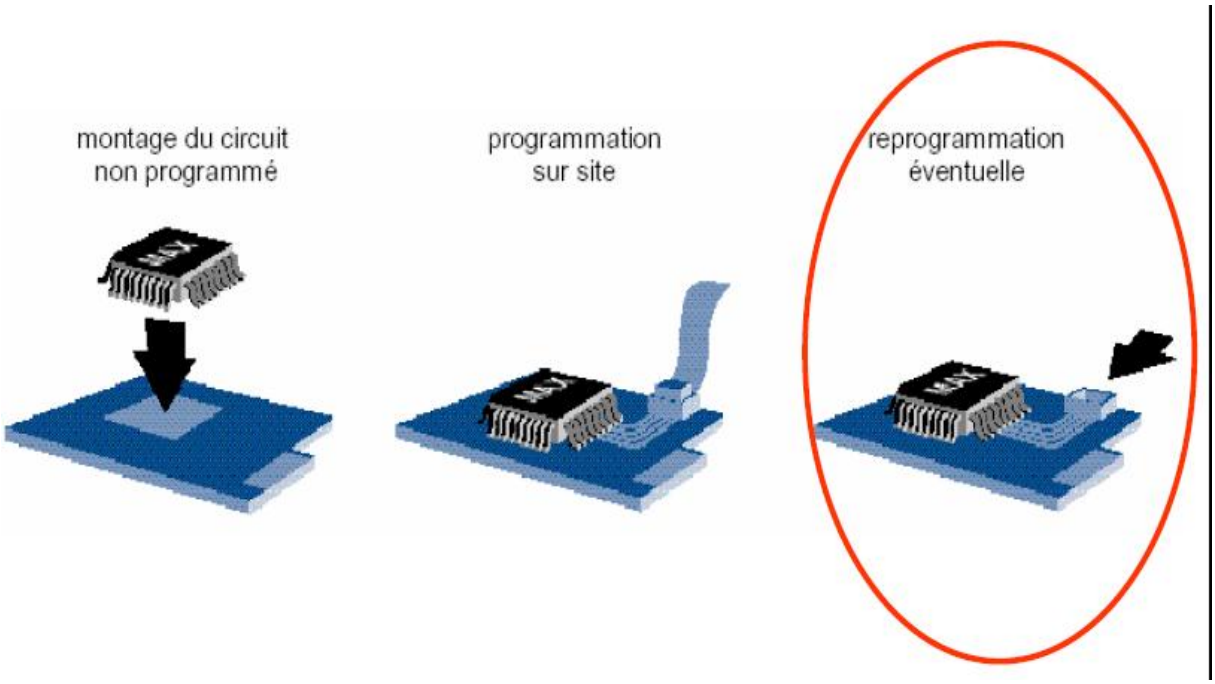
COURS DE PROGRAMMATION DES MICROCONTROLEURS PIC EN C



Mr. Mazoughou GOEPOGUI

Tel: 655 34 42 38 / 669 35 43 10

E-mail: massaleidamagoe@gmail.com

I. NOTION D'ALGORITHME.

I.1. Définitions.

I.1.1. Définition d'un algorithme.

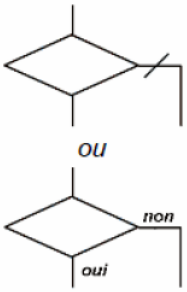
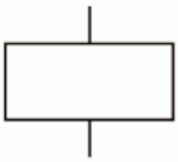
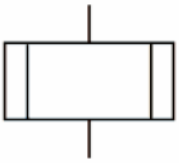
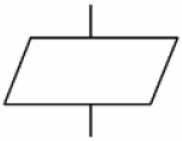
Un algorithme est l'ensemble des règles opératoires ordonnant à un processeur d'exécuter dans un ordre déterminé un nombre d'opérations élémentaires. Il impose une programmation de type structurée.

I.1.2. Définition algorigramme.

Algorigramme est une représentation graphique de l'algorithme utilisant des symboles normalisés. En réalité c'est un diagramme qui permet de représenter et d'étudier le fonctionnement des automatismes de types séquentiels comme les chronogrammes ou le GRAFCET mais davantage réservé à la programmation des systèmes microinformatiques ainsi qu'à la maintenance.

Le diagramme est une suite de directives composées d'actions et de décisions qui doivent être exécutés selon un enchaînement strict pour réaliser une tâche (ou séquence).

Les principaux symboles utilisés sont données ci-dessous.

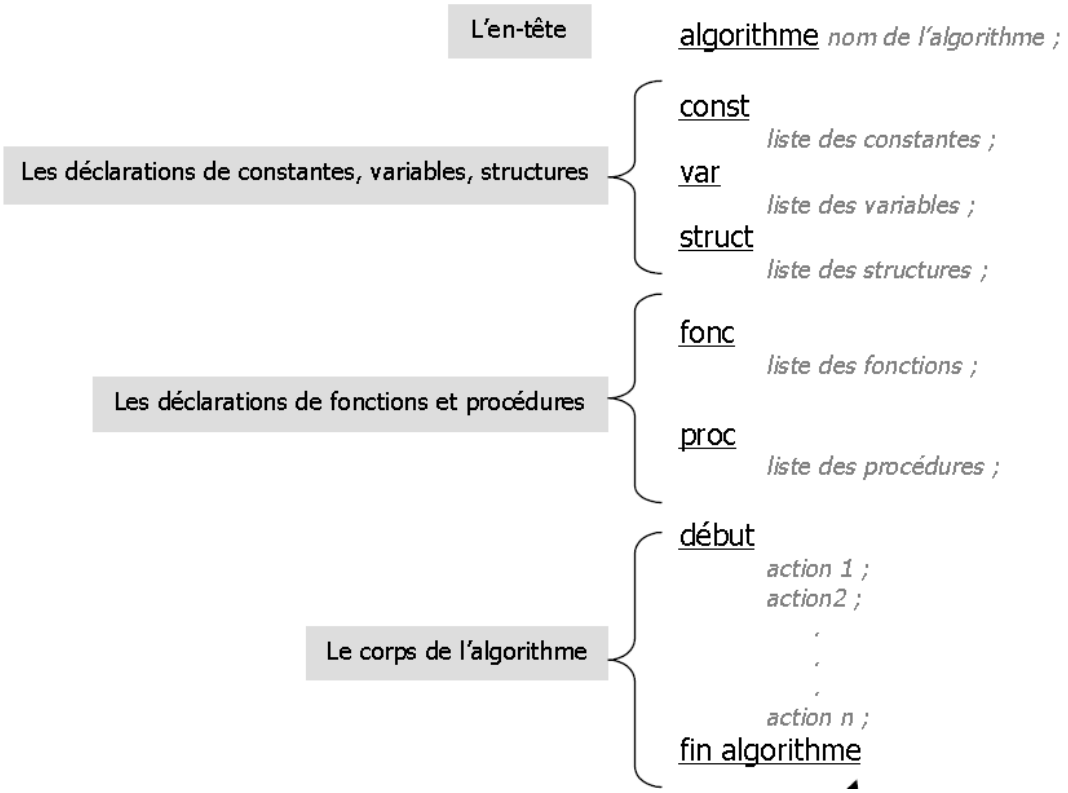
SYMBOLE	DÉSIGNATION	SYMBOLE	DÉSIGNATION
	**début** ou **fin** d'un algorithme		**Test ou Branchement conditionnel** décision d'un choix parmi d'autres en fonction des conditions
	symbole général de **« traitement »** opération sur des données, instructions, … ou opération pour laquelle il n'existe aucun symbole normalisé		**sous-programme** appel d'un sous-programme
	**entrée / sortie**		**Liaison** Les différents symboles sont reliés entre eux par des lignes de liaison. Le cheminement va de haut en bas et de gauche à droite. Un cheminement différent est indiqué à l'aide d'une flèche
	**commentaire**		

Remarques.

