

SOMMAIRE

I) Présentation de la carte	3
II) Environnement de travail	4

TPs

TP 1	5
-------------	----------

I) Création d'un 1^{er} projet et analyse d'un exemple

- 1) Création d'un projet
- 2) Compilation du projet
- 3) Programmation du PIC
- 4) Analyse du fichier source
- 5) Simulation du fichier source

II) Ecriture d'un 1^{er} programme

III) Programme CHENILLARD1

TP 2	17
-------------	-----------

I) Programme CHENILLARD1 (version C)

II) Programme CHENILLARD2 (tempo avec *timer*)

TP 3	21
-------------	-----------

I) Programme CHENILLARD3 (tempo avec *timer* déclenchant une interruption)

II) Programme CHENILLARD4 (tempo avec le "chien de garde")

III) Programme MESSAGE

ANNEXE 1	<u>Schéma de la carte</u>	25
ANNEXE 2	<u>Présentation des PICs</u>	26

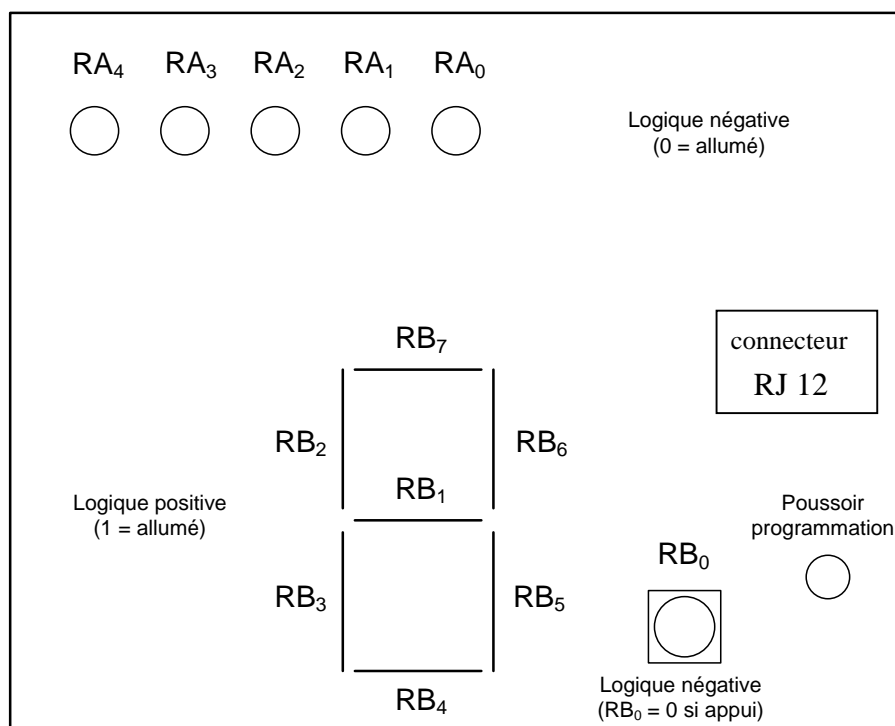
MICROCHIP PICs

Travaux Pratiques

Objectifs : Ces TPs ont pour but d'illustrer le fonctionnement des PICs et de découvrir les outils de développement proposés par Microchip pour programmer ce type de microcontrôleurs. On s'intéresse ici au PIC 16F84a qui fait partie des PICs d'entrée de gamme de Microchip, donc plus facile à prendre en main. On programmera dans un 1^{er} temps en langage assembleur avec le jeu d'instructions propre au processeur afin de mieux comprendre son fonctionnement. On verra ensuite comment programmer les PICs en langage C de plus haut niveau. Pour finir, un projet de synthèse utilisant le PIC16F877a plus évolué permettra de mettre en application les concepts vus dans les 3 TPs.

I) Présentation de la carte

Le PIC 16F84a (= 16F84 avec mémoire FLASH améliorée) est ici intégré sur une carte d'application permettant la commande d'un afficheur 7 segments et de 5 LEDs. Un bouton poussoir est relié au bit 0 du port B (RB₀) ce qui peut permettre de générer une interruption matérielle. Un autre poussoir est aussi présent et sert à la programmation du PIC.



Le connecteur RJ12 sert à relier la carte à l'interface ICD 3 qui permet ici la programmation du PIC et également l'alimentation de la carte (jusqu'à 200 mA). La programmation se fait suivant le protocole ICSP (In-Circuit Serial Programming) défini par Microchip (voir datasheet *DS30277d* du fabricant) : la broche /MCLR est maintenue à 12 V pour faire entrer le PIC en mode programmation et les instructions sont transmises bit par bit de façon synchrone sur la broche PGD au rythme de l'horloge appliquée sur PGC. Le schéma électrique de la carte est donné dans l'annexe 1.

II) Environnement de travail

La chaîne d'outils utilisée pour le développement est MPLAB X, fournie gratuitement par MICROCHIP. Il s'agit d'un IDE (Integrated Development Environment) qui intègre tous les outils nécessaires pour la mise au point d'un projet :

- éditeur de fichier source
- compilateur + assembleur
- éditeur de lien (permet d'obtenir le fichier exécutable de sortie)
- "débugueur" (simulateur et/ou vrai débugeur si le circuit cible le permet)
- programmeur

Un IDE offre l'avantage de pouvoir effectuer toutes ces opérations dans un même environnement de travail et facilite grandement les interactions entre les modules (gestion et mise à jour automatique des fichiers d'échanges, exécution d'un seul clic des modules permettant l'obtention du fichier exécutable...) d'où un gain évident de productivité.

MPLAB X peut communiquer avec le PIC sur sa carte d'application grâce au module d'interface ICD 3, relié d'un côté au port USB du PC et de l'autre au PIC via les 3 broches de contrôle mentionnées plus haut. Cette interface permet la programmation des PICs et aussi d'exploiter le module ICD (In Circuit Debugger) intégré dans la plupart des PICs. L'ICD autorise un "vrai" debuggage en exécutant le programme sur le PIC et en renvoyant le contenu des registres à MPLAB X depuis l'interface ICD 3. On peut ainsi tester le programme en mode pas à pas, continu... comme avec un simulateur mais avec l'intérêt de contrôler véritablement les sorties du PIC et d'être en interaction avec le système. Nous verrons l'usage du simulateur en TP avec le PIC16F84a et les possibilités de l'ICD avec le PIC16F877a dans le projet de synthèse qui suivra.

Tous les documents et logiciels utiles pour ce module EN111 sont disponibles sur la page :

www.enseirb-matmeca.fr/~bedenes (onglet MICROCHIP)

TP 1

- Objectifs
- Découverte de MPLAB X
 - Manipulation des instructions des PICs et gestion des entrées / sorties

I) Création d'un 1^{er} projet et analyse d'un exemple

Cette 1^{ère} partie va sans doute sembler la plus difficile et la plus lourde car il va s'agir de se familiariser avec les outils de développement et le fonctionnement des PICs avant d'être capable de mettre en œuvre les premiers programmes. Nous allons commencer la prise en main de MPLAB X en programmant le PIC à partir d'un fichier d'exemple. Au préalable :

- Ouvrir une session **GNOME classique** et télécharger le répertoire **TPs_PIC** accessible depuis la page WEB www.enseirb-matmeca.fr/~bedenes (onglet MICROCHIP) et le placer dans un répertoire de travail .../EN111 par exemple
- Vérifier que l'interface ICD 3 est bien connectée au PC et à la carte d'application
- Lancer MPLAB X depuis le menu Applications > Développement > MPLAB IDE

1) Création d'un projet

→ FILE > NEW PROJECT

- 1- Choose project : choisir les options *Microchip Embedded* et *Standalone Project*
- 2- Select Device : choisir le *PIC16F84A*
dans la famille *Mid-Range 8-bit MCUs (PIC10/12/16/MCP)*
- 3- Select Tool : choisir l'interface *ICD 3* en cliquant sur son numéro de série

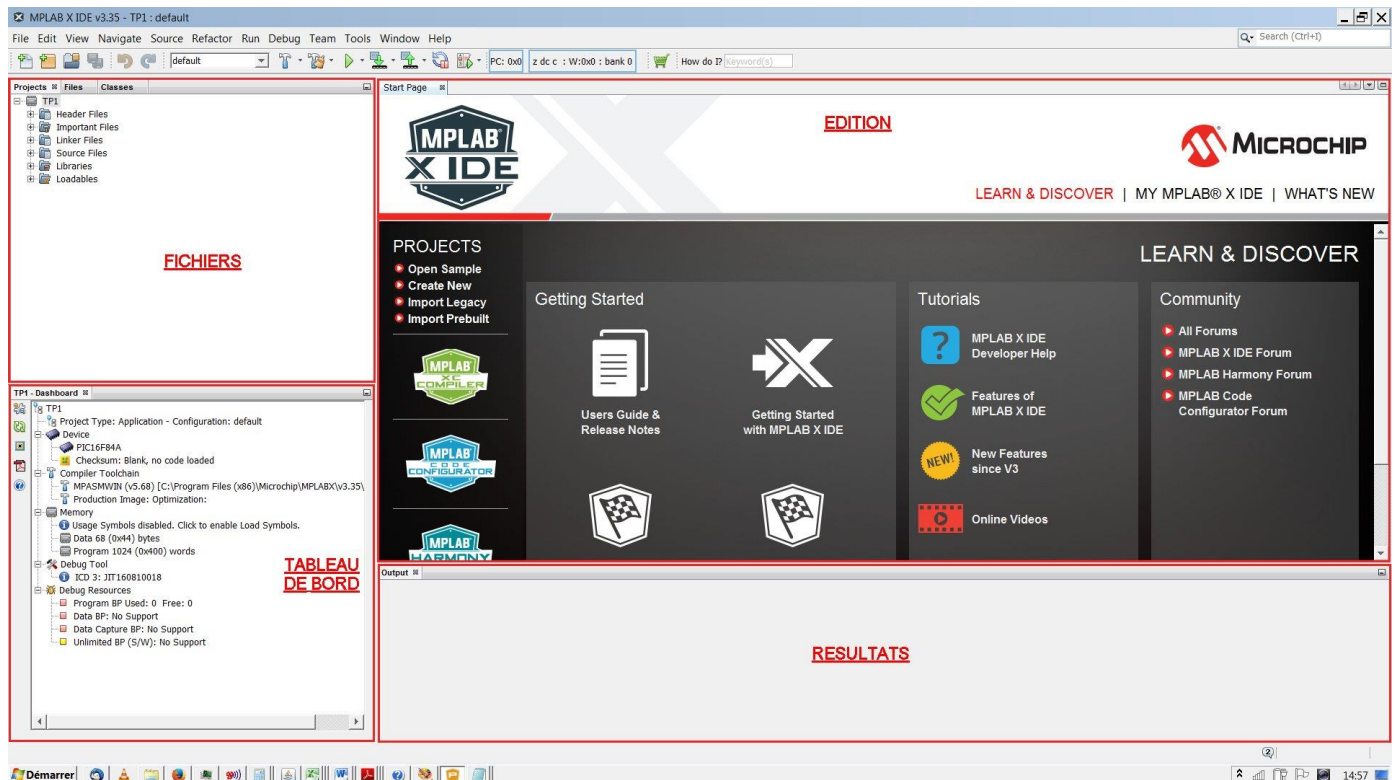
Remarque : A ce stade, on peut s'assurer de la liaison avec l'ICD3 si l'on observe bien son numéro de série



Si ce n'est pas le cas vérifier à nouveau le câblage.

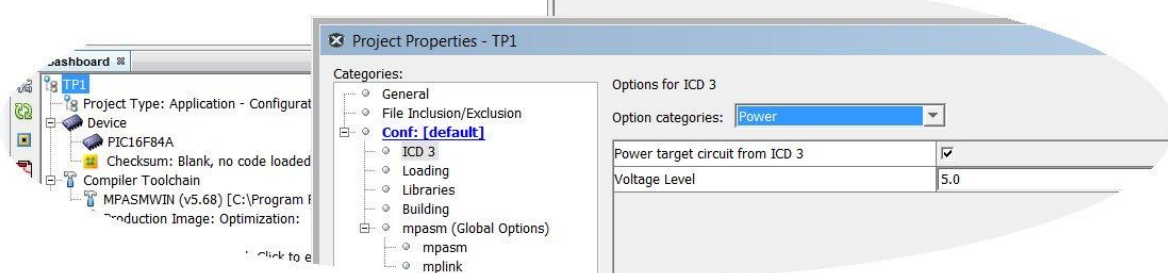
- 6- Select Compiler : choisir le compilateur assembleur *mpasm (5.68)*
- 7- Select Project Name and Folder : définir le nom du projet (**TP1**)
et son emplacement

Une fois le projet créé, l'environnement de travail de MPLAB X se compose de 4 zones d'affichage arrangées selon la figure ci-dessous :



- La fenêtre "FICHIERS" permet d'accéder à tous les fichiers qui composent le projet, classés par catégories
- La fenêtre "EDITION" permet de visualiser voire modifier les fichiers sélectionnés par un double clic
- La fenêtre "TABLEAU DE BORD" résume les paramètres du projet (ceux-ci sont modifiables à tout moment en double cliquant sur le nom du projet)
- La fenêtre "RESULTATS" affiche toutes les informations liées au déroulement du projet. C'est ici que s'affichent automatiquement tous les résultats à l'issue de chaque phase de la conception (compilation, programmation...). On peut également y ajouter différents onglets (via le menu WINDOW) permettant de configurer le PIC, obtenir des informations sur la simulation, voir la mémoire du PIC...

La carte devant être alimentée par l'ICD 3 on va commencer par configurer cette option dans le projet :

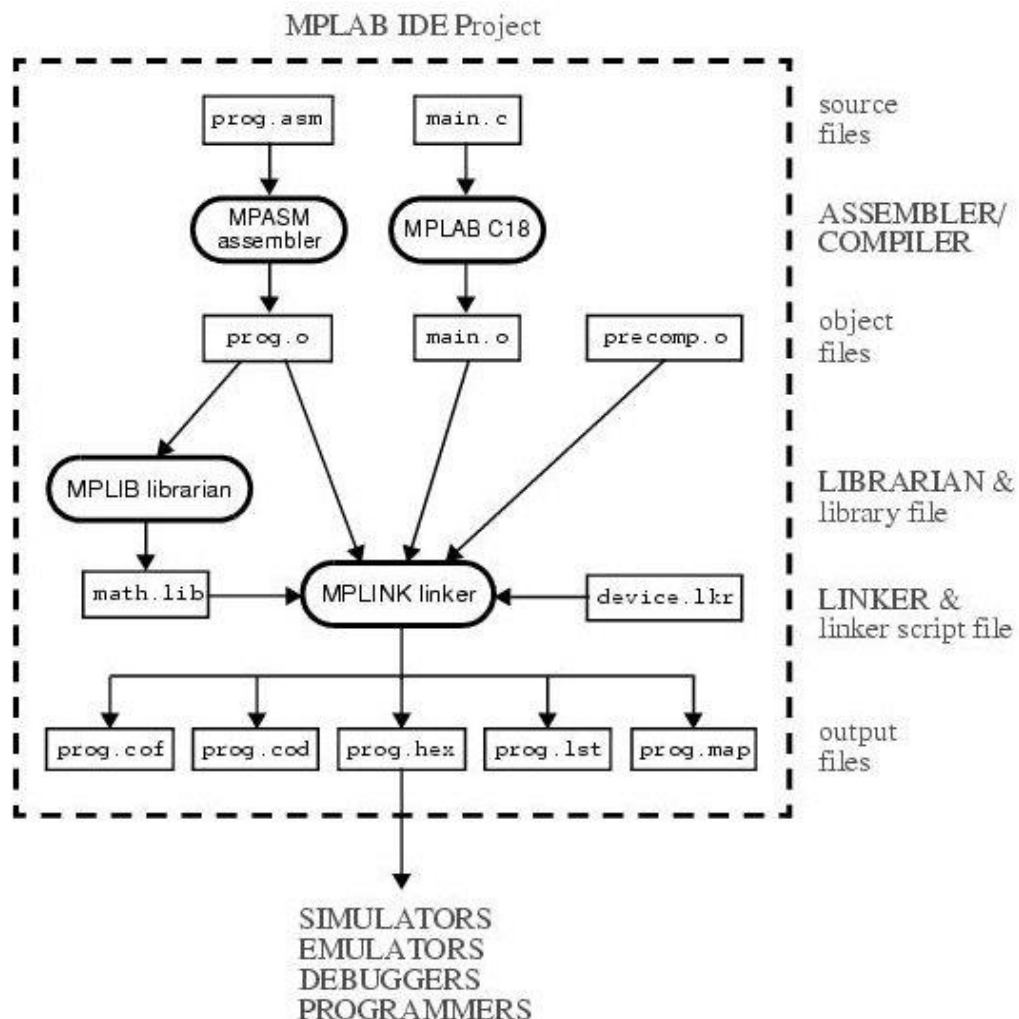


→ Double-clic sur le nom du projet (TP1), sélectionner la catégorie *ICD 3* – option *Power* et cocher la case *Power target circuit from ICD 3*.

On ajoute maintenant le fichier source **exemple1.asm** fourni dans **TPs_PIC** depuis la fenêtre "FICHIERS" : → clic droit sur *Sources Files* puis *Add Existing Item...* en validant la case *copy* pour copier le fichier source dans le répertoire du projet (TP1.X créée automatiquement lors de la création du projet TP1).

2) Compilation du projet

La compilation du projet a pour but d'obtenir, à partir du ou des fichiers sources (**.asm**, **.c**) voire de fichiers objets (**.o**, **.lib**), un fichier de sortie (**.hex**) correspondant au code exécutable qui pourra ensuite être implanté en mémoire programme du PIC lors de la phase de programmation. D'autres fichiers de sortie sont également disponibles dans le répertoire du projet comme on peut le voir sur la figure suivante qui représente toutes les étapes de la compilation : mapping mémoire (**.map**), données pour le débogage (**.cof**)...




