position b7.

Pourquoi b7 ? Et bien tout simplement parce que c'est également b7 dans le PORTB que nous devrons modifier. Nous utilisons en effet RB7. La procédure utilisée permet de positionner en même temps le bit de parité.

Effectuez l'opération manuellement sur papier pour vous en convaincre. Tentez d'écrire une routine utilisant « btfss » et « btfsc » et comparez les résultats obtenus.

Un petit mot concernant l'envoi du bit proprement dit, c'est à dire les 3 instructions :

```
xorwf    PORTB , w          ; Garder 1 si changement sur SERIAL
andlw    0x80               ; on ne modifie que RB7
xorwf    PORTB , f          ; si oui, inverser RB7
```

Nous commençons ici par lire le PORTB et nous effectuons un « xorlw » avec le bit à envoyer contenu dans « W ». Comme « W » ne contient que ce bit, RB0 à RB6 ne seront pas modifiés par les opérations suivantes.

Si le bit7 contenu dans « W » est différent de celui présent sur RB7, nous obtenons b7 = 1 dans W. Dans le cas où b7 de W est identique à RB7, nous obtenons 0. N'oublions pas en effet que le « ,W » permet de placer le résultat dans « W ».

Si nous appliquons « W » sur le PORTB en effectuant un « xorwf », nous inverserons RB7 uniquement si b7 de « W » vaut 1, c'est à dire, en d'autres mots : Nous inverserons RB7 uniquement si son niveau actuel est différent du niveau que nous devons envoyer.

Ceci paraît un peu « tordu », mais si vous essayez d'écrire cette routine autrement, vous verrez qu'en effectuant plusieurs essais, vous arriverez à un résultat identique, tout ceci à cause de la parité à gérer. Sauf si vous acceptez de modifier les autres lignes du PORTB.

Remarquez que vous pouvez ignorer la vérification du bit de parité en réception, c'est votre problème. Par contre, vous êtes obligé de positionner correctement celle-ci à l'émission, car il y a de fortes chances pour que le maître vérifie cette parité.

19.9 Initialisation

Nous allons maintenant étudier le corps de notre programme principal. Comme tout programme qui se respecte, nous commencerons par l'initialisation. Celle-ci va être très simple, elle se limite à initialiser le registre OPTION.

```
;*****************************************************************
;                       INITIALISATIONS                          *
;*****************************************************************
   org 0x000                    ; Adresse de départ après reset
init
   BANK1                        ; passer banque1
   movlw    OPTIONVAL           ; charger masque
   movwf    OPTION_REG          ; initialiser registre option
   BANK0                        ; passer banque 0
```

19.10 Envoi de l'ATR

ATR signifie Answer To Reset, c'est à dire réponse à un reset. C'est une commande envoyée par la carte lors d'une mise sous tension ou lors d'un reset généré par le maître via la broche « MCLR ».

Tout d'abord, nous allons attendre un peu que le maître soit prêt à recevoir notre ATR. Il peut être nécessaire d'ajuster ce temps en fonction des caractéristiques du maître, qui a probablement d'autres choses à faire au démarrage que de s'occuper directement de votre carte.

```
;*****************************************************************
;                       PROGRAMME PRINCIPAL
*
;*****************************************************************
start
                  ; on commence par attendre un peu
                  ; -------------------------------
   call     temp_1bd     ; attendre 1 bit et demi
```

Ensuite, nous pouvons envoyer l'ATR. Pour des raisons de facilité, nous avons écrit notre ATR dans la zone eeprom interne. J'ai choisi un ATR de 5 caractères que j'ai inventé. Consultez les caractéristiques du maître pour connaître les ATR valides de votre application.

```
;===============================================================
;                         ENVOI DE L'ATR                       =
;===============================================================
;---------------------------------------------------------------
; Envoi d'un ATR fictif : l'ATR est dans les 5 octets de 0x04 à
```

```
; 0x00 de l'eeprom interne. L'ATR est écris en sens inverse
;-----------------------------------------------------------
   movlw    0x5               ; pour 5 octets
   movwf    cmpt1             ; dans compteur de boucles = adresse
ATR_loop
   decf     cmpt1 , w         ; adresse à lire = compteur de boucles-1
   call     Rd_eeprom         ; Lire un octet eeprom interne
   call     Send              ; Envoyer sur le décodeur
   decfsz   cmpt1 , f         ; décrémenter compteur
   goto     ATR_loop          ; pas fini, suivant
```

Remarques

L'utilisation de la commande « decfsz » facilite l'écriture des boucles, mais, comme le compteur de boucles est en même temps l'offset de la position en eeprom, l'ATR sera écrit à l'envers dans l'eeprom, c'est à dire du dernier vers le premier octet.