- Les symboles de début et de fin de programme ne sont pas toujours représentés.

I.2. Structure d'un algorithme.

La structure générale d'un algorithme est donnée ci-dessous.



1. **L'entête.** Il permet tout simplement d'identifier l'algorithme.
2. **Les déclarations.** C'est une liste exhaustive d'objets, de grandeurs utilisés et manipulés dans le corps de l'algorithme. Cette liste est placée en début d'algorithme.
3. **Le corps.** C'est dans cette de l'algorithme que placées les tâches (instructions) à exécuter.
4. **Les commentaires.** Ils permettent une interprétation aisée de l'algorithme. L'utilisation de commentaires est vivement conseillée.

I.3. Les structures algorithmiques fondamentales.

Les opérations élémentaires relatives à la résolution d'un problème peuvent, en fonction de leur enchainement, être organisées suivant quatre familles de structures algorithmiques fondamentales.

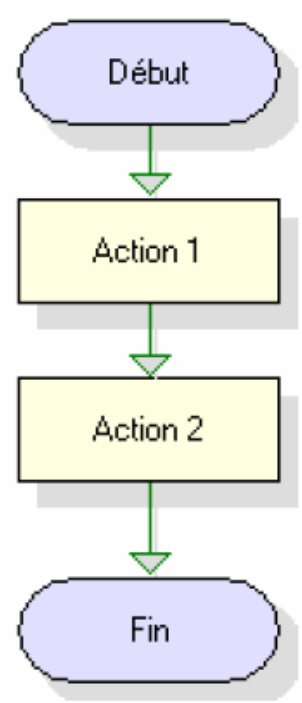
1. Structures linéaires.
2. Structures alternatives.
3. Structures de choix.
4. Structure itératives (ou répétitives).

I.3.1. Structure linéaire.

La structure linéaire se caractérise par une suite d'actions à exécuter successivement dans l'ordre énoncé.

Notation :

Début
Action 1
Action 2
Fin



Exemple en langage C.
{ Action 1 ; }
{ Action 2 ; }

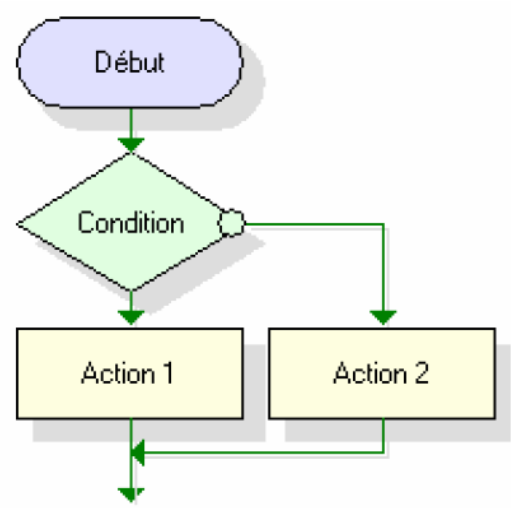
I.3.2. Structure alternative.

Cette structure offre le choix entre deux séquences s'excluant mutuellement. On peut rencontrer deux types de structures alternatives : la structure alternative complète et la structure alternative simple.

a) Structure alternative complète.

Notation :

Début
Si Condition
Alors Action 1
Sinon Action 2

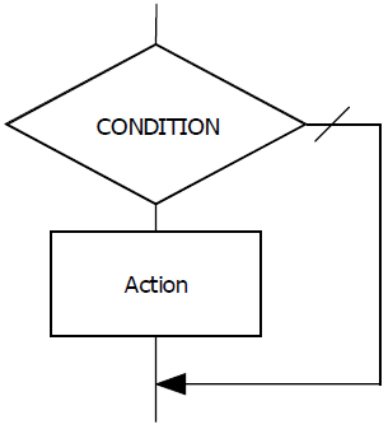


Exemple en langage C.
If (Condition)
{ Action 1 ; }
Else
{ Action 2 ; }

b) Structure alternative réduite.

Notation :

Début
Si Condition
Alors Action



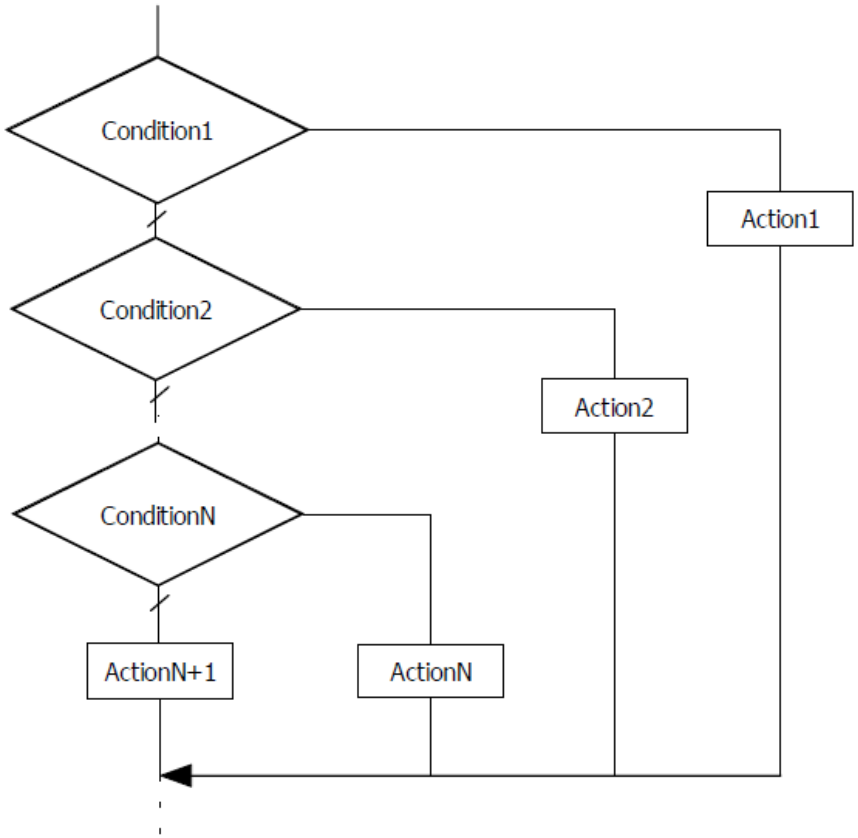
Exemple en langage C.
If (Condition)
{ Action ; }

I.3.3. Structure de choix.

La structure de choix permet, en fonction de plusieurs conditions de type booléen, d'effectuer des actions différentes suivant les valeurs que peut prendre une même variable.