MPLAB IDE utilise la chaîne d'outils "MPASM suite" qui intègre les 3 modules :

MPASM : logiciel d'assemblage qui permet à partir d'un fichier source **.asm** en assembleur d'obtenir un fichier objet **.o** contenant le code exécutable correspondant, non implanté à des adresses fixes en mémoire programme pour le moment (code relogeable) et dont les registres sont désignés par des symboles (références) qui n'ont pas encore d'adresse en mémoire RAM. Un compilateur C peut aussi être utilisé pour obtenir un fichier objet à partir d'un fichier source en langage C (sur la figure MPLAB C18, mais pour nous ce sera MPLAB XC8).

MPLIB : logiciel pour créer une librairie (association) de fichiers objets.

MPLINK : logiciel d'édition de liens pour relier tous les fichiers objets (voire des librairies) pour obtenir le fichier exécutable **.hex**. Ce module a besoin pour cela d'un fichier de commande **.lkr** situé dans le répertoire d'installation, qui définit les sections où le code peut être alloué en mémoire programme ainsi que l'adresse des registres.

Toutes ces étapes sont ordonnancées par le logiciel MAKE à partir de Makefiles situés dans le répertoire NBPROJECT du projet, exécuté automatiquement lors de la compilation (plus grande efficacité qu'un simple script puisque seules les étapes nécessaires sont effectuées, par exemple seuls les fichiers qui ont été modifiés sont recompilés) :

→ Cliquer sur  ou bien lancer RUN > BUILD PROJECT

Tous les fichiers modifiés sont automatiquement sauvegardés puis recompilés et l'onglet *output – TP1 (Build, Load,...)* s'ouvre dans la fenêtre "TACHES" permettant de visualiser le bon déroulement de la compilation avec le message final BUILD SUCCESSFUL.


Ici la compilation indique une erreur car le fichier **p16F84A.inc** inclus ligne 1 doit être spécifié en minuscules "**p16f84a.inc**" (cf commentaires) → **Effectuer la modification et recompiler.**

On notera que plusieurs projets peuvent être ouverts en même temps dans la fenêtre "FICHIERS". Celui qui sera compilé est celui visible dans la fenêtre "TABLEAU DE BORD". Pour éviter les confusions, il est conseillé de le définir clairement en tant que **projet principal.** C'est par défaut le dernier projet ouvert, sinon pour faire passer un des projets sous le statut **principal** sélectionner son nom dans la fenêtre "FICHIERS" puis clic droit > SET AS MAIN PROJECT, il apparaît maintenant en caractères gras et dans le TABLEAU DE BORD.

→ Observer avec la commande FILE > OPEN FILE les fichiers générés dans les répertoires *.\TP1.X\build\default\production* et *.\TP1.X\dist\default\production*

On pourra ouvrir par curiosité les fichiers **exemple1.lst** (code source avec résultat de l'assemblage) et **TP1.X.production.hex** (code exécutable qui sera chargé sur la cible).



3) Programmation du PIC

→ Cliquer sur 

MPLAB X relance la compilation et transfère le fichier **TP1.hex** dans la mémoire programme (FLASH) du PIC via l'interface ICD 3. Il vérifie ensuite par une relecture que l'opération s'est bien effectuée.

L'onglet *output – ICD 3* s'ouvre dans la fenêtre "TACHES" permettant de suivre le bon déroulement de ces opérations (message *Programming / Verify Complete* si tout est OK).

Remarque : Les broches RB₆ (PGC) et RB₇ (PGD) connectées à l'ICD 3 sont aussi utilisées sur notre carte pour la commande de l'afficheur ce qui peut poser des problèmes d'interférences. Si tel est le cas recommencer la programmation en maintenant enfoncé le bouton poussoir rouge sur la carte (déconnection de l'afficheur) **durant toute la phase de programmation/vérification.**

Une fois la programmation correctement effectuée le PIC commence à exécuter le programme. Un clic sur  permet d'activer le RESET du PIC (broche /MCLR = 0), il se transforme en  pour désactiver le RESET et permettre le démarrage du PIC.

4) Analyse du fichier source

Pour comprendre le fichier **exemple.asm** il est nécessaire de connaître les spécificités des microcontrôleurs PIC et d'être capable de répondre aux questions ci-dessous. Le document de référence sera toujours la datasheet du PIC16F84a (désignée par **DS** dans la suite) disponible en salle de TP et sur la page WEB www.enseirb-matmeca.fr/~bedenes - onglet MICROCHIP > Documents. Pour vous aider, vous pouvez consulter la présentation synthétique des PICs de l'annexe 2 et bien sûr le cours EN114 pour les notions générales.

A
P
R
E
P
A
R
E
R

- Comment est organisée la mémoire programme (taille, point d'entrée au démarrage...) ?
- Comment est organisée la mémoire de données (taille, notion de banque...) ?
- Comment sont codées les instructions (taille, différence entre adressage immédiat et direct) ?
- A quoi correspondent les registres SFRs, W, PC ?
- Comment sont-ils accessibles par les instructions du processeur ?
- Combien de ports sont disponibles, comment peut-on lire ou écrire sur les broches ?

- Comment est générée l'horloge du processeur, quelle est sa fréquence ?
- Comment se déroule l'exécution d'une instruction ?
- Qu'est-ce que le registre CONFIG ?

Test en
début
de 1^{ère}
séance

Dans le fichier source à analyser, on distingue les directives d'assemblage, qui sont des commandes à l'attention de l'outil de compilation MPASM X, des instructions en langage assembleur qui correspondent réellement au programme qui sera exécuté par le PIC.

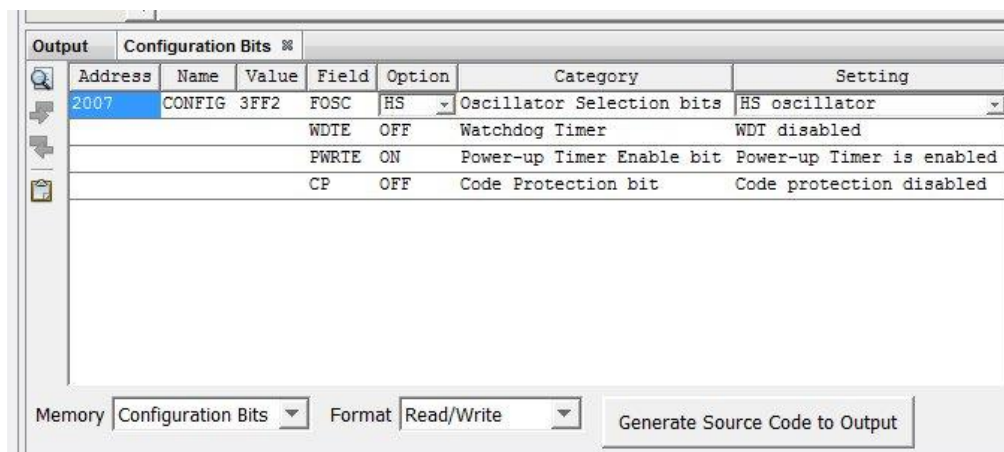
a) Observations des directives d'assemblages

Elles sont commentées dans le listing pour en donner la signification.

→ Indiquer clairement leur rôle dans notre application.

Remarques :

- La directive `__CONFIG` peut être paramétrée automatiquement. Il suffit pour cela de faire apparaître la fenêtre de configuration WINDOW > PIC MEMORY VIEWS > CONFIGURATION BITS, de sélectionner les bonnes valeurs pour chaque paramètre et de cliquer sur le bouton *Generate Source Code to Output* pour obtenir le code à intégrer dans le fichier source.



- Le fichier **p16f84a.inc** se trouve en fait dans le répertoire d'installation `\opt\Microchip\MPLABX\3.35\mpasmx`. Ouvrir ce fichier et observer son contenu.

b) Observation des instructions

→ Répondre aux questions ci-dessous en se demandant à chaque fois :

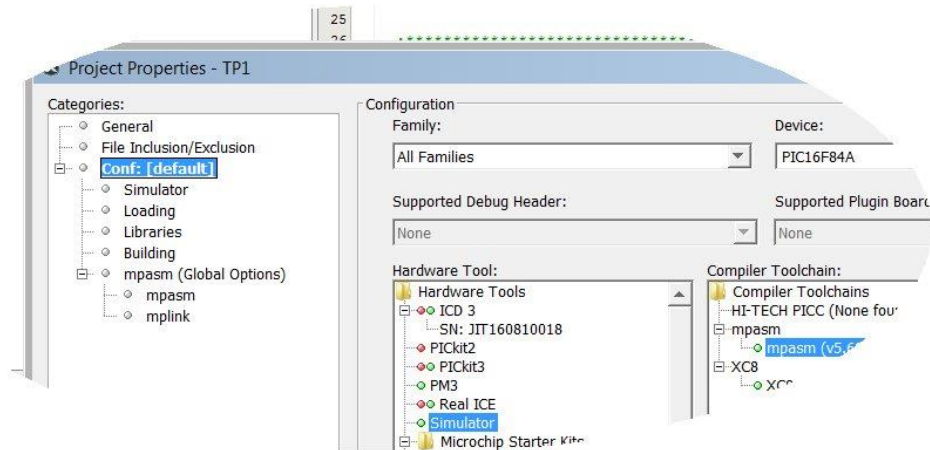
Que fait l'instruction ?			Quel est le rôle de cette opération dans le programme ?
<i>bsf</i>	<i>STATUS,RP0</i>	?	Pourquoi agir sur RP0 ?
<i>clrf</i>	<i>TRISA</i>	?	Quel est l'effet sur le fonctionnement du PIC ?
<i>movlw</i>	<i>0x0A</i>	?	Pourquoi passe-t-on par W ?
<i>movwf</i>	<i>PORTA</i>	?	Quel est l'effet sur le PIC ?
<i>goto</i>	<i>debut</i>	?	Pourquoi utilise-t-on une étiquette ?
			Quelle adresse représente-t-elle ici ?
			Que se passera-t-il si on enlevait cette instruction ?

Plus globalement, quel est l'intérêt d'utiliser des symboles tels que *STATUS*, *RP0*, *TRISA*... ?

5) Simulation du fichier source

Pour vérifier toutes les fonctionnalités d'un programme il est important de pouvoir observer son déroulement en cours d'exécution et ainsi tester son comportement. MPLAB X permet notamment de visualiser tous les registres du microcontrôleur après l'exécution de chaque instruction ou bloc d'instructions. Quand le PIC dispose d'un module de débogage ICD (In Circuit Debugger), on a la possibilité d'avoir une vraie exécution du programme par le PIC ce qui sera préférable pour plus de réalisme. Le PIC 16F84a n'en disposant pas, on se contentera ici d'une simulation par MPLAB X.

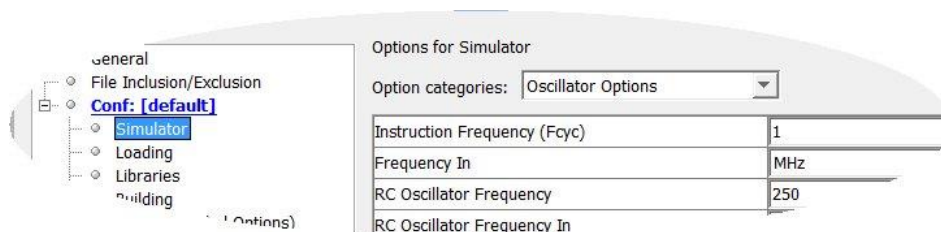
→ Afficher les propriétés du projet (double-clic sur le nom dans le TABLEAU DE BORD)




→ Choisir *Simulator* dans la section *Hardware Tools* (on aurait conservé le choix *ICD 3* pour faire un débogage sur un PIC possédant un module ICD)

Différentes options du simulateur peuvent être paramétrées dans cette même fenêtre :

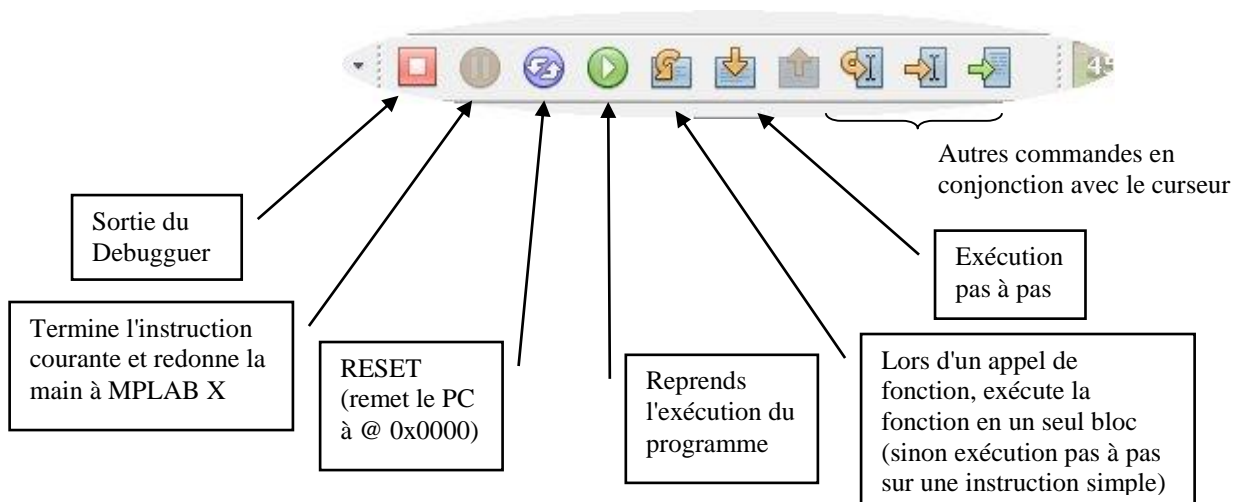
→ On vérifiera que la fréquence d'exécution des instructions est bien de 1 MHz ou 1 Mips (1 instruction toutes les 4 périodes d'horloge et $F_H = 4$ MHz)





→ Cliquer sur  pour lancer le debugger (ouvrir au préalable le fichier source)

MPLAB X compile le projet et démarre automatiquement la simulation.

Dans la barre supérieure se trouvent plusieurs boutons de contrôle du simulateur :




→ Cliquer sur  pour stopper la simulation. (un clic sur  la relance, etc...)

Une ligne verte indique sur le fichier source la prochaine instruction qui sera exécutée.

→ Cliquer sur  le simulateur avance d'une instruction suivant le déroulement du programme.

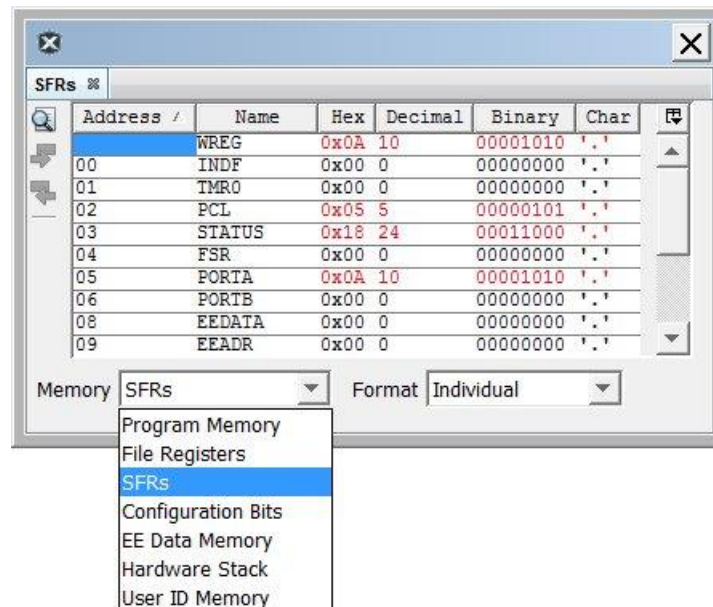
→ Cliquer sur le numéro de ligne d'une instruction plus loin dans le programme : la ligne est surlignée en rouge pour signifier la pose d'un point d'arrêt.

→ Cliquer **sur**  pour reprendre l'exécution, le simulateur s'arrête au point d'arrêt ce qui permet de simuler des blocs d'instructions.

Pour observer le comportement du programme, un grand nombre d'informations sont observables dans la fenêtre "RESULTATS". Plusieurs onglets peuvent ainsi être ajoutés grâce aux menus :

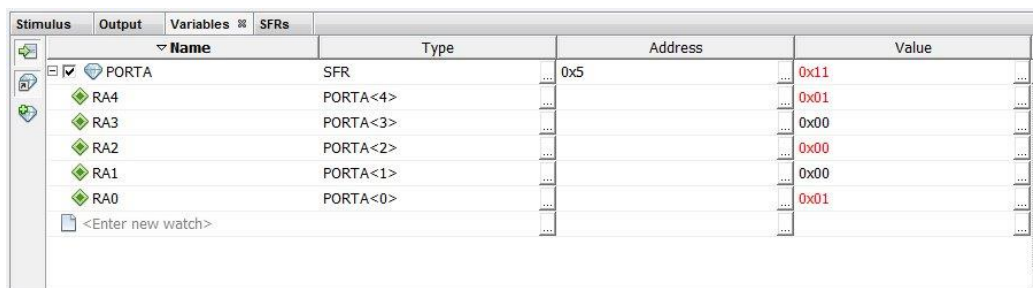
- WINDOW > DEBUGGING
- WINDOW > PIC MEMORY VIEWS
- WINDOW > SIMULATOR

→ Utiliser la commande WINDOW > PIC MEMORY VIEWS > SFRS afin d'observer les registres SFRs. Les autres zones mémoires (mémoire programme, pile système,...) sont directement accessibles avec le menu déroulant *Memory* mais rien n'empêche bien sûr d'ouvrir plusieurs fenêtres et de les visualiser simultanément en les détachant de la fenêtre "RESULTATS" : clic droit sur le nom de la fenêtre puis FLOAT (DOCK pour rattacher)



La couleur rouge signifie que la valeur a changé depuis la dernière exécution.