A l'endroit de l'appel de la sous-routine « Rd_eeprom », le compteur de boucles variera de 5 à 1. Or, notre adresse eeprom variera de 4 à 0. Donc, l'opération « decf » permet de charger dans « W » la valeur du compteur de boucles – 1.

Le sous-routine « Rd_eeprom » n'est rien d'autre que notre macro de lecture de mémoire eeprom transformée en sous-programme. La voici :

```
;*****************************************************************
;                     Lecture d'un octet en eeprom interne       *
;*****************************************************************
;-----------------------------------------------------------------
; Lecture d'un octet de l'eeprom interne. L'adresse est passée dans w.octet lu
est  dans W
;-----------------------------------------------------------------
Rd_eeprom
   movwf    EEADR             ; adresse à lire dans registre EEADR
   bsf      STATUS , RP0      ; passer en banque1
   bsf      EECON1 , RD       ; lancer la lecture EEPROM
   bcf      STATUS , RP0      ; repasser en banque 0
   movf     EEDATA , w        ; charger valeur lue dans W
   return                          ; retour
```

Pensons également à inscrire notre ATR dans la zone eeprom :

```
;*****************************************************************
;                  DECLARATIONS DE LA ZONE EEPROM                *
;*****************************************************************
   org 0x2100                ; adresse début zone eeprom

ATR  DE     0x07              ; Réponse à l'ATR
     DE     0xAB              ; B7 01 BB AB 07
     DE     0xBB
     DE     0x01
     DE     0xB7
```

19.11 L'envoi du status

La norme ISO 7816 demande que chaque émission d'une réponse de la carte soit suivi de 2 octets de status qui indiquent la manière dont a été interprétée la commande.

J’ai inventé des status pour cet exercice. Le status « 80 00 » indiquera que la commande a été exécutée correctement. J’utiliserai également le status « 60 40 » pour indiquer que la commande n’existe pas.

Nous allons donc créer 2 sous-programme. Un qui envoie le status standard, l’autre qui envoie n’importe quel status.

```
;=============================================================
;                          ENVOI DU STATUS STANDARD           =
;=============================================================
;-------------------------------------------------------------
; Envoie le status standard, dans ce cas on a pris 0x80 0X00
;-------------------------------------------------------------
Statstd
   movlw    0x80              ; prendre 1er octet status
   call     Send              ; l'envoyer
   clrw                       ; effacer w
   call     Send              ; envoyer 00
   goto     classe            ; et traiter classe

;=============================================================
;                          ENVOI D'UN STATUS SPECIFIQUE       =
;=============================================================
;-------------------------------------------------------------
; Envoie d'abord l'octet contenu dans w, puis l'octet contenu dans status2
;-------------------------------------------------------------
Statxx
   call     Send              ; on envoie valeur
   movf     status2 , w       ; charger byte à envoyer
   call     Send              ; on envoie 2ème octet du status
```

19.12 Réception de la classe

Maintenant, notre carte passe en mode réception et attend sa première commande. Notre programme, pour des raisons de facilité ne gère qu’une seule classe. Nous nous contentons donc de lire l’octet, sans le vérifier ni le traiter. Il s’agit en effet d’un exercice didactique.

```
;=============================================================
;                          LECTURE DE LA CLASSE               =
;=============================================================
;-------------------------------------------------------------
; on considère dans cet exemple qu'il n'y a qu'une seule classe valide.
; on attend l'arrivée de la classe et on ne la traite pas
;-------------------------------------------------------------
classe
   call     Receive           ; Lire le byte venant du maître
```

19.13 Réception de INS, P1, P2, et LEN

Examinons maintenant notre routine de réception de l’instruction, des paramètres P1 et P2, ainsi que de la longueur de la chaîne.

```
;=============================================================
;                        LECTURE DE INS,P1,P2,LEN             =
;=============================================================
;-------------------------------------------------------------
; INS sera placé dans la variable Ser_ins P1 sera placé dans
; Ser_P1 et P2 dans Ser_P2
; La longueur du champs de data sera dans Ser_len
;-------------------------------------------------------------
   movlw    Ser_Ins           ; pointer sur emplacement instruction
   movwf    FSR               ; initialiser pointeur indirection
read_loop
   call     Receive           ; Lire un octet
   movwf    INDF              ; sauver dans emplacement prévu
   incf     FSR , f           ; pointer sur suivant
   btfss    FSR , 0x4         ; Tester si adresse 0x10 atteinte
   goto     read_loop         ; non, octet suivant
```

Vous pouvez constater l'utilisation de l'adressage indirect pour sauvegarder les octets reçus dans 4 emplacements consécutifs. Nous choisirons les adresses 0x0C à 0x0F, ce qui nous permet facilement de détecter la fin de la commande. En effet, une fois le 4ème octet sauvé, FSR pointe sur 0x10, il suffit donc de tester son bit 4 pour tester la fin de la boucle, sans avoir besoin d'un compteur de boucles supplémentaire.

