UNIVERSITE DE LIEGE

Faculté des Sciences Appliquées

# *Introduction aux microcontrôleurs et à leur assembleur*

## Illustration par le *PIC16F84*

Année académique 2004-2005

# Table des matières

**Introduction** **4**

**1 Microcontrôleurs** **5**
1.1 Définitions . . . . . 5
1.2 Les PICs . . . . . 5

**2 Architecture** **7**
2.1 Mémoire . . . . . 7
2.1.1 Mémoire de programme . . . . . 7
2.1.2 Mémoire RAM . . . . . 8
2.1.3 Mémoire EEPROM . . . . . 9
2.2 Pile matérielle . . . . . 9
2.3 Registres . . . . . 11
2.3.1 Registre *W* . . . . . 11
2.3.2 Registre *TMR0* . . . . . 11
2.3.3 Registre *OPTION* . . . . . 11
2.3.4 Registre *PCL* . . . . . 11
2.3.5 Registre *PCLATH* . . . . . 12
2.3.6 Registre *STATUS* . . . . . 12
2.3.7 Registre *FSR/INDF* . . . . . 13
2.3.8 Registres *PORTA/PORTB* . . . . . 13
2.3.9 Registre *TRISA/TRISB* . . . . . 13
2.3.10 Registre *EEDATA* . . . . . 13
2.3.11 Registre *EEADR* . . . . . 13
2.3.12 Registre *EECON1* . . . . . 13
2.3.13 Registre *EECON2* . . . . . 14
2.3.14 Registre *INTCON* . . . . . 14
2.4 Timer 0 . . . . . 14
2.5 Entrées/sorties . . . . . 15

**3 Programmation du PIC** **17**
3.1 Préparation du projet . . . . . 17
3.2 Structure d'un fichier *.asm* . . . . . 18
3.2.1 Les commentaires . . . . . 18
3.2.2 Les directives . . . . . 19
3.2.3 Les instructions . . . . . 21
3.3 Assemblage . . . . . 21
3.4 Simulation . . . . . 22

**4 Instructions** **24**
4.1 Instructions orientées octets . . . 25
4.1.1 ADDWF: ADD W and F . . . 25
4.1.2 ANDWF: AND W with F . . . 25
4.1.3 CLRF: CLear F . . . 25
4.1.4 CLRW: CLear W . . . 25
4.1.5 COMF: COMplement F . . . 26
4.1.6 DECF: DECrement F . . . 26
4.1.7 DECFSZ: DECrement File, Skip if Z . . . 26
4.1.8 INCF: INCrement F . . . 26
4.1.9 INCFSZ: INCrement F, Skip if Z . . . 27
4.1.10 IORWF: Inclusive OR W with File . . . 27
4.1.11 MOVF: MOVe f . . . 27
4.1.12 MOVWF: MOVe W to F . . . 27
4.1.13 NOP: No Operation . . . 28
4.1.14 RLF: Rotate Left through Carry . . . 28
4.1.15 RRF: Rotate Right through Carry . . . 28
4.1.16 SUBWF: SUBstract W from F . . . 28
4.1.17 SWAPF: SWAP nibbles in F . . . 29
4.1.18 XORWF: eXclusive OR W with File . . . 29
4.2 Instructions orientées bits . . . 29
4.2.1 BCF: Bit Clear File . . . 29
4.2.2 BSF: Bit Set File . . . 29
4.2.3 BTFSC: Bit Test F, Skip if Clear . . . 30
4.2.4 BTFSS: Bit Test F, Skip if Set . . . 30
4.3 Instructions générales . . . 30
4.3.1 ADDLW: ADD Literal and W . . . 30
4.3.2 ANDLW: AND Literal with W . . . 31
4.3.3 CALL: CALL subroutine . . . 31
4.3.4 CLRWDT: CLeaR WatchDog Timer . . . 31
4.3.5 GOTO: GO TO address . . . 31
4.3.6 IORLW: Inclusive OR Literal with W . . . 32
4.3.7 MOVLW: MOV Literal to W . . . 32
4.3.8 RETFIE: RETurn From IntErupt . . . 32
4.3.9 RETLW: RETurn with Literal in W . . . 32
4.3.10 RETURN: RETURN from subroutine . . . 33
4.3.11 SLEEP . . . 33
4.3.12 SUBLW: SUBstract W from Literal . . . 33
4.3.13 XORLW: eXclusive OR Literal with W . . . 34

**5 Interruptions** **35**
5.1 Rappel . . . 35
5.2 Les interruptions sur le *16F84* . . . 35

**6 Exemples d'applications** **39**
6.1 Exemple simple d'utilisation du *16F84* . . . . . 39
6.2 Exemple avancé . . . . . 41
6.2.1 Timer0/Interrputions . . . . . 47
6.2.2 EEPROM . . . . . 47

**7 Evolution: vers le *16F87x*...** **48**

**Références** **49**

# Introduction

Ce document est un complément aux cours d'électronique (principalement ceux d'*électronique numérique* et de *microélectronique*) du Professeur Destiné. Il offre une introduction à l'étude des microcontrôleurs et de leur assembleur.

Si nous avons essayé de limiter au maximum les références à ces cours afin de garder une certaine généralité à notre propos, ce manuel nécessite néanmoins quelques pré-requis. Nous supposons donc que le lecteur possède des connaissances élémentaires en électronique numérique (théorique et pratique) et en structure des ordinateurs; ainsi que sur les manipulations et opérations sur les nombres binaires et hexadécimaux. Nous n'envisagerons pas non plus les aspects plus pratiques liés à l'utilisation des microcontrôleurs: alimentation, manipulation des composants, soudure... Pour de plus amples détails sur ces sujets, le lecteur peut consulter [2], [4] ou encore [5].

Par contre, nous considérons que le lecteur est entièrement néophyte en assembleur. Nous repartirons donc des bases dans ce domaine. Il ne s'agira cependant pas de réaliser un cours complet d'introduction à l'assembleur (d'autres s'en sont chargés mieux que nous, voir [4] par exemple) mais de présenter les concepts les plus courants qui pourraient nous être utiles dans l'utilisation des microcontrôleurs.

Plutôt qu'une étude théorique des microcontrôleurs en général, nous présenterons ceux-ci à l'aide d'un exemple, le PIC; et plus particulièrement un membre de cette famille: le *16F84*. Nous avons choisi ce microcontrôleur pour diverses raisons: sa simplicité, parfaitement adaptée à l'apprentissage; son usage relativement répandu; et la présence de matériel dédié à celui-ci à Montefiore (notamment 2 cartes de développement de type *PICDEM 2 Plus*).

Encore une fois, ce manuel n'a pas la prétention d'être un cours exhaustif sur le *16F84* mais plutôt une introduction utile et un manuel pratique, rapide et facile à consulter, offrant une aide lors de la réalisation de laboratoire ou de manipulations sur le PIC. Le lecteur intéressé par une étude plus en profondeur peut consulter le Bigonoff ([3]), une référence clef en matière de PICs; ou mieux encore, les datasheets ([1]), ce qui reste encore le meilleur moyen d'obtenir les informations les plus complètes.

# 1 Microcontrôleurs

## 1.1 Définitions

Il convient de commencer par définir le concept même de microcontôleur. Un **microcontôleur** est une unité de traitement d'information, de type microprocesseur, à laquelle on a ajouté des périphériques internes. Le type de processeur et de périphériques, ainsi que le nombre de ces derniers, varient d'un microcontrôleur à l'autre. Nous étudierons plus spécifiquement dans ce document les caractéristiques du *PIC16F84* (voir section suivante).

Cette configuration permet de réaliser des fonctions complexes sans devoir ajouter de composants externes autres que l'alimentation et l'oscillateur (bien entendu obligatoires!). Il est néanmoins évident qu'ajouter des circuits externes n'est en aucun cas interdit et sera même fréquemment nécessaire (LED, affichage LCD, PC...).

## 1.2 Les PICs

Les PICs sont un type de microcontrôleurs commercialisés par *Microchip*. *Microchip* fournit également gratuitement sur son site le programme *MPLAB*, dont nous reparlerons à la section 3, software indispensable à la programmation des PICs.

Actuellement, on distingue 3 grandes familles de PICs, selon la taille des mots d'instruction (cf section 2) utilisée: *Base-Line* (mots de 12 bits), *Mid-Range* (14 bits) et *High-End* (16 bits). Tous les PICs dont nous parlerons (dont le *16F84*) sont du type *Mid-Range*.

Chaque type de PIC est identifié de façon unique par un numéro. Celui-ci est composé de 3 (éventuellement 4) parties:

1. les 2 premiers chiffres indiquent la catégorie. 16 indique un *PIC Mid-Range*. Ces deux chiffres sont parfois suivis d'un *L* indiquant que le PIC peut fonctionner avec une plage de tension beaucoup plus tolérante;
2. la (les) lettre(s) suivante(s) indique(nt) le type de mémoire programme:
   - C: mémoire EPROM ou EEPROM (plus rare);
   - CR: mémoire ROM;
   - F: mémoire FLASH.
3. les chiffres qui identifient précisément le PIC (84 dans notre cas);
4. éventuellement, on peut encore trouver sur certaines puces le suffixe *-XX* où *XX* représente la fréquence d'horloge maximale permise par le PIC. Nous tenons ici à insister qu'il s'agit d'une fréquence **maximale** d'utilisation et que le PIC fonctionnera correctement à n'importe quelle fréquence inférieure à celle-ci.

Rappelons que seules les mémoires FLASH et EEPROM sont reprogrammables électriquement. Une ROM n'est pas reprogrammable; tandis qu'une mémoire EPROM s'efface uniquement par exposition aux UV, ce qui nécessite la présence d'une fenêtre transparente sur la puce (en option

sur ce type de puces).

Le *PIC16F84* est donc un microcontrôleur (8 bits) de chez *Microchip* du type *Mid-Range*, avec mémoire programme de type FLASH et dont le numéro d'identification est 84.

# 2 Architecture

Au niveau du processeur, on distingue deux grandes familles d'architecture. Les processeurs **RISC** (pour *Reduced Instruction Set Computer*) possèdent un jeu d'instructions (simples) réduit. Le principal avantage est une rapidité de traitement accrue puisque le décodage de chaque instruction est plus court (théoriquement un seul temps d'horloge). Les processeurs **CISC** (*Complex Instruction Set Computer*) quant à eux contiennent plus d'instructions, plus complexes et plus puissantes, ce qui réduit d'autant leur vitesse de traitement.

Au niveau de la division entre code programme et données, on distingue également deux familles: l'architecture de type **Von Neumann** utilise une mémoire unique pour stocker les programmes et les données; tandis que dans une architecture de type **Harvard** ceux-ci sont situés dans des mémoires physiquement séparées. L'avantage de cette seconde approche est qu'elle permet d'adresser les deux types de mémoire dans le même cycle d'horloge. La vitesse de traitement est donc améliorée aux dépends de l'espace physique et du nombre de composants internes.