On peut aussi choisir de visualiser des registres (utilisateur ou SFRs) spécifiques de façon directe en pointant le registre dans le fichier source avec la souris puis clic droit > NEW WATCH. Après validation, la fenêtre *Variables* s'ouvre et le registre s'ajoute à la liste.



On a de plus la possibilité dans cette fenêtre de pouvoir modifier directement la valeur du registre.

→ Avec le menu WINDOW > DEBUGGING faire apparaître les fenêtres DISASSEMBLY et STOPWATCH

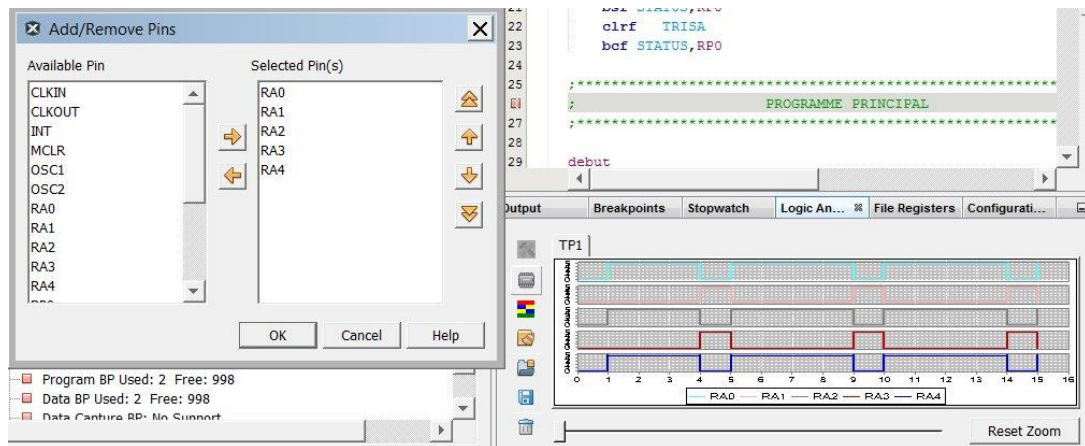
- DISASSEMBLY présente le code exécutable "désassemblé" en-dessous de chaque instruction du code source (très utile pour se repérer lorsque l'on simule des sources en C)
- STOPWATCH permet d'observer le temps qui s'écoule entre chaque exécution

→ Effectuer un RESET et exécuter le programme pas à pas en observant l'affectation des registres SFRs.

→ Observer la fenêtre STOPWATCH et noter le temps d'exécution de chaque instruction de la boucle. Remarque ?

Beaucoup d'autres informations peuvent être observées. On peut par exemple utiliser un analyseur logique virtuel pour observer l'évolution temporelle de signaux :

→ Actionner la commande WINDOW > SIMULATOR > ANALYSER pour ouvrir l'analyseur et observer par exemple tous les bits du PORTA (avec)



→ Effectuer un RESET et exécuter le programme pas à pas pour vérifier l'état des 5 signaux.

→ "Décommenter" l'instruction *comf* du fichier source. Relancer le debugger et simuler à nouveau. Expliquer la forme des signaux obtenus et leur périodicité.

D'autres fonctionnalités intéressantes pourront être utilisées particulièrement pour la mise au point de programmes plus complexes. On citera la fonction TRACE qui permet de visualiser sous forme de listing la succession des instructions exécutées par le processeur à chaque cycle machine et de vérifier d'un seul coup d'œil le bon déroulement d'une routine simulée.

On pourra consulter la documentation de MPLAB X pour plus d'informations (HELP > TOOL HELP CONTENTS > SIMULATOR HELP)

II) Ecriture d'un 1^{er} programme

On va maintenant s'appuyer sur l'exemple précédent pour créer un programme **BP.asm** qui fera évoluer le caractère de l'afficheur 7 segments en fonction de l'état du bouton poussoir. On souhaite par exemple afficher le chiffre "1" lorsqu'on appuie sur le bouton poussoir et "0" au repos.

On remarquera que le poussoir est connecté sur la broche RB₀ sans résistance de tirage et qu'il relie cette broche à la masse lorsqu'on appuie. Ce choix s'explique parce que les PICs de la série 16XX disposent de résistances de pull up internes sur les 8 bits du PORT B permettant ainsi de fixer les bits qui sont en entrée à l'état logique haut lorsqu'ils sont flottants (poussoir non actionné dans notre cas). Ces résistances ne sont pas en service à la mise sous tension.

→ Que doit-on faire pour activer la résistance de pull-up du bit RB₀ ? (cf DS p.17)

→ Quelle instruction peut-on utiliser pour tester l'état de RB₀ ?

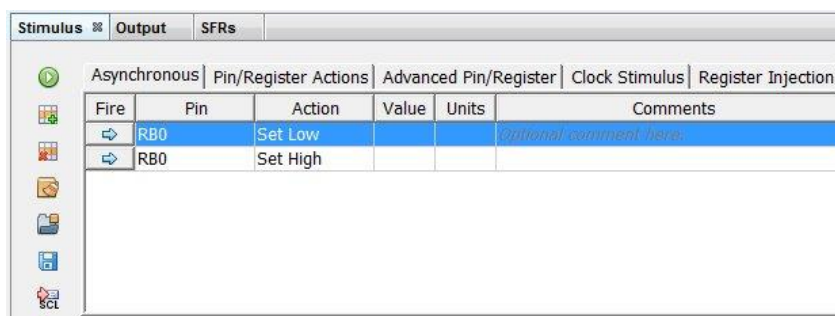
1) Ecrire le programme **BP.asm** à partir du fichier **exemple.asm**. On renommera ce dernier en le sélectionnant dans la fenêtre "FICHIERS" puis FILE > SAVE AS (**exemple.asm** devient donc **BP.asm** dans le répertoire **TP1.X** mais est toujours présent dans **TPs_PIC**).

2) Programmer le PIC sur la carte d'application et tester le programme

3) Simuler le programme (même si celui-ci fonctionne !) :


➤ On visualisera le contenu des registres SFR.

➤ On simulera l'appui sur le poussoir avec la commande WINDOWS > SIMULATOR > STIMULUS. Choisir l'onglet *asynchronous* pour activer une entrée à l'aide de la souris indépendamment du déroulement du programme. Définir la broche dans la colonne *Pin* ainsi que l'action à effectuer dans la colonne *Action*.



Un clic dans la colonne *Fire* effectue l'action correspondante immédiatement (l'affichage des registres n'est cependant mis à jour qu'après l'exécution de la prochaine instruction).

III) Programme CHENILLARD 1

- 1) Ouvrir le fichier **chen1.asm** fourni dans **TPs_PIC**. Observer les différentes parties de ce fichier et identifier le rôle des directives d'assemblage utilisées.
- 2) Ajouter ce nouveau fichier source au projet **TP1** et supprimer le lien sur le fichier **BP.asm** (clic droit + REMOVE FROM PROJECT).
Lancer le simulateur et mesurer la durée de la temporisation *tempo*. Pour cela, observer la fenêtre STOPWATCH en plaçant des points d'arrêt judicieusement (on peut aussi utiliser la commande  sur l'instruction *call tempo* pour exécuter d'un seul bloc la routine).
→ On pourra aussi observer la pile système (commande WINDOW > PIC MEMORY VIEWS > HARDWARE STACK) pour voir son utilisation par le processeur lors de l'appel de sous-programme.
- 3) Ecrire le programme principal pour qu'il fasse allumer l'une après l'autre une LED de gauche à droite puis de droite à gauche...etc sur la carte d'application. La durée d'allumage d'une LED sera calibrée par l'appel au sous-programme *tempo*. On utilisera un compteur ou bien une détection de "butée", au choix, et le décalage sera obtenu grâce aux instructions de rotation *rrf* et *rlf* (cf DS p.41).

ATTENTION :

- Le port A ne comporte que 5 bits, on utilisera donc par commodité une variable intermédiaire *LEDS* qu'on manipulera avec les 2 instructions de rotation et qui sera transférée dans le PORTA pour obtenir le décalage. Le registre *LEDS* pourra être codé en logique positive (bit à 1 = LED correspondante allumée) par simplicité, l'inversion en logique négative se faisant directement lors du transfert dans le PORTA si l'on utilise l'instruction **COMF LEDS, w** au lieu d'un classique **MOVF LEDS, w** avant transfert de W dans PORTA.
 - Les instructions *rrf* et *rlf* opèrent en réalité un décalage sur 9 bits car elles font entrer le bit de retenue C (bit 5 du registre STATUS) qu'il sera donc prudent de neutraliser à l'initialisation (la valeur de C étant inconnue au démarrage, cf DS p.7 – TABLE 2-1).
 - **Définir sur papier l'algorithme de ce programme** avant de se jeter sur le clavier !... ceci afin d'éviter de se retrouver ensuite à "bricoler" un code qui risque dès le départ d'être mal structuré (risque d'autant plus important en assembleur que la pauvreté du langage).
- 4) Compiler, programmer, tester. Effectuer une simulation en observant les registres *LEDS*, *PORTA* ainsi que la pile système. On pourra aussi s'aider de l'émulateur PICSIMLAB (voir *./~bedenes* onglet MICROCHIP) pour tester le programme si on ne dispose pas de la carte.

TP 2

- Objectifs
- Programmation en C
 - Gestion du *timer*

Nous allons désormais programmer en langage C pour plus de simplicité, tout en gardant une visibilité sur le code assembleur comme le permet MPLAB X.

I) Programme CHENILLARD1 (version C)

- 1) Créer un nouveau projet **TP2** en choisissant cette fois l'outil de compilation *XC8 (V1.30)* à l'étape 6. Ajouter ensuite les fichiers source **chen1.c** et **tempo.c** fournis et ouvrez-les.
 - Le fichier **tempo.c** contient uniquement une boucle de temporisation incluse dans la fonction *tempo ()*, de même durée que celle fournie dans **chen1.asm**.
 - Le fichier **chen1.c** est le fichier principal car contenant la fonction *main* indispensable et unique au sein d'un projet. Il faut logiquement lui associer les paramètres de configuration du PIC selon la même procédure qu'au TP1 :
- utiliser le bouton *Generate Source Code to Output* de la fenêtre CONFIGURATION BITS et intégrer le code généré au début du fichier.

XC8 est le compilateur de Microchip dédié aux PICs 8 bits. Il est compatible avec le standard C-ANSI mais ne permet pas la récursivité du fait de la taille de la pile système très limitée sur cette famille de microcontrôleurs. Il faudra aussi en pratique tenir compte du peu d'espace mémoire en RAM et fixer la taille des variables au plus juste, tout en limitant bien sûr l'usage de variables globales. Il gère par contre automatiquement les changements de pages mémoires (*BANK*) et possède en plus les spécificités liées à la programmation de microcontrôleurs : fonction d'interruption, macros, prise en charge du type *bit*.

- 2) Ouvrir le fichier **pic16f84a.h** situé dans le répertoire d'installation `\opt\Microchip\xc8\v1.30\include` afin de visualiser la syntaxe des symboles (registres, bits dédiés...).

Remarque : il est possible d'ouvrir automatiquement ce fichier et de visualiser la définition d'un symbole en pointant celui-ci dans le fichier source avec un clic droit puis *Navigate > Go to Declaration/Definition* (faire l'essai en tapant "PORTA" dans la fonction *main* de **chen1.c**)

En observant ce fichier, on remarque que pour désigner un bit (RA4 par exemple) on peut utiliser le type *bit* reconnu par XC8 (voir ligne 629). On écrira alors :

RA4 = 1; par exemple pour mettre à 1 le bit 4 du PORTA

Le type *bit* n'étant pas normalisé (standard ANSI) on préférera une écriture plus conventionnelle avec l'emploi de la structure **PORTAbits** dont le type est défini par **PORTAbits_t** (voir **p16f84a.h** lignes 160 à 169) et alloué en mémoire à l'adresse 0x05 (celle de PORTA). On écrira dans ce cas :

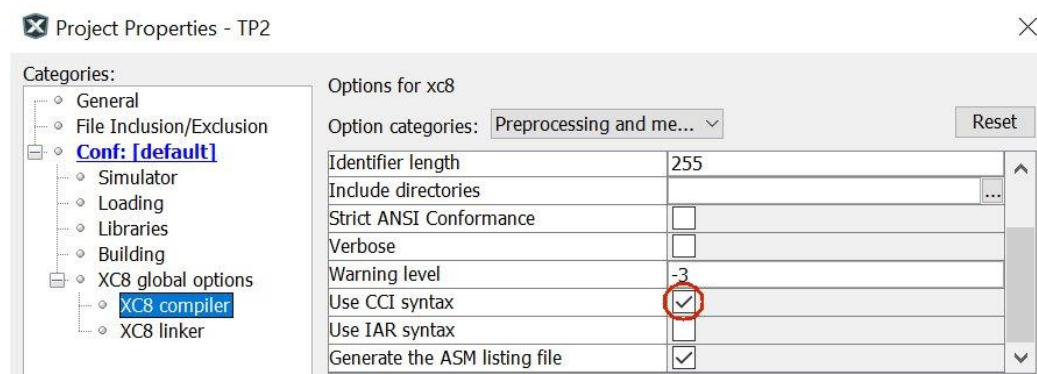
PORTAbits.RA4 = 1; pour effectuer la même opération

On voit que de toutes façons il sera nécessaire de regarder de près le fichier de définition **p16f84a.h** afin d'être en mesure d'accéder à un bit. Une dernière façon de faire plus universelle mais plus lourde consiste à utiliser un masque pour forcer à 0 ou 1 les bits désirés. Avec le même exemple, on écrirait :

PORTA = PORTA | 0b10000; pour fixer à 1 le bit 4 sans modifier les autres bits

Cette dernière méthode permet la meilleure portabilité du code mais sera développée en 2^e année car elle impose aussi certaines précautions (lisibilité du PORTA...??). On retiendra donc la 2^e solution avec l'emploi de structures et on prendra soin d'améliorer la portabilité du code en activant l'option CCI (Common C Interface) propre à tous les compilateurs XCxx de MICROCHIP. Celle-ci impose l'emploi de macros et directives communes qui facilitent par exemple la portabilité d'un code développé sur un PIC 8 bits (compilé avec XC8) vers un PIC 16 bits (compilateur XC16).

→ Activer l'option CCI dans les propriétés du projet **TP2**



2) Transposer en langage C le code assembleur des routines d'initialisation et du programme principal du fichier **chen1.asm** et l'écrire dans le fichier **chen1.c**.

➤ On adoptera le même algorithme que pour **chen1.asm** sachant que le décalage à gauche en C se fait avec l'instruction `<< 1` pour un décalage de 1 bit (c'est un décalage arithmétique donc il rentre forcément un 0 à droite d'où l'intérêt de travailler en logique `> 0` avec le registre *LEDS*).

LEDS <<= 1; effectue donc le décalage de 1 bit à gauche de LEDS

➤ le changement de pages mémoires (BANK) est géré automatiquement par le compilateur XC8 donc on peut accéder directement aux registres de la BANK1 (TRISA...) sans s'en préoccuper.

3) Compiler, programmer, tester en n'hésitant pas bien sûr à utiliser le simulateur en cas de difficultés.

On s'intéresse pour finir à quelques fonctionnalités intéressantes du simulateur.

4) Lancer le simulateur, le mettre en pause et effectuer un RESET.

→ Pointer la fonction *main* avec la souris puis clic droit > SHOW CALL GRAPH, la fenêtre *Call Graph* s'ouvre et présente sur un diagramme les liens entre fonctions (applicable bien sûr depuis toute fonction de départ). Ce graphe peut être très utile pour voir l'imbrication des appels dans des programmes plus complexes, de manière à prévenir un éventuel débordement de la pile système.

→ Actionner la commande WINDOW > DEBUGGING > DISASSEMBLY, la conversion en assembleur apparaît en-dessous de chaque instruction en C dans la fenêtre *assembly* qui s'ouvre. Effectuer une exécution pas à pas, elle se fait instruction par instruction au niveau assembleur (si on sélectionne la fenêtre du fichier source en C, elle se fait au niveau du code en C).

Par curiosité, comparer le code assembleur généré par le compilateur avec celui écrit au départ dans le fichier **chen1.asm**.

II) Programme **CHENILLARD2** (tempo avec *timer*)

Pour avoir une information sur le temps qui s'écoule, il est plus simple et précis d'utiliser un *timer* plutôt qu'une boucle logicielle. Le **TIMER0** du PIC peut fonctionner en compteur d'événements ou bien en *timer* classique incrémenté par les fronts d'horloge du PIC après division par 4 + *prescaler* (cf ANNEXE2 et DS p.19).

1) Par simplicité, rassembler les 2 fichiers précédents en un seul et sauvegardez-le sous le nom **chen2.c** (commande FILE > SAVE AS puis DISCARD pour ne pas modifier **chen1.c**). Ajouter cette nouvelle source au projet **TP2** et retirer les 2 anciennes.

2) Modifier la fonction de temporisation pour que celle-ci soit calibrée par le **TIMER0**.

- Après avoir analysé le fonctionnement du **TIMER0** on commencera par le configurer dans la fonction d'initialisation **init()** pour obtenir la période de débordement la plus longue possible (consulter le schéma p.44 du poly). On remarquera qu'il faudra faire déborder le *timer* plusieurs fois pour obtenir la même durée que dans **chen1.c**, même en réglant le *prescaler* sur le rapport de division le plus élevé (256). On utilisera donc un compteur de débordement en introduisant une variable **cmpt** qu'on incrémentera à chaque débordement repéré par le flag **TOIF**. On n'oubliera pas de compléter la fonction d'initialisation pour configurer convenablement le *timer*.
- La fonction de temporisation **tempo()** consistera ainsi à attendre le débordement du **TIMER0** en testant le passage à 1 du bit **TOIF** (attention, penser à le remettre à 0 après chaque débordement !) et à attendre un nombre de débordements suffisants (comptage avec la variable **cmpt** comme préconisé) avant de ressortir de la fonction **tempo()**.
- Le programme principal est identique à celui déjà écrit dans **chen1.c**

3) Tester le fonctionnement sur la carte et débbugger avec le simulateur si nécessaire.

