TP MICROCONTROLEUR, FAMILLE PIC

Séances 1 et 2

Thomas Grenier,
Dominique Tournier,
Olivier Bernard,
David Lévèque.

Objectifs

But global :
Acquérir l'expérience d'un développement d'applications à base de micro contrôleur.

Séance 1 :
Prise en main du matériel et des logiciels de développement.

Séance 2 :
Notions avancées
Réalisation d'une petite application.

Présentation du matériel

1- Logiciel MPLAB de Microchip

Il s'agit de l'environnement informatique de développement pour PIC fourni par le constructeur Microchip. Il est gratuitement téléchargeable et utilisable.

MPLAB permet d'éditer le code assembleur (reconnaissance des instructions et des variables internes), compiler le code (le lier à des librairies si besoin), simuler le comportement d'un PIC (ceci permet de tester le bon fonctionnement d'un programme (débogage) par simulation) et de piloter des outils de développement supplémentaires comme MPLAB ICD 2.

Ce programme est présent sur chaque PC de la salle de TP

2- Kit MPLAB ICD 2

Ce kit permet de programmer et de déboguer un microcontrôleur PIC via le logiciel MPLAB IDE. Ce kit est composé :

D'un module ICD 2 (circulaire) contenant l'électronique de communication,
D'un câble USB permettant la communication entre le bloc circulaire et le PC,
Un petit câble RJ11 permettant la communication entre le bloc et le PIC.

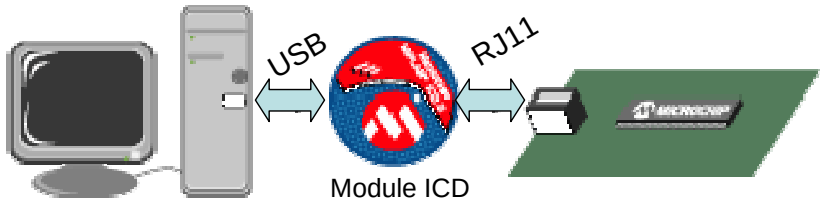


Figure 1: Câblage du kit ICD 2

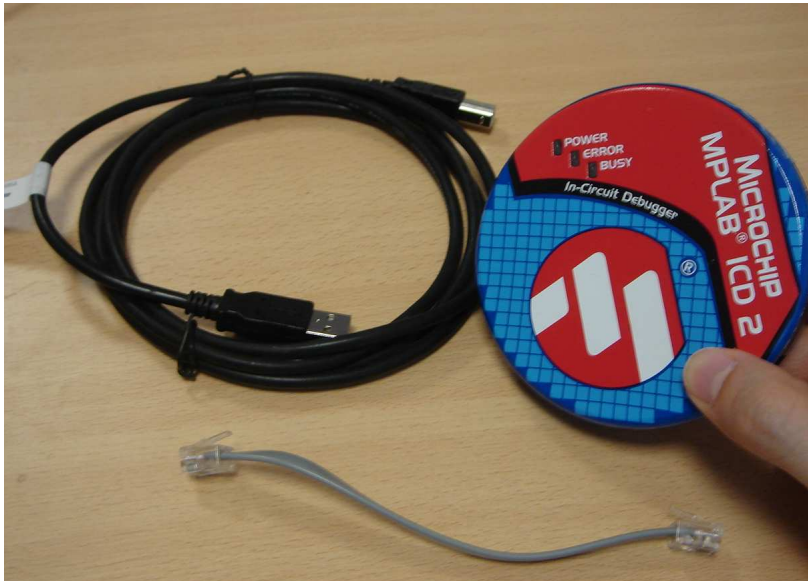


Figure 2: Les 3 éléments du kit ICD 2 ; le câble RJ11 est en bas de l'image

Ne pas connecter d'alimentation au module ICD 2 !!! Le module est alimenté par USB.

3- Carte à PIC

Deux cartes sont nécessaires à ce TP : la carte « *process* » et la carte « *mini* ».

La carte « *process* » est la carte sur laquelle le PIC 16F877A est présent. On trouve sur cette carte (Figure 23) :

- l'alimentation (**il est nécessaire d'alimenter cette carte**),
- un connecteur RJ11 pour le débogage/programmation du PIC via le kit ICD,
- un interrupteur permettant de basculer entre fonctionnement autonome du PIC (position basse) ou fonctionnement par ICD et programmation (position haute, la LED rouge est allumée),
- l'oscillateur à quartz (20MHz)
- un connecteur vers d'autres cartes.

Sur la carte « Mini » sont présents : 8 LEDs, 4 boutons poussoir et un buzzer (voir schéma Figure 21).

I. – Prise en main (1° séance)

➔ **Avant de commencer le TP, vérifier que vous disposez :**

- d'un PC avec MPLAB IDE,
- d'un module MPLAB ICD 2,
- des deux cartes, l'une avec un PIC 16F877A (à 20MHz), l'autre avec les 8 LEDs, les 4 boutons poussoirs et le buzzer,
- d'une alimentation pour la carte PIC,
- d'un câble USB,
- d'un petit câble RJ11,

Dans un premier temps ne pas connecter le kit ICD2 ni les cartes : tout se fait sur PC (questions 1- et 2-).

1- Présentation de l'application : Bouton-poussoir intelligent

Le but est de réaliser la commande d'un éclairage à partir d'un simple bouton poussoir. En appuyant une première fois sur le bouton poussoir, on allume l'éclairage. Une seconde pression l'éteindra. Cependant, si on ne pense pas à éteindre les lampes, une minuterie les éteindra automatiquement.

Voici l'organigramme de l'application (le code assembleur est présent sur les PC et est fourni en annexe) :

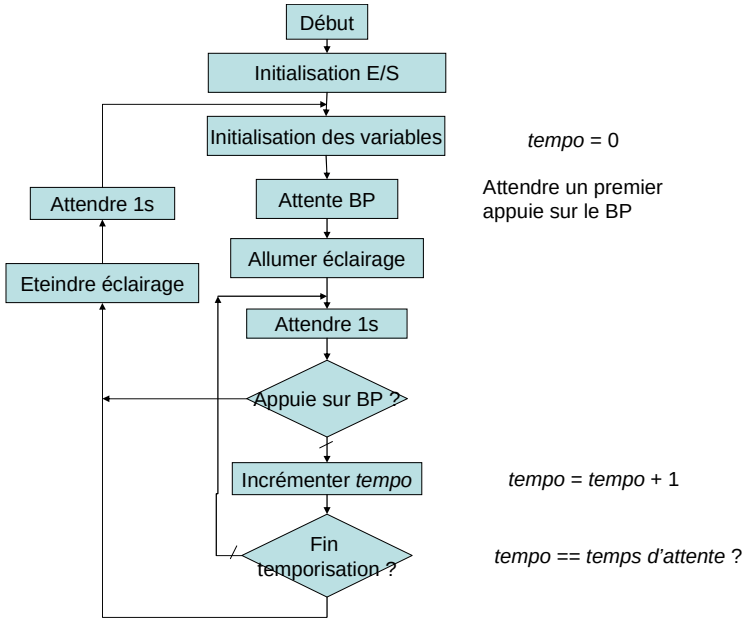


Figure 3 : Organigramme de l'application Bouton-poussoir intelligent.

Q- Pendant combien de temps faudra t'il maintenir le BP appuyé pour garantir l'extinction de l'éclairage ?

Q- Quel est le rôle de la temporisation effectuée après « éteindre éclairage » ?

2- Manipulations 1: MPLAB IDE

- Lancement MPLAB IDE

Lancer le logiciel de développement MPLAB.

Démarrer → *Programmes* → *Microchip* → *MPLAB IDE* → *MPLAB IDE*

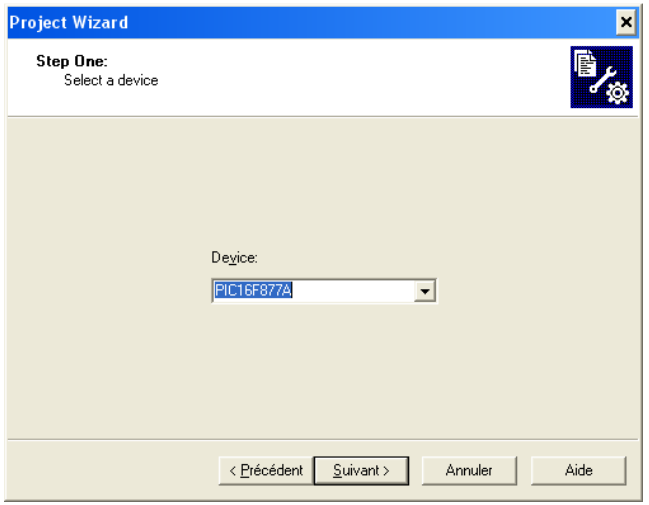
- Création d'un nouveau projet

Au lancement de MPLAB, un projet par défaut est utilisé. Pour chaque nouvelle application il est conseillé de créer un nouveau projet. Un projet permet de gérer les fichiers de code assembleur, la référence et les paramètres du microcontrôleur ainsi que les options et outils de débogage.

Pour créer un nouveau projet :

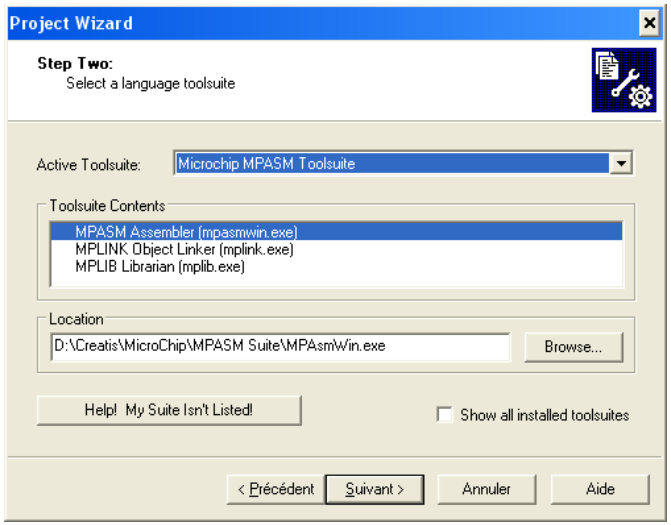
Project → *Project Wizard…*

La fenêtre wizard apparaît. Cliquer sur « Suivant », la fenêtre ci après apparaît. Sélectionner le microcontrôleur présent sur votre platine (16F877A) et passer à la fenêtre suivante.



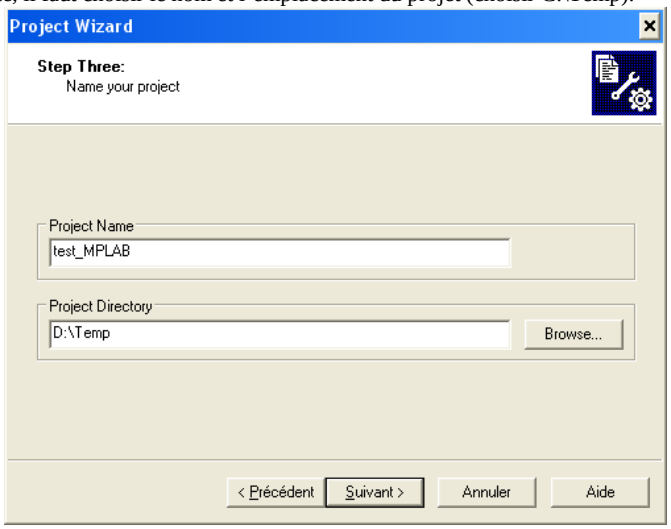
Sélection du microcontrôleur

La fenêtre qui suit permet de choisir les programmes de développement. On gardera l'ensemble logiciel proposé par MicroChip (vérifier que le contenu de la fenêtre ressemble à celui présenté ci-après).

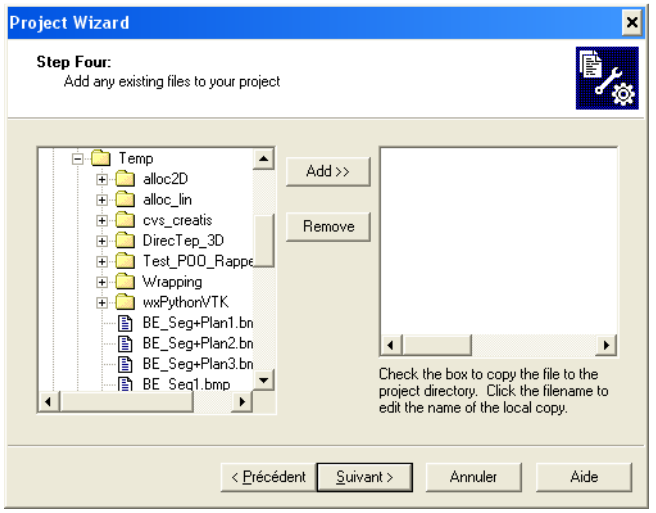


Sélection compilateur et lieur

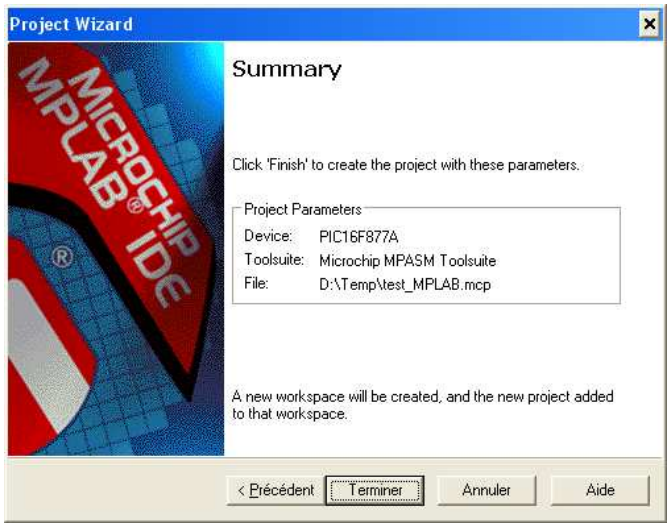
Ensuite, il faut choisir le nom et l'emplacement du projet (choisir C:\Temp).