Dans le cas du *16F84*, nous sommes en présence d'une architecture *pseudo-RISC* de type *Harvard*. Que signifie architecture pseudo-RISC? En fait, le *16F84* ne possède que 35 instructions, mais comme l'horloge fournie au PIC est prédivisée par 4, chaque instruction prend en réalité 4 temps d'horloge pour être exécutée (sauf les instructions de saut qui nécessitent 8 temps d'horloge). C'est donc cette combinaison d'un nombre réduit d'instructions, exécutée chacune en plus d'un temps d'horloge, qui nous conduit à définir ce PIC comme *pseudo-RISC*.

La figure 1 montre l'architecture interne du *16F84*. On y voit clairement les deux mémoires séparées: FLASH pour le programme, EEPROM et RAM pour les données.

Nous allons maintenant présenter plus en détails les principaux composants du *16F84* représentés sur cette figure 1.

## 2.1 Mémoire

Comme nous l'avons mentionné, l'architecture est de type *Harvard*: il y a deux zones de mémoires indépendantes dans le circuit, la mémoire de programme (FLASH) et les mémoires de données (RAM et EEPROM).

### 2.1.1 Mémoire de programme

Il s'agit d'une mémoire de type FLASH (elle peut donc être reprogrammée électriquement) constituée de 1K mots de 14 bits (cf type *Mid-Range*). Les fichiers programmes auront donc une taille de 2Kbytes puisqu'il faut 2 bytes pour coder 14 bits [1]. Comme une instruction est codée sur un mot, il est possible d'écrire des programmes contenant jusqu'à 1K instructions, ce qui offre

1. Pour rappel, 1 byte=1 octets=8 bits.

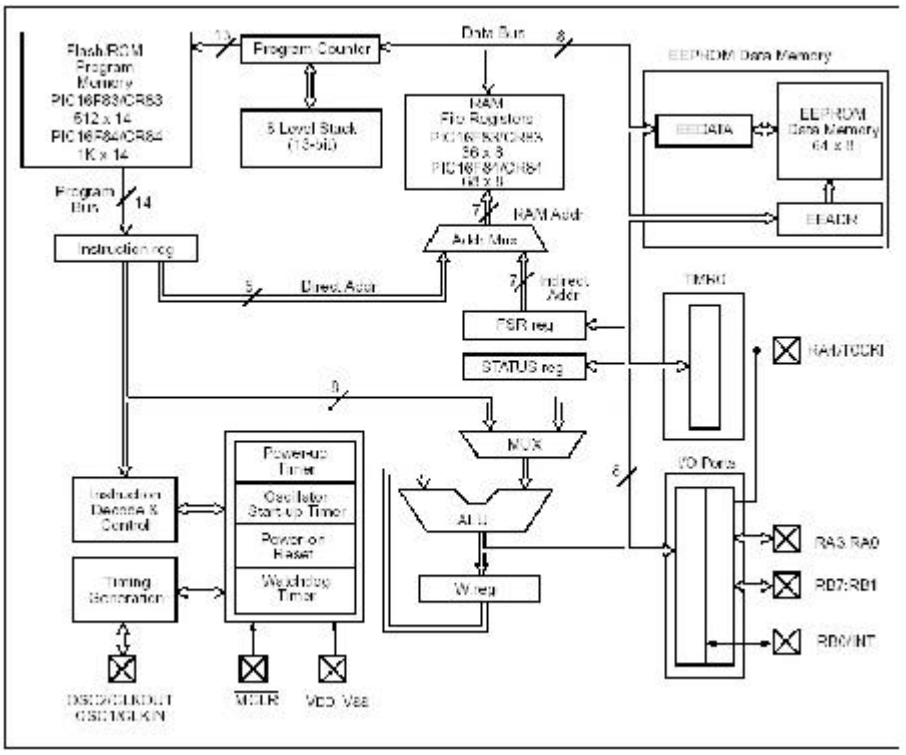


FIG. 1 – *Architecture interne du* 16F84

bien des possibilités.

Cette mémoire est adressée par le compteur de programme (PC, 13 bits donc 8192 mots adressables...), inaccessible directement par le programmeur. Notons que ce compteur pointe toujours sur l'instruction suivante à exécuter.

### 2.1.2 Mémoire RAM

Il s'agit de la mémoire dynamique utilisée continuellement dans les programmes. Cette mémoire étant de type RAM, toutes les données qui y sont stockées seront perdues en cas de coupure de courant ou d'arrêt de l'alimentation.

La structure de la RAM du *16F84* est donnée à la figure 2. On voit qu'elle contient 256 octets au total, divisés en deux parties appelées banques (0, adresses 0x00 à 0x7F et 1, adresses 0x80 à 0xFF). Ces banques sont à nouveau divisées en différents types de registres:

- Les *registres à fonction spéciale*. Suivant la valeur de leur contenu, le PIC entre dans un mode de fonctionnement spécial ou active une fonction particulière. Nous reparlerons de la plupart d'entre eux à la section 2.3.
  Notons également que certains registres sont identiques dans les 2 banques (PCLATH par exemple). Y accéder depuis la banque 0 ou la banque 1 ne fera pas de différence.

- 68 octets libres dans la banque 0, disponibles pour le programmeur (adresses 0x0C à 0x4F).
- Les 68 octets correspondants dans la banque 1 (adresses 0x8C à 0xCF) sont *mapped in bank 0*, ce qui indique qu'ils correspondent en fait à la même case mémoire.
- Finalement, les derniers registres de chaque banque sont des emplacements non utilisés (et non utilisables). Il est également impossible d'accéder à l'adresse 0x00.

Cette mémoire est adressée de manière directe ou indirecte. Ces concepts d'adressage sont supposés connus (sinon, se référer à [5]). Les particularités liées à l'utilisation de l'adressage indirect seront exposées à la section 2.3.

**Remarque**
L'intérêt d'utiliser deux banques au lieu d'une seule apparaîtra clairement dans le chapitre sur les instructions. En effet, sur 14 bits, il s'avère que pour coder toutes les instructions, il faut utiliser au maximum 7 bits, ce qui n'en laisse que 7 pour adresser la RAM. La solution a alors été choisie de diviser la RAM en 2 banques, adressables chacune entièrement à l'aide de 7 bits, le choix de la banque étant déterminé par le bit `RP0` du registre `STATUS`.

### 2.1.3 Mémoire EEPROM

Le *16F84* dispose également d'une mémoire EEPROM périphérique de 64 octets (de données!). Comme il s'agit d'une mémoire de type EEPROM, elle est programmable électriquement. En outre, ce type de mémoire est semi-permanente [2]: les données sont conservées en cas de coupure de courant.

L'accès en EEPROM (surtout en écriture) nécessite une procédure spéciale que nous illustrerons à la section 6. En effet, il ne s'agit pas de mémoire RAM mais d'une ROM de type spécial (ce qui lui permet de conserver des données sans tension appliquée). Il est donc plus rapide d'y lire que d'y écrire. C'est pourquoi on l'utilisera principalement pour conserver les données non dynamiques de notre programme. On essayera aussi de limiter au maximum les accès en EEPROM car ils sont très lents. Le nombre de cycles d'écriture en EEPROM étant limité par la durée de vie de celle-ci (environ 10 millions de cycles), il faudra veiller à ne pas écrire un programme qui écrit de façon continue dans l'EEPROM sous peine d'user le PIC en moins de 28h!!!

Notons dès à présent que cet espace mémoire commence à l'adresse 0x2100, c'est-à-dire hors de l'espace d'adressage normal puisque l'adresse maximale sur 8Kmots est 0x1FFF. En fait, il existe toute une zone hors de cet espace normal où sont stockées des informations comme les paramètres du registre *_CONFIG* ou l'identification du composant.

## 2.2 Pile matérielle

Limitée à 8 niveaux. Il est donc interdit d'avoir plus de 8 routines ou sous-routines imbriquées. Cette pile n'est pas accessible par le programmeur, car non présente dans l'espace d'adres-

2. On utilise ici le terme semi-permanente par opposition aux mémoires de type ROM qui sont des mémoires permanentes. En effet, dans ces mémoires, les données sont également conservées en cas de coupure de courant, mais il n'existe pas de possibilités de les effacer.

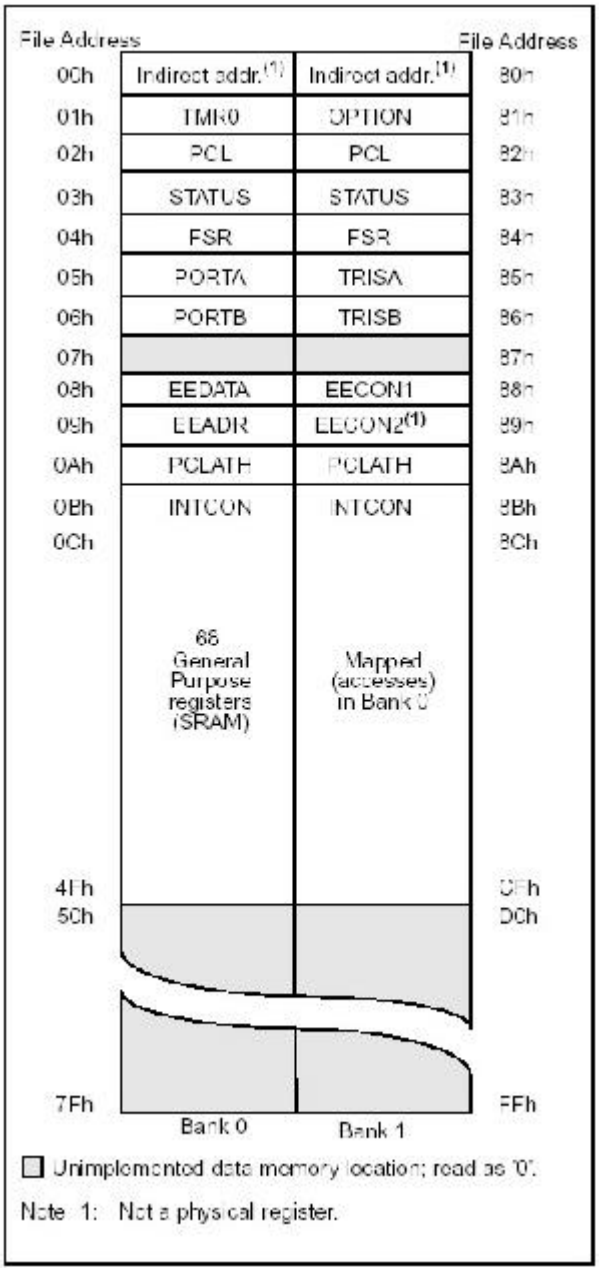


FIG. 2 – *Structure de la RAM du* 16F84

sage du PIC. Il faudra donc uniquement être attentif à ne pas la faire déborder lors de l'utilisation de ***CALL***, ***RETURN***, ***RETFIE***, ***RETLW*** ou de l'appel de la routine de traitement des interruptions (voir sections 4 et 5). Notons qu'il est toujours utile (voire nécessaire!) de vérifier ce point lors de la programmation d'un *PIC16F84*.