Notation :

suivant valeur **faire**
valeur 1 : action 1
valeur 2 : action 2
.
.
.
valeur N : action N
sinon action N+1



Exemple en langage C.
switch (valeur)
{
case valeur 1: action 1;
break;
case valeur 2: action 2;
break;
.
.
.
case valeur N: action N;
break;
default : action N+1
}

I.3.4. Structure itérative (ou répétitive).

Cette structure répète l'exécution d'une opération ou d'un traitement. Deux cas peuvent arriver.

I.3.4.1. Le nombre de répétition n'est pas connu ou est variable.

Là également deux cas peuvent arriver.

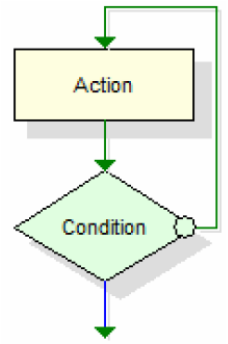
a) Structure « *répéter jusqu'à* ».

Dans cette structure le traitement est exécuté une première fois puis sa répétition se poursuit jusqu'à ce que la condition soit vérifiée.

Notation :

répéter
action ;
jusqu'à condition vraie ;

Remarque. En faisant de sorte que la condition soit toujours vraie, l'action se répétera de façon infinie : c'est la boucle infinie.



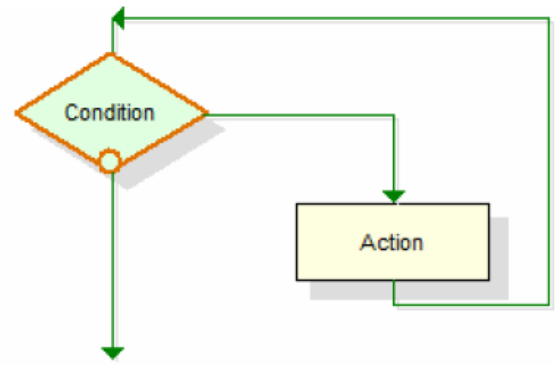
Exemple en langage C.
action ;
while condition ;

b) Structure *« tant que … faire »*.

Dans cette structure, on commence par tester la condition ; si elle vraie, le traitement est exécuté.

Notation :

tant que condition **faire**
action ;
fin tant que ;



Exemple en langage C.
while (condition)
action ;

I.3.4.2. Le nombre de répétition est connu.

Dans cette structure, la sortie de la boucle d'itération s'effectue lorsque le nombre de répétition souhaité est atteint.

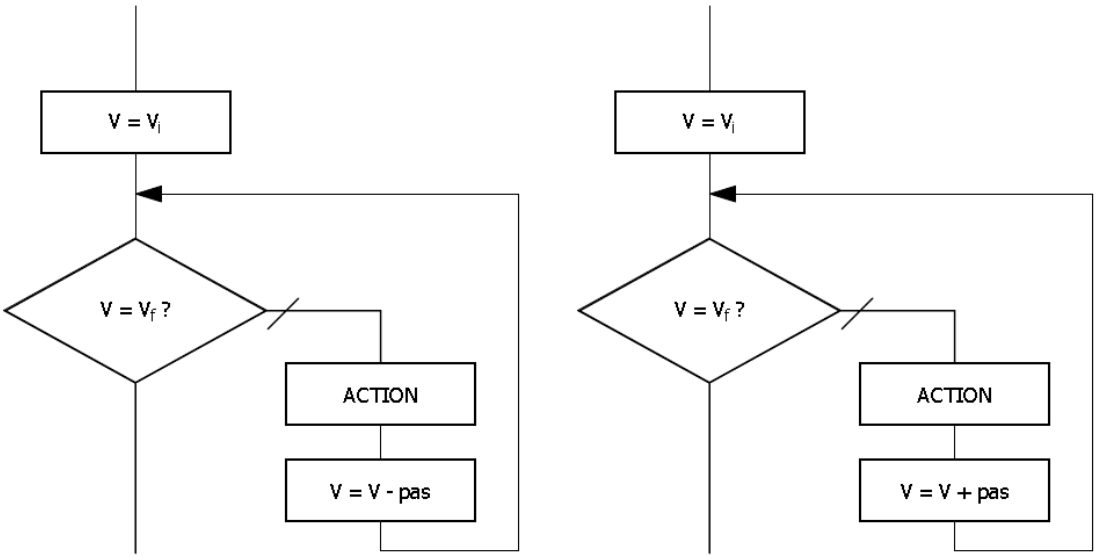
On utilise donc une variable (ou indice) de contrôle d'itération caractérisée par :

- Sa valeur initiale ;
- Sa valeur finale ;
- Son pas de variation.

Si la valeur finale de l'indice est inférieure à sa valeur initiale, le pas de variation est négatif, la structure est dite « ***pour décroissante*** » ; dans le cas contraire, le pas est positif et la structures est dite « ***pour croissante*** »

Notation :

pour variable **de** début **à** fin **pas** n **faire**
action ;
fin pour ;



V : variable ;
V_i : valeur initiale de V ;
V_f : valeur finale de V ;

Exemple en langage C.
for ([expression_1] ; [expression_2] ; [expression_3])
action ;

Les crochets signifient que leur contenu est facultatif. Lorsque *expression_2* est absente, elle est considérée comme vraie. D'une manière générale, en C, la structure **for** peut être remplacée par **while** comme indiqué ci-dessous.

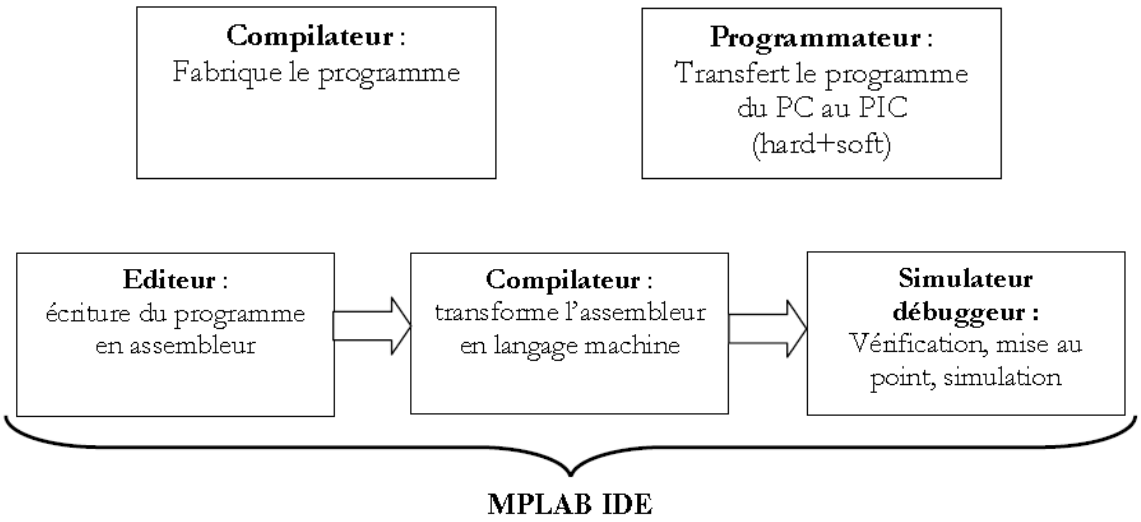
expression_1 ;
***while** (expression_2)*
{instruction ;
expression_3 ;}

II. INSTALLATION DES PROGRAMMES ET PREMIERS SIMULATIONS.

Les microcontrôleurs PIC sont des microcontrôleurs fabriqués par la société Microchip qui fournit par ailleurs gratuitement la plate-forme logiciel de développement MPLAB IDE.