TP 3

- Objectifs
- Programmation avec interruption
 - Utilisation du mode SLEEP, réveil par WATCHDOG
 - Utilisation de la mémoire EEPROM

I) Programme CHENILLARD3 (tempo avec *timer* déclenchant une interruption)

L'utilisation du *timer* pour temporiser apporte une précision et une simplicité au niveau de l'écriture du code. Mais le processeur est mal exploité puisqu'il ne fait qu'attendre le débordement du *timer* (scrutation de *TOIF*) sans pouvoir effectuer d'autres tâches. Le *TIMER0* étant capable de déclencher une interruption du processeur, il sera donc préférable d'utiliser ce procédé pour n'occuper le processeur qu'au moment du débordement du *timer*. On consultera l'annexe 2 pour obtenir des informations sur le concept d'interruptions et leur mise en œuvre sur les PICs 8 bits.

- 1) Créer un nouveau projet **TP3** et ajouter la source précédente **chen2.c** en la copiant dans le répertoire du projet (cocher la case *copy*). Renommer le fichier **chen2.c** sous le nom **chen3.c**.
- 2) Compléter la fonction d'initialisation pour autoriser les interruptions sur la source *TIMER0*. Ecrire la fonction d'interruption (voir modèle en commentaires) de manière à décaler la LED allumée tous les **n** débordements du *timer* (**n** à définir afin d'obtenir une temporisation d'environ 0,5 s). Le programme principal pourra ainsi continuer à se dérouler entre chaque débordement du *timer*. Pour matérialiser visuellement ce pseudo-parallélisme on fera exécuter dans le programme principal la routine principale du programme **BP.asm** du TP1 qu'on recopiera en utilisant les directives *#asm* et *#endasm* (voir exemple dans les illustrations de **chen1.c**). On notera de façon plus générale que les variables utilisées en assembleur doivent être déclarées en variables globales et précédées du préfixe **_** dans le code assembleur (*_cpt* dans l'exemple).
- 3) Tester le programme sur la carte en vérifiant que le symbole affiché évolue bien lorsqu'on appuie sur le poussoir, sans perturber le défilement régulier du chenillard.
- 4) Simuler le programme même s'il fonctionne de manière à visualiser le déroulement de l'interruption. On placera pour cela un point d'arrêt en début de fonction d'interruption.

- Observer notamment la valeur du registre *TMR0* et des bits *GIE*, *TOIF*.
- Visualiser la pile système et remarquer l'évolution du pointeur de pile et du PC (visible directement dans la barre d'outil rapide) en sortant de la fonction d'interruption (utiliser la commande d'exécution pas à pas).
- Observer dans la fenêtre STOPWATCH l'intervalle de temps entre 2 entrées dans la fonction d'interruption et vérifier qu'on obtient bien la valeur attendue.

II) Programme **CHENILLARD4** (tempo avec le "chien de garde")

Une dernière façon d'obtenir une temporisation, plus marginale, consiste à utiliser un autre *timer* interne du PIC : le *watchdog timer WDT*. Ce *timer* est particulier puisqu'il réinitialise le PIC lorsqu'il déborde. Ceci permet de réinitialiser l'application lorsque le programme reste en attente pendant une durée anormale (cf annexe 2). Le débordement du chien de garde permet aussi un réveil du PIC lorsque celui-ci est en mode veille commandé par l'instruction *SLEEP*. Cette possibilité peut être exploitée, si le PIC n'a rien d'autre à faire que d'attendre, pour obtenir une temporisation tout en minimisant la consommation électrique.

Le principe est simple :

- après le décalage d'une LED on placera le PIC en mode *SLEEP*. Le réveil du PIC sera ici provoqué par le débordement du chien de garde ou plutôt par celui du *prescaler* qui est utilisé ici en post-diviseur.
- on choisira une durée de débordement de l'ordre de 0,5 s en prenant comme référence 18 ms pour la période de débordement du *WDT*.

1) Ouvrir le fichier **chen2.c** déjà écrit et le sauvegarder sous le nom **chen4.c**. On l'associera au projet **TP2** à la place de **chen3.c**. Supprimer la fonction de temporisation et remplacer l'appel *tempo()* par l'instruction *asm ("SLEEP")*. Modifier la fonction d'initialisation pour affecter le *prescaler* au *WDT* avec le bon rapport de division. Ne pas oublier d'activer le chien de garde par *WDTE = ON* au lieu de *OFF* dans la section initiale de configuration.

2) Vérifier le bon fonctionnement du programme. A noter que le simulateur prend en charge le *WDT* (message "*W0004-CORE: Watchdog Timer has caused a Reset.*" affiché à chaque débordement dans l'onglet OUTPUT > Simulator de la fenêtre "RESULTATS", bien que le Reset n'ait pas toujours lieu), et l'instruction *SLEEP* cause un blocage de la simulation sur le début de l'instruction suivante tant qu'un des événements causant le réveil n'a pas lieu.

III) Programme **MESSAGE**

L'objectif de cette dernière partie est la synthèse des connaissances précédentes avec la mise en œuvre d'un programme plus complexe utilisant de plus une machine d'états logicielle. Il faudra faire preuve de méthode et rigueur et l'on commencera par définir l'algorithme du programme et le diagramme d'états avant de s'attaquer à la programmation. On voit également ici comment utiliser la mémoire EEPROM du PIC.

Le programme à concevoir doit permettre l'affichage d'un message de **n** caractères séquentiellement sur l'afficheur 7 segments. On souhaite pouvoir accélérer la vitesse d'affichage à chaque appui sur le bouton poussoir. La temporisation de défilement sera mémorisée dans l'EEPROM de données du PIC 16F84a; ainsi l'affichage reprendra à une même cadence après une coupure de l'alimentation (l'intérêt est bien sûr ici de voir comment contrôler de façon logicielle la mémoire EEPROM du PIC !).

- Le message sera stocké dans l'EEPROM au moment de la programmation grâce à la directive `__EEPROM_DATA` (`DATA 1, DATA 2, ..., DATA 8`) qui permet de programmer des octets 8 par 8. On codera les caractères à afficher par l'octet : **(a b c d e f g 1)₂** de manière à pouvoir charger directement cette valeur dans le port B. Le bit de poids faible vaut 1 (indifférent pour RB₀ configuré en entrée) ce qui permettra de réserver le code 0x00 pour définir un caractère de fin de chaîne (= code ASCII de '\0').
- La temporisation sera obtenue à partir du débordement du *TIMER0* qui génèrera une interruption. Le changement de caractère ne sera effectué que tous les 16 débordements de manière à obtenir un intervalle de temps d'environ 1 seconde au démarrage (*prescaler* fixé initialement à 1/256).
- L'appui sur le poussoir influera sur la vitesse en modifiant le rapport de division du *prescaler* (1/256, 1/128, ... 1/2, 1/256, ...), ce qui provoquera consécutivement la mise à jour de sa valeur de sauvegarde dans l'EEPROM. La détection de l'appui se fera par scrutation du bit RB0 dans le programme principal (l'opération d'écriture en EEPROM bloquant le processeur durant 10 ms, cette opération ne doit donc pas être exécutée dans la routine d'IT).

Un fichier source **message.c** préformé est disponible dans le répertoire **TPs_PIC**, dans lequel la ligne `__EEPROM_DATA` charge l'EEPROM avec une 1^{ère} donnée prévue pour être directement stockée initialement dans le registre *OPTION* et les 6 données suivantes correspondent aux caractères du message "**16F84** " (la 8^e donnée est nulle et pourra éventuellement être utilisée comme caractère de fin de chaîne afin de pouvoir gérer des messages de taille variable).

Le développement de ce programme se fera de façon incrémentale :

- ① on écrira une 1^{ère} version qui permettra de valider le bon fonctionnement de la temporisation et l'affichage correct du message (pas de test du poussoir ni d'écriture en EEPROM pour le moment).
- ② on ajoute ensuite la boucle de scrutation du bit *RBO* suivi de l'écriture en EEPROM. On constate à l'exécution de ce programme un problème prévisible : la vitesse change de plusieurs paliers en actionnant le poussoir puisque d'une part on n'a pas neutralisé les rebonds (durée de quelques ms à l'appui et au relâchement) mais aussi parce que l'on teste l'état d'appui et non l'action d'appuyer comme il le faudrait.
- ③ on corrige le défaut précédent en utilisant une machine d'état logicielle permettant de ne prendre en compte l'appui que si *RBO* était auparavant inactif. Le changement d'état ne pourra se faire qu'à chaque passage dans la boucle du programme principal dont la durée ne devra pas être inférieure à 10 ms environ pour ne pas être gêné par les rebonds (cette tempo sera obtenue par une boucle selon le modèle de la fonction *tempo* du programme **chen1.c**).

Remarque :

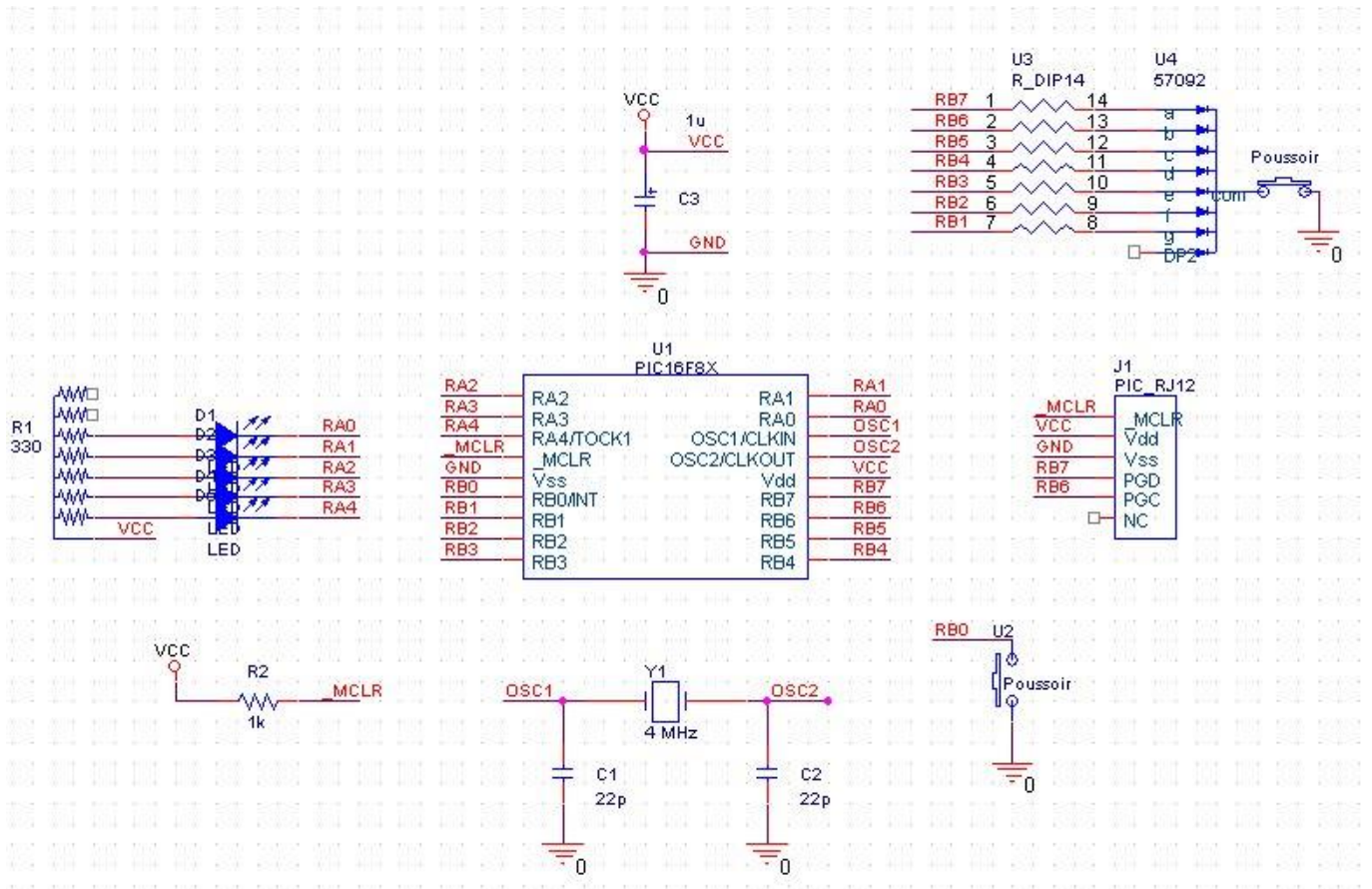
Bien que le diagramme d'états soit très simple, on commencera par le dessiner sur papier avant de commencer à programmer cette machine. Sa transposition en langage C se fera avec une affectation sélective, à l'image de sa description en VHDL, en utilisant de préférence l'instruction *switch* pour plus de lisibilité :

```
switch ( etat )
{
  case etat_1 : .....; break;
  case etat_2 : .....; break;
  .....
  case etat_n : .....; break;
}
```

On pourra aussi avantageusement définir un nouveau type énuméré (nommé par exemple ETATS) pour dissocier les différents états grâce à l'instruction *enum* :

```
enum ETATS (etat_1, ..., etat_n) etat;
```


ANNEXE 1 : schéma de la carte



ANNEXE 2 : Présentation des PICs

I) DESCRIPTION

1) Architecture	28
2) Différentes familles	28
3) Espace mémoire	30

II) PROGRAMMER AVEC LES PICS

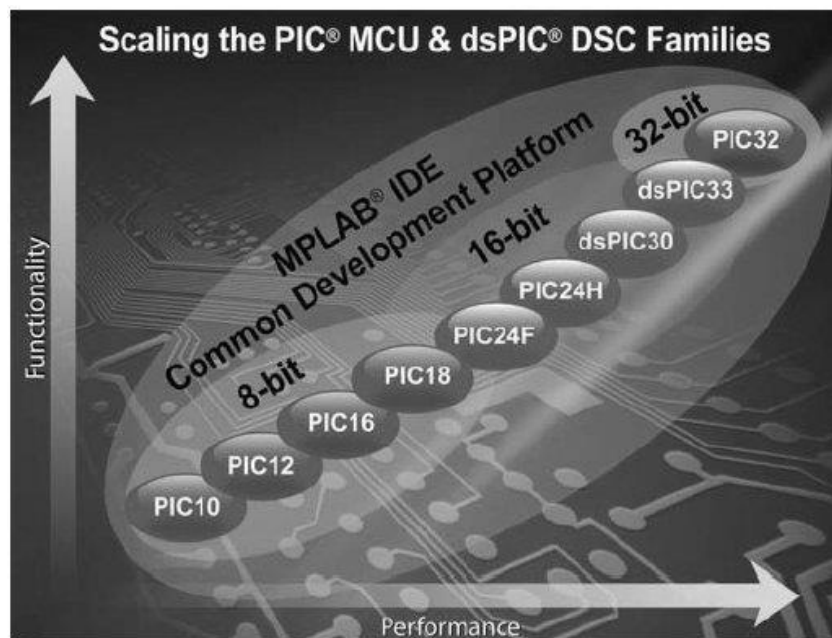
1) Principaux registres	33
2) Modes d'adressage	36
3) Types d'instruction	37
4) Jeu d'instructions	38
5) Déroulement d'une instruction	39

III) ETUDE DU PIC 16F84

1) Description rapide	41
2) Registres SFR	42
3) Architecture interne	43
4) Les ports d'entrée / sortie A et B	43
5) Le <i>timer</i> TIMER 0	44
6) Les interruptions	47
7) Le chien de garde	52
8) Le mode SLEEP	53
9) Les accès en mémoire EEPROM	54

MICROCHIP propose une gamme importante de microcontrôleurs sous l'appellation "PIC", depuis le début des années 90. Il a bâti son succès grâce à son PIC 8 bits, disponible en de multiples versions, permettant ainsi à l'utilisateur de disposer d'un petit microcontrôleur RISC avec des ressources internes répondant au mieux à son application, pour un prix taillé au plus juste (à partir de 0,3 E !). On trouve ainsi des PICs incorporant des ressources élémentaires telles que EEPROM, *timers*, contrôleurs de bus série, etc ..., mais aussi des modules plus évolués permettant le contrôle d'écran tactile, la transmission de données par liaison RF... Les PICs peuvent aussi intégrer des fonctions analogiques telles que CAN, ampli OP, référence de tension, PLL, oscillateurs. Il n'est donc pas étonnant de retrouver ces produits dans bon nombre de systèmes embarqués grand public.

MICROCHIP a progressivement étoffé sa gamme de produits avec des microcontrôleurs 16 et 32 bits ainsi que des DSP et microprocesseurs.



Il fabrique également un grand nombre de circuits intégrés (capteurs, fonctions analogiques variées...) destinés à être interfacés avec ses microcontrôleurs... (voir illustration du constructeur en dernière page). En 2016 Microchip a racheté son principal concurrent, ATMEL avec ses microcontrôleurs AVR, étendant encore un peu plus sa gamme de références. Le site du constructeur www.microchip.com illustre bien la diversité de ses produits et la volonté de proposer des solutions "clés en main" dans un domaine d'application donné.

Dans cette présentation, nous nous limiterons à l'étude des PICs 8 bits et plus précisément la famille PIC16 utilisée dans le cadre de l'activité pratique du module EN111 : 16F84a pour la partie initiation, et 16F877a en ce qui concerne le projet.

I) DESCRIPTION

Il y a plusieurs centaines de références de PICs 8 bits, disponibles en boîtiers de 6 à 100 broches. Cette catégorie, aujourd'hui encore la plus répandue, couvre en effet un grand domaine d'applications par la variété des ressources internes disponibles, limitées tout de même en performance par la puissance de calcul restreinte du processeur 8 bits.