## 2.3 Registres

Nous allons maintenant aborder les divers registres spéciaux présents dans le *16F84*. A l'exception du registre *W*, tous les registres que nous traiterons sont situés dans la mémoire RAM (voir figure 2).

### 2.3.1 Registre *W*

Registre de travail utilisé par le PIC pour effectuer la plupart des calculs. La destination d'un résultat peut en fait être soit un emplacement RAM (f) ou ce registre de travail (W).

### 2.3.2 Registre *TMR0*

Ce registre contient la valeur actuelle du Timer0. Nous reparlerons du Timer0 à la section 2.4.

### 2.3.3 Registre *OPTION*

Il s'agit d'un registre dont chaque bit (ou flag) a une signification particulière. Du LSB [3] au MSB [4], on a:

- **PS0, PS1, PS2**: les 3 bits de poids faible déterminent la valeur de prédivison du Timer0 ou du watchdog (8 valeurs possibles);
- **PSA**: détermine si la prédivision s'effectue sur le Timer0 ou le Watchdog;
- **TOSE**: indique si le Timer0 est incrémenté sur flanc montant ou descendant;
- **TOCS**: détermine le fonctionnement du Timer0. Celui-ci sera incrémenté par les variations de l'horloge interne lorsque TOCS=0. Sinon, il sera incrémenté via les impulsions reçues sur une de ses broches (RA4 en l'occurrence, voir ci-dessous);
- **INTDEG**: indique le flanc de déclenchement des interruptions sur RB0 si elles sont activées: sur flanc montant si INTDEG=1, sur flanc descendant sinon.
- **RBPU**: bit actif bas, permet de placer une résistance tire-haut sur toutes les broches de PORTB.

Nous reparlerons du Timer0 à la section 2.4.
Le tableau 1 montre les différentes combinaisons possibles de PS0, PS1, PS2 et PSA , ainsi que les valeurs de prédivision correspondantes.

### 2.3.4 Registre *PCL*

Comme les registres ne font que 8 bits, deux registres sont nécessaires pour accéder à l'entièreté des adresses programme. Ce registre contient donc les 8 LSB du PC. Il est accessible en lecture et en écriture (au contraire du PC).

3. *Least Significant Bit*, bit de poids faible
4. *Most Significant Bit*, bit de poids fort

| PSA | PS2 | PS1 | PS0 | /tmr0 | /WD | Temps tmr0 | Temps typique Watchdog (minimal) |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 2 | 1 | 512 $\mu$s | 18 ms (7ms) |
| 0 | 0 | 0 | 1 | 4 | 1 | 1024 $\mu$s | 18 ms (7ms) |
| 0 | 0 | 1 | 0 | 8 | 1 | 2048 $\mu$s | 18 ms (7ms) |
| 0 | 0 | 1 | 1 | 16 | 1 | 4096 $\mu$s | 18 ms (7ms) |
| 0 | 1 | 0 | 0 | 32 | 1 | 8192 $\mu$s | 18 ms (7ms) |
| 0 | 1 | 0 | 1 | 64 | 1 | 16384 $\mu$s | 18 ms (7ms) |
| 0 | 1 | 1 | 0 | 128 | 1 | 32768 $\mu$s | 18 ms (7ms) |
| 0 | 1 | 1 | 1 | 256 | 1 | 65536 $\mu$s | 18 ms (7ms) |
| 1 | 0 | 0 | 0 | 1 | 1 | 256 $\mu$s | 18 ms (7ms) |
| 1 | 0 | 0 | 1 | 1 | 2 | 256 $\mu$s | 36 ms (14ms) |
| 1 | 0 | 1 | 0 | 1 | 4 | 256 $\mu$s | 72 ms (28ms) |
| 1 | 0 | 1 | 1 | 1 | 8 | 256 $\mu$s | 144 ms (56ms) |
| 1 | 1 | 0 | 0 | 1 | 16 | 256 $\mu$s | 288 ms (112 ms) |
| 1 | 1 | 0 | 1 | 1 | 32 | 256 $\mu$s | 576 ms (224ms) |
| 1 | 1 | 1 | 0 | 1 | 64 | 256 $\mu$s | 1.152 sec (448ms) |
| 1 | 1 | 1 | 1 | 1 | 128 | 256 $\mu$s | 2.304 sec (996ms) |

TAB. 1 – *Valeur possibles du prédiviseur en fonction de PS0, PS1, PS2 et PSA*

### 2.3.5 Registre *PCLATH*

Ce registre accessible en lecture et en écriture, contient 5 bits utiles.

- lors d'un saut, le contenu du PC est chargé directement avec les 11 bits de destination, complétés par les bits 3 et 4 de *PCLATH* qu'il convient donc de positionner;
- lors d'une modification de *PCL*, PC est complété par les 5 bits de *PCLATH*

### 2.3.6 Registre *STATUS*

Il s'agit d'un registre dont chaque bit (ou flag) a une signification particulière. Du LSB au MSB, on a:

1. **C (Carry)**: 9$^e$ bit d'une opération (report). Il vaut 1 si le résultat est supérieur à 255 (0xFF).
2. **DC (Digit Carry)**: utilisé lorsqu'on travaille en BCD[5]. Il indique dans ce cas un report du bit 3 vers le bit 4.
3. **Z (Zero)**: positionné à 1 si le résultat d'une opération est nul.
4. **PD (Power Down)**: indique quel événement a entraîné le dernier arrêt du PIC (actif à l'état bas);
5. **TO (Time-Out Bit)**: indique que la mise en service suit un arrêt provoqué par un dépassement de temps ou une mise en sommeil (actif à l'état bas). C'est PD qui permet de faire la distinction.

5. *Binary Coded Decimal*, technique d'encodage des nombres décimaux où chaque chiffre est remplacé par son équivalent binaire sur 4 bits.

6. **RP0 (Register Bank Select0)**: permet d'indiquer dans quelle banque de RAM on travaille.
7. **RP1 (Register Bank Select1)**: permet la sélection des banques 2 et 3 pour les PIC qui le permettent. Inutilisé dans le *16F84*. Doit être mis à 0 pour assurer la portabilité des programmes vers des PIC comportant plus de 2 banques.
8. **IRP (Indirect RP)**: permet de décider quelle banque on adresse en adressage indirect.

Précisons que ces flags ne changent leur valeur que si ce changement est une caractéristique de l'instruction exécutée (voir section 4).
Ce registre est principalement utilisé pour des tests sur le résultat de certaines opérations.

### 2.3.7 Registre *FSR/INDF*

Ces registres sont utilisés en adressage indirect. Le registre *INDF* n'existe pas réellement (il s'agit en fait du registre situé à l'adresse 0x00 de la mémoire RAM) mais est utilisé comme procédé d'accès à *FSR*.
Le procédé est le suivant: on fait pointer *FSR* sur la zone mémoire désirée, et on accède au contenu de celle-ci en lisant le contenu de *INDF*.

### 2.3.8 Registres *PORTA/PORTB*

Ces registres sont liés aux entrées/sorties. Nous en reparlerons à la section 2.5.

### 2.3.9 Registre *TRISA/TRISB*

Ces registres sont liés aux entrées/sorties. Nous en reparlerons à la section 2.5.

### 2.3.10 Registre *EEDATA*

Ce registre, ainsi que les 3 suivants, est lié à la gestion de la mémoire EEPROM.
Dans ce registre particulier transite la donnée à écrire vers/lue en provenance de l'EEPROM.

### 2.3.11 Registre *EEADR*

Ce registre contient l'adresse concernée par cette opération d'écriture/lecture en EERPOM.

### 2.3.12 Registre *EECONI*

Registre de bits. Les 5 LSB sont (ordre croissant):

1. **RD**: Démarrage du cycle de lecture. Remis à 0 automatiquement à la fin du cycle.
2. **WR**: Démarrage du cycle d'écriture. Remis à 0 automatiquement à la fin du cycle.
3. **WREN**: autorisation de démarrage du cycle d'écriture.
4. **WRERR**: bit d'erreur, passe à 1 si une opération d'écriture en EEPROM a été interrompue.
5. **EEIF**: signale la fin d'une opération d'écriture en EEPROM (flag).

Les 3 autres bits ne sont pas utilisés.

```
movlw   0x55            ; charger 0x55
movwf   EECON2          ; envoyer commande
movlw   0xAA            ; charger 0xAA
movwf   EECON2          ; envoyer commande
bsf     EECON1 , WR     ; lancer cycle d'écriture
```

FIG. 3 – *Séquence imposée par Microchip pour l'écriture en EEPROM*

### 2.3.13 Registre *EECON2*

Il s'agit d'un registre "fantôme" (c'est-à-dire qui n'existe pas). Il sert uniquement à envoyer des commandes au PIC. Microchip impose d'insérer les lignes reprises figure 3 dans toute routine d'écriture en EEPROM juste après l'instruction d'autorisation d'accès en écriture.

### 2.3.14 Registre *INTCON*

Le registre *INTCON*[6] est intimement lié à la gestion des interruptions (cf section 5). C'est un registre de bits. Du LSB au MSB, on a:

1. **RBIF**: signale qu'une des entrées RB4 à RB7 a été modifiée.
2. **INTF**: signale une transition sur la broche RB0 dans le sens déterminé par INTEDG du registre *OPTION*.
3. **TOIF**: signale le débordement du Timer0.
4. **RBIE**: valide l'interruption sur RB4 à RB7.
5. **INTE**: valide l'interruption sur RB0.
6. **TOIE**: valide l'interruption générée par le débordement du Timer0.
7. **EEIE**: valide l'interruption de fin d'écriture en EEPROM.
8. **GIE**: valide toutes les interruptions.

On peut remarquer que les bits dont le nom se termine par E (pour Enable) sont des bits de validation tandis que ceux qui se terminent par F (pour FLAG) sont des indicateurs.

## 2.4 Timer 0

Le Timer0 est en fait un compteur 8 bits, c'est-à-dire un registre qui s'incrémente de 0x00 à 0xFF puis recommence. Selon le mode, il peut s'incrémenter lors de deux événements différents:

- **Mode compteur**: impulsion sur la broche RA4/TOKI
- **Mode "timer"**: transition d'horloge du PIC

La sélection du mode s'effectue via le bit 5 (TOCS) du registre *OPTION* (1 pour mode compteur). En mode compteur, il faut également indiquer sur quel type de transition on compte (bit 4 -TOSE- d'*OPTION*, 1 pour transition descendante).