II.1. Les outils pour réaliser une application.

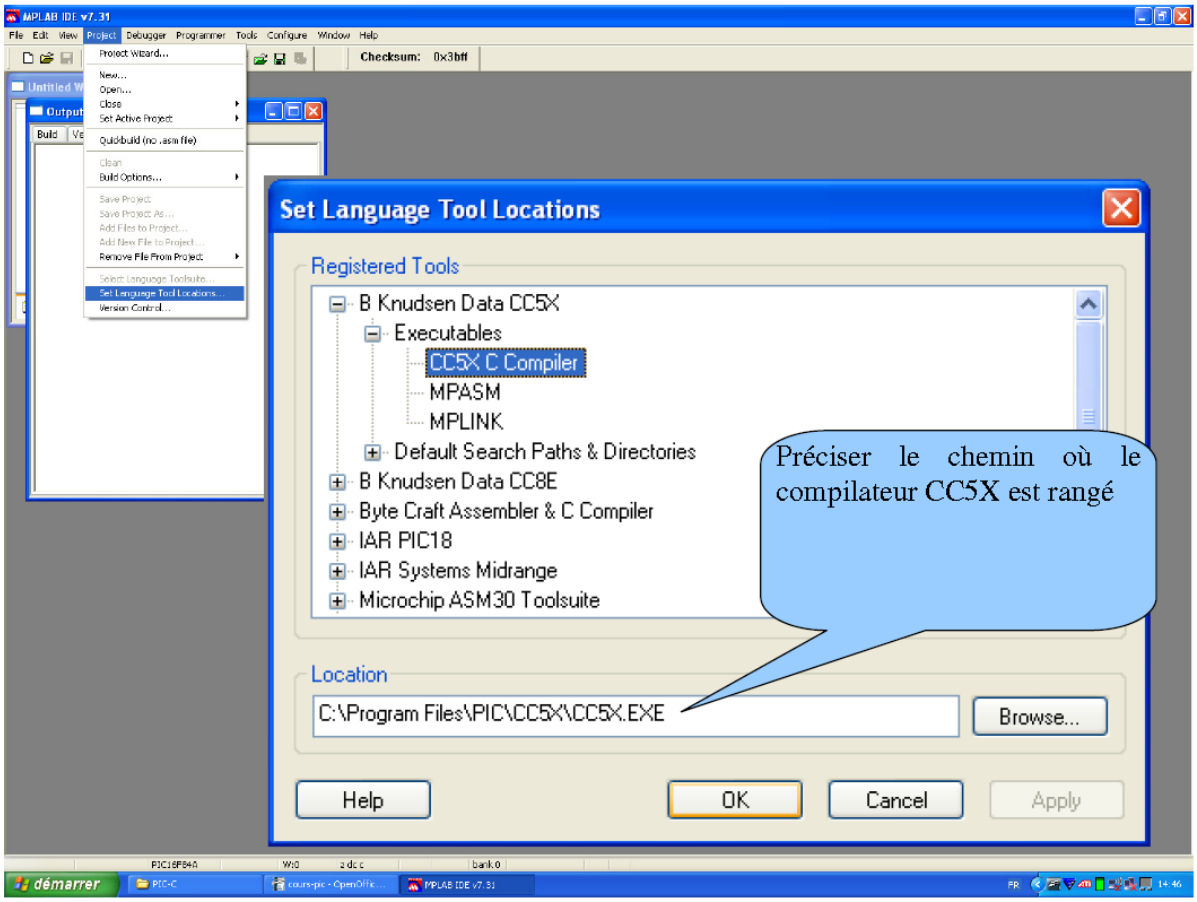
Pour développer une application fonctionnant à l'aide d'un microcontrôleur, il faut disposer d'un ***éditeur de programme***, d'un ***compilateur*** et d'un ***programmateur***.



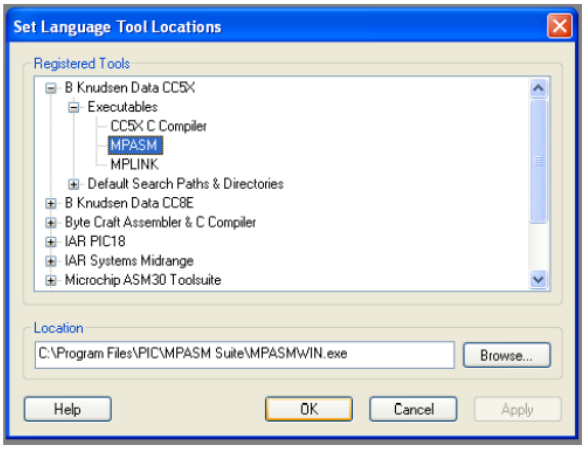
- L'éditeur de programme est un logiciel permettant d'écrire le programme dans un langage de programmation. Nous utiliserons le logiciel ***MPLAB IDE***. Le fabricant Microchip fournit gratuitement ce logiciel téléchargeable sur le site www.microchip.com.
- Le compilateur est un logiciel traduisant un programme écrit dans un langage donné (C, basic, assembleur) en langage machine. Ce logiciel peut aussi comporter un «debugger» permettant la mise au point du programme, et un simulateur permettant de vérifier son fonctionnement. On utilisera le compilateur ***CC5X*** dans sa version gratuite téléchargeable sur www.bknd.com. Cette version gratuite permet d'écrire environ 1ko de programme. On peut alors intégrer CC5X dans l'environnement MPLAB. Ainsi CC5X devient un outil de MPLAB dans lequel l'écriture, la simulation et le debugging du programme en C devient alors possible.
- Le programmateur permet de transférer le programme compilé (langage machine) dans la mémoire du microcontrôleur. Il est constitué d'un circuit branché sur le port COM ou USB du PC, sur lequel on implante le PIC, et d'un logiciel permettant d'assurer le transfert. Il existe différents logiciels, nous utiliserons ***Icprog*** ou ***WinPic800***. De nos jours, il existe des PIC programmable sur site. Les mises à jour du logiciel Icprog sont téléchargeables sur www.seeit.fr.

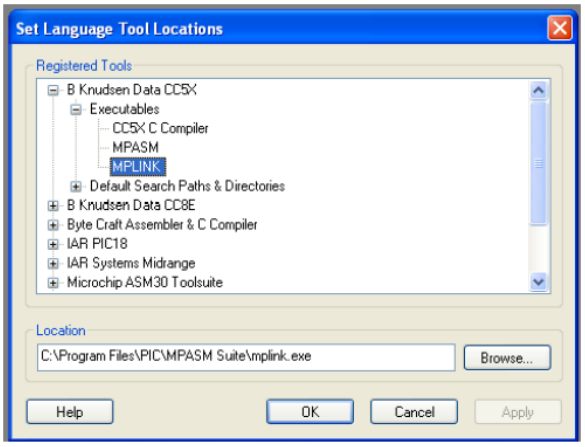
II.2. Déclaration du compilateur CC5X dans MPLAB.