19.14 Contrôle de l'instruction reçue

Une fois la commande reçue, nous devons traiter les différentes commandes reçues. Dans notre exemple didactique, j'ai implémenté une seule instruction. La seule instruction valide est l'instruction 0x25.

Cette instruction calcule la somme de P1 et P2, et renvoie le résultat. Comme LEN contient la longueur de la chaîne de réponse, si LEN est supérieur à 1, la réponse sera complétée par des 0xFF. Toute autre instruction sera considérée comme incorrecte.

Voici notre test :

```
;=============================================================
;                    SWITCH SUIVANT INSTRUCTION RECUE         =
;=============================================================
;-------------------------------------------------------------
; Nous allons imaginer que nous allons réagir à une instruction 0x25
; Toute autre instruction sera considérée comme incorrecte
;-------------------------------------------------------------
                        ; tester instruction reçue
                        ; ------------------------
   movf     Ser_Ins , w       ; charger instruction reçue
   sublw    0x25              ; comparer avec 0x25
   btfsc    STATUS , Z        ; tester si identique
   goto     Ins25             ; oui, traiter instruction 25

                  ; traiter instruction incorrecte
                  ; ------------------------------
   movlw    0x40              ; charger octet2 status à envoyer
   movwf    status2           ; placer dans variable
   movlw    0x60              ; charger octet 1 status
   goto     Statxx            ; envoyer status
```

Nous voyons ici que si l'instruction reçue est 0x25, nous sautons au traitement de l'instruction. Dans le cas contraire, nous envoyons le status « 60 40 » qui signifie dans notre cas « instruction incorrecte ».

19.15 Traitement d'une instruction

Nous en arrivons maintenant au traitement de notre instruction proprement dite.

On va traiter cette instruction de la manière suivante :

- Comme dans toute instruction, on renvoie l'instruction reçue
- La carte renvoie la somme de P1 et de P2.
- La trame d'envoi est complété par des 0xFF pour atteindre une longueur totale de data identique à Ser_Len
- Ensuite le status standard est envoyé « 80 00 »

```
;=============================================================
;                     TRAITER INSTRUCTION 25                  =
;=============================================================
Ins25
                  ; envoyer écho de la commande
                  ; ---------------------------
   movf     Ser_Ins , w       ; charger commande reçue
   call     Send              ; renvoyer en écho

                  ; renvoyer P1 + P2
                  ; ----------------
   movf     Ser_P1 , w        ; charger P1
   addwf    Ser_P2 , w        ; + P2
   call     Send              ; envoyer résultat

            ; Tester longueur de la réponse
            ; ----------------------------------
   decf     Ser_Len , f       ; car déjà résultat envoyé
   btfsc    STATUS , Z        ; tester si complet
   goto     Statstd           ; oui, envoyer status standard

     ; compléter avec des 0xFF
     ; ----------------------
Insloop
   movlw    0xFF              ; valeur à envoyer
   call     Send              ; envoyer 0xFF
   decfsz    Ser_Len , f      ; décrémenter compteur de boucles
   goto     Insloop           ; pas fini, suivant

     ; envoyer status standard
     ; ---------------------------
   goto     Statstd           ; envoyer status standard
```

19.16 Les variables

Il nous reste maintenant à déclarer les variables utilisées. J'ai décidé ici d'utiliser des variables locales lorsque c'était possible :

```
;*****************************************************************
;                    DECLARATIONS DE VARIABLES                    *
;*****************************************************************

   CBLOCK   0x00C       ; début de la zone variables
   Ser_Ins : 1          ; instruction ISO7816
   Ser_P1    :  1       ; paramètre 1 ISO7816
   Ser_P2    : 1        ; paramètre 2 ISO7816
   Ser_Len : 1          ; longueur data ISO7816
   local1 : 1           ; variable locale 1
   local2 : 1           ; variable locale 2
   local3 : 1           ; variable locale 3
   local4 : 1           ; variable locale 4
   temp_sauvw : 1       ; sauvegarde de W pour temp
   ENDC                 ; Fin de la zone

                  ; routine ATR
                  ; -----------
#DEFINE     cmpt1       local1      ; compteur d'octets pour ATR

            ; sous-routine send et receive
            ; ----------------------------
#DEFINE     caract      local2      ; caractère à envoyer
#DEFINE     parite      local3      ; bit de parité
#DEFINE     cmptbts     local4      ; compteur de bits

                  ; pour STATUS
                  ; -----------
#DEFINE     status2     local1      ; octet 2 du status

                  ; pour instruction 25
                  ; -------------------
#DEFINE     cmpt2       local1      ; compteur d'octets
```