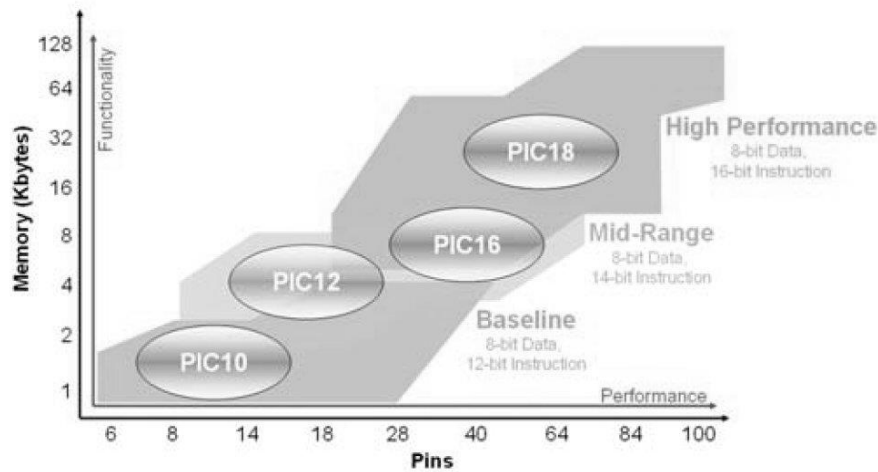
1) Architecture

Les PICs dits "8 bits" sont bâtis autour d'un processeur 8 bits, c'est-à-dire que les registres et le bus de données internes ont un format de 8 bits. Ils ont une architecture Harvard et possèdent donc un bus "programme" distinct du bus de données ce qui leur permet d'accéder et manipuler une donnée et simultanément aller lire l'instruction suivante. Ce parallélisme est possible grâce à un pipeline à 2 niveaux (voir section III-5 "déroulement d'une instruction").

Le processeur est de type RISC (Reduced Instruction Set Computer) et ne possède donc que peu d'instructions (35 pour les 2 PICs que nous utiliserons). Cette caractéristique permet de les coder dans un seul emplacement mémoire pour chaque instruction, ce qui permet un accès et un décodage plus rapide. Une instruction s'exécute ainsi en 1 cycle machine soit 4 périodes d'horloge. Les PICs permettent ainsi une cadence de plusieurs MIPS (Million Instructions Per Second), jusqu'à 5 MIPS pour la série PIC16 que nous utiliserons. La contrepartie de ce gain de rapidité se situe au niveau de la simplicité des instructions qui nécessite d'en associer souvent plusieurs pour réaliser une opération même basique. Un autre inconvénient concerne l'accès aux registres plus complexe puisqu'on ne code qu'une partie de leur adresse dans le champ de l'instruction afin de limiter la taille des emplacements de la mémoire programme, l'autre partie définissant une page mémoire ("banque") sélectionnée au moyen d'un registre spécifique.

2) Différentes familles

Les PICs 8 bits sont découpés en 4 séries (PIC10, PIC12, PIC16 et PIC18) classées par "puissance" en termes de ressources internes, d'entrées/sorties et, ce qui est étroitement lié, en capacité de mémoire programme d'autant plus étendue qu'il y aura de ressources à contrôler. Les PICs ont une pérennité réduite afin de répondre aux besoins du marché, certaines gammes disparaissant (PIC14, PIC16C5...) remplacées par de nouvelles généralement compatibles et plus performantes. Il est bien clair que les PICs sont avant tout destinés aux produits grands publics à faible durée de vie plutôt qu'aux applications militaires...



MICROCHIP a défini 3 classes d'architecture pour ses PICs 8 bits, qui se distinguent par la taille des instructions :

Baseline : instructions codées sur 12 bits

Mid-Range et *Enhanced Midrange* : instructions codées sur 14 bits

High performance (ou *high end*) : instructions codées sur 16 bits

Une taille plus élevée permet un plus grand nombre d'instructions mais surtout d'augmenter le champ alloué à l'adresse du registre à manipuler et donc de disposer de plus de registres. Ceux-ci sont effet d'autant plus nombreux qu'il y a de ressources (entrées/sorties, *timer*...) à contrôler.

8-bit PIC[®] Architectures

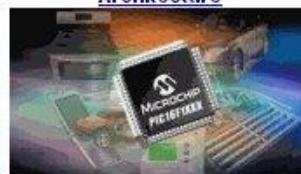
Baseline Architecture



Midrange Architecture



Enhanced Midrange Architecture



PIC18 Architecture



Pin count	6 – 40	8 – 64	8 – 64	18 – 100
Interrupts	No	Single Interrupt Capability	Single Interrupt Capability with Hardware Context Save	Multiple Interrupt Capability w Hardware Context Save
Operating Performance	5 MIPS	5 MIPS	8 MIPS	10 – 16 MIPS
Instructions	33, 12-bit instructions	35, 14-bit instructions	49, 14-bit instructions	75 - 83, 16-bit instructions
Program Memory	Up to 3 KB	Up to 14 KB	Up to 56 KB	Up to 128 KB
Data Memory	Up to 138 Bytes	Up to 368 Bytes	Up to 4 KB	Up to 4 KB
Features	<ul style="list-style-type: none"> • Smallest form factor • Lowest cost • Ideal for battery operated or space constrained applications • Easy to learn & use 	<ul style="list-style-type: none"> • Optimal cost-to-performance ratio • Integrated peripherals including SPI, I²C™, UART, LCD, ADC 	<ul style="list-style-type: none"> • C-code Optimized • Enhanced 16 Level Hardware Stack • Enhanced Indirect Addressing • Reduced Interrupt Latency • Simplified Memory Map 	<ul style="list-style-type: none"> • 32 level deep stack, 8x8 hardware multiplier • C-code optimized • Advanced peripherals including CAN, USB, Ethernet touch sensing, and LCD drive
Families	Includes PIC10 , PIC12 and PIC16	Includes PIC12 and PIC16	Includes PIC12F1xxx & PIC16F1xxx	PIC18 J-series for cost-sensitive applications w high levels of integration

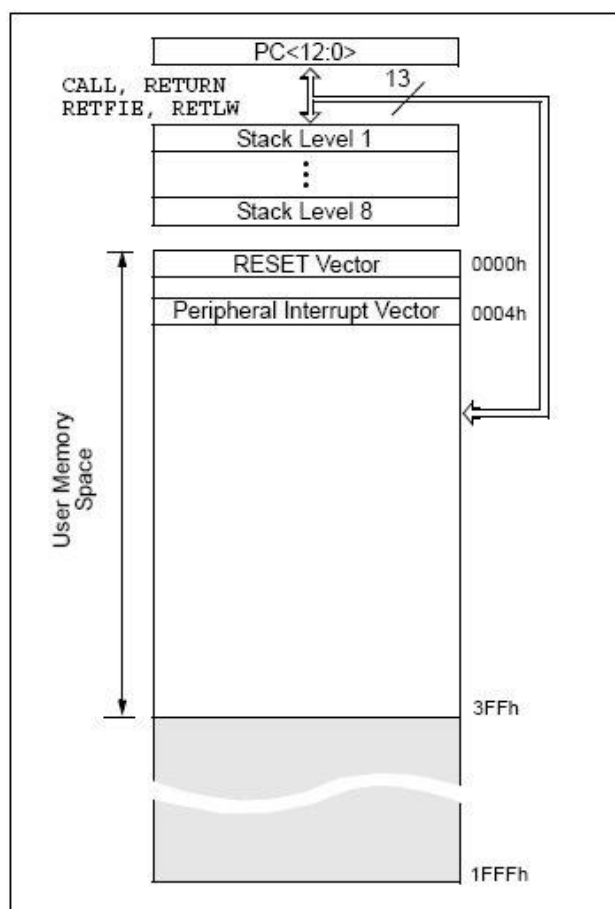
Les 2 PICs étudiés dans le module EN111 font partie de la famille intermédiaire *mid-range* qui s'adresse essentiellement aux produits grand public. Nous nous limiterons à l'exemple du PIC16F84 pour définir l'espace mémoire de programme et de données, les autres références étant calquées sur la même organisation, avec plus ou moins d'emplacements.

3) Espace mémoire

a) Mémoire code programme

Il s'agit d'une mémoire de type FLASH (= EEPROM à accès rapide) sur les PICs de type F tel que le 16F84. On trouve aussi des versions industrielles OTPROM (One Time Programming ROM), et ROM (type CR) programmé en usine par MICROCHIP.

Mémoire programme PIC 16F84



Le PIC 16F84 possède une mémoire FLASH de 1 Kmots (= 1024 mots) de 14 bits et l'espace maximum adressable pour la famille *midrange* est de 8 Kmots (cas du 16F877). Le compteur programme PC (Program Counter) est donc un registre de 13 bits dont seuls les 10 bits de poids faibles sont utiles sur le 16F84. Le registre PC contient à tout instant l'adresse de l'instruction qui sera exécutée au prochain cycle d'horloge.

La pile ("Stack") est une zone mémoire qui sert à stocker des adresses utiles au déroulement du programme. Par exemple lorsqu'on effectue un appel de sous-programme (ou appel de fonction en langage C) l'adresse de l'instruction courante est stockée automatiquement dans la pile permettant ainsi de reprendre le déroulement du programme après cette instruction une fois le sous-programme exécuté. Le processeur effectue pour cela des copies PC→pile (appel) ou pile→PC (retour). La pile est une mémoire à accès séquentiel de type LIFO (Last In First Out) dont le nom fait référence à une pile d'assiette : la dernière posée sera la première à être reprise. Cette zone mémoire n'est pas accessible par l'utilisateur sur les PICs, et ne contient que 8 emplacements ce qui est très peu. Toute récursivité est pour cela interdite par les compilateurs C. La pile étant ici circulaire, la 9^e écriture écrase la 1^{ère} définitivement perdue.

2 adresses sont particulières pour les PICs :

0000h : adresse de l'instruction qui sera exécutée à la mise sous tension ou après un RESET

0004h : adresse de l'instruction qui sera exécutée si une interruption survient (voir partie ...)

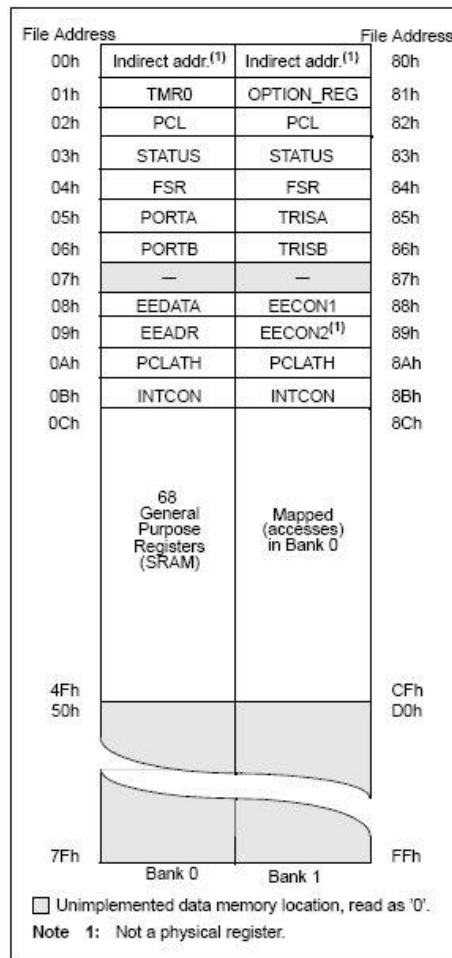
Remarque : Microchip utilise le terme "vecteur" pour désigner ces 2 emplacements ce qui est un abus de langage car un vecteur est en principe un pointeur en microinformatique, c'est-à-dire qu'il contient l'adresse de la case mémoire qu'il pointe. Les PICs accèdent à ces 2 adresses de la même manière que pour tout autre emplacement en mémoire programme, c'est-à-dire qu'il décode la valeur lue pour obtenir une instruction. En pratique, on place à ces 2 adresses une instruction *GOTO add* où *add* désigne l'adresse des routines RESET et ISR (Interrupt Service Routine).

Certains PICs plus haut de gamme (à partir de la série PIC24 – MCU de 16 bits) possèdent une table de vecteurs dédiées aux interruptions, permettant ainsi l'exécution d'autant de fonctions d'interruptions différentes liées à des sources de déclenchement spécifiques, chaque vecteur pointant sur l'adresse du point d'entrée de la fonction correspondante.

b) Mémoire données (8 bits)

① RAM

Mémoire de données du PIC 16F84



Comme on peut le voir avec l'exemple du PIC 16F84 la RAM est divisée en pages mémoires de 128 octets (soit 7 bits d'adresse) que MICROCHIP désigne par le terme "banques" (2 banques BANK0 et BANK1 sur le PIC16F84). C'est une des grandes particularités des PICs et une conséquence directe du type RISC de leur processeur intégré dans une architecture à faible coût. En effet, les instructions sont peu nombreuses dans le but de chacune pouvoir les coder sur un seul emplacement de la mémoire programme, et la taille commune de toutes instructions a donc été volontairement restreinte (14 bits sur le PIC16F84) dans un souci d'économie et tout en respectant les exigences de l'ensemble des instructions. Cela induit par contre une limitation sur l'espace (champ de 7 bits sur le PIC16F84) alloué dans le code de l'instruction pour spécifier l'adresse d'un registre utilisé comme argument.

Les registres sont donc divisés en 2 banques (voire 4 sur certains PICs) sélectionnables par des bits supplémentaires qui correspondraient aux bits de poids fort de l'adresse complète : RP0 (+RP1) du registre **STATUS** ou directement par un registre dédié (BANK REGISTER sur les PICs de la famille *High performance*).

La présélection de la bonne banque est donc une contrainte à ne pas oublier avant l'accès à un registre en RAM sous peine de ne pas manipuler le bon registre !

La RAM contient les registres à fonction dédiée (SFR) qui permettent le contrôle du processeur ou des périphériques (les SFRs les plus utilisés sont situés en BANK0 qui est la banque de base "par défaut"). A la suite de ces registres se trouve de la RAM utilisable pour stocker des variables "GPR" (General Purpose Registers).

On peut remarquer que certains registres SFR sont accessibles dans toutes les banques (le *STATUS register* puisqu'il permet de changer de banque + quelques autres), c'est aussi le cas pour les 68 GPRs du PIC16F84 (attention, ce n'est pas vrai avec tous les PICs).

② EEPROM (selon circuit)

Certains PICs disposent d'une mémoire EEPROM pour stocker des constantes accessibles au moment de la programmation mais aussi par le programme. Attention, cette dernière procédure est assez lourde (surtout en écriture) et le temps d'accès en écriture est relativement long (≈ 10 ms). Ces octets ne sont donc pas à utiliser pour stocker des variables temporaires évoluant souvent dans le programme.

Le PIC 16F84 dispose de 64 octets situés à partir de l'adresse 0x2100

II) PROGRAMMER AVEC LES PICS

1) Principaux registres

Certains registres sont indispensables au déroulement du programme.

a) Le registre de travail W (Working register)

Le processeur ne peut déplacer une donnée ou effectuer une opération arithmétique ou logique sur un registre (SFR ou GPR) qu'en utilisant le registre de travail W. C'est un registre 8 bits indépendant des registres SFR et qui est directement associé à l'ALU (Arithmetic Logic Unit). On le qualifie aussi d'*accumulateur* car, comme on peut le remarquer sur le schéma de l'architecture interne (cf partie III-3 pour le PIC16F84a), il permet de réinjecter directement dans l'ALU le résultat de l'opération précédente ce qui réduit le temps de calcul des opérations portant sur plus de 2 opérandes (addition de 3 nombres par exemple).

b) Le registre d'état **STATUS** (0x03/0x83)

C'est un registre qui fournit de manière générale des indications liées à l'exécution de la dernière instruction par le processeur. On y trouve en particulier les bits d'état qui renseignent sur la valeur de la dernière opérande (=donnée manipulée par une instruction). Comme déjà évoqué, ce registre contient également les bits qui permettent de sélectionner les différentes pages mémoires en RAM ("banques").

STATUS REGISTER (ADDRESS 03h, 83h)

R/W-0	R/W-0	R/W-0	R-1	R-1	R/W-x	R/W-x	R/W-x
IRP	RP1	RP0	\overline{TO}	\overline{PD}	Z	DC	C
bit 7							bit 0

C (b0) : Carry (report).	Bit de retenue, correspond au 9 ^{ème} bit de l'opération.
DC (b1) : Digit Carry.	Utilisé lorsque l'on travaille avec des nombres BCD : il indique un report du bit 3 vers le bit 4 dans une opération.
Z (b2) : Zero.	Vaut 1 si le résultat de la dernière opération vaut 0.
/PD (b3) : Power down	Indique quel événement a entraîné le dernier arrêt du PIC (instruction sleep ou dépassement du temps du watchdog).
/TO (b4) : Time-Out 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.
RP0 (b5) : Register Bank Select 0	Sélectionne la banque de RAM qui sera utilisé au prochain accès (0 = banque 0).
RP1 (b6) : Register Bank Select 1	Permet la sélection des banques 2 et 3. Inutilisé pour le 16F84, il doit être laissé à 0 pour garantir la compatibilité ascendante (portabilité du programme).
IRP (b7) : Indirect RP	Permet de décider quelle banque on adresse dans le cas de l'adressage indirect pour les PICs disposant de plus de 2 banques (inutile pour le 16F84).

c) Le compteur programme PC

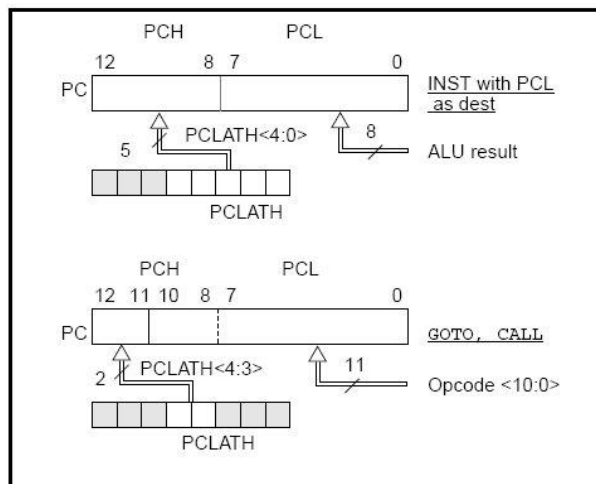
Il pointe sur la prochaine instruction à exécuter. Comme il est codé sur 13 bits (accès à toute la mémoire de codes programmes) il nécessite d'être représenté par 2 registres :

PCL	(PC Low)	(0x02/0x82) :	8 bits de poids faible
PCLATH	(PC LATch counter High)	(0x0A/0x8A) :	5 bits de poids fort b4 à b0

NB : pour le 16F84, 1K mots de mémoire programme \Rightarrow b4, b3, b2 de PCLATH non utilisés.