La **déclaration** du compilateur s'effectue **une seule fois** :
Lancez MPLAB IDE ; ouvrez la fenêtre Projet > Set Language Tool Locations…



Déclarer également le chemin de MPASM et MPLINK :

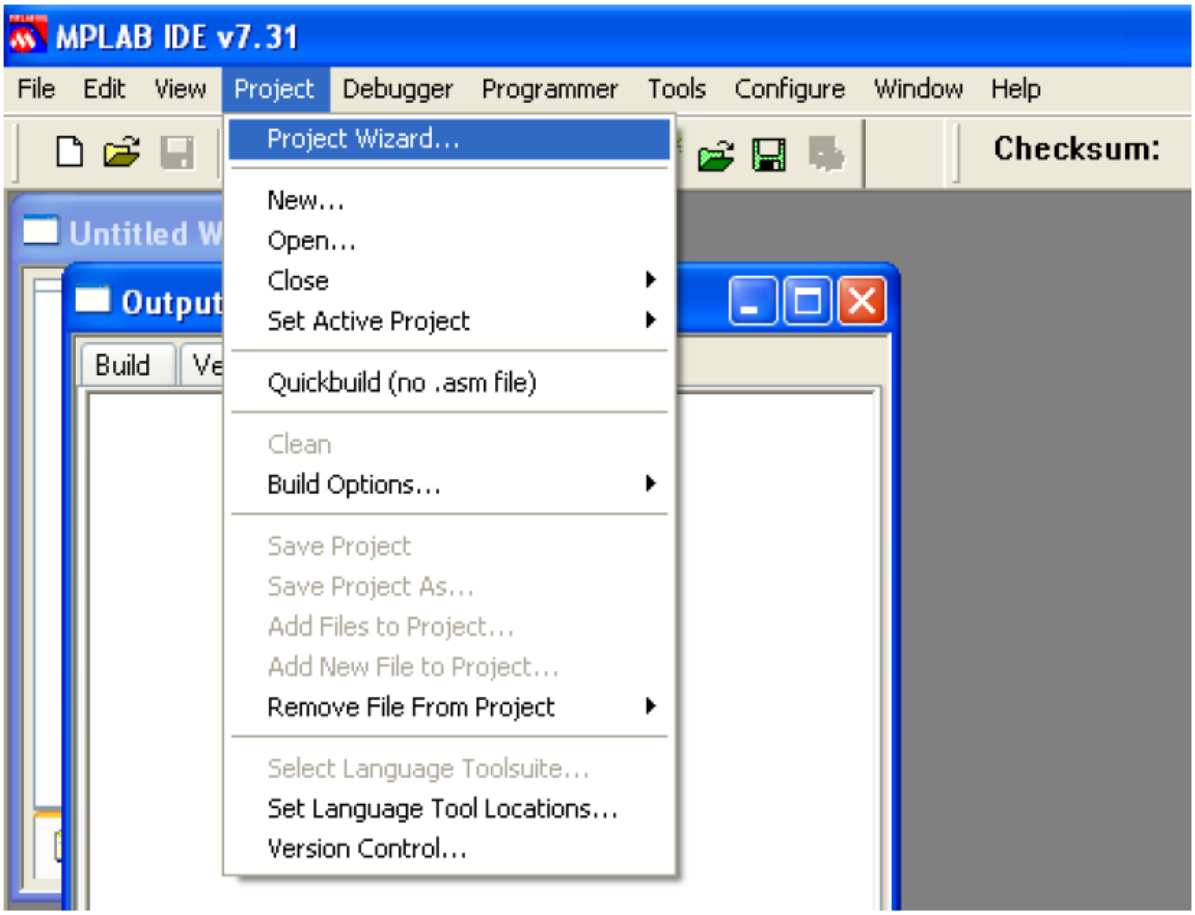


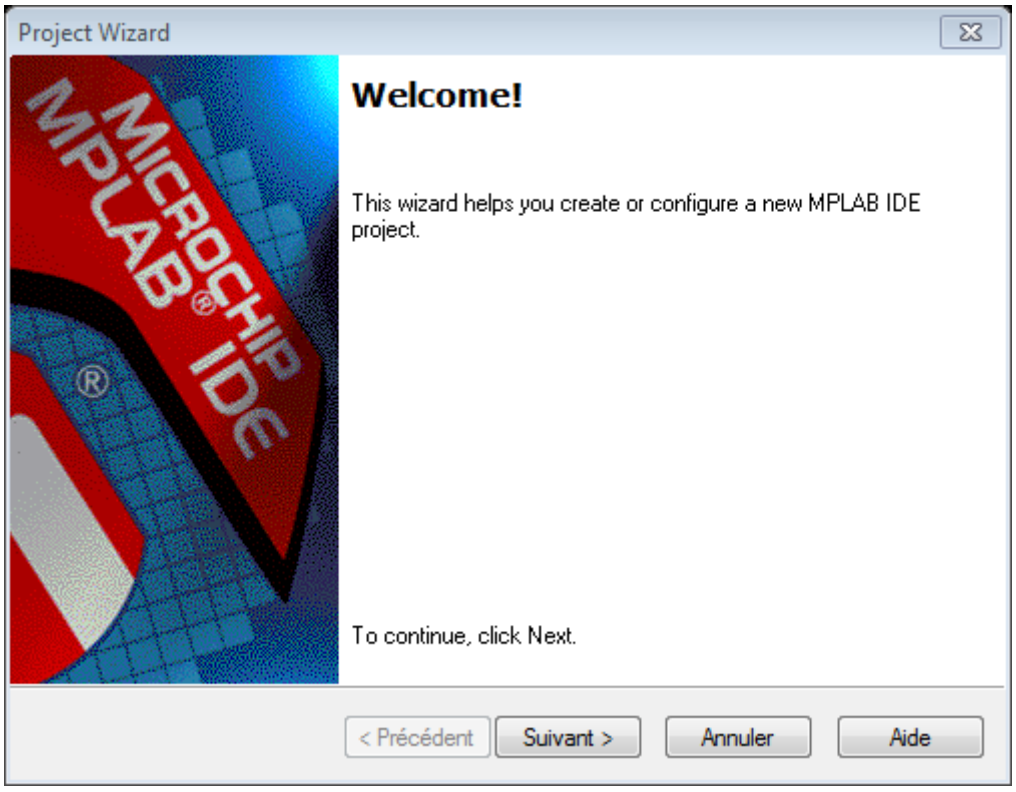


II.3. Création d'un nouveau projet.