19.17 Conclusion

Vous savez maintenant communiquer en liaison série asynchrone. De plus vous avez les notions de base nécessaire pour réaliser une carte répondant à la norme ISO7816. Nul doute que vous trouviez de nombreuses applications mettant en œuvre la théorie vue ici.

Je vous rappelle qu'il s'agit ici d'un exemple imaginaire. Inutile donc de m'écrire pour me demander comment transformer cet exemple pour l'utiliser en pratique en tant que « clé pour une serrure codée » quelconque. Je rappelle que je ne répondrai pas.

J'ai reçu également un nombreux courrier de lecteurs qui ont cru « avisé » de commencer la lecture du cours par ce chapitre. Il ne faut pas mettre la charrue avant les bœufs, et si ce chapitre est placé en fin de cours, c'est que j'ai estimé qu'il nécessitait la compréhension de ce qui précède.

Annexe1 : Questions fréquemment posées (F.A.Q.)

Je vais tenter de répondre ici à un maximum de questions que se posent les utilisateurs en général.

A1.1 Je trouve que 8 sous-programmes, c'est peu

Attention, vous ne devez pas confondre le nombre de sous-programme et le nombre d'imbrications de sous-programmes. Le nombre de sous-programme est illimité (dans la limite de la mémoire disponible).

Une imbrication, c'est quand un sous-programme appelle un autre sous-programme. Pour compter les niveaux d'imbrications, suivez le chemin de votre programme dans l'ordre d'exécution. Faites +1 pour chaque instruction « call » rencontrée et (-1) pour chaque instruction « return » ou « retlw ».

Si votre programme est correct, les 3 conditions suivantes doivent être remplies :

- On ne doit jamais durant le comptage passer en négatif
- On ne peut jamais dépasser 8 durant le comptage
- A la fin de l'exécution du programme, on doit être revenu à 0

A1.2 Je n'utilise que 8 imbrications, et pourtant mon programme plante.

Vous ne devez pas oublier que les interruptions utilisent aussi la pile. Si vous utilisez les interruptions, vous n'avez droit qu'à 7 niveaux d'imbrication (moins les sous-routines utilisées éventuellement par la routine d'interruption).

A1.3 Mon programme semble ne jamais sortir des interruptions

Vous avez oublié d'effacer le flag qui a provoqué l'interruption.

A1.4 Je n'arrive pas à utiliser le simulateur, les options n'apparaissent pas

Vous avez oublié de signaler à MPLAB® que vous utilisez le simulateur. Allez dans le menu « debugger -> select tool » et sélectionnez «MPLAB® SIM ». Configurez ensuite comme expliqué dans le chapitre concerné.

A1.5 Je reçois un message d'erreur EOF avant instruction END

Vérifiez si votre fichier comporte la directive END.

Si ce n'est pas le cas, ouvrez votre fichier « .asm » dans le bloc-notes de windows et vérifiez qu'il se présente correctement, avec les mises à la ligne correctes. Si ce n'est pas le cas, c'est que le fichier a été écrit avec un éditeur qui ne génère pas les bons retour de ligne.

Dans ce cas, essayer un copier/coller de ce fichier dans un fichier vide créé avec un autre éditeur.

A1.6 Comment désassembler un fichier « .hex » ?

Je vais vous décrire 2 méthodes pour désassembler un fichier en format hexadécimal. La première méthode utilise MPLAB® :

1) Allez dans le menu « File-> import »
2) Sélectionnez le fichier « .hex » à désassembler
3) Sélectionnez « View-> program memory »
4) Sélectionner l'onglet qui correspond à votre désir

La seconde méthode utilise IC-Prog, le célèbre utilitaire de programmation des PIC® disponible partout :

1) Chargez le fichier « .hex » depuis le menu « File->open file »
2) Sélectionnez « view->assembler ». C'est tout.