La fenêtre suivante permet d'ajouter des fichiers existants à votre projet. Pour l'instant, ne pas ajouter de fichier.



La dernière fenêtre du *wizard* apparaît ensuite, elle résume les principaux paramètres du projet.



Après avoir cliqué sur « Terminer » de la fenêtre précédente, votre projet est créé, mais il est vide ! Il va falloir ajouter des fichiers sources (en assembleur) et des outils de simulation, débogage et programmation.

- Ajout de fichier source assembleur (***.asm***)

Commencer par copier le fichier **TPuC_1.asm** (présent sur le bureau) dans le répertoire de votre projet (dans C :\Temp) . Il s'agit du programme assembleur de l'application.

Ensuite, ajouter ce fichier à votre projet. A partir du menu, procéder ainsi :

Project → *Add Files to Project…*

Choisir le fichier **TPuC_1.asm** de **votre** répertoire.

Q- Analyser rapidement le code (faire le lien avec l'organigramme, fonction, solution pour la temporisation, variables,…) puis justifier (se reporter au schéma des cartes), le test de la première détection d'appuie sur le bouton poussoir :

```
; touche ?
     btfsc PORTB, 0
     goto main_prg
```

- Compilation d'un seul fichier

→ But de la compilation

Le fichier assembleur n'est pas exécutable par un microcontrôleur (ni par un processeur). Il est nécessaire de le transformer en langage machine : chaque mnémonique est transformée en un code hexadécimal, les sauts aux étiquettes sont remplacés par des sauts relatifs, … Tout ceci est fait par le compilateur.

Pour que le compilateur génère le code machine, il est évident que le fichier assembleur ne doit comporter plus aucune erreur de syntaxe…

→ Comment compiler

Pour compiler un seul fichier .asm, sélectionner le fichier dans la fenêtre de gestion de projet (si elle a été fermée faire « *View*→ *Projet* ») puis cliquer sur le bouton droit. Ensuite faire « *Assemble* » :

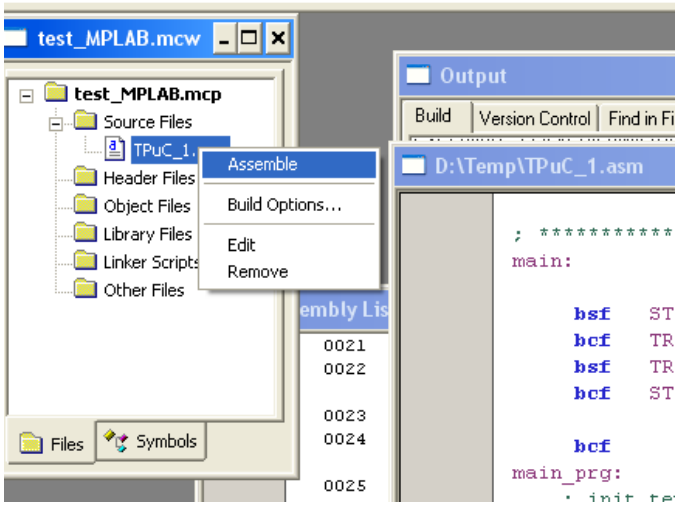


Figure 4 : Compilation d'un seul fichier .asm

→ Messages d'erreur, « *warning* » et « *message*[1] »

Si le compilateur détecte une erreur (respectivement, une ambiguïté) de syntaxe il le signalera par une erreur (respectivement, par un *warning* ou *message*) dans la fenêtre « *Output* ». En assembleur, les erreurs sont :

- utilisation d'une instruction ou d'un registre qui n'existe pas,
- utilisation d'une variable non définie précédemment,
- utilisation d'une étiquette non définie précédemment.

Les *warning* et *message* sont des mises en gardes qu'il est fortement conseillé de vérifier. Notamment, dans le cas de la programmation des PIC :

- l'accès à un registre n'appartenant pas à la banque en cours d'utilisation,
- pour les instructions, l'opérande de destination (f ou W) est facultatif. Si elle n'est pas présente, le compilateur génère un warning signalant que par défaut il a utilisé le « *file register* » (noté *file* ou *f*) comme destination !

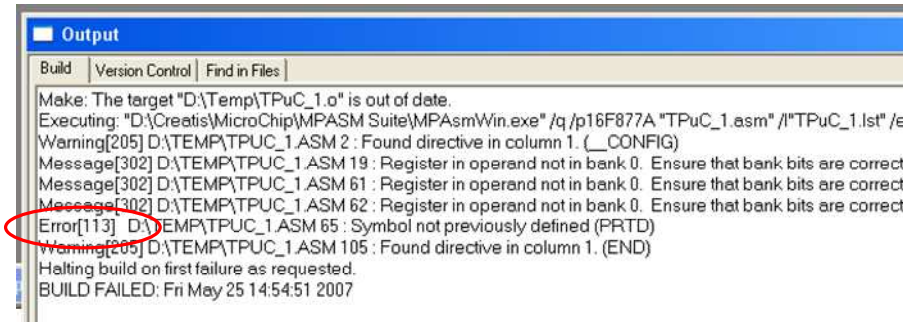


Figure 5 : Fenêtre « Output » des messages de construction et compilation.

Un double clic sur une erreur (ou un *warning*, *message*) renvoie dans le fichier et à la ligne où l'erreur (ou le *warning*, *message*) a été détectée.

Q- Corriger les erreurs du fichier TP_uC1.asm. (4 erreurs)

Q- Prendre en compte les warnings et modifier le code si nécessaire. (1 *message* est crucial !)

- Construction (Build)

→ But de la construction

La construction d'un projet va créer le programme complet de l'application en langage machine.

Quand un projet contient plusieurs fichiers et qu'il existe des liens entre ces fichiers, il est nécessaire de construire (« *Build* ») le projet. La construction consiste à compiler tous les fichiers puis à faire les liens (« *link* ») entre les différents fichiers et librairies utilisés.

S'il n'y a aucune erreur (pour la compilation et l'éditeur de liens) le langage machine de l'application est généré.

[1] Les « message » et les warning sont équivalents : il s'agit de mises en gardes signalées par le compilateur.

→ Comment construire un projet

Plusieurs solutions :

- clic droit sur le nom du projet dans la fenêtre de gestion projet puis « *Build All* » (ou *Make*),
- par le menu : Project → *Build All* (ou *Make*),
- par les raccourcis : ctrl+F10 pour *Build All* (F10 pour *Make*).

→ Messages d'erreur et de mise en garde

Mêmes démarches que pour la compilation d'un seul fichier avec en plus le problème des liens entre fichiers et librairies si le projet en contient.

Q- Construire le projet. Si les erreurs de syntaxe assembleur ont été corrigées dans la question précédente, la construction du projet ne doit pas poser de problème.

- Etude du .hex

Le langage machine généré par la construction d'un projet est généralement stocké dans un fichier « *.hex* » au format texte (« .o » ou « .exe » pour le format binaire). Sous MPLAB ce fichier est situé dans le répertoire de votre projet, et il porte le même nom de votre projet avec l'extension « *.HEX* ».

Q- Ouvrir le fichier .HEX de votre projet dans un éditeur de texte.

Q- A l'aide de la Figure 6**, déterminer le code machine correspondant à la première instruction du programme principal :**

```
bsf STATUS, RP0.
```

(Compléter en binaire la valeur « *Encoding* » en trouvant l'adresse mémoire du registre STATUS et le numéro du bit représenté par RP0 ; cf. Figure 13 page 29)

Q- Où se situe ce code dans le .HEX ?

BSF **Bit Set f**

Syntax:	[*label*] BSF f,b
Operands:	0 ≤ f ≤ 127 0 ≤ b ≤ 7
Operation:	1 → f<b>
Status Affected:	None
Encoding:	01 | 01bb | bfff | ffff
Description:	Bit 'b' in register 'f' is set.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write register 'f'

Figure 6 : Détail de codage de l'instruction BSF

- Etude du .lst

Le fichier *listing* (.lst) est un fichier au format texte dont le contenu permet de passer facilement du langage assembleur (mnémonique) au langage machine et inversement. Lors d'une compilation, ce fichier est toujours créé, même en cas d'erreur de syntaxe. Il s'agit du fichier listing (ou log) du compilateur. Dans ce ficher on trouve tous les commentaires du compilateur (erreurs, parfois warning, …) ainsi que le codage ligne à ligne des instructions.

Pour ouvrir le fichier .lst du projet : à partir du menu faire *View* → *Disassembly Listing* ; ou trouver le fichier .lst portant le nom de votre projet et l'ouvrir dans un éditeur de texte.

Q- Ouvrir le fichier .lst et vérifier le code de l'instruction `bsf STATUS, RP0` **ainsi que vos conclusions.**

3- Manipulations 2: débogage et simulation

L'environnement de développement MPLAB avec le kit ICD 2 permet 3 types d'exécution :

- simulation du PIC sur PC avec débogage (pratique pour mettre au point un programme).
- exécution avec débogage sur PIC (grâce au débogueur du kit ICD),
- exécution sur PIC autonome, finalité du développement…

Nous allons étudier et exploiter ces trois modes d'exécution dans l'ordre précédent (qui est l'ordre logique de déploiement…).

- Simulation sur PC et débogage

Il faut commencer par activer l'outil MPLAB SIM : dans le menu

Debugger → *Select Tool* → *MPLAB SIM*

Une barre d'outils propre au débogage apparait et de nombreuses options sont maintenant disponibles dans le menu *Debugger*.

Il est maintenant possible d'exécuter le programme :

Debugger → *Run* (ou F9 ou  de la barre d'outils)

Le programme tourne… mais il n'est pas possible de modifier l'état du bouton poussoir ni d'interagir avec les registres pendant l'exécution…

Pour interagir avec l'exécution du programme, une solution consiste à l'arrêter avant l'exécution de certaines instructions. Pour cela on place des « *breakpoint* » (point d'arrêt) dans le programme assembleur. Lorsque l'exécution s'arrête sur un *breakpoint*, on peut lire et modifier les registres du microcontrôleur.

Pour placer un *breakpoint* sur une ligne, double cliquer sur cette ligne.

→ Dans la fonction main, placer un *breakpoint* à chaque écriture du bit 0 du PORTD ainsi que sur les `call Tempo_Ws`**.**

→ Afficher le contenu de registres :

View → *Watch*

Ajouter les registres de la SFR (en utilisant le bouton *Add SFR)* :

- WREG (nom complet du registre W)

- PORTB
- PORTD

La fenêtre Watch doit ressembler à ceci :

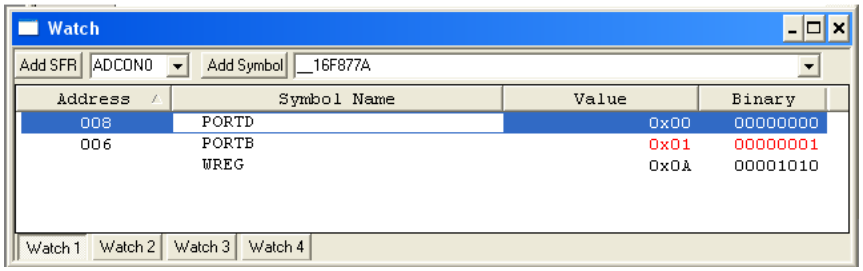


Figure 7 : Fenêtre Watch avec 3 registres visualisés

➔ **Faire un RESET du processeur** :

Debugger → *Reset* → *MCLR Reset*

ou: *Debugger* → *Reset* → *Processor Reset* (ou : F6, )

➔ **Relancer l'exécution du programme (*Run*). On peut voir la valeur des registres à chaque breakpoint.**

Apres un *breakpoint* on peut continuer d'exécuter le programme jusqu'au prochain *breakpoint* (refaire *Run*) ou exécuter une seule instruction à la fois :

Debugger → *Step Into* (F7)

Debugger → *Step Over* (F8) pour ne pas aller dans le code d'une fonction appelée via un CALL.

Q- Donner l'évolution des valeurs des registres PORTB et PORTD à chaque *breapoint*. Justifier ces valeurs.

Q- Pourquoi le programme s'exécute entièrement ? (comme si on appuyait tout le temps sur le bouton poussoir)

On va maintenant interagir avec le bit PORTB<0> (RB0) sans modifier le code grâce à l'utilisation de stimuli. On va initialiser RB0 à la valeur 1 et permettre une mise à 0 momentanée.