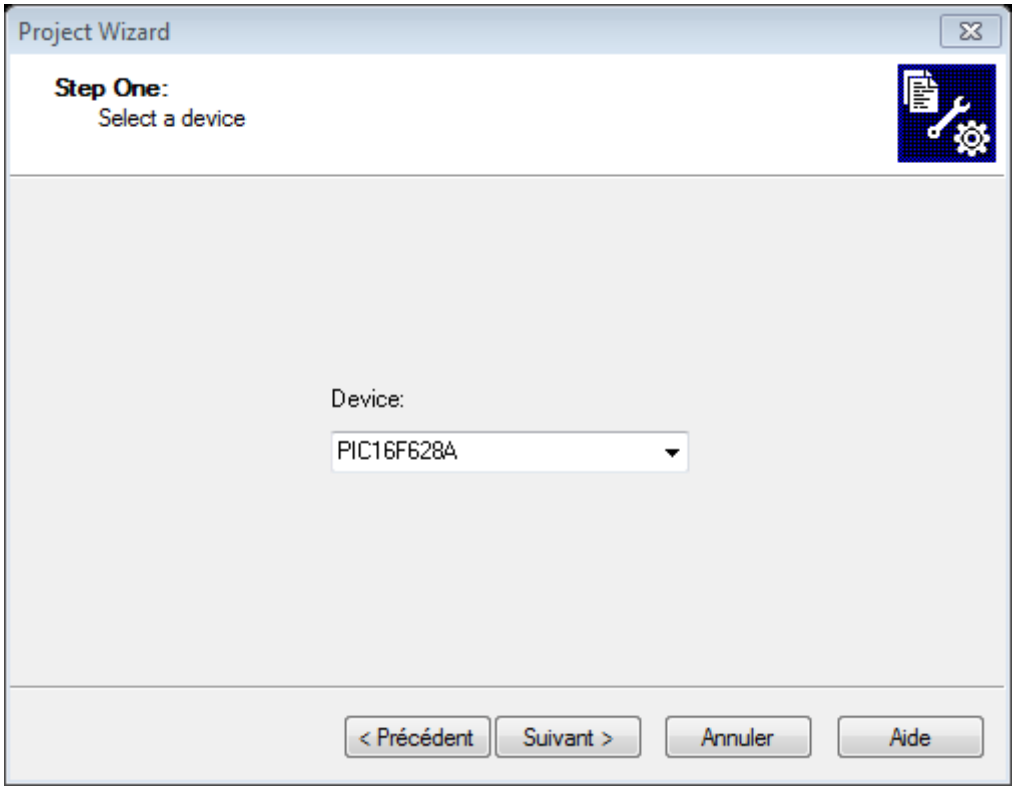
II.3.1. Définition du projet avec l'assistant.

Dans le menu Project, sélectionner Project Wizard. Cela lance un assistant permettant de définir certaines options du projet.

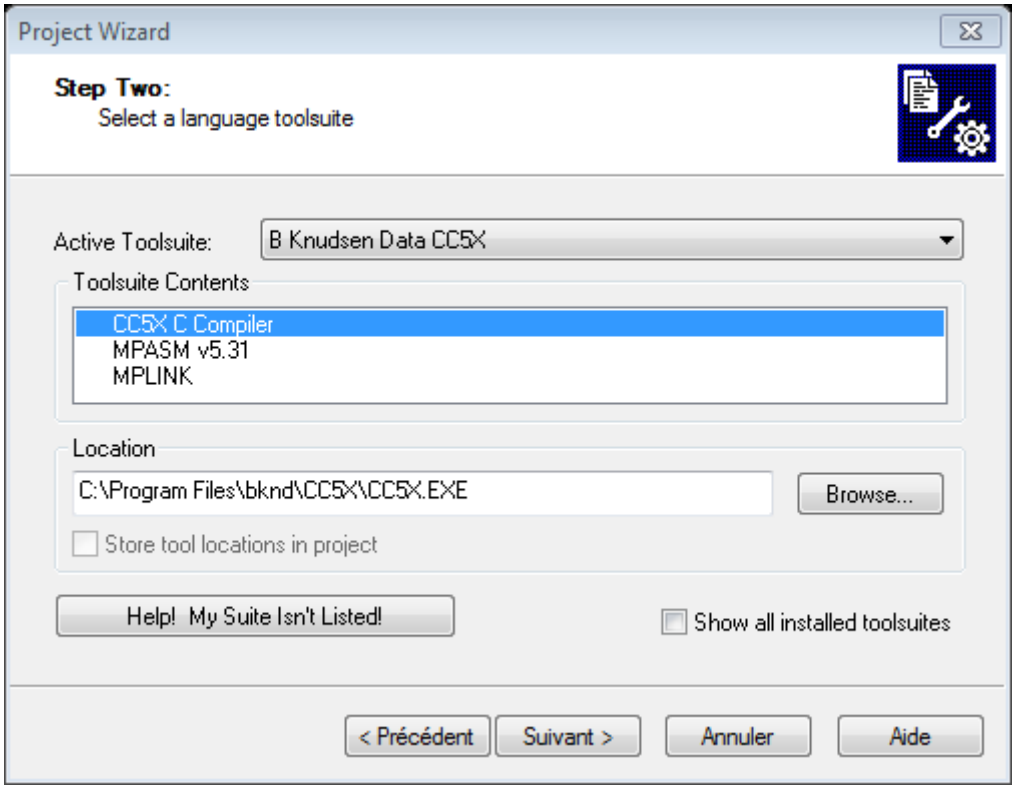




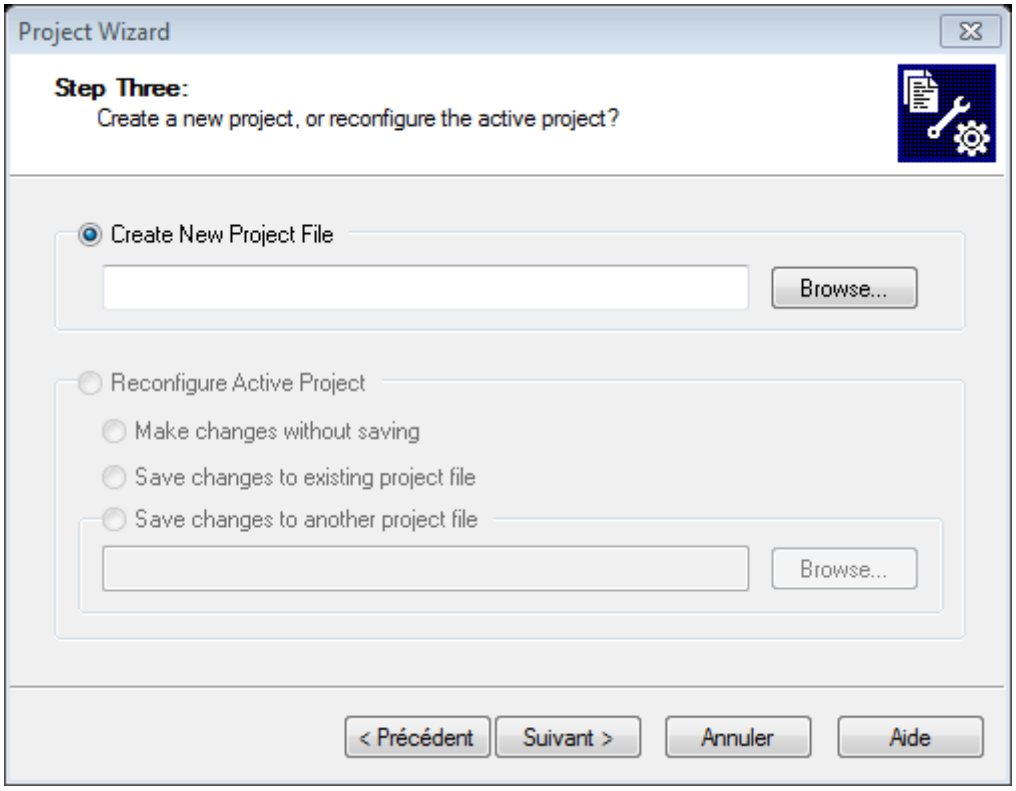
Sélectionner un microcontrôleur.