6. INTerrpution CONtrol

Le Timer0 est généralement utilisé de 3 manières différentes.

- **Mode de lecture simple**: on lit la valeur du Timer0 contenue dans le registre *TMR0*.
- **Mode de scrutation du flag**: on regarde le flag TOIF d'*INTCON* pour voir si le Timer0 a débordé.
- **Mode d'interruption**: on peut choisir d'activer une interruption à chaque débordement du timer0.

Pour une description plus complète du flag de débordement et des interruptions, voir section 5.

Comme nous l'avons indiqué ci-dessus, il est possible de choisir via les bits PS1 et PS2 de *OPTION*, un prédiviseur qui diminue la fréquence d'horloge perçue par le Timer0 d'un facteur choisi. Ceci permet notamment de déclencher des interruptions avec une période au choix. Notons que ce prédiviseur unique sera affecté soit au Watchdog, soit au Timer0 via le bit PSA de *OPTION*.

## 2.5 Entrées/sorties

La figure 4 montre les 18 broches du *16F84* et leur attribution. On peut y observer:

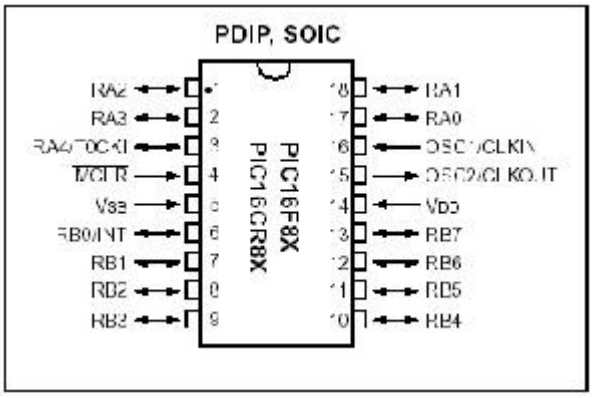


FIG. 4 – *Connexions du* 16F84 *avec le monde extérieur*

- **MCLR**: Master clear Reset (Active bas). Permet d'effectuer un reset;
- **Vdd**: entrée de la source d'alimentation (5V);
- **Vss**: entrée pour la connexion à la masse;
- **OSCIN/OSCOUT**: connexion des 2 bornes de l'oscillateur externe;
- **RA (0-4)**: ces broches constituent le moyen de communication avec le monde extérieur. Les 5 LSB du registre *TRISA* configure ces broches en entrée (si le bit correspondant dans *TRISA* vaut 1) ou en sortie.
  Lorsqu'une de ces broches est configurée en sortie, c'est la valeur correspondante dans le registre *PORTA* qui apparaîtra sur cette broche.
  Une remarque de type électronique s'impose ici: les broches RA0 à RA3 peuvent être

placées en niveau haut ou bas (montage push-pull). Elles possèdent également une diode de protection vers le 0V et le 5V. Elles peuvent donc recevoir en entrée ou sortir des niveaux de 0 ou 5V. RA4 par contre, ne possède qu'une diode vers la masse. En sortie, elle est de type drain ouvert et en entrée elle comporte un trigger de Schmitt. Elle ne peut donc pas envoyer du 5V en sortie. Le fonctionnement des ports étant toujours du type: *lecture complète->modification->écriture complète*, il faudra se méfier des modifications non désirées de RA4 liées à une électronique extérieure...

- **RB (0-7)**: ces broches ont les mêmes fonctions que les broches RA, à trois différences près. Premièrement, le comportement de RB4 est identique à celui des autres broches, au contraire de RA4. Ensuite, comme nous l'avons mentionné section 2.3, il est possible de placer une résistance de rappel au 5V sur ces broches via le bit de poids fort du registre *OPTION*. Enfin, il est possible de configurer RB0 comme source d'interruption comme nous l'expliquerons à la section 5.
  Les registres *PORTB* et *TRISB* agissent sur ces broches de la même façon que PORTA et *TRISA* sur les broches RA.

Le *16F84* possède encore d'autres périphériques et fonctionnalités plus avancées (mode sleep...) dont nous ne parlerons pas ici. Nous renvoyons à nouveau le lecteur intéressé aux meilleures sources en la matière: [3] et [1].

# 3 Programmation du PIC

Afin de faciliter la compréhension et l'écriture de programmes, les informaticiens et les électroniciens ont développé un langage qui permet d'éviter l'écriture binaire du programme: le langage d'assemblage (ou assembleur [7]). La conversion vers la version binaire du programme (ou OPCODE) sera réalisée à l'aide d'un programme d'assemblage. Il est cependant toujours possible au programmeur d'écrire son programme directement en binaire ou en hexadécimal.

Sur son site, *Microchip* met gratuitement à disposition le logiciel MPLAB, qui contient un gestionnaire de projet, un éditeur de code et un programme d'assemblage. C'est ce programme que nous utiliserons comme référence dans cette section.

## 3.1 Préparation du projet

MPLAB travaille avec le concept de projet. Pour commencer, il convient donc de créer un nouveau projet (menu *project->new project*). Il est alors possible de choisir le nom du projet (par défaut, le fichier *.hex* généré portera ce nom) ainsi que le répertoire où ils era situé. A ce moment, divers fichiers sont crées dans le répertoire indiqué, notamment un *nom_du_projet.mcp* pour le projet, et un *nom_du_projet.mcw* pour l'espace de travail. Ce dernier est immédiatement affiché dans une fenêtre sur l'écran. La liste *Source Files* contient les différents fichiers (ou noeuds) utilisés dans le projet. Pour en ajouter, il suffit de cliquer à l'aide du bouton droit sur *Source Files* puis de choisir *Add files...*.

Il existe différentes façons de construire le fichier *.hex* à partir des fichiers en assembleur (fichiers *.asm*) du projet. Ces options de compilation sont réglables via le menu *Project->Build Options...* Le choix de *Project* ou d'un fichier *.asm* définit à quelle partie du projet elles s'appliquent.

Pour l'ensemble du projet, il est possible d'indiquer

- *Include Path*: chemin des fichiers de définitions (fichiers .INC), qui contiennent toutes les constantes relatives au PIC utilisé dans le projet;
- *Library Path*: chemin des librairies utilisées;
- *Linker Script Path*: chemin des fichiers de lien utilisés.

Pour l'ensemble du projet ou pour chaque fichier individuellement *Defaut radix* permet de choisir comment seront interprétés les nombres sans préfixe.

Notons enfin qu'il est indispensable d'indiquer à MPLAB sur quel type de microcontrôleur on travaille via le menu *Configure->Select Device...*

7. Pour être tout à fait précis, il existe une confusion dans le langage courant pour le terme ASSEMBLEUR, utilisé aussi bien pour le langage de programmation utilisé pour écrire le code (langage d'assemblage) et le programme utilisé pour traduire ce langage en binaire (programme d'assemblage). Nous utiliserons ici le terme assembleur dans le sens *langage d'assemblage*

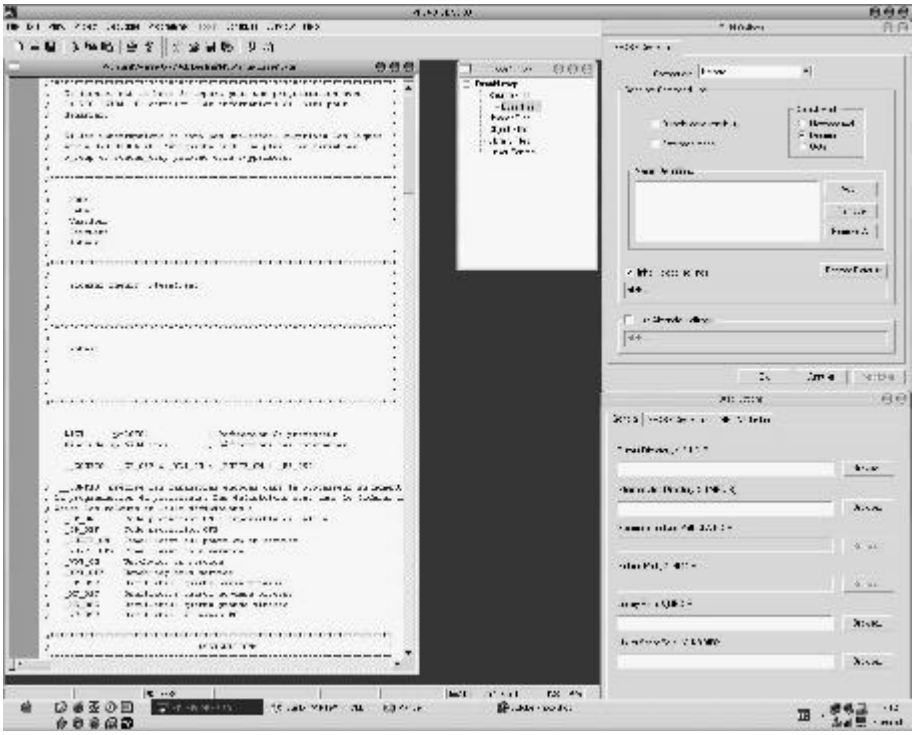

FIG. 5 – *Editeur de projet de MPLAB (projet* Essai1*): fichier* Essai1.asm*, espace de travail, fenêtres d'options*

**Remarque**
Certaines versions de MPLAB sont allergiques aux caractères étendus/accentués dans le noms de fichiers. Pour assurer une portabilité totale du programme, il est conseillé d'éviter ce type de caractères dans les noms de fichiers.

## 3.2 Structure d'un fichier *.asm*

Il convient de distinguer 3 types de commandes en assembleur, selon la façon dont elles seront traitées par le programme d'assemblage.

### 3.2.1 Les commentaires

Ces zones du code seront entièrement ignorées lors de l'assemblage. Comme leur nom l'indique, elles servent uniquement à commenter et expliciter le code du programme. Leur utilisation est fortement recommandée; aussi bien pour faciliter la compréhension du code par d'autres utilisateurs que par le programmeur lui-même.

Sur une ligne, tous ce qui suit le symbole **;** est considéré comme commentaire.

### 3.2.2 Les directives

Ces lignes du code servent uniquement à indiquer à l'assembleur comment travailler. Elles ne font pas partie du programme et ne seront donc pas traduite en OPCODE.