➔ Pour faire ceci, lancer la fenêtre *Stimulu* à partir du menu :

Debugger → *Stimulus* → *New Workbook*

➔ Initialisation de RB0 : Sur l'onglet *Pin /Register Action* ajouter un signal sur RB0. Choisir l'instant de modification (t=0) puis la valeur que l'on souhaite appliquer à RB0 (1), enfin valider (Apply). On obtient ceci :

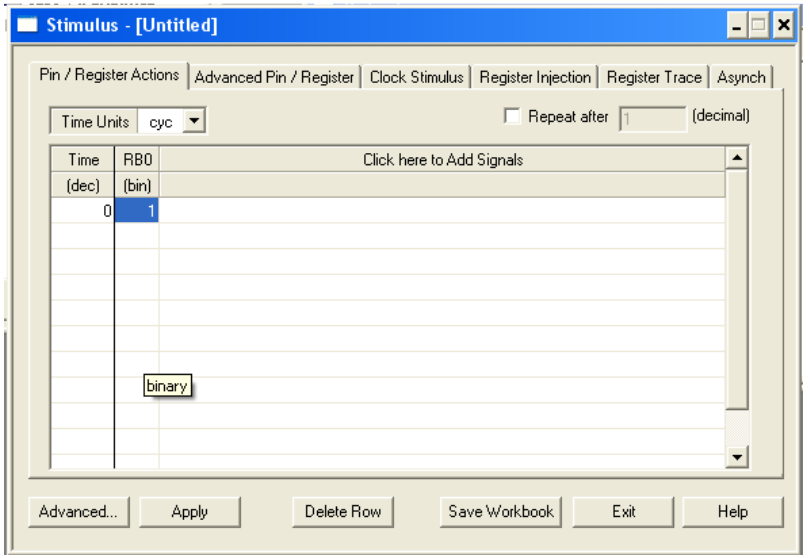


Figure 8 : Initialisation de RB0 à 1

➔ Création de la mise à l'état bas : aller sur l'onglet *Asynch* de cette même fenêtre pour ajouter une interaction non synchronisée qui correspondra à une pression sur le bouton poussoir (la pression sur le bouton poussoir peut se produire n'importe quand, d'où le « *Asynch* »).

➔ Sur la première ligne choisir RB0, et l'action adaptée… Puis valider (*Apply*).

On obtient ceci :

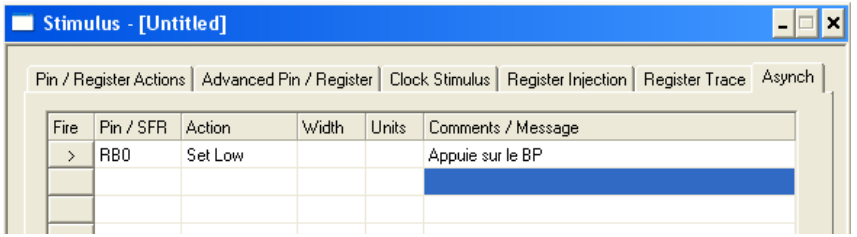


Figure 9 : Evénement non synchrone sur RB0

Sur l'exemple de la Figure 9, toute pression avec la souris sur le bouton  **provoquera une mise à l'état bas de RB0. Cet état est maintenu indéfiniment…**

➔ Initialiser le processeur (*Reset*) et exécuter à nouveau le programme.

Q- Vérifier que le programme fonctionne correctement. Modifier le programme assembleur si besoin

4- Manipulations 3 : Exécution avec débogage sur le PIC (ICD)

Le kit ICD permet de déboguer un programme sur le circuit réel. Il est recommandé d'avoir validé globalement son programme avant de le tester sur les cartes (notamment vérifier que les configurations des ports d'E/S correspondent aux câblages électronique).

Pour permettre le débogage sur le circuit, une partie des ressources du PIC est utilisée par le kit ICD et le code transmis possède une surcouche permettant de faire le débogage.

- Alimentation des cartes « *process* » et « *Mini* »

→ Quitter MPLAB.
→ Connecter l'alimentation à la carte « *process* » PIC.
➔ **Mettre l'interrupteur en position haute** (la LED rouge est allumée).

- Connexion PC, kit ICD 2, carte à PIC

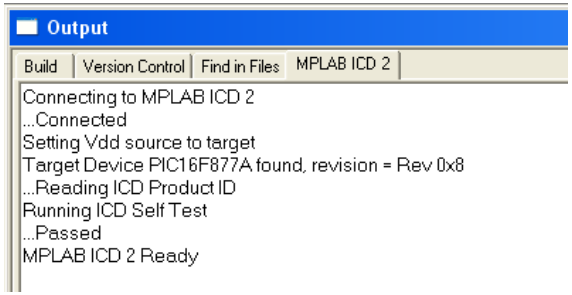
→ Connecter le module ICD 2 à la carte *process* avec le cordon RJ11.
→ Connecter le PC au module ICD 2 avec le cordon USB

Rem : Si *windows* détecte un nouveau périphérique :
- Fermer MPLAB (sauvegarder votre travail).
- Suivre les instructions de windows pour l'installation des pilotes. Les pilotes se trouvent dans le répertoire :
C:\ Program Files\Microchip\MPLAB IDE\ICD 2\Drivers
- Relancer MPLAB IDE et ouvrir votre projet.

- Choisir ICD 2 comme outil de débogage

➔ **Dans le menu faire :**
Debugger → *Select Tool* → *MPLAB ICD 2*
(Si besoin, préciser que le kit est connecté en USB.)
De nouvelles options sont alors disponibles dans le menu *Debugger*.
➔ **Commencer par initialiser la communication avec le module ICD 2 :**
Debugger → *Connect*

Si le kit est correctement configuré et connecté, on obtient l'affichage suivant dans la fenêtre Output (sinon vérifier que l'interrupteur est en position haute et qua la LED rouge est allumée) :

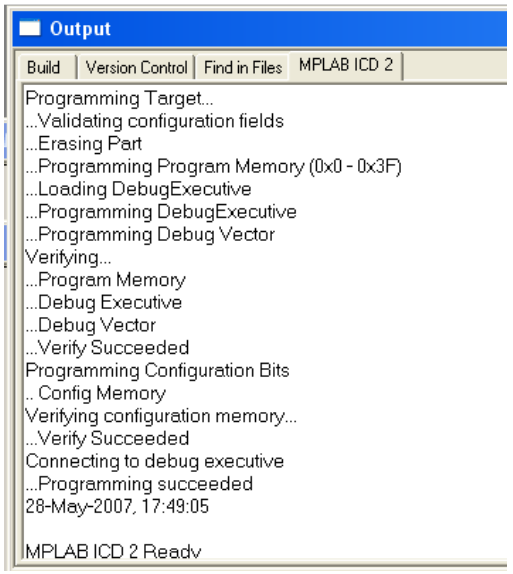


- Construction et programmation du PIC

➔ **Enlever les breakpoint de votre code** (double clic sur les lignes avec un breakpoint).
➔ **Faire une construction de votre projet** (*Build All*).

➔ **Programmer le PIC** : (interrupteur en position haute, LED rouge allumée) il est nécessaire d'envoyer le code machine dans le PIC. Procéder ainsi, dans le menu
Degugger → *Program*

Si la programmation s'est correctement déroulée, on obtient l'affichage suivant dans la fenêtre *Output* :



Le programme est maintenant chargé dans le PIC mais le PC reste le superviseur de l'exécution.

- Exécution

➔ Lancer le programme en faisant *Degugger* → *Run* (F9)
Pour arrêter l'exécution du programme faire *Debugger* → *Halt* (F5)

Q- Vérifier le fonctionnement de l'application.

- Débogage

De même qu'en simulation, il est possible de suivre l'évolution des registres et du programme pas à pas. Il faut placer des *breakpoint* sur des lignes puis exécuter le programme (F9). Il est ensuite possible de contrôler l'exécution du programme (Pas à pas : *Step Into* et *Step Over*) et le contenu des registres (*View* → *Watch*).

Q- Mettre un *breakpoint* sur la ligne `clrf tempo` et suivre l'évolution des registres PORTB et PORTD jusqu'à l'appel de la fonction Tempo_Wms (`call Tempo_Ws`).

5- Manipulations 4 : Programmation du PIC et fonctionnement autonome

Il s'agit de la programmation finale. Celle-ci permettra l'exécution du programme par le PIC sans aucune connexion au module ICD.

Pour programmer le PIC il faut choisir le kit ICD 2 comme programmateur. Il faut noter que le kit ICD peut soit être utilisé comme débogueur soit comme programmateur, mais pas les deux en même temps.

- Choisir ICD 2 comme outils de programmation

➔ Enlever ICD 2 comme débogueur (si besoin) :

Debugger → Select Tool → None (ou MPLAB SIM)

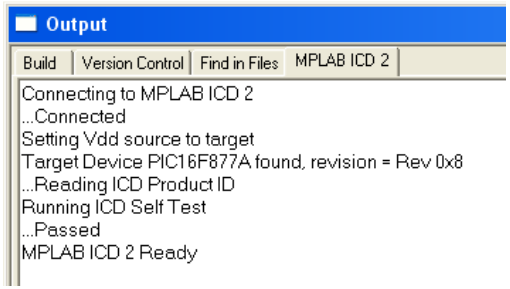
➔ Choisir ICD 2 comme programmeur :

Programmer → Select Programmer → MPLAB ICD 2

➔ Initialiser la communication avec le module :

Programmer → Connect

Si tout se passe bien, on obtient l'affichage suivant :



- Programmation du PIC

➔ Construire votre projet

➔ Programmer le PIC : (interrupteur en position haute, LED rouge allumée)

Programmer → Program

- Exécution du programme sur le PIC

➔ Pour déconnecter électriquement le kit ICD sans déconnecter le câble RJ11, basculer l'interrupteur en position basse (la LED rouge s'éteint). Le programme s'exécute sur le PIC (faire un reset : appuyer sur le bouton au dessous du connecteur RJ11).

Q- Vérifier le fonctionnement de l'application en autonomie.

Q - Proposer et tester une solution plus convenable pour la gestion du bouton poussoir.

Fin du premier TP : vous devez être familiarisé avec MPLAB IDE/ICD et maitriser les principes de mise au point d'un programme.

II - Projet Chenille Lumineuse et notions avancées (2° séance)

Le but de cette séance de TP est de réaliser une petite animation à LED où l'utilisateur pourra faire varier la vitesse de défilement ainsi que choisir le programme d'animation.

Cette application permettra de s'intéresser aux notions avancées de la programmation des PIC :

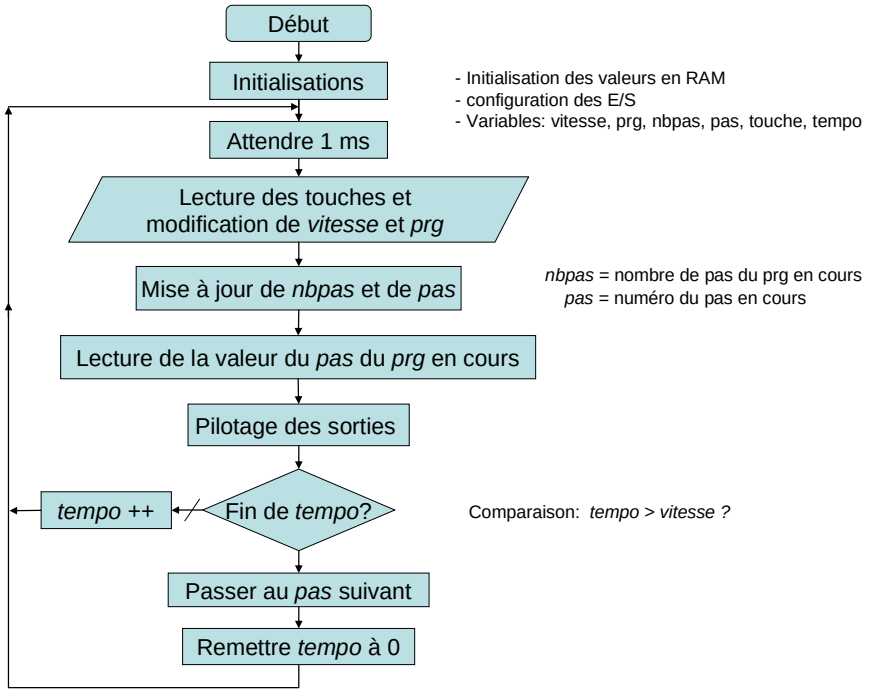
- adressage indirect,
- bits de configuration,
- réservation statique d'espace mémoire,
- utilisation du linker,

1- Présentation

- Description de l'application (cahier des charges)

Il s'agit de réaliser une chenille lumineuse (chenillard) à 8 voies (commande indépendante de 8 sorties). Plusieurs séquences d'animation seront programmées. L'utilisateur pourra changer de séquence d'animation ainsi que la vitesse de défilement grâce aux 4 boutons poussoirs.

- Organigramme global de l'application



- Choix technologiques

L'application sera basée sur les cartes « process » et « mini » (PIC 16F877A à 20MHz).
Les séquences d'animation seront stockées en mémoire RAM.
On ne mettra pas en œuvre d'interruptions (rendez vous en 4GE).

2- Manipulations 1 : création du projet

→ Créer un nouveau répertoire dans c: /temp.
→ Copier dans ce répertoire les fichiers suivant (présent dans « Mes Documents ») :
- TPuC_2.asm
- IDASM16.ASM
- 16f877a.lkr

→ Ouvrir MPLAB et créer un projet dans le répertoire C:\Temp.
→ Ajouter à votre projet en tant que « *source code* » **les copies** des deux fichiers .asm.
→ Ajouter à votre projet comme « *linker script* » la copie de *16f877a.lkr*.