Le compteur programme est normalement incrémenté d'une unité après l'exécution de chaque instruction du programme, sauf si celle-ci agit directement sur lui-même. C'est le cas avec les instructions de test (modification automatique du PC selon résultat du test) et les instructions de branchement qui spécifient dans leur argument l'adresse de la prochaine instruction qui devra être exécutée :

Chargement du PC dans différentes situations



Comme le montre cette figure, le compteur programme peut en effet être modifié directement par des instructions spécifiques telles que les déroutements (GOTO) ou les appels de sous-programme (CALL). Mais attention, la totalité des bits définissant la nouvelle adresse du PC ne pouvant être codée en une seule fois dans les 14 bits d'une instruction, il faudra peut-être définir les bits de poids forts par un accès spécifique à PCLATH.

2) Modes d'adressage

Le mode d'adressage désigne la manière de désigner la donnée à manipuler (opérande). Il n'existe que 3 modes d'adressage sur les PICs, qui sont les plus usuels sur les processeurs :

- literal (= immédiat) : la valeur de l'opérande est directement donnée dans l'instruction

ex: MOVLW k *charge la valeur k dans W*

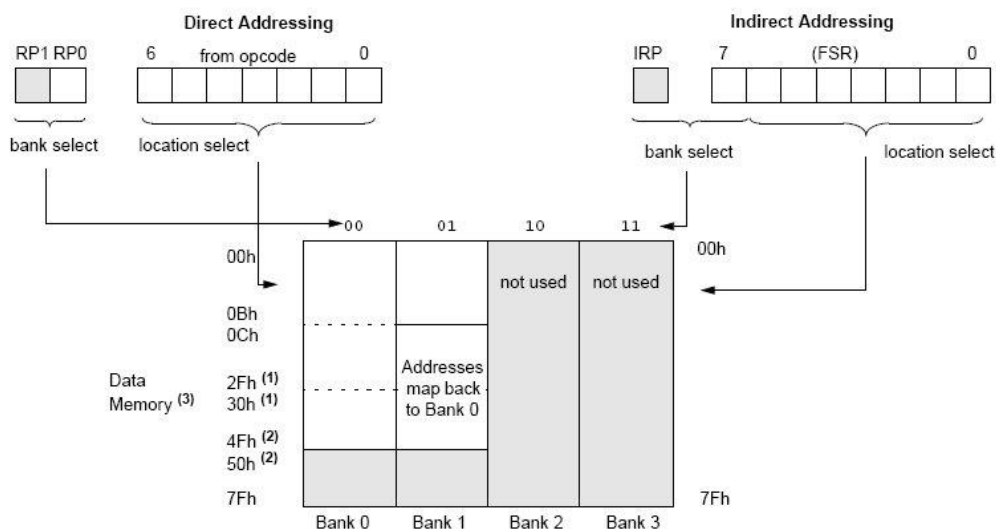
A noter que ce mode d'adressage n'est utilisable qu'avec W car la taille réduite des instructions ne permet pas de coder sur ses 14 bits l'adresse d'un registre en RAM en plus des 8 bits de la donnée k (cf partie suivante)

- direct : l'opérande est le contenu de l'adresse (registre) spécifié

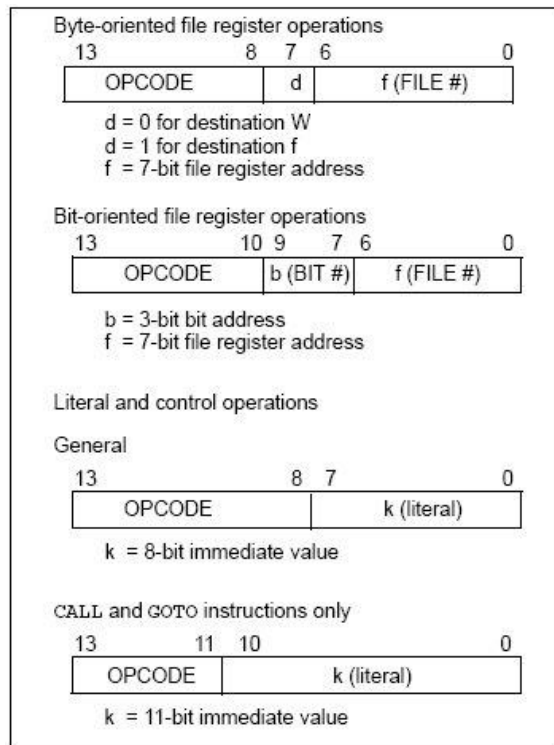
ex: MOVWF 0x05 charge le contenu de W dans l'adresse 05h
 (= registre PORTA si banque 0)

- indirect : l'opérande est le contenu de la case mémoire dont l'adresse est elle-même contenue dans le registre spécifié (= pointeur en langage C). Sur les PICs l'utilisation de ce mode d'adressage est particulier : le pointeur est unique, il s'agit du registre FSR (0x04/0x84). Pour utiliser ce mode d'adressage on fait appel dans l'instruction à un registre spécial, le registre INDF (0x00/0x80) qui n'existe pas vraiment.

<u>ex</u> :	MOVLW	0x05	<i>charge la valeur 05h (adresse de PORTA) dans W</i>
	MOVWF	0x04	<i>transfert la valeur 05h dans le registre FSR (on a ainsi initialisé le pointeur FSR sur PORTA)</i>
	MOVF	INDF,W	<i>effectue l'accès indirect via le pointeur FSR (le contenu de PORTA est copié dans W)</i>



3) Types d'instruction



a) Instruction qui porte sur 1 octet : manipule l'octet contenu dans le registre f dont l'adresse est spécifiée sur 7 bits (= taille du bus d'adresse de la RAM) en accord avec la banque sélectionnée.

Remarque : le bit d (destination) désigne le registre qui sera utilisé pour stocker le résultat de l'opération avec seulement 2 possibilités : soit W (si $d=0$), soit le registre source f (si $d=1$) puisque la taille réduite de l'instruction ne permet pas de définir l'adresse d'un registre supplémentaire.

b) Instruction qui porte sur un bit : manipule directement le bit numéro b parmi les 8 bits d'un registre f spécifié.

c) Instruction literal : manipule des données codées dans l'instruction même (adressage immédiat).

d) Sauts et appel de sous-programme : 3 bits pour coder l'instruction, les 11 bits restant pour coder l'adresse de destination. Si la mémoire programme est supérieure à 2 K mots (16F877...) on positionne les bits 3, 4 de **PCLATH**.

4) Jeu d'instructions (exemple de la famille *mid-range* avec 35 instructions)

Mnemonic, Operands	Description	Cycles	14-Bit Opcode				Status Affected	Notes	
			MSb		LSb				
BYTE-ORIENTED FILE REGISTER OPERATIONS									
ADDWF	f, d	Add W and f	1	00	0111	dfff	ffff	C,DC,Z	1,2
ANDWF	f, d	AND W with f	1	00	0101	dfff	ffff	Z	1,2
CLRF	f	Clear f	1	00	0001	1fff	ffff	Z	2
CLRW	-	Clear W	1	00	0001	0xxx	xxxx	Z	
COMF	f, d	Complement f	1	00	1001	dfff	ffff	Z	1,2
DECf	f, d	Decrement f	1	00	0011	dfff	ffff	Z	1,2
DECFSZ	f, d	Decrement f, Skip if 0	1 (2)	00	1011	dfff	ffff		1,2,3
INCF	f, d	Increment f	1	00	1010	dfff	ffff	Z	1,2
INCFSZ	f, d	Increment f, Skip if 0	1 (2)	00	1111	dfff	ffff		1,2,3
IORWF	f, d	Inclusive OR W with f	1	00	0100	dfff	ffff	Z	1,2
MOVF	f, d	Move f	1	00	1000	dfff	ffff	Z	1,2
MOVWF	f	Move W to f	1	00	0000	1fff	ffff		
NOP	-	No Operation	1	00	0000	0xx0	0000		
RLF	f, d	Rotate Left f through Carry	1	00	1101	dfff	ffff	C	1,2
RRF	f, d	Rotate Right f through Carry	1	00	1100	dfff	ffff	C	1,2
SUBWF	f, d	Subtract W from f	1	00	0010	dfff	ffff	C,DC,Z	1,2
SWAPF	f, d	Swap nibbles in f	1	00	1110	dfff	ffff		1,2
XORWF	f, d	Exclusive OR W with f	1	00	0110	dfff	ffff	Z	1,2
BIT-ORIENTED FILE REGISTER OPERATIONS									
BCF	f, b	Bit Clear f	1	01	00bb	bfff	ffff		1,2
BSF	f, b	Bit Set f	1	01	01bb	bfff	ffff		1,2
BTFSC	f, b	Bit Test f, Skip if Clear	1 (2)	01	10bb	bfff	ffff		3
BTFSS	f, b	Bit Test f, Skip if Set	1 (2)	01	11bb	bfff	ffff		3
LITERAL AND CONTROL OPERATIONS									
ADDLW	k	Add literal and W	1	11	111x	kkkk	kkkk	C,DC,Z	
ANDLW	k	AND literal with W	1	11	1001	kkkk	kkkk	Z	
CALL	k	Call subroutine	2	10	0kkk	kkkk	kkkk		
CLRWDT	-	Clear Watchdog Timer	1	00	0000	0110	0100	$\overline{TO}, \overline{PD}$	
GOTO	k	Go to address	2	10	1kkk	kkkk	kkkk		
IORLW	k	Inclusive OR literal with W	1	11	1000	kkkk	kkkk	Z	
MOVLW	k	Move literal to W	1	11	00xx	kkkk	kkkk		
RETFIE	-	Return from interrupt	2	00	0000	0000	1001		
RETLW	k	Return with literal in W	2	11	01xx	kkkk	kkkk		
RETURN	-	Return from Subroutine	2	00	0000	0000	1000		
SLEEP	-	Go into standby mode	1	00	0000	0110	0011	$\overline{TO}, \overline{PD}$	
SUBLW	k	Subtract W from literal	1	11	110x	kkkk	kkkk	C,DC,Z	
XORLW	k	Exclusive OR literal with W	1	11	1010	kkkk	kkkk	Z	

Note 1: When an I/O register is modified as a function of itself (e.g., `MOVF PORTB, 1`), the value used will be that value present on the pins themselves. For example, if the data latch is '1' for a pin configured as input and is driven low by an external device, the data will be written back with a '0'.

2: If this instruction is executed on the TMR0 register (and, where applicable, d = 1), the prescaler will be cleared if assigned to the Timer0 Module.

3: If Program Counter (PC) is modified or a conditional test is true, the instruction requires two cycles. The second cycle is executed as a `NOOP`.

Field	Description
f	Register file address (0x00 to 0x7F)
w	Working register (accumulator)
b	Bit address within an 8-bit file register
k	Literal field, constant data or label
x	Don't care location (= 0 or 1) The assembler will generate code with x = 0. It is the recommended form of use for compatibility with all Microchip software tools.
d	Destination select; d = 0: store result in W, d = 1: store result in file register f. Default is d = 1
PC	Program Counter
TO	Time-out bit
PD	Power-down bit

5) Déroulement d'une instruction

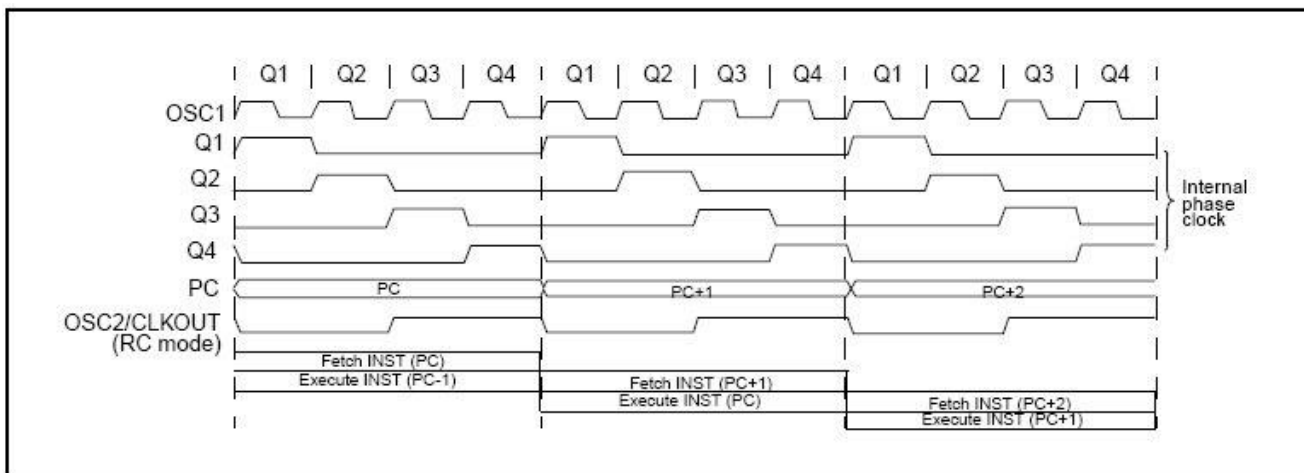
La fréquence du signal d'horloge externe est divisée par 4 à l'intérieur du PIC pour obtenir 4 horloges internes en quadrature nommées Q1, Q2, Q3 et Q4 qui délimitent en fait les 4 phases de l'exécution d'une instruction. Pour commencer le PC est incrémenté par Q1 et l'instruction est ensuite chargée dans le registre instruction par Q4 (délai nécessaire à cause du temps d'accès de la mémoire programme). Le cycle suivant effectue l'exécution de l'instruction avec, dans le cas le plus fréquent :

Q1 : décodage de l'instruction

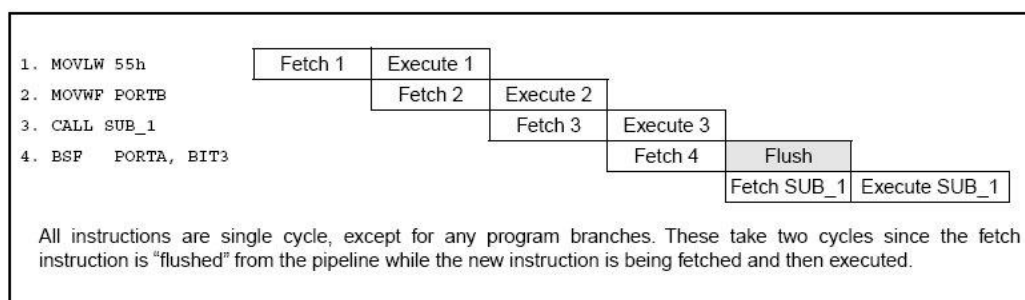
Q2 : lire l'opérande (W, registre en RAM, donnée immédiate...)

Q3 : effectuer un éventuel traitement (addition, ...)

Q4 : stocker le résultat (dans l'opérande source, W...)



Une instruction nécessite donc 1 cycle pour que le PIC aille la chercher en mémoire programme et un autre pour l'exécuter. Un pipeline à 2 niveaux permet de mener ces 2 types d'opération en parallèle : dans un même cycle de 4 périodes d'horloge, le PIC va chercher l'instruction n tout en exécutant l'instruction $n-1$ pré-chargée durant le cycle précédent. Une instruction s'exécute donc en un seul cycle, sauf lorsqu'il y a déroutement (changement du PC par une instruction, cf exemple ci-dessous) où il faut rajouter un cycle "mort" pour charger la nouvelle instruction avant son exécution.