Les directives les plus courantes sont illustrées dans l'exemple de la figure 6.

```
    LIST      p=16F84           ; Définition de processeur
    #include <p16F84.inc>       ; Définitions de variables

    __CONFIG   _CP_ON & _WDT_ON & _PWRTE_ON & _HS_OSC

; '__CONFIG' précise les paramètres encodés dans le processeur au moment de
; la programmation du processeur. 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 vitesse
;   _XT_OSC     Oscillateur quartz moyenne vitesse
;   _HS_OSC     Oscillateur quartz grande vitesse
;   _RC_OSC     Oscillateur à réseau RC

;*****************************************************************************
;                               ASSIGNATIONS                                  *
;*****************************************************************************

OPTIONVAL   EQU H'0040' ; Valeur registre option
                        ; Résistance pull-up ON
                        ; Interrupt flanc montant RB0
                        ; Préscaler timer à 2 (exemple)

INTERMASK   EQU H'0090' ; Masque d'interruption
                        ; Interruptions sur RB0 (exemple)


;*****************************************************************************
;                                 DEFINE                                      *
;*****************************************************************************

#DEFINE LED1    PORTB,1     ; LED de sortie 1 (exemple)

;*****************************************************************************
;                                 MACRO                                       *
;*****************************************************************************

LIREIN macro            (exemple)
        comf PORTB,0
        andlw   1
```

```
    endm
;

;*****************************************************************
;                    DECLARATIONS DE VARIABLES                   *
;*****************************************************************

    CBLOCK 0x00C        ; début de la zone variables
    w_temp :1           ; Zone de 1 byte
    status_temp : 1     ; zone de 1 byte
    ENDC                ; Fin de la zone

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

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

FIG. 6 – *Exemple de structure de fichier* .asm

- ***LIST*** indique quel type de processeur est utilisé dans ce programme;
- ***#include*** indique le(s) fichier(s) où sont regroupées les assignations;
- ***CONFIG*** est une directive extrêmement importante: elle contient la valeur qui fixe le fonctionnement du PIC. La valeur écrite ici sera intégrée au fichier *.hex* et sera encodée automatiquement à l'adresse spécifique du PIC. Cette directive permet de choisir si on désire activer la protection de code (impossible de le relire si activée), le watchdog et le timer reset sur power on (cf section 2.4); et d'indiquer le type d'oscillateur utilisé.
- les ***ASSIGNATIONS*** permettent de remplacer un nombre par une chaîne de caractères afin de faciliter les manipulations.
  Syntaxe:
  ```
  assignations  EQU   nombre_a_remplacer
  ```
- les ***DEFINITIONS*** ont la même fonction que les assignations mais pour remplacer un texte plus complexe. Syntaxe:
  ```
  #DEFINE   definition   chaîne_a_substituer
  ```
- les ***MACROS*** ont la même fonction mais pour remplacer des morceaux de code.
  Syntaxe:
  ```
  nom_de_macro MACRO
      code_de_macro
  ```

```
endm
```

- *CBLOCK*, suivi d'une adresse, permet de définir le début d'une zone de variable.
  Syntaxe:

```
CBLOCK adresse_de_debut_de_zone
    nom_de_variable : taille_de_zone(en bytes)
    ...
    ENDC
```

- *ORG*, suivi d'une adresse, précise à quelle adresse les instructions qui suivent seront placées dans le PIC. Notons qu'après un reset ou une mise sous tension, le PIC redémarre toujours à l'adresse 0x00. Le début du programme doit donc toujours se trouver là.
- *END* précise l'endroit où doit s'arrêter l'assemblage du programme. C'est la seule directive OBLIGATOIRE dans tous les programmes. Tout ce qui suit cette directive sera simplement ignoré à l'assemblage.

### 3.2.3 Les instructions

Il s'agit de la partie du programme réellement traduite en OPCODE par le programme d'assemblage. La syntaxe de ces commandes doit respecter l'ordre suivant:

- étiquette (facultative): il s'agit d'un repère pour le programme qui permet d'éviter le calcul d'adresses lors des sauts;
- espace(s)/tabulation(s): il est en effet interdit d'écrie autre chose dans la première colonne qu'une étiquette où un ; (commentaire);
- mnémonique: symbole littéral remplaçant le code binaire d'une instruction en assembleur;
- tabulation(s)/espace(s);
- opérande/valeur;
- virgule éventuelle de séparation;
- bit de destination W, f ou numéro du bit de 0 à 7 si nécessaire selon l'instruction;
- espace(s)/tabulations(s);
- point-virgule (facultatif si pas de commentaire);
- commentaire (facultatif).

Rappelons que toute ligne commençant par un ; est également considérée entièrement comme commentaire.

Un exemple d'instruction simple avec étiquette est donné figure 7. Les instructions sont décrites individuellement de manière détaillée à la section 4.

## 3.3 Assemblage

C'est à cette étape que le programme d'assemblage (MPLAB dans notre cas), transforme le code "assembleur" en code binaire directement exécutable par le PIC (OPCODE). Cette opéra-