Contrairement au TP précédent, ***mplink*** (le linkeur) sera utilisé pour construire ce projet (le fait d'inclure un fichier .lkr au projet provoque automatiquement l'utilisation de *mplink*). L'utilisation de *mplink* ajoute de nombreuses commandes et macro, notamment pour la gestion et la réservation des variables.

Pour ce TP, seul le fichier TPuC_2.asm est à modifier.

3- Analyse du code assembleur

Q- Repérer les 3 parties du programme : configuration, déclaration (variables, fonctions), implémentation.

- Fonctions :

Q- Combien de fonctions sont utilisées par le programme principal ?
Q- Donner leur nom et leur rôle et où elles sont implémentées (où est le code de la fonction).
Q- Justifier la déclaration `extern copy_init_data` **dans TPuC2.asm.**

- Variables et mémoires

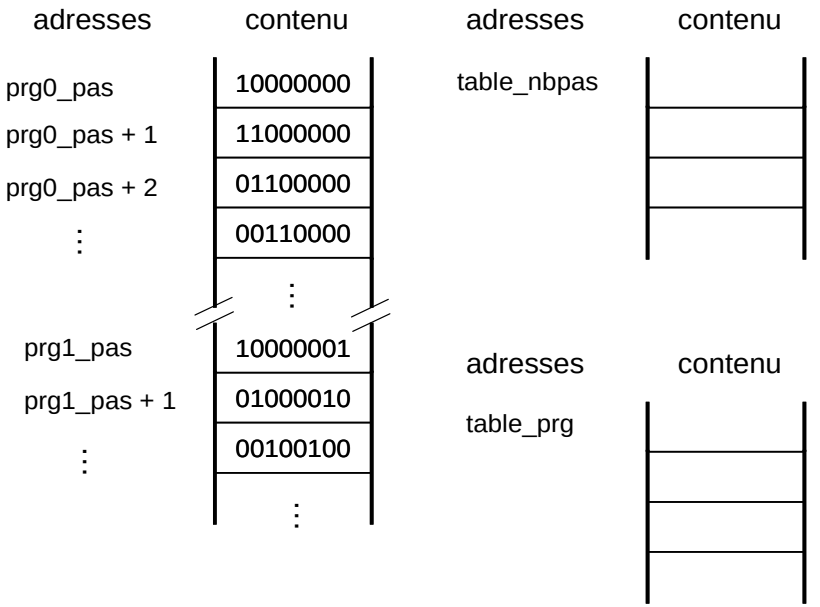
Grâce à *mplink*, il est possible de réserver des espaces en mémoire RAM pour les variables. Ceci est fait dans des sections comme ***udata*** et ***idata*** (il en existe d'autre).

La section ***udata*** permet de réserver des espaces en mémoire RAM dont les valeurs ne sont pas initialisées. On doit préciser la taille de l'espace mémoire (en octet) à réserver.

La section ***idata*** permet de réserver et d'initialiser des espaces mémoires RAM. La taille de l'espace mémoire à allouer est directement déduite par le nombre des valeurs initiales.

Quelque soit le type de section utilisé, chaque espace mémoire est nommé, c'est-à-dire qu'il est identifié par une variable. Cette variable est équivalente à l'adresse de l'espace mémoire (ou du premier élément dans le cas d'un tableau).

Q- Compléter la figure suivante représentant la mémoire RAM : ajouter le contenu des variables *table_nbpas* et *table_prg*.



Q- Quel est le rôle de chacun de ces tableaux ?

Remarque : pour la famille PIC16 les valeurs initiales sont stockées dans la ROM et doivent ensuite être copiées dans la RAM au début du programme. D'où l'utilisation de la fonction `copy_init_data`.

Q- Analyser le code de la fonction copy_init_data. Quelles techniques de programmation sont utilisées par cette fonction ?

4- Modifications et validation du code

- Fonction `Tempo_Ws`

Q- Compléter le code suivant provoquant le warning (Message):

```
Tempo_Wms_B1:
  movf      Tempo_Wms_value ;
  btfsc     STATUS, Z
```

Q- Compléter le code de la fonction Tempo_Wms : il manque la valeur de configuration du TIMER1 (T1CON) ainsi que les valeurs de TMR1H et TMR1L. Cette fonction doit permettre de faire une pause de W millisecondes.

Q- Vérifier par simulation le temps d'attente de votre fonction :

→ Passer en débogage par simulation (MPLAB SIM).

→ Ouvrir l'outil ***StopWatch*** qui permet de compter le nombre de cycles (et le temps) exécutés :

Debugger → StopWatch.

→ Mettre un *breakpoint* sur la ligne `call Tempo_Wms.`

→ Exécuter le programme (*Run*).

→ Quand le programme bloque sur le *breakpoint*, cliquer sur « Zero » de la fenêtre *StopWatch* (remise à 0 des compteurs de cycles et de temps).

→ Exécuter la fonction (conseil : sans faire du pas à pas dans le code de la fonction).

→ Le temps écoulé est donné dans la fenêtre *Stopwatch*.

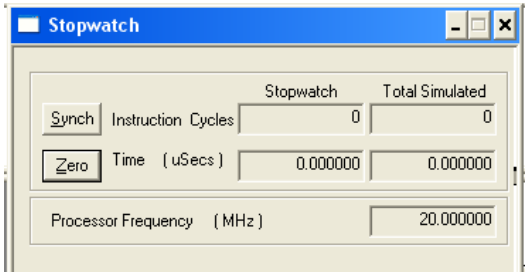


Figure 10 : Fenêtre Stopwatch.

Q- Ajuster les valeurs du TIMER1 pour obtenir un temps d'attente d'environ W ms. Donner l'erreur de votre fonction pour une attente des 1 ms et de 250 ms. Conclusions.

- Analyse des touches

La gestion des touches est faite dans la fonction `Analyse_BP`. Cette fonction n'est pas à modifier.

Q- Quel est le but de la première instruction : `movf touche, f` **?**

Q- Donner l'organigramme de la fonction *Analyse_BP*.

Q- Expliquer le rôle de la variable « *touche* ».

Q- Justifier la présence des trois `return` **de cette fonction en termes d'algorithme, de vitesse d'exécution et de lisibilité du programme. Juger la pertinence de chacun de ces trois critères.**

- Accès au nombre de pas de la séquence en cours :

Q- Compléter le programme principal en insérant le code permettant de mettre dans la variable *nbpas* le nombre de pas de la séquence en cours. Le nombre de pas de la séquence encours *prg* est la valeur *table_nbpas[prg]* (se reporter à la question 3 : Analyse du code assembleur).

Rappels : - `movlw table_nbpas`

charge la valeur littérale de *table_nbpas* dans W, c'est-à-dire la valeur équivalente au nom « *table_nbpas* ». Il s'agit d'une adresse. Cette instruction correspond en langage C à &(table_nbpas[0]) -> W (ou plus simplement table_nbpas -> W).

- `movf table_nbpas, W`

charge dans W le contenu du registre *table_nbpas*, c'est-à-dire la valeur contenue à l'adresse mémoire représentée par *table_nbpas*. Ceci correspond à table_nbpas[0] -> W en langage C.

- pour accéder aux éléments d'un tableau, il faut utiliser l'adressage indirect… (cf. annexe page 30).

Q- Valider par simulation sur PC le code ajouté.

- Accès au pas de la séquence en cours :

Q- Même travail pour l'accès à la valeur du pas de la séquence en cours : charger dans W la valeur de *table_pas[prg][pas]*.

Q- Valider par simulation sur PC le code ajouté.

5- Programmation du PIC

- Etude des bits de configuration

→ Compiler le programme. Ouvrir hors de MPLAB le fichier .lst du **nom de votre projet**.

Q- Qu'est ce que l'adresse 2007 ?

Q- Qu'elle est la valeur mise à cette adresse ? A quoi correspond cette valeur ?

- Etude de la gestion des espaces mémoires

→ Ouvrir hors de MPLAB le fichier *.map* .

Q- Que contient ce fichier ?

Q- Qu'elles sont les adresses attribuées à *prg0_pas* et *table_nbpas* en RAM ?

Q- A qu'elle adresse ROM commence le stockage des valeurs initiales des tableaux ?

Q- Combien d'octets en RAM sont encore libres dans la bank0 ?

- Validation du fonctionnement

Q- Valider votre programme sur les cartes PIC. Faire des tests !

6- Améliorations

Q- Ajouter une séquence lumineuse de votre composition.

Q- Modifier le code pour permettre une incrémentation et une décrémentation rapides si on maintient les boutons poussoirs enfoncés.

Fin du deuxième TP : les microcontrôleurs ? ça fait pas peur !

Annexes :

Extraits de la documentation du 16F877

Schéma des cartes « Mini » et « process » et Implémentation des cartes

Code assembleur TPuC_1.asm (séance 1)

Code assembleur TPuC_2.asm (séance 2)

FIGURE 2-3: PIC16F876A/877A REGISTER FILE MAP

Bank 0	File Address	Bank 1	File Address	Bank 2	File Address	Bank 3	File Address
Indirect addr.(*)	00h	Indirect addr.(*)	80h	Indirect addr.(*)	100h	Indirect addr.(*)	180h
TMR0	01h	OPTION_REG	81h	TMR0	101h	OPTION_REG	181h
PCL	02h	PCL	82h	PCL	102h	PCL	182h
STATUS	03h	STATUS	83h	STATUS	103h	STATUS	183h
FSR	04h	FSR	84h	FSR	104h	FSR	184h
PORTA	05h	TRISA	85h		105h		185h
PORTB	06h	TRISB	86h	PORTB	106h	TRISB	186h
PORTC	07h	TRISC	87h		107h		187h
PORTD(1)	08h	TRISD(1)	88h		108h		188h
PORTE(1)	09h	TRISE(1)	89h		109h		189h
PCLATH	0Ah	PCLATH	8Ah	PCLATH	10Ah	PCLATH	18Ah
INTCON	0Bh	INTCON	8Bh	INTCON	10Bh	INTCON	18Bh
PIR1	0Ch	PIE1	8Ch	EEDATA	10Ch	EECON1	18Ch
PIR2	0Dh	PIE2	8Dh	EEADR	10Dh	EECON2	18Dh
TMR1L	0Eh	PCON	8Eh	EEDATH	10Eh	Reserved(2)	18Eh
TMR1H	0Fh		8Fh	EEADRH	10Fh	Reserved(2)	18Fh
T1CON	10h		90h		110h		190h
TMR2	11h	SSPCON2	91h		111h		191h
T2CON	12h	PR2	92h		112h		192h
SSPBUF	13h	SSPADD	93h		113h		193h
SSPCON	14h	SSPSTAT	94h		114h		194h
CCPR1L	15h		95h		115h		195h
CCPR1H	16h		96h		116h		196h
CCP1CON	17h		97h	General Purpose Register 16 Bytes	117h	General Purpose Register 16 Bytes	197h
RCSTA	18h	TXSTA	98h		118h		198h
TXREG	19h	SPBRG	99h		119h		199h
RCREG	1Ah		9Ah		11Ah		19Ah
CCPR2L	1Bh		9Bh		11Bh		19Bh
CCPR2H	1Ch	CMCON	9Ch		11Ch		19Ch
CCP2CON	1Dh	CVRCON	9Dh		11Dh		19Dh
ADRESH	1Eh	ADRESL	9Eh		11Eh		19Eh
ADCON0	1Fh	ADCON1	9Fh		11Fh		19Fh
General Purpose Register 96 Bytes	20h	General Purpose Register 80 Bytes	A0h	General Purpose Register 80 Bytes	120h	General Purpose Register 80 Bytes	1A0h
	7Fh	accesses 70h-7Fh	EFh F0h FFh	accesses 70h-7Fh	16Fh 170h 17Fh	accesses 70h - 7Fh	1EFh 1F0h 1FFh

Unimplemented data memory locations, read as '0'.
* Not a physical register.

Note 1: These registers are not implemented on the PIC16F876A.
2: These registers are reserved; maintain these registers clear.