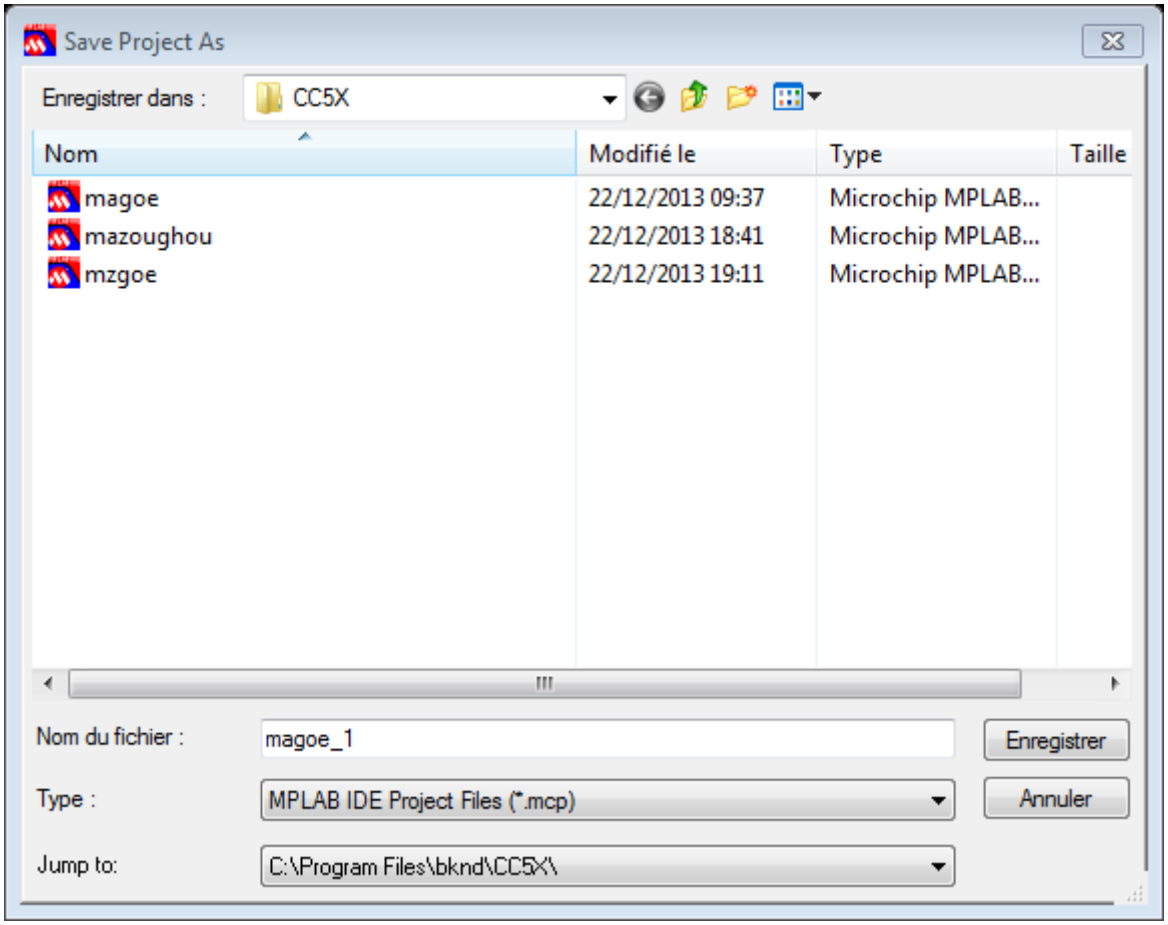
Si la déclaration du compilateur n'a pas été faite, il convient de le faire.

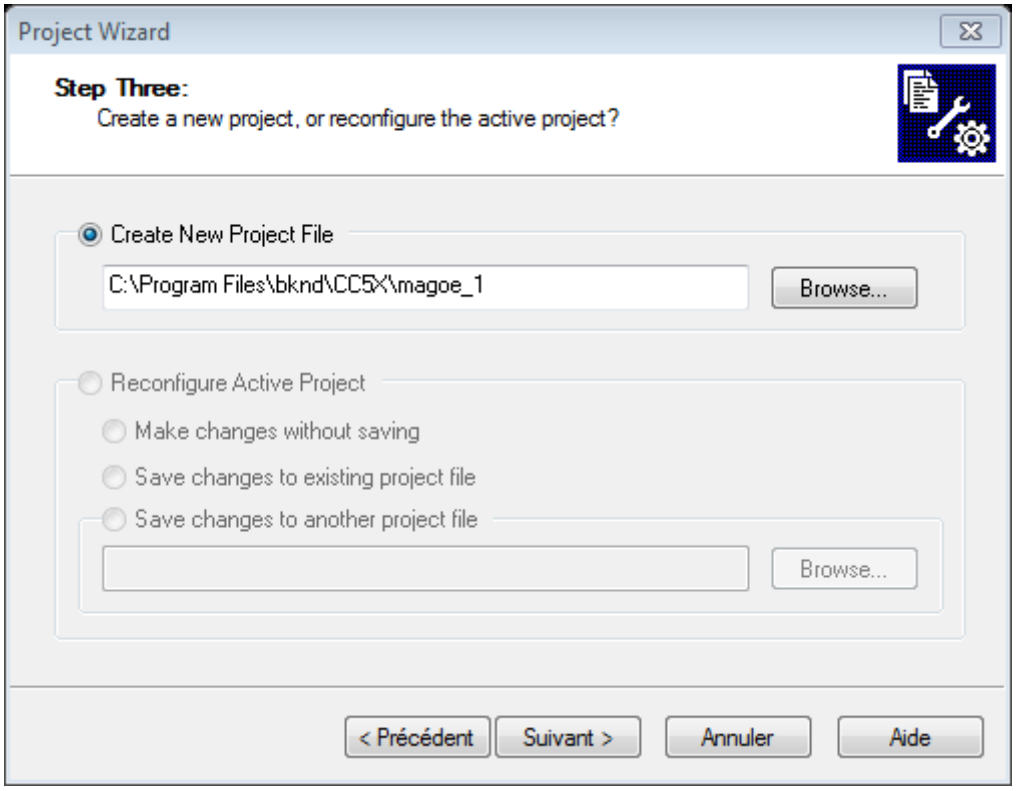


Définir un chemin (browse…) pour la sauvegarde du projet.