```
restorereg
    swapf   status_temp,w   ; swap ancien status, résultat dans w
```

FIG. 7 – *Exemple d'instruction simple avec étiquette et commentaire*

tion est réalisée avec MPLAB en pressant la touche F10.

A chaque compilation, MPLAB construit un rapport contenant différentes informations utiles. En cas d'échec de la compilation, il est indispensable de consulter ce rapport pour déterminer les sources d'erreur.

Deux types d'éléments nous intéressent. Tout d'abord, les **warnings**. Ces messages sont destinés à attirer notre attention mais ne génèrent pas d'impossibilité d'assemblage. Il peut s'agir par exemple d'oubli dans le code que le programme d'assemblage a réussi à combler par lui-même. Comme ils n'empêchent pas l'assemblage dans un premier temps, ils peuvent être ignorés; mais dans la phase de finalisation du programme, il est intéressant d'essayer de trouver leur source et de les supprimer.

On remarquera également des lignes **error**. Ce sont celles qui renseignent sur les événements qui ont empêchés la compilation de se produire correctement; ainsi que sur leurs positions dans le code.

## 3.4 Simulation

MPLAB possède un simulateur/débogeur intégré qui permet de vérifier le fonctionnement du programme (et donc de le débogeur en cas de dysfonctionnement) avant une implémentation sur le PIC. Cette fonctionnalité est indispensable dans le cas des PIC basé sur une mémoire ROM qui n'acceptent qu'une seule et unique programmation.

Pour activer le simulateur de MPLAB, il suffit d'ouvrir le menu *Debugger->Select Tool*) et de sélectionner la case MPLAB-SIM.

Nous pouvons maintenant, à l'aide du menu *View*, choisir les informations que nous souhaitons afficher à l'écran (registres spéciaux via *Special function registers* et variables au choix via *Watch*).

Il est maintenant possible de lancer la simulation à l'aide de la touche <F9>. Il existe différents modes de débogage possibles (voir le menu *debug->run*):

- appuyer sur <F7> fera avancer le programme instruction par instruction;
- appuyer sur <F8> fera également avancer le programme instruction par instruction mais chaque sous-routine sera exécutée comme une instruction unique;

- <F9> permet de lancer le programme rapidement sans visualisation et <F5> de l'arrêter là où il est arrivé. Notons qu'avec cette méthode, il est également possible de définir, à l'aide d'un double clic, des point d'arrêts ("break points") pour la simulation dans le programme (menu *debug*).Si on presse alors <F9>, le programme se déroulera jusqu'à cette ligne;
- la même procédure avec <CTRL><F9> donne un exécution animée, mais plus lente.

Il est possible de donner une valeur particulière à un élément (variable, registre spécial...), en le modifiant simplement dans sa fenêtre d'observation, pour déterminer le comportement du programme dans ce cas de figure particulier.

# 4 Instructions

Avant de présenter chaque instruction, remarquons qu'il est possible de répartir les instructions du *16F84* (codées, rappelons le, sur 14 bits chacune) en 4 grandes catégories:

1. **Instructions "orientées octets"**
   Elles manipulent les données sous forme d'octets.
   Codage:
   - 6 bits pour l'instruction (en effet, 35 instructions nécessitent 6 bits de codage);
   - 1 bit de destination: soit le registre *W*, soit l'opérande (*F* pour File);
   - 7 bits pour l'adresse de l'opérande.

   On voit ici l'utilité du bit *RP0* du registre *STATUS* qui, en indiquant la banque, permet d'avoir un accès complet à la mémoire avec seulement 7 bits d'adresse pour l'opérande dans l'instruction.
2. **Instructions "orientées bits"**
   Elles manipulent directement les bits d'un registre particulier.
   Codage:
   - 4 bits pour l'instruction;
   - 3 bits pour indiquer le numéro du bit (en effet, 8 bits par registre nécessitent 3 bits de codage);
   - 7 bits pour l'opérande.
3. **Instructions générales**
   Elles manipulent des données codées directement dans l'instruction.
   Codage:
   - 6 bits pour l'instruction;
   - 8 bits pour une valeur immédiate (de 0 à 255 donc).
4. **Instructions de saut et d'appel de sous-routines**
   Elles provoquent une rupture dans la séquence de déroulement du programme.
   Codage:
   - 3 bits pour l'instruction;
   - 11 bits pour la destination (amplement suffisant puisqu'il suffit de 10 bits pour coder l'entièreté des adresses de la mémoire programme).

   Ce sont les seules instructions qui nécessitent 8 temps d'horloge au lieu de 4.

Nous allons maintenant décrire brièvement les 35 instructions disponibles sur le *16F84*. Pour une description absolument complète d'une instruction particulière, la meilleure référence est encore une fois le datasheet [1].

## 4.1 Instructions orientées octets

### 4.1.1 ADDWF: ADD W and F

Cette instruction ajoute le *contenu* du registre de travail *W* au *contenu* du registre désigné par *f*. Le résultat est placé dans *d*. Il s'agit ici d'un adressage direct et il ne faut donc pas confondre avec l'instruction ADDLW.

**Syntaxe**
*addwf* f, d; (w) + (f) → (d)

**Bits de statut affectés**
C,DC,Z

### 4.1.2 ANDWF: AND W with F

Cette instruction réalise un "et" logique bit à bit entre le *contenu* de l'emplacement mémoire désigné par *f* et le contenu du registre de travail W. Le résultat sera stocké dans l'emplacement spécifié par *d*. Il s'agit donc d'un adressage direct.

**Syntaxe**
*andwf* f, d; (f) AND (W) → (d)

**Bits de statut affectés**
Z

### 4.1.3 CLRF: CLear F

Cette instruction efface l'emplacement mémoire spécifié par *f*.

**Syntaxe**
*clrf* f; (f)=0

**Bits de statut affectés**
Z: Z vaut toujours 1 après cette opération

### 4.1.4 CLRW: CLear W

Cette instruction efface *W*.

**Syntaxe**
*clrw* f; (w)=0

**Bits de statut affectés**
Z: Z vaut toujours 1 après cette opération

### 4.1.5 COMF: COMplement F

On calcule le complément à 1 de l'emplacement mémoire spécifié par *f* , c.à.d on inverse tous les bits de *(f)*. Le résultat est stocké dans d.

**Syntaxe**

*comf* f, d; NOT (f) → (d)

**Bits de statut affectés**

Z:

### 4.1.6 DECF: DECrement F

*DECF* provoque la décrémentation de l'emplacement mémoire spécifié par *f* et le résultat est stocké dans *d*.

**Syntaxe**

*decf* f, d; (f) −1 → (d)

**Bits de statut affectés**

Z

### 4.1.7 DECFSZ: DECrement File, Skip if Z

Cette instruction décrémente le contenu de l'emplacement mémoire *f*, stocke le résultat dans l'emplacement de destination *d* et saute si le résultat vaut 0.

**Syntaxe**

*decfsz* f, d; *quad* (f)-1 → (d). Saut si (d)=0

**Bits de statut affectés**

aucun

### 4.1.8 INCF: INCrement F

Cette instruction provoque l'incrémentation de l'emplacement spécifié par *f* et le résultat est stocké dans *d* qui peut valoir soit *f* ou *W*. Dans le cas où d vaut *f* le résultat est stocké dans l'emplacement de mémoire mentionné ci-dessus. Si par contre d vaut *W*, le résultat est laissé dans l'accumulateur est *f* est laissé inchangé.

**Syntaxe**

*incf* f, d

**Bits de statut affectés**

Z

### 4.1.9 INCFSZ: INCrement F, Skip if Z

Cette instruction incrémente le contenu de l'emplacement mémoire *f*, stocke le résultat dans l'emplacement de destination *d* et saute si le résultat vaut 0.

**Syntaxe**

*decfsz* f,d; *quad* (f)+1 → (d). Saut si (d)=0

**Bits de statut affectés**

aucun

### 4.1.10 IORWF: Inclusive OR W with File

Cette opération effectue un "ou" logique bit à bit entre le registre de travail *W* et le contenu de l'emplacement mémoire spécifié par *f*. Le résultat sera stocké dans l'emplacement spécifié par *d*. C'est un adressage direct.

**Syntaxe**

*iorwf* f, d; (w) OR (f) → (d)

**Bits de statut affectés**

Z

### 4.1.11 MOVF: MOVe f

Cette instruction charge le *contenu* de l'emplacement spécifié par *f* dans la destination spécifiée par d.

**Syntaxe**

*movf* f, d; (f) →(d)

**Bits de statut affectés**

Z

### 4.1.12 MOVWF: MOVe W to F

L'instruction en question permet de sauvegarder le contenu du registre de travail *W* dans un emplacement de mémoire spécifié par *f*.

**Syntaxe**

*movwf* f; (W) → (f)

**Bits de statut affectés**

aucun

### 4.1.13 NOP: No Operation

Cette instruction ne fait 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 ait pu se faire, par exemple.

**Syntaxe**

*nop* f;

**Bits de statut affectés**

aucun

### 4.1.14 RLF: Rotate Left through Carry

On réalise une rotation vers la gauche à travers le carry *C* du contenu de l'emplacement mémoire spécifié par *f*. L'opération de rotation s'effectue de la manière 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 mémorisé dans la première étape. Il s'agit donc d'une rotation sur 9 bits. Le PIC ne possède pas d'opération de décalage. Il faut donc faire une rotation avec le carry préalablement initialisé à 0 pour réaliser un décalage (vers la gauche dans ce cas-ci). Le résultat est stocké dans *d*.

**Syntaxe**

*rrf* f, d; (f) rotation à gauche avec *C* → (d)

**Bits de statut affectés**

C

### 4.1.15 RRF: Rotate Right through Carry

On réalise une rotation vers la droite à travers le carry *C* du contenu de l'emplacement mémoire spécifié par *f*. Le détails de l'opération RRF est le même que pour RLF sauf que tout est déplacé vers la droite.

**Syntaxe**

*rrf* f, d; (f) rotation à gauche avec *C* → (d)

**Bits de statut affectés**

C

### 4.1.16 SUBWF: SUBstract W from F

L'instruction soustrait *W* du *contenu* de l'emplacement mémoire désigné par *f*. Il s'agit donc d'un adressage direct.

**Syntaxe**

*subwf* k; (f)-(W) → (d)

**Bits de statut affectés**
C,DC,Z

### 4.1.17 SWAPF: SWAP nibbles in F

Cette instruction inverse le demi-octet de poids fort avec le demi-octet de poids faible du contenu de l'emplacement mémoire *f* et stocke le résultat dans *d*.

**Syntaxe**
*swapf* f, d;  inversion des b0/b3 de (f) avec b4/b7 → (d)

**Bits de statut affectés**
aucun; Cette particularité sera très utile lors des interruptions.

### 4.1.18 XORWF: eXclusive OR W with File

Cette opération effectue un "ou exclusif" logique bit à bit entre le registre de travail *W* et le contenu de l'emplacement mémoire spécifié par *f*. Le résultat sera stocké dans l'emplacement spécifié par *d*. C'est un adressage direct.

**Syntaxe**
*xorwf* f,d;  (w) XOR (f) → (d)

**Bits de statut affectés**
Z

## 4.2 Instructions orientées bits

### 4.2.1 BCF: Bit Clear File

C'est une instruction qui permet de forcer directement le *bit* numéro *b* d'un emplacement mémoire *f* à 0.

**Syntaxe**
*bsf* f ,b;  le bit n° *b* du contenu de f est positionné à 0

**Bits de statut affectés**
aucun

### 4.2.2 BSF: Bit Set File

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

**Syntaxe**
*bsf* f ,b;  le bit n° *b* du contenu de f est positionné à 1

**Bits de statut affectés**
aucun

### 4.2.3 BTFSC: Bit Test F, Skip if Clear

En traduisant littéralement le mnémonique de l'instruction on obtient: "Teste le bit du contenu de l'emplacement *f* et saute s'il vaut 0." Le saut se rapporte à l'instruction suivante. En d'autres termes, si le bit *b* vaut 0, l'instruction suivant BTFSC n'est pas exécuté. Si le dit bit, par contre, vaut 1, l'instruction suivant BTFSC est exécutée. Il s'agit d'un saut conditionnel puisqu'il n'aura lieu que si la condition est remplie.

**Syntaxe**

| | |
|---|---|
| *btfsc* f,b; | on teste le bit *b* de *f* |
| *instruction suivante*; | Cette instruction n'est pas exécuté si le bit en question vaut 0 |
| *suite du programme*; | Le PIC saute donc vers ici si le bit vaut 0 |

**Bits de statut affectés**
aucun

### 4.2.4 BTFSS: Bit Test F, Skip if Set

Si le bit *b* du contenu de l'emplacement *f* vaut 1, l'instruction suivant BTFSs n'est pas exécuté. Si le dit bit, par contre, vaut 0, l'instruction suivant BTFSS est exécutée. Il s'agit ici aussi d'un saut conditionnel.

**Syntaxe**

| | |
|---|---|
| *btfss* f,b; | on teste le bit *b* de *f* |
| *instruction suivante*; | Cette instruction n'est pas exécuté si le bit en question vaut 1 |
| *suite du programme*; | Le PIC saute donc vers ici si le bit vaut 1 |

**Bits de statut affectés**
aucun

## 4.3 Instructions générales

### 4.3.1 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; $(W) + k \rightarrow (w)$

**Bits de statut affectés**
Z,C,DC

### 4.3.2 ANDLW: AND Literal with W

Cette opération effectue une opération logique "AND" bit à bit entre le *Contenu* de W et la valeur littérale qui suit.

**Syntaxe**

*andlw* k; avec k=octet : (W) AND k → (W)

**Bits de statut affectés**

Z

### 4.3.3 CALL: CALL subroutine

Cette instruction effectue un saut inconditionnel vers une sous-routine référencié par l'*étiquette*. Lors de l'exécution de cette instruction, l'adresse de l'instruction qui, dans le code, suit immédiatement CALL est mémorisé sur la pile. Lorsque la sous-routine est terminée (par RETURN), l'adresse sauvegardée est retirée de la pile et placé dans le PC. Le programme poursuit alors depuis l'endroit d'où il avait fait le saut.

**Syntaxe**

*CALL* étiquette; appel de la sous-routine à l'adresse étiquette

**Bits de statut affectés**

aucun

### 4.3.4 CLRWDT: CLeaR WatchDog Timer

On réalise la miseà 0 du chien de garde.

**Syntaxe**

*clrwdt*; remet le timer du watchdog à 0

**Bits de statut affectés**

aucun

### 4.3.5 GOTO: GO TO address

Cette instruction réalise un saut inconditionnel, encore appelé rupture séquence synchrone inconditionnelle. *Goto* charge l'adresse de saut dans le PC. La suite du programme s'effectue alors à cette nouvelle adresse.

**Syntaxe**

*goto* étiquette

**Bits de statut affectés**

aucun

### 4.3.6 IORLW: Inclusive OR Literal with W

Cette opération effectue une opération logique "ou" bit à bit entre le *Contenu* de W et la valeur littérale qui suit. Le résultat sera stocké dans le registre de travail.

**Syntaxe**
*iorlw* k;  (W) OR k → (W)

**Bits de statut affectés**
Z

**Syntaxe**
*goto* étiquette

**Bits de statut affectés**
aucun

### 4.3.7 MOVLW: MOV Literal to W

Cette instruction charge la *valeur* spécifié (appelé valeur litérale, ou encore valeur immédiate) dans le registre de travail *W*.

**Syntaxe**
*movlw* k;  k représente une valeur de $0x00$ à $0xFF$

**Bits de statut affectés**
aucun

### 4.3.8 RETFIE: RETurn From IntErupt

Cette instruction indique un retour d'interruption . Elle 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

**Bits de statut affectés**
aucun

### 4.3.9 RETLW: RETurn with Literal in W

On effectue un retour de sous-routine en plaçant en même temps une valeur littérale dans le registre de travail *W*. Cette instruction est équivalente à RETURN, mais permet de sortir d'une sous-routine avec une valeur spécifié dans *W*. Cette instruction peut être utilisée pour créer une lookup-table en mémoire programme.

**Syntaxe**
*retlw* k;  (w)=k puis return

**Bits de statut affectés**
aucun

### 4.3.10 RETURN: RETURN from subroutine

Cette instruction permet de réaliser un retour de sous-routine vers la partie appelante de la partie du programme. Elle doit se trouver à la fin de la sous-routine. Le PC est rechargé depuis la pile et le programme poursuit à l'adresse qui suit la ligne CALL. Elle va toujours de pair avec un instruction CALL.

**Syntaxe**
*return*;  fin de sous-routine

**Bits de statut affectés**
aucun

### 4.3.11 SLEEP

Le PIC est placé en mode de veille. Il ne se réveillera que sous certaine conditions que nous verrons pas dans ce document.

**Syntaxe**
*sleep*;  arret du PIC

**Bits de statut affectés**
aucun

### 4.3.12 SUBLW: SUBstract W from Literal

L'instruction devrait en toute logique s'appeler SUBWL, car on soustrait *W* de la valeur littérale et non la valeur littérale de *W*.

**Syntaxe**
*sublw* k;  k-(W) → (W)

**Bits de statut affectés**
C,DC,Z

Attention à l'utilisation du bit *C*. Si le résultat est positif et qu'il n'y a pas de débordement C vaudra 1 et s'il y a débordement C devient 0.

#### 4.3.13 XORLW: eXclusive OR Literal with W

Cette opération effectue une opération logique "ou exclusif" bit à bit entre le *Contenu* de W et la valeur littérale qui suit. Le résultat sera stocké dans le registre de travail.

**Syntaxe**

*xorlw* k; (W) XOR k → (W)

**Bits de statut affectés**

Z

# 5 Interruptions

Les interruptions et leur gestion sont une caractéristique importante et indispensable dans la programmation des microcontrôleurs.

## 5.1 Rappel

Avant de développer ce sujet, il est peut-être utile de rappeler brièvement ce qu'est une interruption.

Les interruptions sont utilisées pour gérer des événements spécifiques qui peuvent se produire à un instant non déterminable (ou du moins non déterminé) du programme. Le traitement de ces événements nécessite une modification du cours du programme principal, qui doit donc être interrompu pendant ce traitement. Une *interruption* est donc une rupture de séquence asynchrone.

Le mécanisme d'une interruption est le suivant:

- le programme se déroule normalement;
- l'événement survient,
- le programme achève l'instruction en cours de traitement (une instruction commencée n'est donc jamais interrompue!!!);
- le programme saute à l'adresse de traitement de l'interruption (et sauve son contexte);
- le programme traite l'interruption;
- le programme (après avoir restauré le contexte) saute à l'instruction qui suit la dernière instruction exécutée dans le programme principal.

Bien entendu, ce mécanisme ne se déclenche que pour certains événements prédéfinis pour chaque microcontrôleurs. Il est également possible au programmeur de ne pas autoriser l'exécution des interruptions.

Notons que les interruptions entraînent généralement l'exécution d'une sous-routine qui pourrait éventuellement modifier l'état des registres. En outre, le programme principal ne sait pas quand il est interrompu. Il est donc crucial de sauver le contexte (les registres) avant de traiter l'interruption puis de le restaurer avant de reprendre le cours du programme.

## 5.2 Les interruptions sur le *16F84*

Après ce bref rappel, examinons maintenant les particularités des interruptions pour le *16F84*. Le schéma d'exécution d'une telle interruption est donné figure 8.

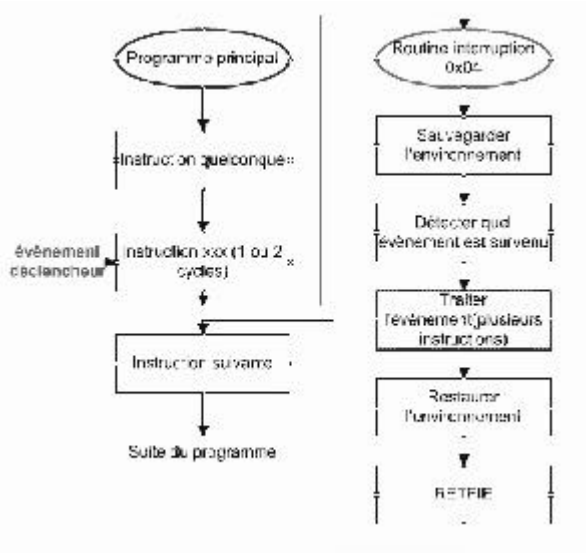


FIG. 8 – *Exécution d'une routine d'interruption sur les PICs*

Tout d'abord, notons qu'il n'y a qu'une seule interruption matérielle sur le *16F84*. Cependant, celle-ci peut être déclenchée par 4 sources périphériques différentes:

- débordement du **Timer0** (cf section 2.4), c'est-à-dire lorsque celui-ci passe de **0xFF** à **0x00**;
- **EEPROM**, c'est-à-dire lorsque l'écriture dans une case EEPROM est terminée;
- **RB0/INT**: si cette broche est configurée en entrée, il est possible de la définir comme source d'interruption (variation de niveau sur cette broche, le type de variation -montante ou descendante- étant imposée par le programmeur via le bit 6 -INTEDG- du registre *OPTION*);
- **PORTB**: même chose pour les broches RB4 à RB7, sauf qu'il n'est pas possible de limiter ce comportement à une seule de ces broches.

Nous devons signaler ici qu'un problème qui se pose toujours lors de l'utilisation d'interruptions sur les broches connectées à un bouton poussoir est celui du rebond. En effet, la fréquence d'horloge du PIC est assez élevée pour qu'une simple pression sur le bouton dure plusieurs périodes d'horloge. Il faut donc que le programmeur tienne compte de l'effet de rebond d'u bouton poussoir en insérant une temporisation adéquate pour la reprise des interruptions, qui permet d'éviter que le PIC ne considère cette poussée unique comme plusieurs interruptions différentes. Nous renvoyons le lecteur intéressé à la bibliographie pour plus de détails.

La source d'interruption étant unique, le programme saute toujours à l'adresse **0x04** après une interruption. Comme les sources peuvent en être variées, nous devrons, dans la routine de traitement, prévoir des tests qui déterminent le périphérique source de l'interruption afin de déterminer le traitement adéquat. La détermination de la source de l'interruption se fait via l'état

des 3 LSB du registre *INTCON* appelés flags d'interruptions, comme expliqué à la section 2.3.

Lors de l'appel à la routine d'interruption, celles-ci sont automatiquement désactivées (comme au début du programme) mais il est possible au programmeur de les réactiver toutes (pour rappel, ces activations/désactivations se font par l'intermédiaire du registre *INTCON*) ou seulement une partie d'entre elles. Si on ne les réactive pas, il est toujours possible de savoir si une interruption s'est produite en vérifiant un à un les flags d'interruptions (qui se positionnent même si les interruptions sont désactivées). Notons cependant qu'il faudra également repositionner ces flags à 0 un par un.

Il est également possible de définir des priorités dans le traitement des interruptions:

- par l'ordre des tests présents dans la routine de traitement;
- en réactivant explicitement, si nécessaire, les interruptions lors de l'appel à la routine de traitement afin de permettre le traitement immédiat des routines de priorité supérieure.

Les PIC en se connectant à cette adresse ne sauvent rien automatiquement, hormis le contenu du PC, sauvé sur la pile interne (n'oublions à nouveau pas qu'elle est limitée à 8 niveaux...). Le programmeur doit donc demander au PIC de sauver le contexte. Dans le cadre du *16F84*, le contexte est assez limité. En effet, il suffit de sauver *W* (le registre de travail) et *STATUS*, ainsi que *FSR* si on travaille en adressage indirect. Afin de ne pas modifier *STATUS* lors de la sauvegarde du contexte (en effet, rappelons que certaines interruptions modifient ce registre, cf section 4), il est nécessaire d'utiliser une procédure particulière. Celle-ci est donnée, avec la structure générale d'une routine d'interruption à la figure 9. Notons l'instruction RETFIE qui doit toujours terminer une routine d'interruption.

**Remarque**
L'adresse du début de la routine d'interruption étant 0x04 et tous les programmes commençant obligatoirement à l'adresse 0x00; il est nécessaire de commencer ses programmes par un saut vers une adresse supérieure où le programme est localisé afin de laisser l'adresse 0x04 libre.