A1.7 Utilisation des minuscules et des majuscules

Par défaut, MPLAB® effectue la distinction entre minuscules et majuscules. Si cela vous pose problème, vous pouvez le modifier dans les propriétés de votre projet.
Pour effectuer cette opération :

- Sélectionnez «project -> build options -> project »
- Sélectionnez l'onglet « MPASM assembler »
- Cochez la case « disable case sensitivity »

A1.8 Le choix d'un programmateur

Je ne veux pas entrer ici dans le concret, car je veux pas faire de publicité pour un ou l'autre produit sur le marché.

Je me limiterai donc à un conseil : Utilisez un programmateur série ou parallèle, mais choisissez de préférence un modèle disposant de sa propre alimentation, de préférence à un modèle tirant son alimentation du port série.

Je suis sidéré de voir que dans la presse dite « spécialisée », certains auteurs continuent de préconiser l'utilisation de programmateurs qui ne répondent pas à ces prescriptions. Ceci me vaut un abondant courrier de lecteurs qui se retrouvent dans l'impossibilité de programmer certains de leurs composants (dont les 16F84A qui sont en mode « code protect »).

En tout état de cause, FUYEZ les programmateurs qui tirent leur alimentation du port série, ils sont sources de la plupart des problèmes de programmation.

Par contre, pour le logiciel, je vous recommande le meilleur d'entre-eux (à mon avis), et, de plus, entièrement gratuit.

Ce logiciel est IC-PROG, et son auteur, Bonny Gijzen, est d'une gentillesse à toute épreuve, n'hésitez pas à le remercier par email (en anglais) si vous décidez d'utiliser son programme. Il a eu la grande gentillesse de mettre son programme à jour spécialement pour moi (donc pour vous), afin de permettre certaines options utilisées dans la seconde partie du cours. Vous pouvez donc lui dire que vous avez trouvé son lien dans mon petit cours.

L'adresse de téléchargement de IC-PROG est « www.IC-PROG.com ».

A1.9 J'ai une erreur de « stack »

Je reçois également pas mal de courrier de personnes qui me disent : « Lorsque j'exécute mon programme en pas-à-pas dans MPLAB®, je reçois à un moment donné un message de type : « Stack overflow » ou « Stack underflow » ».

Que signifie ces messages ?

En fait un message « stack overflow » peut signifier que vous avez dépassé les 8 niveaux de sous-programme autorisés. En effet, je rappelle qu'il n'y a que 8 emplacements sur la pile (stack), et que donc, une nouvelle tentative d'empilement, suite à une sous-routine ou une interruption provoquera l'écrasement de la pile.

Dans la plupart des cas, cependant, il s'agit d'une erreur dans la structure de votre programme. Le message « stack overflow » intervient si vous empilez plus de 7 emplacements, le message « stack underflow » intervient si vous dépilez plus que ce que vous avez empilé (par exemple, « return » sans « call »).

Pour résoudre ces problèmes, si vous n'avez pas dépassé les 8 niveaux (voir A1.2), vérifiez les points suivants :

- Chaque appel via un call doit revenir au programme appelant par un « return » ou un « retlw »
- Chaque « return » ou « retlw » rencontré doit avoir été précédé du call correspondant
- La sortie d'une sous-routine avec un « goto » doit mener à un point où on trouvera un « return »
- Une routine ne doit pas s'appeler elle-même (sauf fonctions récursives : rares avec une telle taille de pile).

A1.10 Quelles sont les différences entre 16F84 et 16F84A ?

Voici une question qui revient régulièrement dans mon courrier. Je vous livre ici les principales différences entre ces composants :

- La fréquence maximale disponible passe de 10 à 20 Mhz.
- La tension maximale pour Vdd est ramenée de 6V à 5.5V.
- Le reset nécessite une durée d'impulsion de 2µs sur Mclr au lieu de 1µs
- Temps de montée de la tension d'alimentation prise en compte pour le démarrage
- Le courant typique lors de la programmation flash passe de 7,3 à 3 mA.
- Le courant typique consommé double entre un 16F84 à 10Mhz et un 16F84A à 20Mhz
- Plusieurs modifications des caractéristiques électriques (tensions des niveaux, courants…)
- Division par 2.5 des temps d'écriture et d'effacement eeprom
- Division par 2.5 des temps d'écriture et d'effacement en mémoire flash

En sommes rien qui concerne la programmation, ce sont des différences au niveau électrique.