Figure 11 : Organisation mémoire

TABLE 2-1: SPECIAL FUNCTION REGISTER SUMMARY

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on: POR, BOR	Details on page:
Bank 0											
00h(3)	INDF	Addressing this location uses contents of FSR to address data memory (not a physical register)								0000 0000	31, 150
01h	TMR0	Timer0 Module Register								xxxx xxxx	55, 150
02h(3)	PCL	Program Counter (PC) Least Significant Byte								0000 0000	30, 150
03h(3)	STATUS	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx	22, 150
04h(3)	FSR	Indirect Data Memory Address Pointer								xxxx xxxx	31, 150
05h	PORTA	—	—	PORTA Data Latch when written: PORTA pins when read						--0x 0000	43, 150
06h	PORTB	PORTB Data Latch when written: PORTB pins when read								xxxx xxxx	45, 150
07h	PORTC	PORTC Data Latch when written: PORTC pins when read								xxxx xxxx	47, 150
08h(4)	PORTD	PORTD Data Latch when written: PORTD pins when read								xxxx xxxx	48, 150
09h(4)	PORTE	—	—	—	—	—	RE2	RE1	RE0	---- -xxx	49, 150
0Ah(1,3)	PCLATH	—	—	—	Write Buffer for the upper 5 bits of the Program Counter					---0 0000	30, 150
0Bh(3)	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF	0000 000x	24, 150
0Ch	PIR1	PSPIF(3)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000	26, 150
0Dh	PIR2	—	CMIF	—	EEIF	BCLIF	—	—	CCP2IF	-0-0 0--0	28, 150
0Eh	TMR1L	Holding Register for the Least Significant Byte of the 16-bit TMR1 Register								xxxx xxxx	60, 150
0Fh	TMR1H	Holding Register for the Most Significant Byte of the 16-bit TMR1 Register								xxxx xxxx	60, 150
10h	T1CON	—	—	T1CKPS1	T1CKPS0	T1OSCEN	$\overline{T1SYNC}$	TMR1CS	TMR1ON	--00 0000	57, 150
11h	TMR2	Timer2 Module Register								0000 0000	62, 150
12h	T2CON	—	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0	-000 0000	61, 150
13h	SSPBUF	Synchronous Serial Port Receive Buffer/Transmit Register								xxxx xxxx	79, 150
14h	SSPCON	WCOL	SSPOV	SSPEN	CKP	SSPM3	SSPM2	SSPM1	SSPM0	0000 0000	82, 82, 150
15h	CCPR1L	Capture/Compare/PWM Register 1 (LSB)								xxxx xxxx	63, 150
16h	CCPR1H	Capture/Compare/PWM Register 1 (MSB)								xxxx xxxx	63, 150
17h	CCP1CON	—	—	CCP1X	CCP1Y	CCP1M3	CCP1M2	CCP1M1	CCP1M0	--00 0000	64, 150
18h	RCSTA	SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D	0000 000x	112, 150
19h	TXREG	USART Transmit Data Register								0000 0000	118, 150
1Ah	RCREG	USART Receive Data Register								0000 0000	118, 150
1Bh	CCPR2L	Capture/Compare/PWM Register 2 (LSB)								xxxx xxxx	63, 150
1Ch	CCPR2H	Capture/Compare/PWM Register 2 (MSB)								xxxx xxxx	63, 150
1Dh	CCP2CON	—	—	CCP2X	CCP2Y	CCP2M3	CCP2M2	CCP2M1	CCP2M0	--00 0000	64, 150
1Eh	ADRESH	A/D Result Register High Byte								xxxx xxxx	133, 150
1Fh	ADCON0	ADCS1	ADCS0	CHS2	CHS1	CHS0	GO/$\overline{DONE}$	—	ADON	0000 00-0	127, 150

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

Note 1: The upper byte of the program counter is not directly accessible. PCLATH is a holding register for the PC<12:8>, whose contents are transferred to the upper byte of the program counter.
2: Bits PSPIE and PSPIF are reserved on PIC16F873A/876A devices; always maintain these bits clear.
3: These registers can be addressed from any bank.
4: PORTD, PORTE, TRISD and TRISE are not implemented on PIC16F873A/876A devices, read as '0'.
5: Bit 4 of EEADRH implemented only on the PIC16F876A/877A devices.

Figure 12 : Description des registres

TABLE 2-1: SPECIAL FUNCTION REGISTER SUMMARY (CONTINUED)

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on: POR, BOR	Details on page:
Bank 1											
80h(3)	INDF	Addressing this location uses contents of FSR to address data memory (not a physical register)								0000 0000	31, 150
81h	OPTION_REG	$\overline{RBPU}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111	23, 150
82h(3)	PCL	Program Counter (PC) Least Significant Byte								0000 0000	30, 150
83h(3)	STATUS	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx	22, 150
84h(3)	FSR	Indirect Data Memory Address Pointer								xxxx xxxx	31, 150
85h	TRISA	—	—	PORTA Data Direction Register						--11 1111	43, 150
86h	TRISB	PORTB Data Direction Register								1111 1111	45, 150
87h	TRISC	PORTC Data Direction Register								1111 1111	47, 150
88h(4)	TRISD	PORTD Data Direction Register								1111 1111	48, 151
89h(4)	TRISE	IBF	OBF	IBOV	PSPMODE	—	PORTE Data Direction bits			0000 -111	50, 151
8Ah(1,3)	PCLATH	—	—	—	Write Buffer for the upper 5 bits of the Program Counter					---0 0000	30, 150
8Bh(3)	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF	0000 000x	24, 150
8Ch	PIE1	PSPIE(2)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000	25, 151
8Dh	PIE2	—	CMIE	—	EEIE	BCLIE	—	—	CCP2IE	-0-0 0--0	27, 151
8Eh	PCON	—	—	—	—	—	—	$\overline{POR}$	$\overline{BOR}$	---- --qq	29, 151
8Fh	—	Unimplemented								—	—
90h	—	Unimplemented								—	—
91h	SSPCON2	GCEN	ACKSTAT	ACKDT	ACKEN	RCEN	PEN	RSEN	SEN	0000 0000	83, 151
92h	PR2	Timer2 Period Register								1111 1111	62, 151
93h	SSPADD	Synchronous Serial Port (I^2C mode) Address Register								0000 0000	79, 151
94h	SSPSTAT	SMP	CKE	D/$\overline{A}$	P	S	R/$\overline{W}$	UA	BF	0000 0000	79, 151
95h	—	Unimplemented								—	—
96h	—	Unimplemented								—	—
97h	—	Unimplemented								—	—
98h	TXSTA	CSRC	TX9	TXEN	SYNC	—	BRGH	TRMT	TX9D	0000 -010	111, 151
99h	SPBRG	Baud Rate Generator Register								0000 0000	113, 151
9Ah	—	Unimplemented								—	—
9Bh	—	Unimplemented								—	—
9Ch	CMCON	C2OUT	C1OUT	C2INV	C1INV	CIS	CM2	CM1	CM0	0000 0111	135, 151
9Dh	CVRCON	CVREN	CVROE	CVRR	—	CVR3	CVR2	CVR1	CVR0	000- 0000	141, 151
9Eh	ADRESL	A/D Result Register Low Byte								xxxx xxxx	133, 151
9Fh	ADCON1	ADFM	ADCS2	—	—	PCFG3	PCFG2	PCFG1	PCFG0	00-- 0000	128, 151

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

Note 1: The upper byte of the program counter is not directly accessible. PCLATH is a holding register for the PC<12:8>, whose contents are transferred to the upper byte of the program counter.

2: Bits PSPIE and PSPIF are reserved on PIC16F873A/876A devices; always maintain these bits clear.

3: These registers can be addressed from any bank.

4: PORTD, PORTE, TRISD and TRISE are not implemented on PIC16F873A/876A devices, read as '0'.

5: Bit 4 of EEADRH implemented only on the PIC16F876A/877A devices.

TABLE 2-1: SPECIAL FUNCTION REGISTER SUMMARY (CONTINUED)

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on: POR, BOR	Details on page:
Bank 2											
100h(3)	INDF	Addressing this location uses contents of FSR to address data memory (not a physical register)								0000 0000	31, 150
101h	TMR0	Timer0 Module Register								xxxx xxxx	55, 150
102h(3)	PCL	Program Counter's (PC) Least Significant Byte								0000 0000	30, 150
103h(3)	STATUS	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx	22, 150
104h(3)	FSR	Indirect Data Memory Address Pointer								xxxx xxxx	31, 150
105h	—	Unimplemented								—	—
106h	PORTB	PORTB Data Latch when written: PORTB pins when read								xxxx xxxx	45, 150
107h	—	Unimplemented								—	—
108h	—	Unimplemented								—	—
109h	—	Unimplemented								—	—
10Ah(1,3)	PCLATH	—	—	—	Write Buffer for the upper 5 bits of the Program Counter					---0 0000	30, 150
10Bh(3)	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF	0000 000x	24, 150
10Ch	EEDATA	EEPROM Data Register Low Byte								xxxx xxxx	39, 151
10Dh	EEADR	EEPROM Address Register Low Byte								xxxx xxxx	39, 151
10Eh	EEDATH	—	—	EEPROM Data Register High Byte						--xx xxxx	39, 151
10Fh	EEADRH	—	—	—	—(5)	EEPROM Address Register High Byte				---- xxxx	39, 151
Bank 3											
180h(3)	INDF	Addressing this location uses contents of FSR to address data memory (not a physical register)								0000 0000	31, 150
181h	OPTION_REG	$\overline{RBPU}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111	23, 150
182h(3)	PCL	Program Counter (PC) Least Significant Byte								0000 0000	30, 150
183h(3)	STATUS	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx	22, 150
184h(3)	FSR	Indirect Data Memory Address Pointer								xxxx xxxx	31, 150
185h	—	Unimplemented								—	—
186h	TRISB	PORTB Data Direction Register								1111 1111	45, 150
187h	—	Unimplemented								—	—
188h	—	Unimplemented								—	—
189h	—	Unimplemented								—	—
18Ah(1,3)	PCLATH	—	—	—	Write Buffer for the upper 5 bits of the Program Counter					---0 0000	30, 150
18Bh(3)	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF	0000 000x	24, 150
18Ch	EECON1	EEPGD	—	—	—	WRERR	WREN	WR	RD	x--- x000	34, 151
18Dh	EECON2	EEPROM Control Register 2 (not a physical register)								---- ----	39, 151
18Eh	—	Reserved; maintain clear								0000 0000	—
18Fh	—	Reserved; maintain clear								0000 0000	—

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

Note 1: The upper byte of the program counter is not directly accessible. PCLATH is a holding register for the PC<12:8>, whose contents are transferred to the upper byte of the program counter.

2: Bits PSPIE and PSPIF are reserved on PIC16F873A/876A devices; always maintain these bits clear.

3: These registers can be addressed from any bank.

4: PORTD, PORTE, TRISD and TRISE are not implemented on PIC16F873A/876A devices, read as '0'.

5: Bit 4 of EEADRH implemented only on the PIC16F876A/877A devices.

2.2.2.1 Status Register

The Status register contains the arithmetic status of the ALU, the Reset status and the bank select bits for data memory.

The Status register can be the destination for any instruction, as with any other register. If the Status register is the destination for an instruction that affects the Z, DC or C bits, then the write to these three bits is disabled. These bits are set or cleared according to the device logic. Furthermore, the $\overline{TO}$ and $\overline{PD}$ bits are not writable, therefore, the result of an instruction with the Status register as destination may be different than intended.

For example, `CLRF STATUS`, will clear the upper three bits and set the Z bit. This leaves the Status register as `000u u1uu` (where `u` = unchanged).

It is recommended, therefore, that only `BCF`, `BSF`, `SWAPF` and `MOVWF` instructions are used to alter the Status register because these instructions do not affect the Z, C or DC bits from the Status register. For other instructions not affecting any status bits, see **Section 15.0 "Instruction Set Summary"**.

Note:	The C and DC bits operate as a $\overline{borrow}$ and $\overline{digit\ borrow}$ bit, respectively, in subtraction. See the `SUBLW` and `SUBWF` instructions for examples.

REGISTER 2-1: STATUS REGISTER (ADDRESS 03h, 83h, 103h, 183h)

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

bit 7 **IRP:** Register Bank Select bit (used for indirect addressing)
1 = Bank 2, 3 (100h-1FFh)
0 = Bank 0, 1 (00h-FFh)

bit 6-5 **RP1:RP0**: Register Bank Select bits (used for direct addressing)
11 = Bank 3 (180h-1FFh)
10 = Bank 2 (100h-17Fh)
01 = Bank 1 (80h-FFh)
00 = Bank 0 (00h-7Fh)
Each bank is 128 bytes.

bit 4 **$\overline{TO}$**: Time-out bit
1 = After power-up, `CLRWDT` instruction or `SLEEP` instruction
0 = A WDT time-out occurred

bit 3 **$\overline{PD}$**: Power-down bit
1 = After power-up or by the `CLRWDT` instruction
0 = By execution of the `SLEEP` instruction

bit 2 **Z**: Zero bit
1 = The result of an arithmetic or logic operation is zero
0 = The result of an arithmetic or logic operation is not zero

bit 1 **DC**: Digit carry/$\overline{borrow}$ bit (`ADDWF`, `ADDLW`, `SUBLW`, `SUBWF` instructions)
(for $\overline{borrow}$, the polarity is reversed)
1 = A carry-out from the 4th low order bit of the result occurred
0 = No carry-out from the 4th low order bit of the result

bit 0 **C**: Carry/$\overline{borrow}$ bit (`ADDWF`, `ADDLW`, `SUBLW`, `SUBWF` instructions)
1 = A carry-out from the Most Significant bit of the result occurred
0 = No carry-out from the Most Significant bit of the result occurred

Note: For $\overline{borrow}$, the polarity is reversed. A subtraction is executed by adding the two's complement of the second operand. For rotate (`RRF`, `RLF`) instructions, this bit is loaded with either the high, or low order bit of the source register.

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
- n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

Figure 13 : Description du register STATUS

2.5 Indirect Addressing, INDF and FSR Registers

The INDF register is not a physical register. Addressing the INDF register will cause indirect addressing.