```
;*********************************************************************
;                     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 l'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                ; Réinversion de L et H dans W
                                    ; W restauré sans modifier status
    retfie                          ; return from interrupt
```

FIG. 9 – *Structure de base d'une routine d'interruption*

# 6 Exemples d'applications

Maintenant que nous avons exposé l'architecture et les principes de base de la programmation du *16F84*, nous allons donner quelques exemples simples afin d'illustrer notre propos.

Ces exemples sont entièrement tirés du cours sur les PIC réalisé par BigOnOff [3].

## 6.1 Exemple simple d'utilisation du *16F84*

Commençons par un exemple simple. Le but de ce programme est d'allumer une LED lorsque le bouton connecté à RB2 est enfoncé, et de l'éteindre lorsqu'il est relâché.

```
;*****************************************************************************
;   PROGRAMME D'ALLUMAGE D'UNE LED CONNECTEE SUR LE PORTA.2                   *
;   D'UN PIC16F84 LORS DE LA PRESSION D'UN BOUTON SUR LE PORTB.2.             *
;                                                                             *
;*****************************************************************************
;                                                                             *
;    NOM:       LED-BUTTON                                                    *
;    Date:      2004-2005                                                     *
;    Version:   1.0                                                           *
;    Circuit:                                                                 *
;    Auteur:    Bigonoff                                                      *
;                                                                             *
;*****************************************************************************
;                                                                             *
;    Fichier requis: P16F84.inc                                               *
;                                                                             *
;*****************************************************************************
;                                                                             *
;    Notes:                                                                   *
;                                                                             *
;*****************************************************************************


    LIST      p=16F84           ; Définition de processeur
    #include <p16F84.inc>       ; Définitions de variables

    __CONFIG   _CP_OFF & _WDT_OFF & _PWRTE_ON & _HS_OSC

; '__CONFIG' précise les paramètres encodés dans le processeur au moment de
; la programmation du processeur. 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 vitesse
```

```
;   _XT_OSC     Oscillateur quartz moyenne vitesse
;   _HS_OSC     Oscillateur quartz grande vitesse
;   _RC_OSC     Oscillateur à réseau RC

;*****************************************************************************
;                                ASSIGNATIONS                                 *
;*****************************************************************************

OPTIONVAL   EQU H'0008' ; Valeur registre option
                        ; Résistance pull-up ON
                        ; Pas de préscaler

;*****************************************************************************
;                                 DEFINE                                      *
;*****************************************************************************

#DEFINE LED     PORTA,2     ; Led rouge
#DEFINE BOUTON  PORTB,2     ; Bouton-poussoir


;*****************************************************************************
;                                 MACRO                                       *
;*****************************************************************************

LEDON macro
        bsf LED
    endm

LEDOFF macro
        bcf LED
    endm

;*****************************************************************************
;                    DECLARATIONS DE VARIABLES                                *
;*****************************************************************************

    CBLOCK 0x00C    ; début de la zone variables
        ENDC        ; Fin de la zone