III) ETUDE DU PIC 16F84



PIC16F84A

18-pin *Enhanced* FLASH/EEPROM 8-Bit Microcontroller

High Performance RISC CPU Features:

- Only 35 single word instructions to learn
- All instructions single-cycle except for program branches which are two-cycle
- Operating speed: DC - 20 MHz clock input
DC - 200 ns instruction cycle
- 1024 words of program memory
- 88 bytes of Data RAM
- 64 bytes of Data EEPROM
- 14-bit wide instruction words
- 8-bit wide data bytes
- 15 Special Function Hardware registers
- Eight-level deep hardware stack
- Direct, indirect and relative addressing modes
- Four interrupt sources:
 - External RB0/INT pin
 - TMR0 timer overflow
 - PORTB<7:4> interrupt-on-change
 - Data EEPROM write complete

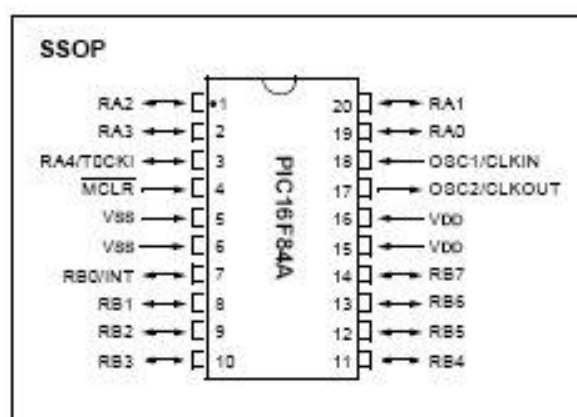
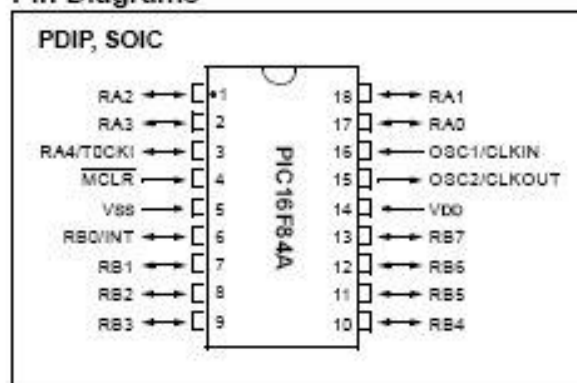
Peripheral Features:

- 13 I/O pins with individual direction control
- High current sink/source for direct LED drive
 - 25 mA sink max. per pin
 - 25 mA source max. per pin
- TMR0: 8-bit timer/counter with 8-bit programmable prescaler

Special Microcontroller Features:

- 10,000 erase/write cycles *Enhanced* FLASH Program memory typical
- 10,000,000 typical erase/write cycles EEPROM Data memory typical
- EEPROM Data Retention > 40 years
- In-Circuit Serial Programming™ (ICSP™) - via two pins
- Power-on Reset (POR), Power-up Timer (PWRT), Oscillator Start-up Timer (OST)
- Watchdog Timer (WDT) with its own On-Chip RC Oscillator for reliable operation
- Code protection
- Power saving SLEEP mode
- Selectable oscillator options

Pin Diagrams



CMOS *Enhanced* FLASH/EEPROM Technology:

- Low power, high speed technology
- Fully static design
- Wide operating voltage range:
 - Commercial: 2.0V to 5.5V
 - Industrial: 2.0V to 5.5V
- Low power consumption:
 - < 2 mA typical @ 5V, 4 MHz
 - 15 µA typical @ 2V, 32 kHz
 - < 0.5 µA typical standby current @ 2V

1) Description rapide

On s'intéresse ici au PIC 16F84A qui remplace le 16F84 en tout point identique mais avec une mémoire FLASH améliorée (10000 écritures possibles au lieu de 1000).

Résumé des caractéristiques du PIC 16F84 (A)

- Boîtier 18 broches (DIP ou SOIC)
- Alimentation : $(2\text{ V} < V_{CC} < 5,5\text{ V})$
- Fréquence d'horloge $\leq 20\text{ MHz}$ (quartz ou circuit RC selon programmation)
(1 cycle d'horloge = 4 périodes d'horloge)
- Mémoire programme : 1 K mots de 14 bits (bit 3 et 4 de PCLATH inutilisés)
de type FLASH (16F84) ou EEPROM (16C84)
- Mémoire de données : ➤ RAM : 2 banques de 128 octets dont certains sont identiques (!) et d'autres inutilisables.
 - 15 registres SFR
 - 68 octets libres pour stocker des variables
- EEPROM : 64 octets
- Périphériques : ➤ 2 ports d'entrée/sortie
 - A (5 bits)
 - B (8 bits)
- 1 *timer* de 8 bits (avec prédiviseur 8 bits programmable)
- 1 chien de garde
- 4 sources d'interruption (1 seul "vecteur")
- pile à 8 niveaux
- mode *standby*

2) Registres SFR

En plus des registres principaux décrits dans la section II-1, on trouve des registres dédiés à la gestion des périphériques internes associés au processeur (cf schéma *architecture interne* section III-3).

Le tableau ci-dessous récapitule tous les registres du PIC16F84 (A), avec la désignation de chacun de leurs 8 bits ainsi que leur état au démarrage et après un reset. Ces différents registres vont être décrits dans les sections suivantes.

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on Power-on Reset	Value on all other resets (Note3)
Bank 0											
00h	INDF	Uses contents of FSR to address data memory (not a physical register)								---- --	---- --
01h	TMR0	8-bit real-time clock/counter								xxxx xxxx	uuuu uuuu
02h	PCL	Low order 8 bits of the Program Counter (PC)								0000 0000	0000 0000
03h	STATUS ⁽²⁾	IRP	RP1	RP0	T0	PD	Z	DC	C	0001 1xxx	000q quuu
04h	FSR	Indirect data memory address pointer 0								xxxx xxxx	uuuu uuuu
05h	PORTA	—	—	—	RA4/T0CKI	RA3	RA2	RA1	RA0	---x xxxx	---u uuuu
06h	PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0/INT	xxxx xxxx	uuuu uuuu
07h		Unimplemented location, read as '0'								---- --	---- --
08h	EEDATA	EEPROM data register								xxxx xxxx	uuuu uuuu
09h	EEADR	EEPROM address register								xxxx xxxx	uuuu uuuu
0Ah	PCLATH	—	—	—	Write buffer for upper 5 bits of the PC ⁽¹⁾				---0 0000	---0 0000	
0Bh	INTCON	GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x	0000 000u
Bank 1											
80h	INDF	Uses contents of FSR to address data memory (not a physical register)								---- --	---- --
81h	OPTION_REG	RSPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111	1111 1111
82h	PCL	Low order 8 bits of Program Counter (PC)								0000 0000	0000 0000
83h	STATUS ⁽²⁾	IRP	RP1	RP0	T0	PD	Z	DC	C	0001 1xxx	000q quuu
84h	FSR	Indirect data memory address pointer 0								xxxx xxxx	uuuu uuuu
85h	TRISA	—	—	—	PORTA data direction register				---1 1111	---1 1111	
86h	TRISB	PORTB data direction register								1111 1111	1111 1111
87h		Unimplemented location, read as '0'								---- --	---- --
88h	EECON1	—	—	—	EEIF	WRERR	WREN	WR	RD	---0 x000	---0 q000
89h	EECON2	EEPROM control register 2 (not a physical register)								---- --	---- --
0Ah	PCLATH	—	—	—	Write buffer for upper 5 bits of the PC ⁽¹⁾				---0 0000	---0 0000	
0Bh	INTCON	GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x	0000 000u

Legend: x = unknown, u = unchanged. - = unimplemented read as '0', q = value depends on condition.

Note 1: The upper byte of the program counter is not directly accessible. PCLATH is a slave register for PC<12:8>. The contents of PCLATH can be transferred to the upper byte of the program counter, but the contents of PC<12:8> is never transferred to PCLATH.

2: The \overline{TO} and \overline{PD} status bits in the STATUS register are not affected by a \overline{MCLR} reset.

3: Other (non power-up) resets include: external reset through \overline{MCLR} and the Watchdog Timer Reset.

ex : TRISB = 0x0F

PB7, PB6, PB5, PB4 en sortie

PB3, PB2, PB1, PB0 en entrée

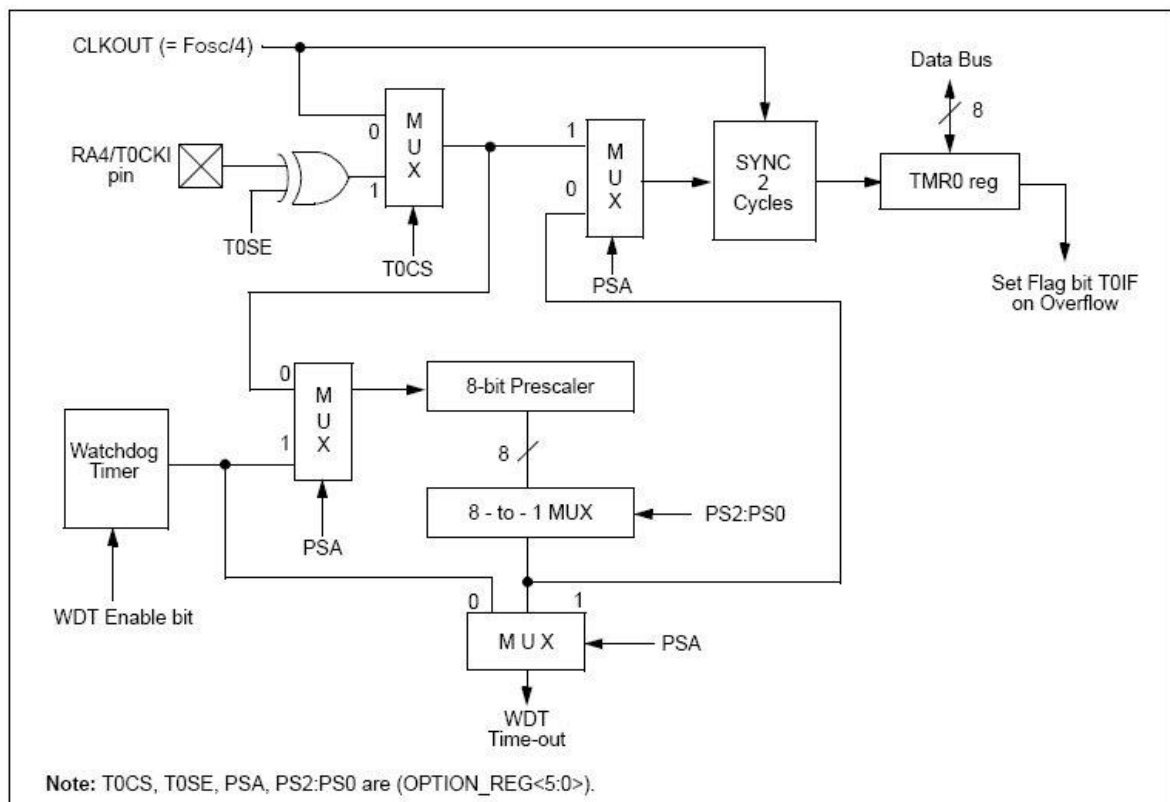
- le registre commun **OPTION** (0x81) qui permet grâce au bit RBPU (b7) de mettre en service (si RBPU = 0) les résistances de rappel à + 5V (pull up) sur tous les bits du port B.

- NB :**
- Certaines broches de ces ports ont une double fonction. Par exemple RB0/INT peut aussi servir pour provoquer une interruption externe (matérielle).
 - D'un point de vue électrique, il y a aussi des différences entre les broches et selon les PORTs. Par exemple RA4 a une sortie en drain ouvert (nécessité de placer une résistance externe de pull up), consulter la datasheet - section 4 pour plus d'infos.
 - Courant max de sortie : 25 mA / broche et total de 50 mA (port A) , 100 mA (port B)

5) Le timer TIMER 0

C'est un compteur 8 bits dont la valeur en sortie est accessible par l'intermédiaire du registre **TMR0** (0x01) en lecture mais aussi en écriture. Il peut être incrémenté de 2 façons :

- ① par les cycles d'horloge du PIC (attention 1 cycle = 4 périodes de H) : c'est le mode **TIMER**
- ② par les impulsions reçues sur la broche RA4/T0CKI : c'est le mode **COMPTEUR**



Le TIMER 0 est lui aussi géré par le registre **OPTION** (0x81) :

- le bit T0CS (b5) permet de choisir le mode :
 - T0CS = 0 → mode TIMER
 - T0CS = 1 → mode COMPTEUR
- le bit T0SE (b4) permet de choisir sur quelle transition de la broche RA4/T0CKI le comptage est effectué en mode COMPTEUR :
 - T0SE = 0 : comptage si l'entrée RA4/T0CKI passe de 0 à 1
 - T0SE = 1 : comptage si l'entrée RA4/T0CKI passe de 1 à 0
- les bits PS2 (b2), PS1 (b1), PS0 (b0) permettent de définir le rapport de division du prédiviseur (**prescaler**)

bit 2-0: **PS2:PS0**: Prescaler Rate Select bits

Bit Value	TMR0 Rate	WDT Rate
000	1 : 2	1 : 1
001	1 : 4	1 : 2
010	1 : 8	1 : 4
011	1 : 16	1 : 8
100	1 : 32	1 : 16
101	1 : 64	1 : 32
110	1 : 128	1 : 64
111	1 : 256	1 : 128

NB : Le *prescaler* peut aussi être utilisé pour le chien de garde (voir plus loin) mais il ne peut plus alors être utilisé par le TIMER 0.

- le bit PSA (b3) permet d'affecter le *prescaler* :
 - au TIMER 0 (PSA = 0)
 - **ou** au chien de garde (PSA = 1)

NB : - Pour désactiver le *prescaler* il faut l'affecter au chien de garde avec un rapport 1 : 1 (PS2=PS1=PS0=0)
- Les 8 bits du *prescaler* ne sont pas accessibles en lecture ni en écriture

ex : $f_{\text{horloge}} = 4 \text{ MHz}$ *cycle d'horloge de durée = 1 μs*
 TOSC = 0 *mode TIMER*
 PSA = 0 *prescaler affecté au TIMER 0*
 (PS2, PS1, PS0) = 010 : *prescaler = 8*

⇒ TIMER 0 incrémenté toutes les 8 μs

Utilisation du TIMER 0

Le TIMER 0 peut être utilisé de 2 façons :

① Par scrutation du flag **T0IF**

Lorsque le TIMER 0 déborde (passage de la valeur 255 à 0) le bit T0IF (b2) du registre **INTCON** (0x0B, 0x8B) passe à 1 (et reste à 1 tant qu'il n'a pas été remis à 0 par le programme !). Il suffit donc de scruter le bit T0IF pour savoir si 256 incréments ont eu lieu.

Inconvénient de cette méthode : on perd du temps à attendre et tester le TIMER 0. De plus, le programme risque de ne pas réagir assez rapidement (si la fréquence des tests est inférieure à celle de l'incrément de TMR0)

Remarque : Si l'on souhaite mesurer une durée correspondant à N incréments du *timer* :

$N < 256$: on préférera initialiser le registre **TMR0** à la valeur de N codée en complément à 2 plutôt que de scruter la valeur de **TMR0** et de la comparer à N (test de T0IF plus rapide)

$N \geq 256$: on utilise le nombre de boucles nécessaires pour comptabiliser les débordements du *timer*, le reliquat étant obtenu en initialisant de la même façon le registre TMR0 avec le complément à 2 de (N modulo 256)

② Par interruption (voir partie suivante)

Cette méthode résout le problème posé par la précédente car le programme principal n'a pas à tester le TIMER 0 : le débordement du TIMER 0 génère une interruption (si le registre **INTCON** est bien configuré...) qui déroute "aussitôt" le PIC du programme principal pour traiter un sous-programme prévu. C'est donc cette méthode qui est de loin la plus utilisée.

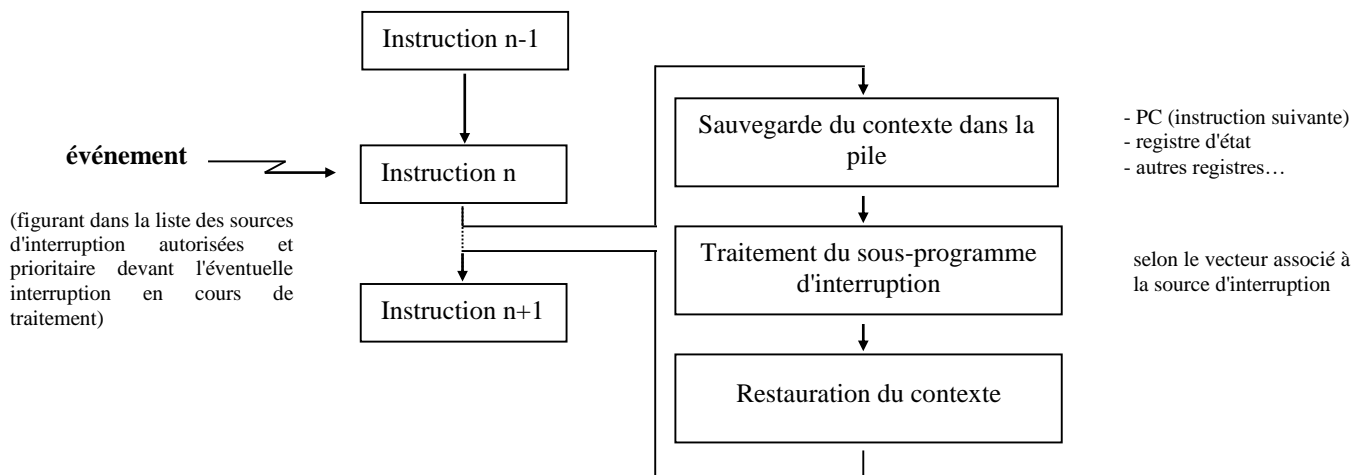
La remarque précédente reste valable.

6) Les interruptions

a) Principe

On parle d'interruption lorsqu'un événement survient et interrompt le déroulement normal du programme afin de pouvoir traiter immédiatement l'événement. Une fois l'événement traité (routine d'interruption terminée) le programme reprend à l'endroit où il avait été interrompu, comme si l'interruption n'avait pas eu lieu.