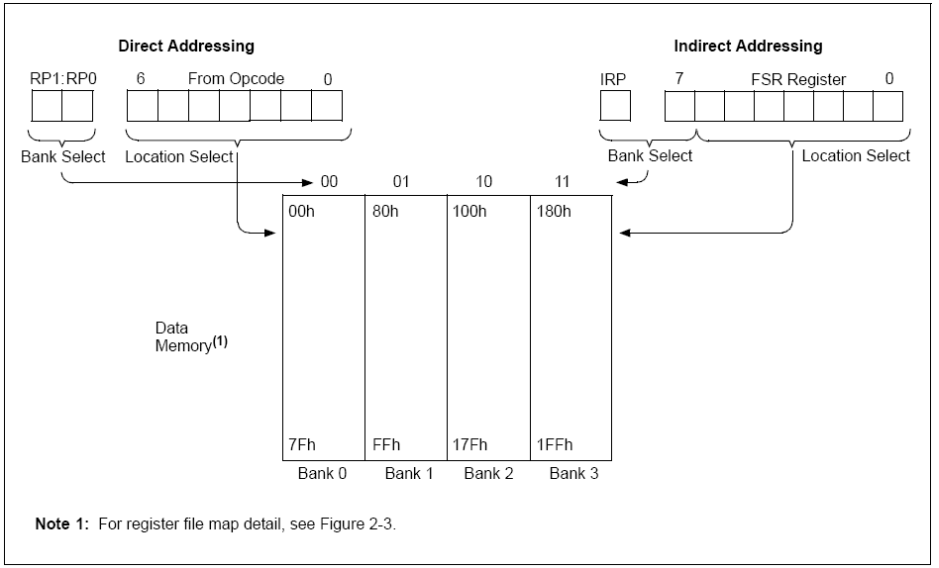
Indirect addressing is possible by using the INDF register. Any instruction using the INDF register actually accesses the register pointed to by the File Select Register, FSR. Reading the INDF register itself, indirectly (FSR = 0) will read 00h. Writing to the INDF register indirectly results in a no operation (although status bits may be affected). An effective 9-bit address is obtained by concatenating the 8-bit FSR register and the IRP bit (Status<7>) as shown in Figure 2-6.

A simple program to clear RAM locations 20h-2Fh using indirect addressing is shown in Example 2-2.

EXAMPLE 2-2: INDIRECT ADDRESSING

```
        MOVLW  0x20     ;initialize pointer
        MOVWF  FSR      ;to RAM
NEXT    CLRF   INDF     ;clear INDF register
        INCF   FSR,F    ;inc pointer
        BTFSS  FSR,4    ;all done?
        GOTO   NEXT     ;no clear next
CONTINUE
        :               ;yes continue
```

FIGURE 2-6: DIRECT/INDIRECT ADDRESSING



Note 1: For register file map detail, see Figure 2-3.

Figure 14 : Adressage indirect

4.0 I/O PORTS

Some pins for these I/O ports are multiplexed with an alternate function for the peripheral features on the device. In general, when a peripheral is enabled, that pin may not be used as a general purpose I/O pin.

Additional information on I/O ports may be found in the PICmicro™ Mid-Range Reference Manual (DS33023).

4.1 PORTA and the TRISA Register

PORTA is a 6-bit wide, bidirectional port. The corresponding data direction register is TRISA. Setting a TRISA bit (= 1) will make the corresponding PORTA pin an input (i.e., put the corresponding output driver in a High-Impedance mode). Clearing a TRISA bit (= 0) will make the corresponding PORTA pin an output (i.e., put the contents of the output latch on the selected pin).

Reading the PORTA register reads the status of the pins, whereas writing to it will write to the port latch. All write operations are read-modify-write operations. Therefore, a write to a port implies that the port pins are read, the value is modified and then written to the port data latch.

Pin RA4 is multiplexed with the Timer0 module clock input to become the RA4/T0CKI pin. The RA4/T0CKI pin is a Schmitt Trigger input and an open-drain output. All other PORTA pins have TTL input levels and full CMOS output drivers.

Other PORTA pins are multiplexed with analog inputs and the analog VREF input for both the A/D converters and the comparators. The operation of each pin is selected by clearing/setting the appropriate control bits in the ADCON1 and/or CMCON registers.

Note:	On a Power-on Reset, these pins are configured as analog inputs and read as '0'. The comparators are in the off (digital) state.

The TRISA register controls the direction of the port pins even when they are being used as analog inputs. The user must ensure the bits in the TRISA register are maintained set when using them as analog inputs.

EXAMPLE 4-1: INITIALIZING PORTA

```
BCF     STATUS, RP0   ;
BCF     STATUS, RP1   ; Bank0
CLRF    PORTA         ; Initialize PORTA by
                      ; clearing output
                      ; data latches
BSF     STATUS, RP0   ; Select Bank 1
MOVLW   0x06          ; Configure all pins
MOVWF   ADCON1        ; as digital inputs
MOVLW   0xCF          ; Value used to
                      ; initialize data
                      ; direction
MOVWF   TRISA         ; Set RA<3:0> as inputs
                      ; RA<5:4> as outputs
                      ; TRISA<7:6>are always
                      ; read as '0'.
```

FIGURE 4-1: BLOCK DIAGRAM OF RA3:RA0 PINS

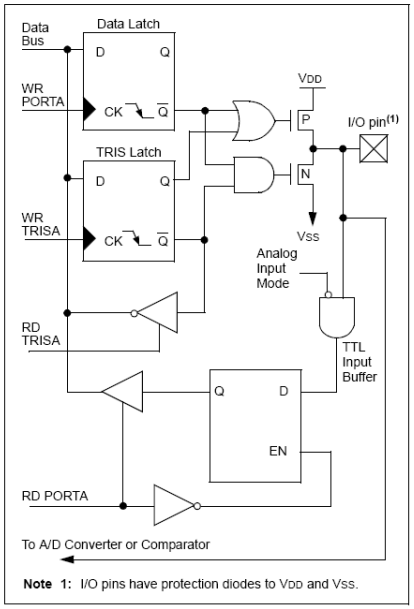


Figure 15 : Description du PORTA

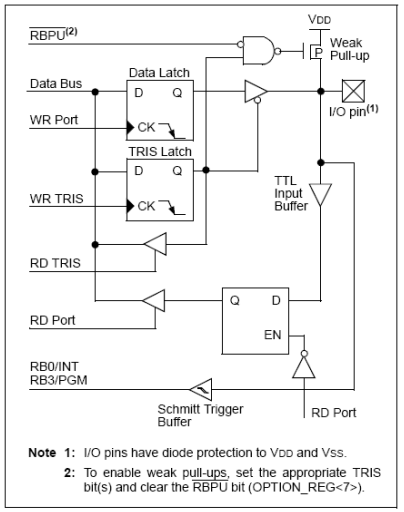
4.2 PORTB and the TRISB Register

PORTB is an 8-bit wide, bidirectional port. The corresponding data direction register is TRISB. Setting a TRISB bit (= 1) will make the corresponding PORTB pin an input (i.e., put the corresponding output driver in a High-Impedance mode). Clearing a TRISB bit (= 0) will make the corresponding PORTB pin an output (i.e., put the contents of the output latch on the selected pin).

Three pins of PORTB are multiplexed with the In-Circuit Debugger and Low-Voltage Programming function: RB3/PGM, RB6/PGC and RB7/PGD. The alternate functions of these pins are described in **Section 14.0 "Special Features of the CPU"**.

Each of the PORTB pins has a weak internal pull-up. A single control bit can turn on all the pull-ups. This is performed by clearing bit RBPU (OPTION_REG<7>). The weak pull-up is automatically turned off when the port pin is configured as an output. The pull-ups are disabled on a Power-on Reset.

FIGURE 4-4: BLOCK DIAGRAM OF RB3:RB0 PINS



Four of the PORTB pins, RB7:RB4, have an interrupt-on-change feature. Only pins configured as inputs can cause this interrupt to occur (i.e., any RB7:RB4 pin configured as an output is excluded from the interrupt-on-change comparison). The input pins (of RB7:RB4) are compared with the old value latched on the last read of PORTB. The "mismatch" outputs of RB7:RB4 are OR'ed together to generate the RB port change interrupt with flag bit RBIF (INTCON<0>).

This interrupt can wake the device from Sleep. The user, in the Interrupt Service Routine, can clear the interrupt in the following manner:

a) Any read or write of PORTB. This will end the mismatch condition.
b) Clear flag bit RBIF.

A mismatch condition will continue to set flag bit RBIF. Reading PORTB will end the mismatch condition and allow flag bit RBIF to be cleared.

The interrupt-on-change feature is recommended for wake-up on key depression operation and operations where PORTB is only used for the interrupt-on-change feature. Polling of PORTB is not recommended while using the interrupt-on-change feature.

This interrupt-on-mismatch feature, together with software configurable pull-ups on these four pins, allow easy interface to a keypad and make it possible for wake-up on key depression. Refer to the application note, *AN552, "Implementing Wake-up on Key Stroke"* (DS00552).

RB0/INT is an external interrupt input pin and is configured using the INTEDG bit (OPTION_REG<6>).

RB0/INT is discussed in detail in **Section 14.11.1 "INT Interrupt"**.

FIGURE 4-5: BLOCK DIAGRAM OF RB7:RB4 PINS

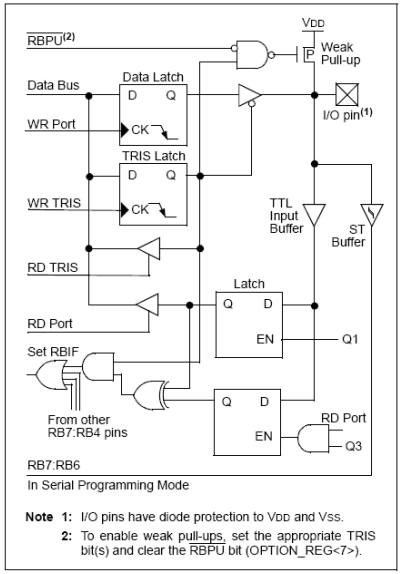


Figure 16 : Description PORTB

4.3 PORTC and the TRISC Register

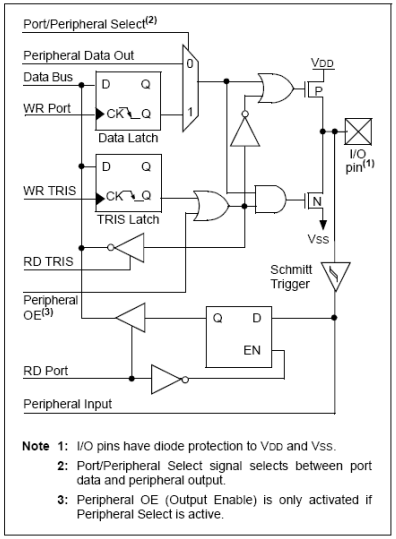
PORTC is an 8-bit wide, bidirectional port. The corresponding data direction register is TRISC. Setting a TRISC bit (= 1) will make the corresponding PORTC pin an input (i.e., put the corresponding output driver in a High-Impedance mode). Clearing a TRISC bit (= 0) will make the corresponding PORTC pin an output (i.e., put the contents of the output latch on the selected pin).

PORTC is multiplexed with several peripheral functions (Table 4-5). PORTC pins have Schmitt Trigger input buffers.

When the I^2C module is enabled, the PORTC<4:3> pins can be configured with normal I^2C levels, or with SMBus levels, by using the CKE bit (SSPSTAT<6>).

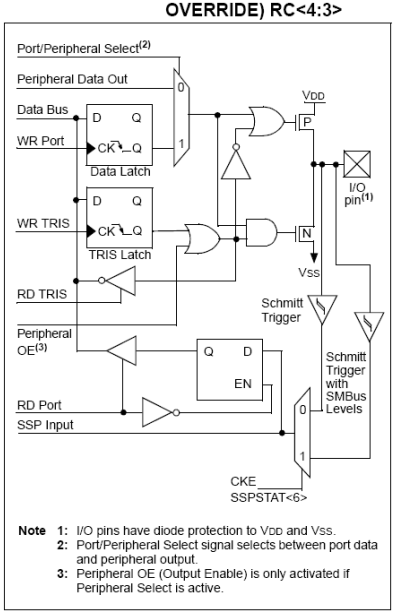
When enabling peripheral functions, care should be taken in defining TRIS bits for each PORTC pin. Some peripherals override the TRIS bit to make a pin an output, while other peripherals override the TRIS bit to make a pin an input. Since the TRIS bit override is in effect while the peripheral is enabled, read-modify-write instructions (BSF, BCF, XORWF) with TRISC as the destination, should be avoided. The user should refer to the corresponding peripheral section for the correct TRIS bit settings.

FIGURE 4-6: PORTC BLOCK DIAGRAM (PERIPHERAL OUTPUT OVERRIDE) RC<2:0>, RC<7:5>



Note 1: I/O pins have diode protection to VDD and VSS.
2: Port/Peripheral Select signal selects between port data and peripheral output.
3: Peripheral OE (Output Enable) is only activated if Peripheral Select is active.

FIGURE 4-7: PORTC BLOCK DIAGRAM (PERIPHERAL OUTPUT OVERRIDE) RC<4:3>



Note 1: I/O pins have diode protection to VDD and VSS.
2: Port/Peripheral Select signal selects between port data and peripheral output.
3: Peripheral OE (Output Enable) is only activated if Peripheral Select is active.

Figure 17 : Description PORTC

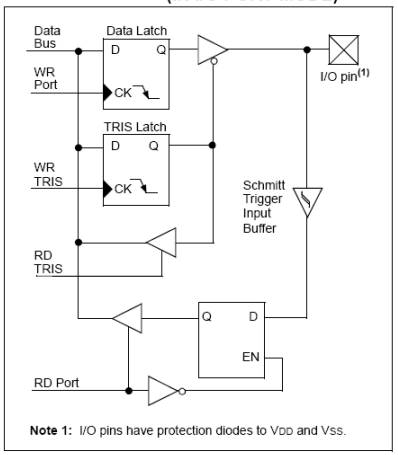
4.4 PORTD and TRISD Registers

Note: PORTD and TRISD are not implemented on the 28-pin devices.

PORTD is an 8-bit port with Schmitt Trigger input buffers. Each pin is individually configurable as an input or output.