A1.11 J'ai une erreur 173 lors de l'assemblage

Au moment de lancer l'assemblage, vous obtenez une erreur 173, du genre :

```
Error[173] Source file path exceeds 62 characters
```

Ceci signifie que le chemin pour accéder à votre source excède 62 caractères, ce qui peut être le cas si vous utilisez des noms trop longs ou trop de sous-répertoires. Par exemple :

```
C:\Mon_repertoire\sources\Sources_MPLAB\Cours\Cours_Part1\Exercices\test1.Asm
```

Il vous suffit dans ce cas de déplacer votre répertoire dans un dossier plus proche de la racine, ou de raccourcir les noms utilisés. Voici un exemple correct :

```
C:\Cours_PIC_1\Exercices\test1.asm
```

Contribution sur base volontaire

La réalisation de ces cours m'a demandé beaucoup de temps et d'investissements (documentations, matériel, abonnements, etc.).

Aussi, pour me permettre de poursuivre, je vous demande, si cela est dans vos possibilités, et si vous appréciez ce que je fais, de contribuer un peu, chacun selon ses possibilités et ses désirs.

J'ai donc besoin de votre aide pour continuer l'aventure. En effet, je ne dispose plus vraiment de la capacité de consacrer l'intégralité de mon temps libre à écrire des cours et des programmes sans recevoir un petit "coup de pouce".

Cependant, je ne voulais pas tomber dans le travers en verrouillant l'accès aux fichiers, et en imposant un payement pour les obtenir. En effet, je tiens à ce qu'ils restent disponibles pour tous.

J'ai donc décidé d'instaurer un système de contribution sur base volontaire en permettant à celui qui le désire, et en fonction de ses propres critères, de m'aider financièrement. Le but n'étant pas de me faire riche, mais plutôt de m'aider à "joindre les 2 bouts".

Il ne s'agit donc pas d'un payement, ni d'une obligation. Il s'agit simplement d'une assistance sans promesse d'aucun sorte, et sans contrainte. Je continuerai à répondre au courrier de tout le monde, sans distinction, et sans interrogation à ce sujet.

Une bonne méthode consiste donc, pour celui qui le désire, à télécharger le document choisi, le lire ou l'utiliser, puis décider si cela vaut ou non la peine de m'aider sur base de l'usage que vous en faites.

Si oui, vous vous rendez sur mon site : www.abcelectronique.com/bigonoff ou www.bigonoff.org, et vous suivez la page « cours-part1 ». Vous y trouverez, dans la page « contributions », la procédure à suivre. Pensez que ces contributions me sont très utiles, et contribuent à me permettre de continuer à travailler pour vous.

Pour remercier ceux qui ont contribué, soit par l'envoi d'une lettre, soit par la création d'un site d'utilité publique, j'offrirai le logiciel BigoPic V2.0.

N'oubliez pas de mettre votre email en caractère d'imprimerie, pour que je puisse vous répondre.

Je réponds toujours au courrier reçu. Aussi, si vous n'obtenez pas de réponse, n'hésitez surtout pas à me contacter pour vérifier s'il n'y a pas de nouveau un problème.

Merci d'avance à tous ceux qui m'ont aidé ou m'aideront à poursuivre ce travail de longue haleine.

Utilisation du présent document

Le présent ouvrage est destiné à faciliter la compréhension de la programmation des PIC® en général, et du 16F84 en particulier. La seconde partie consacrée au reste de la gamme via le 16F876 est disponible sur mon site.

Communiquez à l'auteur (avec politesse) toute erreur constatée afin que la mise à jour puisse être effectuée dans l'intérêt de tous.

J'avais autorisé de proposer la première partie en téléchargement sur des sites de webmasters qui en faisaient la demande. Je remercie tous ceux qui ont joué correctement le jeu.

Cependant, certains ne l'ont pas fait, et n'ont pas mis leur site régulièrement à jour en fonction des nouvelles versions. Ceci implique que je reçois régulièrement du courrier de personnes qui me signalent des erreurs corrigées depuis longtemps, et des utilisateurs qui s'aperçoivent avec dépit qu'ils viennent d'imprimer une version obsolète du cours. Vu le nombre de pages, ça ne fait pas plaisir à tout le monde.

Aussi, pour ces raisons, et par facilité de maintenance pour moi, j'ai décidé que ce cours serait disponible uniquement sur mon site : http://www.abcelectronique.com/bigonoff ou www.bigonoff.org . Aussi, si vous trouvez mon cours ailleurs, merci de m'en avertir.

Malheureusement, malgré avoir été avertis, plusieurs webmasters ont refusés de remplacer le cours-part6 disponible sur leur site par un lien vers mon propre site (probablement pour raisons d'audimat). Malheureusement ça se traduit par des internautes qui impriment un cours devenu obsolète (traitant par exemple de MPLAB® 5), c'est regrettable de faire passer son compteur de visites avant l'intérêt des internautes.