L'intérêt de ce mécanisme est d'abord de minimiser le temps de réaction du processeur vis-à-vis de l'action à mener en lien avec l'événement, comparativement à une solution classique de "polling" (boucle de test sur l'événement). Le 2^e avantage découle du 1^{er}, on libère le processeur de cette phase de scrutation (gain de temps CPU). Le déroulement classique d'une procédure d'interruption a lieu selon le schéma :

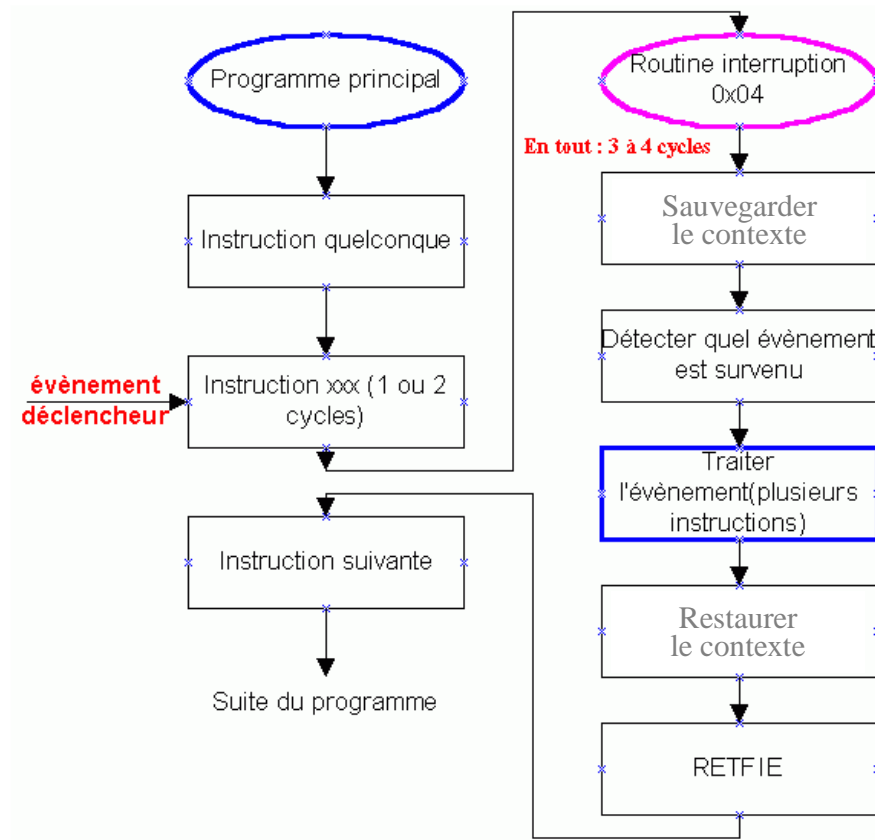


b) Mécanisme d'interruption sur les PICs

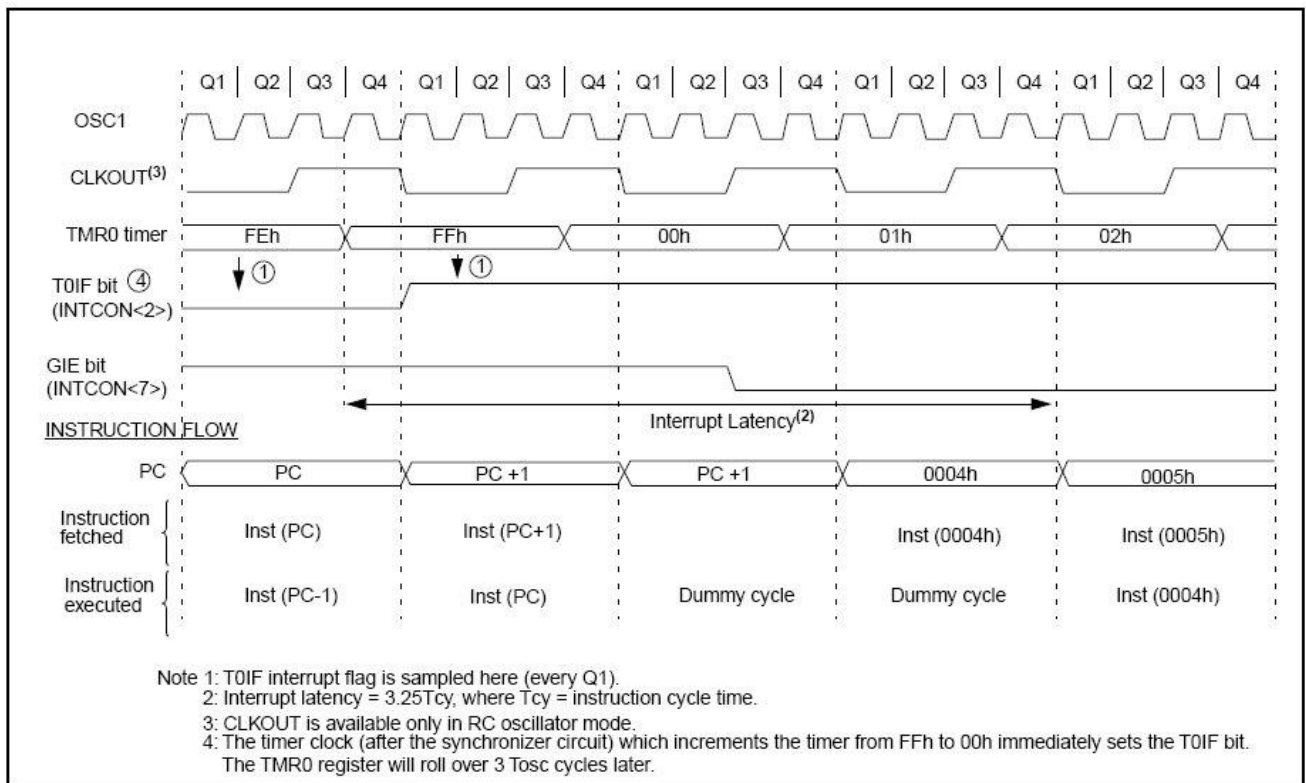
- Tout d'abord l'adresse de début du sous-programme d'interruption est fixe (un seul vecteur), il s'agit toujours de l'adresse 0x04. Toute interruption provoquera donc le saut du programme vers cette adresse.
- Toutes les sources d'interruption déclenchant l'exécution de la même routine à cette adresse, si le programmeur utilise plusieurs sources d'interruptions, il lui faudra déterminer lui-même laquelle il va devoir traiter.
- Les PICs jusqu'à la série PIC16 ne sauvegardent rien automatiquement en cas d'interruption, hormis bien sûr la valeur du PC indispensable pour revenir au déroulement normal du programme. C'est donc à l'utilisateur de se charger de la sauvegarde des registres (opération par contre prise en charge sur les PICs à partir de la série PIC18).

- Le contenu du PC est sauvegardé sur la pile interne (8 niveaux). Donc, si on utilise des interruptions, on ne dispose plus que de 7 niveaux d'imbrication pour les sous-programmes (moins si on utilise des sous-programmes dans les 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 va 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.
- 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 (voir plus loin).
- 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.

On obtient ainsi l'organigramme suivant en cas d'interruption sur les PICs :



Exemple du déclenchement d'une interruption par le débordement du TIMER0



c) Les interruptions avec le PIC 16F84

Le 16F84 est très pauvre à ce niveau, puisqu'il ne dispose que de 4 sources d'interruptions possibles (contre 13 pour le 16F877 par exemple). Les événements susceptibles de déclencher une interruption sont les suivants :

- **TIMER0** : débordement du TIMER 0. Une fois que le contenu de **TMR0** passe de 0xFF à 0x00, une interruption peut être générée.
- **EEPROM** : une interruption peut être générée lorsque l'écriture dans une case EEPROM interne est terminée.
- **RB0/INT** : une interruption peut être générée lorsque la broche RB0, encore appelée INTerrupt pin, étant configurée en entrée, celle-ci reçoit un front. Le front actif est défini par le bit INTEDG (b6) du registre **OPTION** (0x81) :

INTEDG = 1 → front montant

INTEDG = 0 → front descendant

- **PORTB** : de la même manière, une interruption peut être générée si un changement de niveau (front montant ou descendant) se produit sur une des broches RB4 à RB7. Il n'est pas possible de limiter l'interruption à une seule de ces broches. L'interruption sera effective pour les 4 broches ou pour aucune.

Les interruptions sont gérées par le registre **INTCON** (0x0B, 0x8B) :

GIE (b7) : Global Interrupt Enable bit. Il doit être activé pour permettre le déclenchement de toute interruption (il sert surtout à empêcher toute IT lorsqu'il est désactivé).

EEIE (b6) : EEprom write complete Interrupt Enable bit. Ce bit permet d'autoriser une interruption en fin d'écriture en EEPROM (voir paragraphe 7).

T0IE (b5) : Tmr0 Interrupt Enable bit : Autorise une interruption lors du débordement du TIMER 0.

INTE (b4) : INTerrupt pin Enable bit : Autorise une interruption dans le cas où un front actif se présente sur la broche RB0.

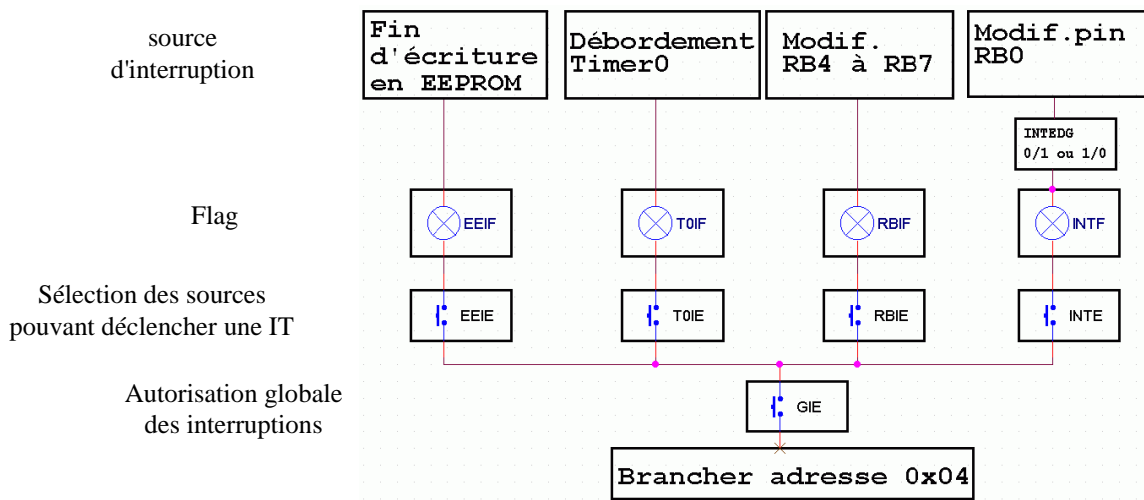
RBIE (b3) : RB port change Interrupt Enable bit : Autorise une interruption s'il y a un changement de niveau sur une des entrées RB4 à RB7.

T0IF (b2) : Tmr0 Interrupt Flag bit : C'est un Flag (indicateur), donc il signale. Ici c'est le débordement du TIMER 0.

INTF (b1) : INTerrupt pin Flag bit : signale une transition sur la pin RB0 dans le sens déterminé par INTEDG (b6) du registre **OPTION**.

RBIF (b0) : RB port Interrupt Flag bit : signale qu'une des entrées RB4 à RB7 a été modifiée.

Le schéma suivant permet de représenter de façon synthétique le rôle des bits du registre **INTCON** :



Remarques :

- Les indicateurs (flags) permettent de connaître la source qui a provoqué l'interruption (attention **ils doivent être remis à 0 par le sous-programme d'IT**).
- Les bits EEIE, TOIE, RBIE, INTE permettent d'autoriser ou non la source correspondante à déclencher une interruption.
- Le bit GIE sert de validation globale et a surtout pour rôle d'interdire tout déclenchement d'IT. Comme les PICs ne peuvent traiter qu'une seule IT à la fois, le bit GIE est automatiquement mis à 0 au début du traitement de l'IT (il est restauré à 1 automatiquement par l'instruction RETFIE)
- Le bit EEIF (indicateur de fin d'écriture en EEPROM) est placé à part, c'est le bit 4 du registre **EECON1** (0x88)

d) Sauvegarde et restauration du contexte

Le PC est sauvegardé dans la pile et restauré automatiquement à la fin de l'IT. Le sous-programme d'IT doit donc gérer la sauvegarde des registres **STATUS** et **W**. On ne peut, comme on le fait en général pour stocker ces registres, utiliser la pile car elle n'est pas accessible. On utilise donc 2 emplacements en RAM.

7) Le chien de garde (Watch Dog)

C'est un mécanisme de protection du programme qui en théorie ne doit pas servir mais qui en pratique peut permettre de sortir d'une boucle d'attente infinie (événement attendu qui ne se produit pas...) ou de vérifier si le programme ne s'est pas égaré. Le Watch-Dog (WD) sert également à réveiller un PIC placé en mode SLEEP (voir paragraphe 6).

a) Description

Il s'agit en fait d'un *timer* (nommé WDT) qui possède sa propre horloge dont la fréquence dépend de la température et de la tension d'alimentation (mais pas de f_{horloge} !). Le constructeur donne :

$$7 \text{ ms} \leq \text{périodicité du débordement du timer WD} = 18 \text{ ms typique} \leq 33 \text{ ms}$$

Le temps de débordement peut être allongé grâce au *prescaler* (qui fonctionne ici en post-diviseur), si $\text{PSA} = 1$, qui peut multiplier ce temps au maximum $\times 128$ selon le tableau :

bit 2-0: **PS2:PS0**: Prescaler Rate Select bits

Bit Value	TMR0 Rate	WDT Rate
000	1 : 2	1 : 1
001	1 : 4	1 : 2
010	1 : 8	1 : 4
011	1 : 16	1 : 8
100	1 : 32	1 : 16
101	1 : 64	1 : 32
110	1 : 128	1 : 64
111	1 : 256	1 : 128

b) Principe

Chaque fois que l'instruction CLRWDT est exécutée par le PIC, le WDT est remis à 0 ainsi que la valeur contenue dans le prédiviseur. Si, pour une raison anormale, cette instruction n'est pas exécutée avant le débordement du WDT, le PIC est redémarré à l'adresse 0x00 et le bit /TO (b4) de **STATUS** est mis à 0. En lisant ce bit en début de programme il est de plus possible de détecter si le PIC vient d'être mis sous tension ou si ce démarrage est dû à un débordement du WDT ("plantage" du programme ?).

c) Utilisation

Le WDT est donc une protection intéressante mais qu'il faut gérer par le programme. Pour cela il suffit de placer des instructions CLRWDT à intervalle inférieure au temps de débordement du WDT.

$$\Delta T_{\max} = \text{rapport } \textit{prescaler} \text{ (d'après PS2 ,PS1, PS0)} \times 7 \text{ ms (cas le plus défavorable)}$$

Il est à choisir en fonction de l'application selon le compromis :

$$\Delta T_{\max} \text{ faible (réaction rapide face au plantage)} \leftrightarrow \text{peu d'instructions CLRWDT à insérer}$$

NB : - Le WD est mis en service par la directive CONFIG _WDT _ON à préciser au moment de la programmation (met le bit WDTE à 1 dans le registre CONFIG) et ne peut être désactivé par le programme.

- Ne jamais placer un CLRWDT dans une routine d'IT sinon l'IT risque de neutraliser la protection offerte par le WDT sans pour autant bien sûr réinitialiser le PIC.

8) Le mode SLEEP

L'instruction SLEEP permet de placer le PIC en mode SLEEP (Standby), c'est-à-dire que le PIC s'arrête juste après l'instruction en attendant d'être réveillé. L'intérêt de ce mode est de diminuer la consommation du PIC ($I_{cc} < 0,5 \mu A$ sans compter le courant fourni par chaque PORT en sortie).

Principe

Lorsque l'instruction SLEEP est exécutée :

- Le WDT 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 TIMER 0 est synchronisé par l'horloge interne, il est également mis dans l'incapacité de compter. Par contre, il est important de se rappeler que le *timer* du watchdog possède son propre circuit d'horloge; il continue donc de compter normalement.

Plusieurs événements peuvent faire sortir le PIC du mode SLEEP :

- Application d'un niveau 0 sur la broche /MCLR. 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. 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 WDT ne provoque pas un reset du PIC, il se contente de le réveiller. L'instruction qui suit est alors exécutée au réveil.
- Une interruption RB0/INT, RB, ou EEPROM est survenue. Pour qu'une telle interruption puisse réveiller le processeur, il faut que les bits de mise en service de l'interruption aient été positionnés. Le PIC se réveille et exécute l'instruction suivant SLEEP. On peut noter que si le bit GIE = 1 (interruptions activées) le PIC traite en plus l'IT qui l'a réveillé, sinon il continue.

9) Les accès en mémoire EEPROM

a) Ecriture lors de la programmation

Le PIC 16F84 dispose de 64 octets implantés à partir de l'adresse 0x2100 accessibles directement uniquement lors de la programmation.

b) Accès par le programme

L'EEPROM peut bien sûr être lue par le programme mais également être programmée grâce à des registres de contrôle :

- **EEDATA (0x08)** : C'est dans ce registre que va transiter la donnée à lire ou écrire .

- **EEADR** (0x09): Dans ce registre doit être précisée sur 8 bits l'adresse concernée par l'opération de lecture ou d'écriture en EEPROM. On voit déjà que pour cette famille de PICs, on ne peut pas dépasser 256 emplacements d'EEPROM. Pour le 16F84, la zone admissible va de 0x00 à 0x3F (codé en relatif par rapport à l'adresse de début de zone physique 0x2100), soit 64 emplacements.

EECON1 (0x88) : Ce registre contient 5 bits qui définissent ou indiquent le fonctionnement des cycles de lecture/écriture en EEPROM (voir datasheet)

EECON2 (0x89): Il s'agit tout simplement d'une adresse, qui sert à envoyer des commandes au PIC concernant les procédures d'écriture en EEPROM. On ne peut l'utiliser qu'en respectant la routine donnée par le constructeur (voir ②).

① Lecture

Pour lire une donnée en EEPROM, il suffit de placer l'adresse concernée dans le registre **EEADR**, puis de positionner le bit RD à 1. On peut ensuite récupérer la donnée lue dans le registre **EEDATA**.

② Ecriture

La procédure à suivre consiste d'abord à placer la donnée dans le registre **EEDATA** et l'adresse dans **EEADR**. Ensuite une séquence spécifique imposée par le constructeur doit être chargée dans le registre **EECON2**.

Remarques

- A la fin de la procédure d'écriture, la donnée n'est pas encore enregistrée dans l'EEPROM. Elle le sera approximativement 10 ms plus tard (ce qui représente tout de même environ 10.000 instructions !). On ne peut 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.

NB : Avant d'écrire dans l'EEPROM il faut vérifier qu'une autre écriture n'est pas en cours en s'assurant que le bit WR du registre **EECON1** est bien à 0.

Microchip Block Diagram Support