PORTD can be configured as an 8-bit wide microprocessor port (Parallel Slave Port) by setting control bit, PSPMODE (TRISE<4>). In this mode, the input buffers are TTL.

FIGURE 4-8: PORTD BLOCK DIAGRAM (IN I/O PORT MODE)



Note 1: I/O pins have protection diodes to VDD and VSS.

TABLE 4-7: PORTD FUNCTIONS

Name	Bit#	Buffer Type	Function
RD0/PSP0	bit 0	ST/TTL(1)	Input/output port pin or Parallel Slave Port bit 0.
RD1/PSP1	bit 1	ST/TTL(1)	Input/output port pin or Parallel Slave Port bit 1.
RD2/PSP2	bit2	ST/TTL(1)	Input/output port pin or Parallel Slave Port bit 2.
RD3/PSP3	bit 3	ST/TTL(1)	Input/output port pin or Parallel Slave Port bit 3.
RD4/PSP4	bit 4	ST/TTL(1)	Input/output port pin or Parallel Slave Port bit 4.
RD5/PSP5	bit 5	ST/TTL(1)	Input/output port pin or Parallel Slave Port bit 5.
RD6/PSP6	bit 6	ST/TTL(1)	Input/output port pin or Parallel Slave Port bit 6.
RD7/PSP7	bit 7	ST/TTL(1)	Input/output port pin or Parallel Slave Port bit 7.

Legend: ST = Schmitt Trigger input, TTL = TTL input
Note 1: Input buffers are Schmitt Triggers when in I/O mode and TTL buffers when in Parallel Slave Port mode.

TABLE 4-8: SUMMARY OF REGISTERS ASSOCIATED WITH PORTD

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on: POR, BOR	Value on all other Resets
08h	PORTD	RD7	RD6	RD5	RD4	RD3	RD2	RD1	RD0	xxxx xxxx	uuuu uuuu
88h	TRISD	PORTD Data Direction Register								1111 1111	1111 1111
89h	TRISE	IBF	OBF	IBOV	PSPMODE	—	PORTE Data Direction Bits			0000 -111	0000 -111

Legend: x = unknown, u = unchanged, - = unimplemented, read as '0'. Shaded cells are not used by PORTD.

Figure 18 : Description PORTD

6.0 TIMER1 MODULE

The Timer1 module is a 16-bit timer/counter consisting of two 8-bit registers (TMR1H and TMR1L) which are readable and writable. The TMR1 register pair (TMR1H:TMR1L) increments from 0000h to FFFFh and rolls over to 0000h. The TMR1 interrupt, if enabled, is generated on overflow which is latched in interrupt flag bit, TMR1IF (PIR1<0>). This interrupt can be enabled/disabled by setting/clearing TMR1 interrupt enable bit, TMR1IE (PIE1<0>).

Timer1 can operate in one of two modes:

- As a Timer
- As a Counter

The operating mode is determined by the clock select bit, TMR1CS (T1CON<1>).

In Timer mode, Timer1 increments every instruction cycle. In Counter mode, it increments on every rising edge of the external clock input.

Timer1 can be enabled/disabled by setting/clearing control bit, TMR1ON (T1CON<0>).

Timer1 also has an internal "Reset input". This Reset can be generated by either of the two CCP modules (**Section 8.0 "Capture/Compare/PWM Modules"**). Register 6-1 shows the Timer1 Control register.

When the Timer1 oscillator is enabled (T1OSCEN is set), the RC1/T1OSI/CCP2 and RC0/T1OSO/T1CKI pins become inputs. That is, the TRISC<1:0> value is ignored and these pins read as '0'.

Additional information on timer modules is available in the PICmicro® Mid-Range MCU Family Reference Manual (DS33023).

REGISTER 6-1: T1CON: TIMER1 CONTROL REGISTER (ADDRESS 10h)

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	T1CKPS1	T1CKPS0	T1OSCEN	$\overline{\text{T1SYNC}}$	TMR1CS	TMR1ON
bit 7							bit 0

bit 7-6 **Unimplemented:** Read as '0'

bit 5-4 **T1CKPS1:T1CKPS0:** Timer1 Input Clock Prescale Select bits
11 = 1:8 prescale value
10 = 1:4 prescale value
01 = 1:2 prescale value
00 = 1:1 prescale value

bit 3 **T1OSCEN**: Timer1 Oscillator Enable Control bit
1 = Oscillator is enabled
0 = Oscillator is shut-off (the oscillator inverter is turned off to eliminate power drain)

bit 2 **$\overline{\text{T1SYNC}}$**: Timer1 External Clock Input Synchronization Control bit
When TMR1CS = 1:
1 = Do not synchronize external clock input
0 = Synchronize external clock input
When TMR1CS = 0:
This bit is ignored. Timer1 uses the internal clock when TMR1CS = 0.

bit 1 **TMR1CS**: Timer1 Clock Source Select bit
1 = External clock from pin RC0/T1OSO/T1CKI (on the rising edge)
0 = Internal clock (Fosc/4)

bit 0 **TMR1ON**: Timer1 On bit
1 = Enables Timer1
0 = Stops Timer1

Legend:		
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'
- n = Value at POR	'1' = Bit is set	'0' = Bit is cleared x = Bit is unknown

Figure 19 : Description TIMER1

6.1 Timer1 Operation in Timer Mode

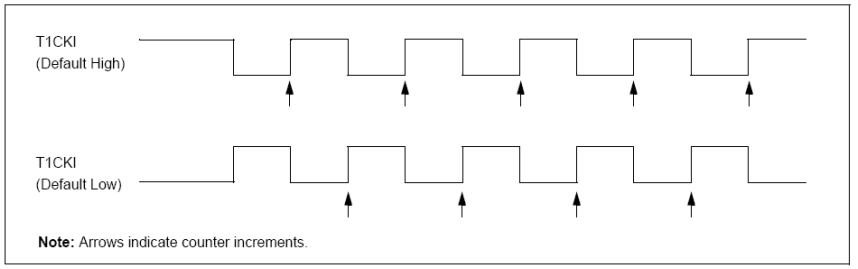
Timer mode is selected by clearing the TMR1CS (T1CON<1>) bit. In this mode, the input clock to the timer is Fosc/4. The synchronize control bit, $\overline{\text{T1SYNC}}$ (T1CON<2>), has no effect since the internal clock is always in sync.

6.2 Timer1 Counter Operation

Timer1 may operate in either a Synchronous, or an Asynchronous mode, depending on the setting of the TMR1CS bit.

When Timer1 is being incremented via an external source, increments occur on a rising edge. After Timer1 is enabled in Counter mode, the module must first have a falling edge before the counter begins to increment.

FIGURE 6-1: TIMER1 INCREMENTING EDGE



Note: Arrows indicate counter increments.

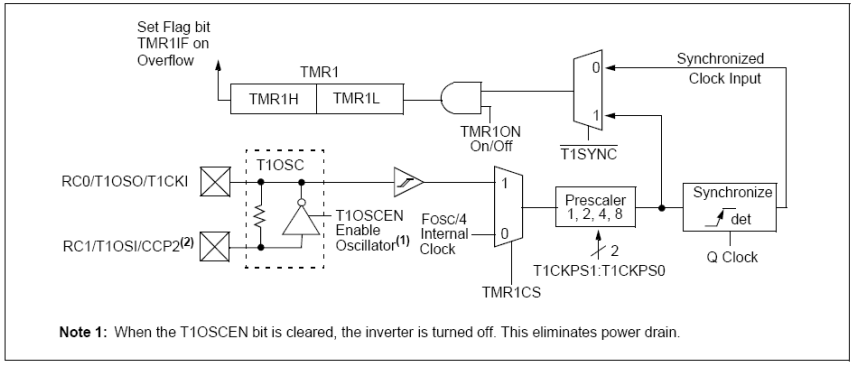
6.3 Timer1 Operation in Synchronized Counter Mode

Counter mode is selected by setting bit TMR1CS. In this mode, the timer increments on every rising edge of clock input on pin RC1/T1OSI/CCP2 when bit T1OSCEN is set, or on pin RC0/T1OSO/T1CKI when bit T1OSCEN is cleared.

If $\overline{\text{T1SYNC}}$ is cleared, then the external clock input is synchronized with internal phase clocks. The synchronization is done after the prescaler stage. The prescaler stage is an asynchronous ripple counter.

In this configuration, during Sleep mode, Timer1 will not increment even if the external clock is present since the synchronization circuit is shut-off. The prescaler, however, will continue to increment.

FIGURE 6-2: TIMER1 BLOCK DIAGRAM



Note 1: When the T1OSCEN bit is cleared, the inverter is turned off. This eliminates power drain.

Mnemonic, Operands		Description	Cycles	14-Bit Opcode MSb			LSb	Status Affected	Notes
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	lfff	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	lfff	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 NOP.

Figure 20: Jeu d'instructions des PIC 16

Annexes :

Schéma des cartes « Mini » et « process »

Implémentation des cartes

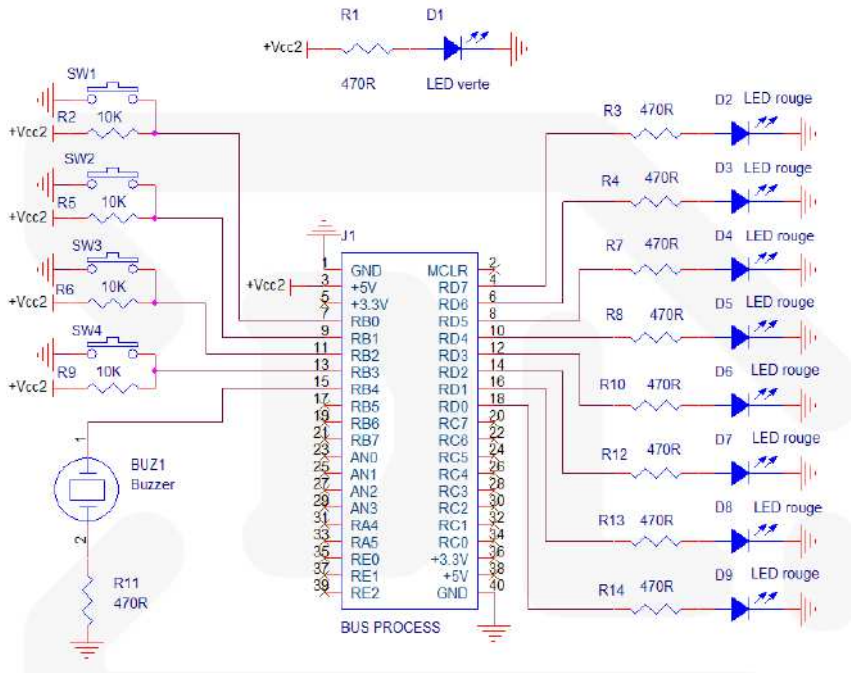
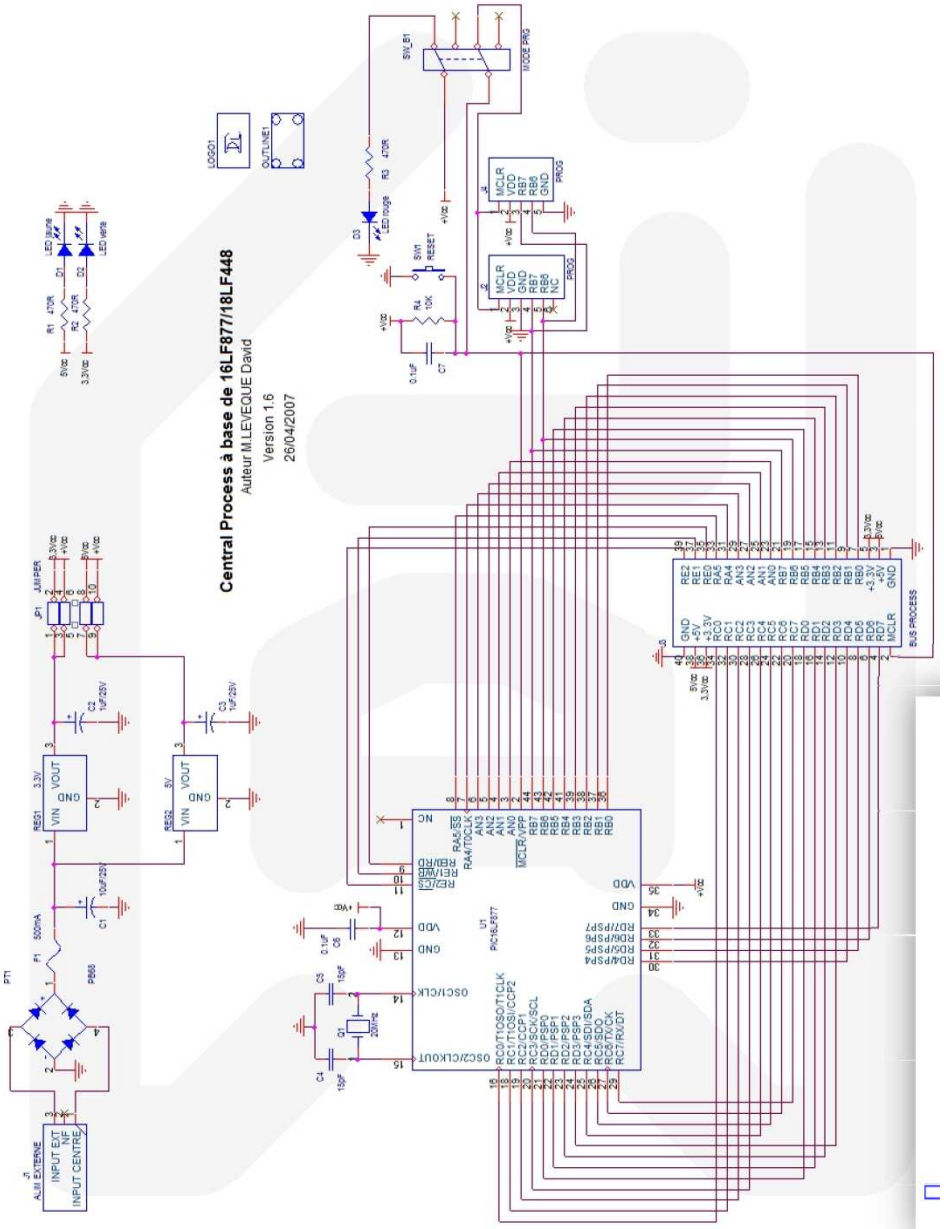