Bien entendu, j'autorise (et j'encourage) les webmasters à placer un lien sur le site, inutile d'en faire la demande. Bien entendu, je ferai de même en retour si la requête m'en est faite. Ainsi, j'espère toucher le maximum d'utilisateurs.

Microchip®. Les datasheets sont également disponibles gracieusement sur le site de cette société, à savoir : http://www.Microchip.com

Si vous avez aimé cet ouvrage, si vous l'utilisez, ou si vous avez des critiques, merci de m'envoyer un petit mail. Ceci me permettra de savoir si je dois ou non continuer cette aventure avec les parties suivantes.

Sachez que je réponds toujours au courrier reçu, mais notez que :

- Je ne réalise pas les programmes de fin d'étude pour les étudiants (même en payant), c'est une demande qui revient toutes les semaines dans mon courrier. Tout d'abord je n'ai pas le temps, et ensuite je ne pense pas que ce soit un bon service. Enfin, pour faire un peu d'humour, si je donnais mes tarifs, ces étudiants risqueraient un infarctus.

- Je n'ai malheureusement pas le temps de debugger des programmes complets. Inutile donc de m'envoyer vos programmes avec un message du style « Ca ne fonctionne pas, vous pouvez me dire pourquoi ? ». En effet, je passe plus de 2 heures par jour pour répondre au courrier, si, en plus, je devais debugger, j'y passerais la journée. Vous comprenez bien que c'est impossible, pensez que vous n'êtes pas seul à poser des questions. Posez plutôt une question précise sur la partie qui vous semble inexacte.

- Si vous avez des applications personnelles, n'hésitez pas à les faire partager par tous. Pour ce faire, vous pouvez me les envoyer par mail.

- Avec cette version, j'essaye de répondre aux demandes légitimes des personnes qui travaillent sur différentes plates-formes (Mac, Linux, Windows, etc.). Si, cependant, la version fournie est inexploitable sur votre machine, merci de me le faire savoir. Notez cependant que ce cours utilise MPLAB® pour les exercices, il faudra donc éventuellement adapter ces exercices en fonction du logiciel qu'il vous sera possible d'utiliser.

Merci au webmaster de www.abcelectronique.com, pour son hébergement gratuit.
Merci à Byte, pour sa proposition d'hébergement gratuit.
Merci à Grosvince pour sa proposition d'hébergement gratuit.
Merci à Thierry pour m'avoir offert mon nom de domaine
Merci à Bruno d'en avoir repris le payement depuis 2007.

Merci à tous ceux qui m'ont envoyé des mails d'encouragement, ça motive plus que vous ne le pensez.

Dernière remarque : il est impossible que vous trouviez trace d'un plagiat ici, tout comme dans les précédents cours, étant donné que je n'ai lu **aucun** ouvrage sur le sujet, autre que les datasheets de Microchip®. Tout est donc issu de mes propres expériences. Si vous trouvez trace d'un plagiat (j'en ai vu), sachez que ce sont les autres qui ont copié, et ceci vaut également pour les ouvrages édités de façon classique (cette remarque n'est pas innocente).

Edition terminée le 09/02/2001.

- Mise à jour révision 2 le 15/06/2001 : correction de quelques erreurs

- Mise à jour révision 3 le 24/07/2001 : correction détaillée avec les Fribottes

- Mise à jour révision 4 le 26/10/2001 : quelques petites erreurs retrouvées

- Mise à jour révision 5 le 27/02/2002 : encore quelques erreurs tenaces

- Mise à jour révision 6 le 20/04/2002 : quelques corrections, améliorations de points suscitant des questions fréquentes, passage en format « pdf » sous compression rar pour permettre l'utilisation sur toutes les machines, masculinisation du terme « PIC® »

- Mise à jour révision 7 le 22/09/2002 : quelques corrections syntaxiques et orthographiques mineures et quelques ajouts.

- Mise à jour révision 8 le 25/10/2002 : Petite correction chapitre 2.4. L'énoncé cité en exemple ne correspond pas à celui effectivement développé. Correction du lien vers le site Microchip® en page 27. Modification de la définition de CISC et RISC page 17.

- Mise à jour révision 9 le 01/11/2002 : Fautes d'orthographe pages 82, 83, 85, 86, 87, 89, 90, 94.

- Mise à jour révision 10 le 04/12/2002 : suppression d'une ligne qui prête à confusion page 183, quelques correctifs mineurs pages 99, 100, 104,108, 122, 124, 128, 129, 131,132, 134, modification de contribution page 209.