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

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

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

init
```

```
    clrf    PORTA       ; Sorties portA à 0
;   clrf    PORTB       ; sorties portB à 0
    bsf     STATUS,RP0  ; sélectionner banque 1
    movlw   OPTION_VAL  ; charger masque
    movwf   OPTION_REG  ; initialiser registre option

            ; Effacer RAM
            ; -----------
    movlw   0x0c        ; initialisation pointeur
    movwf   FSR         ; pointeur d'adressage indirect
initl
    clrf    INDF        ; effacer ram
    incf    FSR,f       ; pointer sur suivant
    btfss   FSR,6       ; tester si fin zone atteinte (>=40)
    goto    initl       ; non, boucler
    btfss   FSR,4       ; tester si fin zone atteinte (>=50)
    goto    initl       ; non, boucler

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

;*****************************************************************
;                     PROGRAMME PRINCIPAL                       *
;*****************************************************************

start                   ; étiquette de début de la boucle principale
    btfss   BOUTON      ; tester RB2, sauter si vaut 1
    LEDON               ; RB2 vaut 0, donc on allume la LED
    btfsc BOUTON        ; tester RB2, sauter si vaut 0
    LEDOFF              ; RB2 vaut 1, donc LED éteinte
    goto start          ; et on recommence

    END                 ; directive fin de programme
```

Le programme étant suffisamment commenté, nous ne développerons pas plus avant mais passerons tout de suite à un exemple un peu plus complexe.

## 6.2 Exemple avancé

Cet exemple illustre l'utilisation des interruptions, du Timer0 et de la mémoire EEPROM. Le but est toujours d'allumer une LED, mais cette fois sans intervention extérieure: elle clignotera à une fréquence choisie. Pour cela, on utilise le Timer0 et l'interruption. En outre, la période, variable, est stockée et lue dans l'EEPROM.

```
;*****************************************************************
;   PROGRAMME D'ALLUMAGE D'UNE LED CONNECTEE SUR LE PORTA.2       *
;   D'UN PIC16F84 SELON UNE FREQUENCE IMPOSEE STOCKEE DANS L'EEPROM  *
```

```
;   ET A L'AIDE DU TIMER0.                                              *
;                                                                       *
;    Fichier requis: P16F84.inc                                         *
;                                                                       *
;***********************************************************************
;                                                                       *
;    NOM:       LED-CLI                                                 *
;    Date:      2004-2005                                               *
;    Version:   1.0                                                     *
;    Circuit:                                                           *
;    Auteur:    M. Dechesne                                             *
;                                                                       *
;***********************************************************************
;                                                                       *
;    Fichier requis: P16F84.inc                                         *
;                                                                       *
;***********************************************************************
;                                                                       *
;    Notes: Démonstration de l'utilisation des données en EEPROM.       *
;           La base de temps de clignotement est contenue dans          *
;           l'EEPROM                                                    *
;                                                                       *
;***********************************************************************


    LIST      p=16F84           ; Définition de processeur
    #include <p16F84.inc>       ; Définitions des constantes

    __CONFIG    _CP_OFF & _WDT_OFF & _PWRTE_ON & _HS_OSC


;***********************************************************************
;                               ASSIGNATIONS                            *
;***********************************************************************

OPTIONVAL   EQU H'0087' ; Valeur registre option
                        ; Résistance pull-up OFF
                        ; Préscaler timer à 256

INTERMASK   EQU H'00A0' ; Interruptions sur tmr0


;***********************************************************************
;                               DEFINE                                  *
;***********************************************************************

#DEFINE LED     PORTA,2     ; LED

;***********************************************************************
;                               MACRO                                   *
;***********************************************************************
```

```
BANK0   macro
    bcf     STATUS , RP0    ; passer banque0
    endm

BANK1   macro
    bsf     STATUS , RP0    ; passer banque1
    endm

READEE  macro   adeeprom    ; macro avec paramètre
    movlw   adeeprom        ; charger adresse 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 banque0
    movf    EEDATA , w      ; charger valeur lue dans W
    endm                    ; fin de la macro

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 banque0
    endm

;*********************************************************************
;                   DECLARATIONS DE VARIABLES                       *
;*********************************************************************

    CBLOCK 0x00C    ; début de la zone variables
    w_temp :1       ; Sauvegarde du registre W
    status_temp : 1 ; Sauvegarde du registre STATUS
    cmpt : 1        ; compteur de passage
    reload : 1      ; valeur à recharger dans compteur
        ENDC        ; Fin de la zone
```

```
;*****************************************************************
;                  DECLARATIONS DE LA ZONE EEPROM                 *
;*****************************************************************
    org 0x2100      ; adresse début zone eeprom
    DE  0x1E        ; valeur de recharge du compteur

#DEFINE eereload    0x00    ; adresse eeprom de eereload

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

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

;*****************************************************************
;                     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
            ; 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
    bcf     INTCON,T0IF ; effacer flag interrupt timer
    goto    restorereg  ; et fin d'interruption
                ; SUPPRIMER CETTE LIGNE POUR
                ; TRAITER PLUSIEURS INTERRUPT
                ; EN 1 SEULE FOIS

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
                ; SUPPRIMER CETTE LIGNE POUR
                ; TRAITER PLUSIEURS INTERRUPT
                ; EN 1 SEULE FOIS

intsw3
    BANK1               ; passer banque1
    btfsc   INTCON,EEIE ; tester si interrupt EEPROM autorisée
    btfss   EECON1,EEIF ; oui,tester si interrupt EEPROM
    goto    restorereg  ; non sauter
    call    inteep      ; traiter interruption eeprom


            ; 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        ; Réinversion de L et H dans W
                            ; W restauré sans modifier status
    retfie                  ; return from interrupt

;*********************************************************************
;                     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
```

```
;*************************************************************************
;                      INTERRUPTION RB0/INT *
;*************************************************************************
intrb0
    return          ; fin d'interruption RB0/INT
                ; peut être remplacé par
                ; retlw pour retour code d'erreur

;*************************************************************************
;                      INTERRUPTION RB0/RB4                              *
;*************************************************************************
intrb4
    return          ; fin d'interruption RB0/RB4
                ; peut être remplacé par
                ; retlw pour retour code d'erreur

;*************************************************************************
;                      INTERRUPTION EEPROM                               *
;*************************************************************************
inteep
    return          ; fin d'interruption eeprom
                ; peut être remplacé par
                ; retlw pour retour code d'erreur


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

init
    clrf    PORTA       ; Sorties portA à 0
    clrf    PORTB       ; sorties portB à 0
    BANK1               ; passer banque1
    movlw   OPTION_VAL  ; 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 (>=40)
    goto    init1       ; non, boucler
    btfss   FSR,4       ; tester si fin zone atteinte (>=50)
    goto    init1       ; non, boucler

            ; initialiser ports
            ; -----------------
```

```
    bcf LED             ; passer LED en sortie
    BANK0               ; passer banque0

        ; initialisations variables
        ; ------------------------
    READEE  eereload    ; lire emplacement eeprom 0x00
    movwf   reload      ; placer dans reload
    movwf   cmpt        ; et initialiser compteur de passages
    movlw   INITERMASK  ; masque interruption
    movwf   INITCOM     ; charger interrupt control
    goto    start       ; sauter programme principal


;*****************************************************************
;                     PROGRAMME PRINCIPAL                        *
;*****************************************************************

start

    nop
    goto    start       ; non, attendre

    END             ; directive fin de programme
```

### 6.2.1 Timer0/Interrputions

Avec une horloge à 4MHz (l'hypothèse pour ce programme), en choisissant la plus grande prédivision sur le Timer0 (cf table 1), celui-ci déborde toutes les (256*256)=65536$\mu$s. Choisir la fréquence d'allumage revient donc à choisir le nombre de passages dans la routine d'interruption avant d'allumer la LED. C'est la valeur stockée dans l'EEPROM qui indique ce nombre de passages.

### 6.2.2 EEPROM

Remarquons qu'on ne lit qu'une seule fois la valeur en EEPROM (lors de l'initialisation) et qu'on la stocke dans la variable `reload`. C'est parce que le processus de lecture en EEPROM est plus complexe, et donc plus lent, qu'une simple lecture en RAM.

L'initialisation de l'EEPROM est également faite une seule fois, lors de la programmation, à l'aide de l'instruction DE.

# 7 Evolution: vers le *16F87x*...

Le *16F84* est le composant le plus simple de la famille des PIC Mid-Range. Les *16F87x* sont un exemple très répandu de modèles un peu plus complexes.

Il existe beaucoup de similitudes entre ces deux types de composants. Nous allons seulement évoquer ici les compléments des *16F87x*:

- plus de mémoire RAM, Flash et EEPROM. La mémoire RAM disponible est de 368 octets répartis dans 4 banques:
  1. 80 octets en banque 0, adresses 0x20 à 0x6F;
  2. 80 octets en banque 1, adresses 0xA0 à 0XEF;
  3. 96 octets en banque 2, adresses 0x110 à 0x16F;
  4. 96 octets en banque 3, adresses 0x190 à 0x1EF;
  5. 16 octets communs aux 4 banques, soit 0x70 à 0x7F = 0xF0 à 0xFF = 0x170 à 0x17F = 0x1F0 à 0x1FF;

  Les octets communs ont l'avantage de permettre l'utilisation de ces emplacements sans devoir connaître l'état de RP0, RP1 et IRP.
- plus de ports I/O (20 broches);
- 3 timers;
- nouvelles fonctionnalités: gestion des ports série et parallèle...
- présence d'un convertisseur A/D à plusieurs canaux d'une résolution de 10 bits;
- possibilité de sortie en PWM[8]

Ces différences entraînent quelques modifications au niveau de la programmation et de l'initialisation. La plus importante concerne la directive *_CONFIG* qui gagne en complexité: elle permet maintenant de définir 14 paramètres au lieu de 4.

Malgré ces différences, tous les programmes écrits pour un *16F84* tourneront sans problèmes sur un *16F87x* à condition qu'on ait bien respecté la condition de toujours positionner RP1 à 0.

8. *Pulse Width Modulation*, technique de modulation d'un signal numérique où la porteuse est un signal carré. La longueur des pulses est déterminée par l'amplitude du signal modulant.

# Références


[1] Microchip, *PIC16F84 Datasheet*,
http://www.microchip.com/download/lit/pline/picmicro/families/16f8x/30430c.pdf

[2] H. Smeets, *Aide à la conception de ciruits électroniques*
http://www.montefiore.ulg.ac.be/ smeets/spe.html, 2001

[3] Bigonoff, *La programmation des PICs, parties 1-2*,
Révision 10

[4] P. Wolper, *Structure des ordinateurs*,
http://www.montefiore.ulg.ac.be/%7Epw/cours/struct.html, 2002-2003

[5] M. Mano et C. Kime, *Logic and computer design fundamentals*,
2$^{e}$ édition, 2000