Figure 21 : Schéma de la carte « Mini »



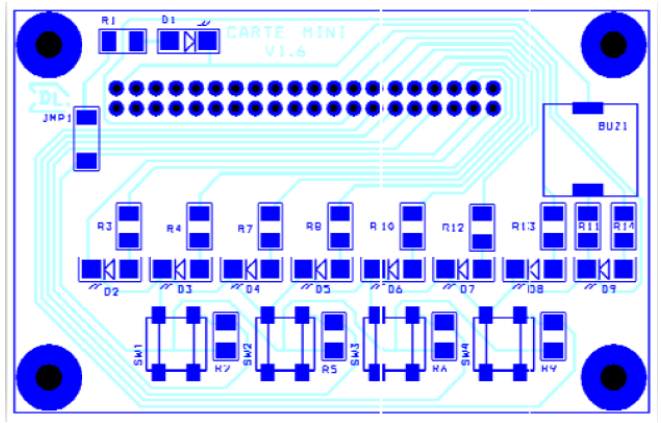


Figure 22 : Implantation de la carte « Mini »

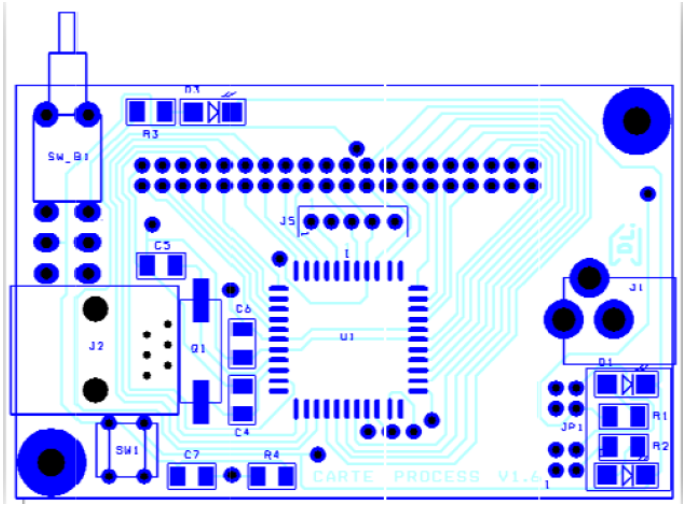


Figure 23 : Implantation de la carte « process »

Annexes : Code assembleur TPuC_1.asm (séance 1)

AVEC LES ERREURS nécessaires au TP…;)

```
#include <p16F877A.inc>  ; processor specific variable definitions
     __CONFIG _HS_OSC & _WDT_OFF & _BODEN_OFF & _LVP_OFF & _PWRTE_OFF &
_CPD_OFF & _WRT_OFF & _CP_OFF

;***** VARIABLE DEFINITIONS
Tempo_Ws_value  equ 0x0020
Tempo_Ws_mem    equ 0x0021
tempo           equ 0x0022
;*****************************************************************
          ORG     0x0000             ; processor reset vector
          goto    main              ; go to beginning of program
          ORG        0x0010

Tempo_Ws:
               movwf      Tempo_Ws_value ;dans W temps en ms à attendre
          ; INIT compteur
               bsf        STATUS, RP0 ; Bank1
               clrf       PIE1 ; Disable peripheral interrupts
               bcf        STATUS, RP0 ; Bank0
               clrf       PIR1 ; Clear peripheral interrupts Flags
               movlw      0x30 ; Internal Clock source with 1:8 prescaler
               movwf      T1CON ; Timer1 is stopped and T1 osc is disabled

               movlw      0x76
               movwf      TMR1H ;
               movlw      0x97
               movwf      TMR1L ;
               bsf        T1CON, TMR1ON ; Timer1 starts to increment

; 20 MHz / 4 = 5MHz
; prescale de 8 : 5MHZ /8 = 625kHz soit 625 000 incrémentations = 1 s
; nombre de remplissage du compteur 16 bits : 625000/65536 = 9,53 .... (>9)
; pour 9 : 9 * 65536 = 589824 incrémentations réalisées
; il manque 625000 - 589824 = 35176 incrémentations
; donc charger le compteur avec 65536 - 35176 = 30359 => 0x7697
; il faut donc partir de cette valeur puis boucler 9 fois (1+9 = 10 it)
Tempo_Ws_B1:
               movlw      0x0A
               movwf      Tempo_Ws_mem
Tempo_Ws_B2:
Tempo_Ws_OVFL_WAIT:
               btfss      PIR1, TMR1IF
               goto       Tempo_Ws_OVFL_WAIT
; Timer has overflowed
               bcf        PIR1, TMR1IF
; 9 *
               decfsz     Tempo_Ws_mem, f; s
               goto       Tempo_Ws_B2;  Tempo_Wms_Value != 0

               decfsz     Tempo_Ws_value, f; s
```



```
               goto  Tempo_Ws_B1;  Tempo_Wms_Value != 0

               bcf        T1CON, TMR1ON ; Timer1 stops to increment
          RETURN

; *****************************************************************
main:

      bsf   STATUS, RP0; Selection bank 1
      bcf   TRISD  ;      mise à 0 du bit 0 de TRISD
      bsf   TRISB, 0 ;      mise à 1 du bit 0 de TRISB
      bcf   STATUS, RP0 ; Selection bank 0

     bcf     PRTD,0
main_prg:
     ; init tempo
          clrf tempo;

     ; touche ?
          btfsc PORTB, 0
          goto main_prg

     ; allumage
           bsf    PORTD,0

main_attente:
     ; attendre 1s
          movl .1
          call Tempo_Ws

     ; touche ?
          btfss PORTB, 0
          goto main_eteindre

     ; incrementer tempo : tempo = tempo + 1
          incf tempo

     ; fin tempo ?
          movf  tempo
          sublw .10
          btfss STATUS, Z
          goto main_attente ; Z=0

main_Eteindre:
     ; eteindre LED
          bcf   PORTD, 0     ; LED eteinte

     ; attendre 1s
          movlw .1
          call Tempo_Ws

     ; retour au debut
          goto main_prg
     END                       ; directive 'end of program'
```

Annexes : Code assembleur TPuC_2.asm (séance 2)

AVEC LES ERREURS nécessaires au TP…;)

```
; CODE INCOMPLET !!!! special TP2...

#include <p16F877A.inc>  ; processor specific variable definitions
      __CONFIG _HS_OSC & _WDT_OFF & _BODEN_OFF & _LVP_OFF & _PWRTE_OFF &
_CPD_OFF & _WRT_OFF & _CP_OFF

#define NB_PRG  4

;***** VARIABLE DEFINITIONS
      extern     copy_init_data

      udata
Tempo_Wms_value res  1
tempo           res  1
pas             res  1
prg             res  1
vitesse         res  1
nbpas           res  1
touche          res 1

      idata
prg0_pas        db   b'10000000', b'11000000', b'01100000', b'00110000',
b'00011000', b'00001100', b'00000110',b'00000011'
                db   b'00000001', b'00000011', b'00000110', b'00001100',
b'00011000', b'00110000', b'01100000', b'11000000'
prg1_pas        db   b'10000001', b'01000010', b'00100100', b'00011000',
b'00011000', b'00100100', b'01000010', b'10000001'
prg2_pas        db   b'10101010',b'01010101'
prg3_pas        db   b'10001000',b'01000100', b'00100010', b'00010001'

table_prg       db   prg0_pas, prg1_pas, prg2_pas, prg3_pas
table_nbpas     db   .16, .8, .2, .4

;*************************************************************
RST     CODE     0x0
      goto    main              ; go to beginning of program

PGM     CODE
Tempo_Wms:
      movwf      Tempo_Wms_value ; dans W : le temps en ms à attendre
      ; INIT compteur
            bsf        STATUS, RP0 ; Bank1
            clrf       PIE1 ; Disable peripheral interrupts
            bcf        STATUS, RP0 ; Bank0
            clrf       PIR1 ; Clear peripheral interrupts Flags
;-->
            movlw      0x?? ;
            movwf      T1CON ;

Tempo_Wms_B1:
      movf       Tempo_Wms_value
```



```
      btfsc      STATUS, Z
      return
      decf       Tempo_Wms_value, f

;-->
            movlw      0x??
            movwf      TMR1H ;
;-->
            movlw      0x??
            movwf      TMR1L ;
            bsf        T1CON, TMR1ON ; Timer1 starts to increment

Tempo_Wms_OVFL_WAIT:
                  btfss   PIR1, TMR1IF
                  goto    Tempo_Wms_OVFL_WAIT

; Timer has overflowed
            bcf   PIR1, TMR1IF
            bcf   T1CON, TMR1ON ; Timer1 stops to increment
      goto  Tempo_Wms_B1


; ********************************************************************
; Lecture de PORTB
; lecture et modification de "touche", "prg", "vitesse"
Analyse_BP:
      ;lecture des touches si: touche = 0?
            movf       touche, f
            btfsc      STATUS, Z
            goto       Analyse_BP0
            ; sinon plus de touches appuyées  sur le portB ?
            movf       PORTB, W
            sublw      0x0F
            btfss      STATUS, Z
            return           ; non : pas de lecture

            ; gestion du relachement des touches
            clrf    touche;

      ; Lecture des touches
Analyse_BP0:
            btfsc      PORTB, 0        ; BP moins vite!
            goto       Analyse_BP1     ; pas appuyé : on va voir un autre bouton

            incf   touche, f; gestion touche
            ; on incremente (ralentir) si vitesse < 255
            movlw      .255
            subwf      vitesse, W
            btfss      STATUS, C
            incf       vitesse, f ;

Analyse_BP1:
            btfsc      PORTB, 1   ;    BP plus vite
            goto       Analyse_BP2; pas appuyé : on va voir le bouton suivant

            incf touche, f; gestion touche
```

```
          ; on decremente (accelerer) si vitesse > 0
          movf      vitesse, f; pour positionner le flag Z
          btfss     STATUS, Z
          decf      vitesse, f

Analyse_BP2:
          btfsc     PORTB, 2        ; BP : prg --
          goto      Analyse_BP3

          incf   touche, f; gestion touche
          ; prg > 0
          movf      prg, f; pour positionner le flag Z
          btfss     STATUS, Z
          decf      prg, f

Analyse_BP3:
          btfsc     PORTB, 3        ; BP : prg ++
          return    ; fin de la fonction

          incf   touche, f; gestion touche
          ; on increment jusqu'à 2
          movlw     NB_PRG-1
          subwf     prg, W
          btfss     STATUS, C
          incf      prg, f
     return



; *****************************************************************
main:
     ; Initialisation des valeurs en RAM
          call       copy_init_data

     ; Init E/S
          clrf      PORTB
          clrf      PORTD

          bsf       STATUS, RP0; bank 1

          movlw     0x0F
          movwf     TRISB ; 4 premiers bits de PORTB en entrée

          clrf      TRISD ; PORTD en sortie

          bcf       STATUS, RP0; bank 0

     ; Init des variables
          clrf      touche

          movlw     .150
          movwf     vitesse; 150 : vitesse par initiale

          clrf      pas; premier pas

          clrf      prg  ; prg0 = prg par defaut
```



```
          movf      table_nbpas, W
          movwf     nbpas;  le nombre de pas du programme

          clrf      tempo;     ; on part de 0

main_prg:
     ; attente de 1 ms
          movlw     .1
          call      Tempo_Wms
          nop  ; pour debug

     ; fonction analyse des touches
          call   Analyse_BP

     ; mise à jour nbpas : table_nbpas[prg] -> nbpas
;         movlw     table_nbpas
;
;
;
;
;         movwf     nbpas

     ; pas = nbpas ?
          movf      pas, W
          subwf     nbpas, W
          btfsc     STATUS, Z
          clrf      pas
     ; pas > nbpas (nbpas - pas < 0) ?
          btfss     STATUS, C
          clrf      pas

     ; lire table[prg][pas] -> W
;         movlw     table_prg
;
;
;
;
;
;
;

     ; W -> PORTD
          movwf     PORTD

     ; tempo > vitesse ? si oui changer de pas
          movf      vitesse, W
          subwf      tempo, W
          btfsc     STATUS, C
          goto      pas_suivant

               ; tempo ++
               incf      tempo, f
               ; retour au debut du prg
               goto      main_prg

pas_suivant:
          ; pas suivant
```

```
        incf    pas, f

        ; remise à 0 de tempo
        clrf    tempo

        ;retour au debut du prg
        goto    main_prg

    END
```