- Mise à jour révision 11 le 19/04/2003 : modification des registres modifiés pour l'instruction sleep page 74, ajout du programmateur dans la liste des composants nécessaires, modif dans le define page 123, correctif mineur page 70.

- Mise à jour révision 12 le 16/06/2003 : Corrections mineures page 108, 198

- Mise à jour révision 13 le 19/09/2003 : Grosses modifications pour le passage de MPLAB® 5.50 à MLPAB IDE 6.30. Numérotation changée, nombreuses modifications sur des chapitres entiers. Réimpression complète nécessaire, fichiers exemples reconstruits.

- Mise à jour révision 14 le 11/12/2003 : Modification du schéma page 134, correctifs pages 201,190,191 (modification de radix suite au passage de MPLAB® 5 à MPLAB® 6).

- Mise à jour révision 15 le 13/02/2004 : Correction d'une erreur répétitive sur les exemples et le cours, au niveau de l'initialisation de EEADR (changement de banque une ligne trop tôt), pages 96,99,123,142 et fichiers d'exemples et de maquette.

- Mise à jour révision 16 le 09/03/2004 : correction page 58,59

- Mise à jour révision 17 le 09/06/2004 : corrections page 34, 198,annexes, ajout d'une astuce de programmation, table des matières, modification de mon adresse courrier.

- Mise à jour révision 18 le 02/08/2004 : Ajout du temps de réveil page 171, remarque page 113

- Mise à jour révision 19 le 27/03/06 : correctifs pages 11, 12, 13, 15, 17, 19, 21, 24, 26, 27, 29, 30, 33, 37, 38, 40, 47, 50, 52, 54, 63, 72, 80, 83, 85, 93, 96, 114, 131, 133, 135, 171, 197, 198, 220, remises en page des portions de codes suite à une modification accidentelle des tabulations, mise à jour page 20, remarque pages 24 et 58. Ajout d'une annexe.

- Mise à jour révision 20 le 03/04/06 : Ajout d'un petit chapitre sur la consommation minimale en mode sleep

- Mise à jour révision 21 le 13/10/06 : correctifs mineurs (surlignages, parenthèses, orthographe…) pages 19, 25, 39, 40, 41, 47, 51, 52, 53, 54, 57, 61, 67, 68, 69, 71, 72, 76, 80, 81, 86, 98, 118, 120, 128, 152, 178, passage des nombres H'00xx ' sur 8 bits. Ajout d'une remarque page 44. Révision 21b, remise en place du schéma de l'optocoupleur qui avait disparu.

- Mise à jour révision 22 le 24/01/07 : Modifications importantes dans tout ce qui traite du timer0 suite à l'oubli de prendre en compte les 2 cycles perdus lors de toute modification de TMR0. Modifications et adaptations pages 29, 44, 52, 95 y compris les fichiers asm concernés. Divers correctifs mineurs (tabulations, espaces, ponctuations, orthographe...), petite modification page 54 concernant le simulateur (l'explication était restée sur la version 5). Modification page 21 au niveau de l'explication de la mention « -xx » que j'avais oublié d'adapter aux diverses expériences effectuées à ce sujet.

- Mise à jour révision 23 le 04/11/07 : Ajout d'une liste de considérations sur le watchdog, au chapitre 15.3. Correctifs orthographiques ou tournures de phrases pages 16, 23, 34, 37, 44, 61, 64, 79, 83, 86, 89, 91, 96, 108, 111, 112, 113, 114, 120, 124, 128, 138. Modification d'un chemin page 38. Précisions sur le terme « positionné » pages 27, 43, 50, 54, 60, 110. Correctif n°de bit page 12. Décalage de certains numéros de pages. A la demande amicale de Microchip®, ajout du symbole ® pour tout terme déposé par Microchip®.

- Mise à jour révision 24 le 08/02/08 : Modification mineure page 47. Remplacement de système de numérotation par numération. Refonte totale des chapitres 12.7.2 à 12.7.4 (ajouts) impliquant la renumérotation de toutes les pages suivantes.

Sur mon site, vous trouverez également des livres de reports d'information, qui vous permettront de vous exprimer sur ce cours, et sur les autres fichiers proposés en téléchargement. Posez de préférence vos questions et envoyez vos correctifs par email.

Réalisation : Bigonoff

Site : http://www.abcelectronique.com/bigonoff (merci à l'hébergeur)
Domaine : www.bigonoff.org (merci à Thierry)
Email : bigocours@hotmail.com (Attention BIGOCOURS PAS BIGONOFF)