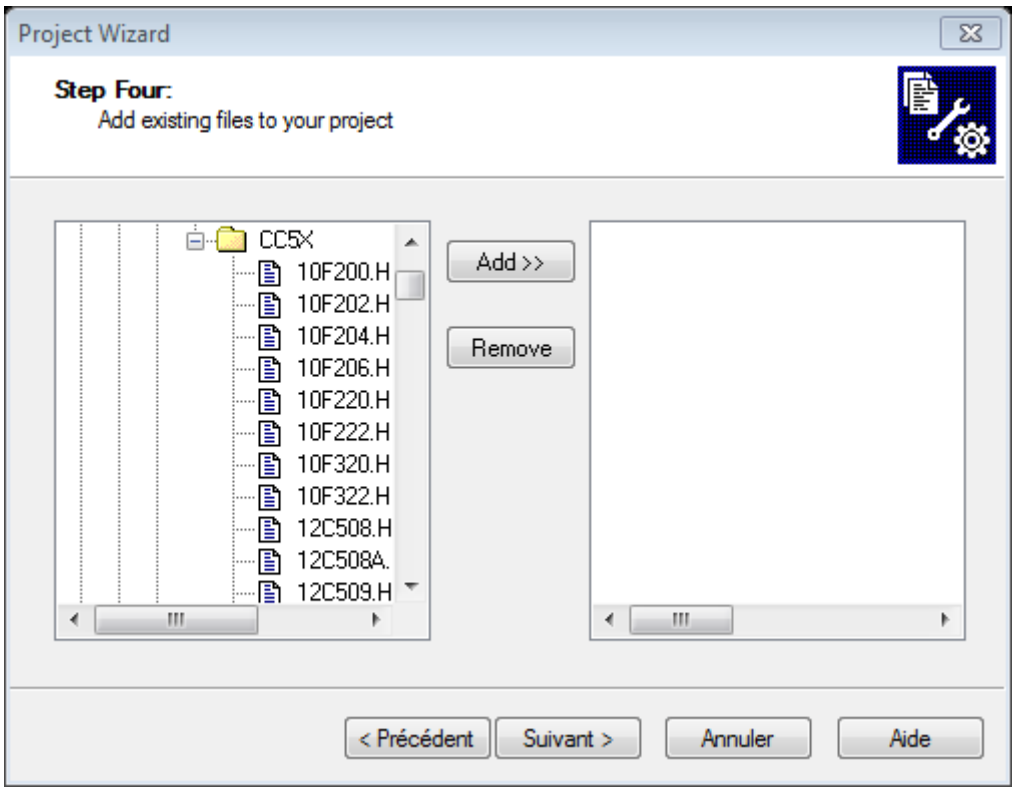


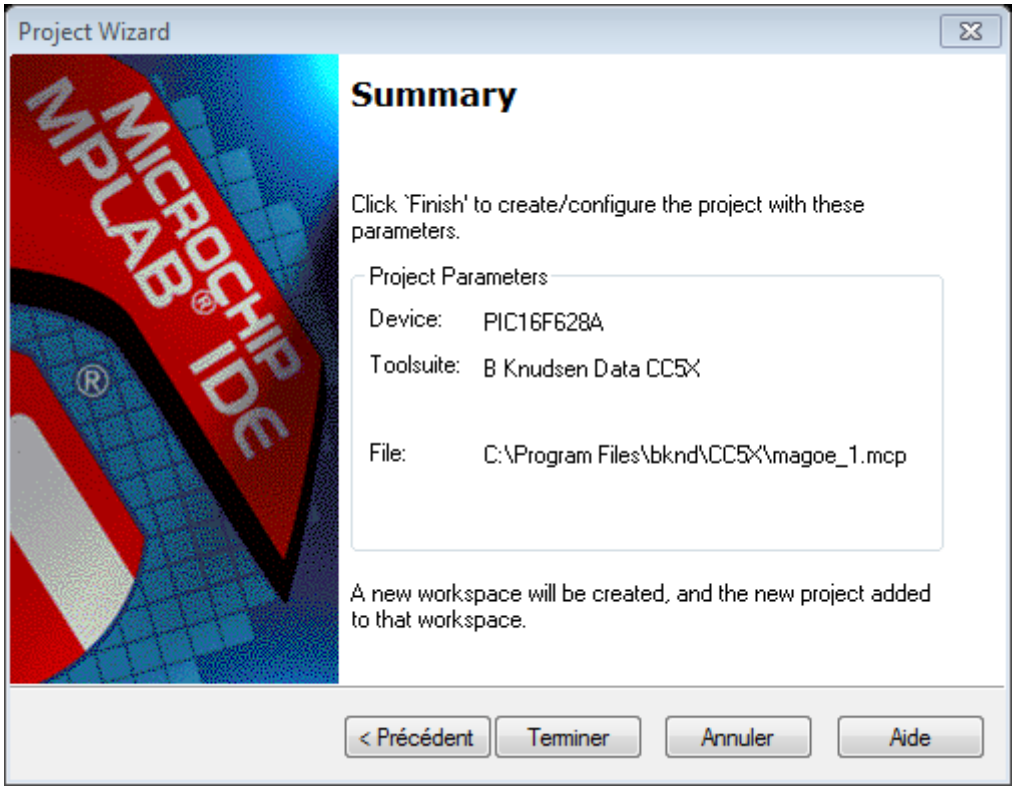
Donner un nom au projet puis enregistrer.





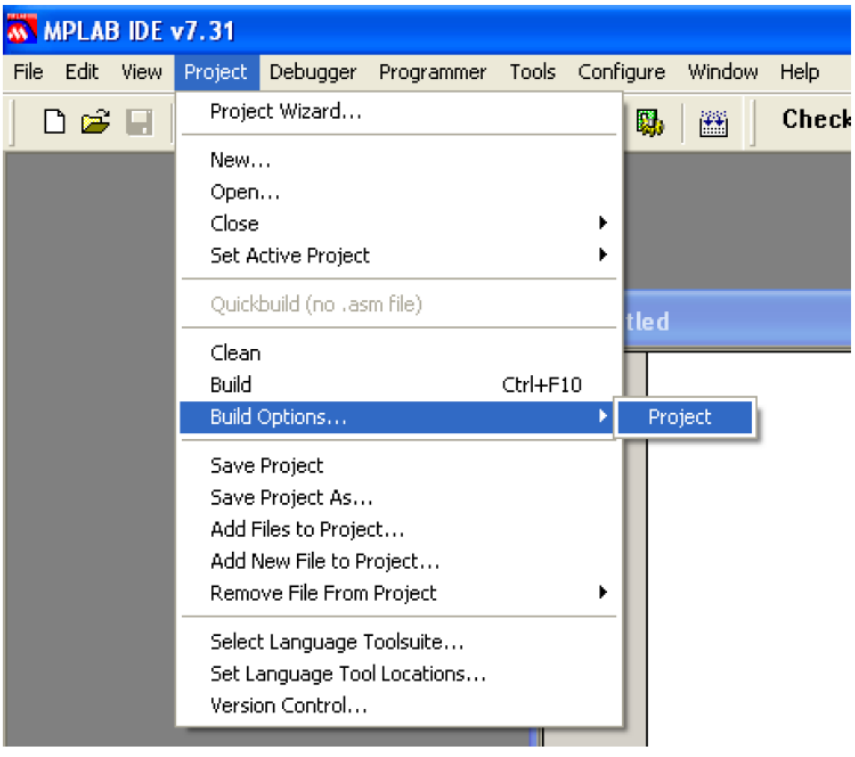
La 4ème étape permet d'ajouter éventuellement un fichier déjà créé, par exemple un programme source en c. Si on désire écrire le programme ultérieurement, il faut cliquer sur suivant puis sur terminer.



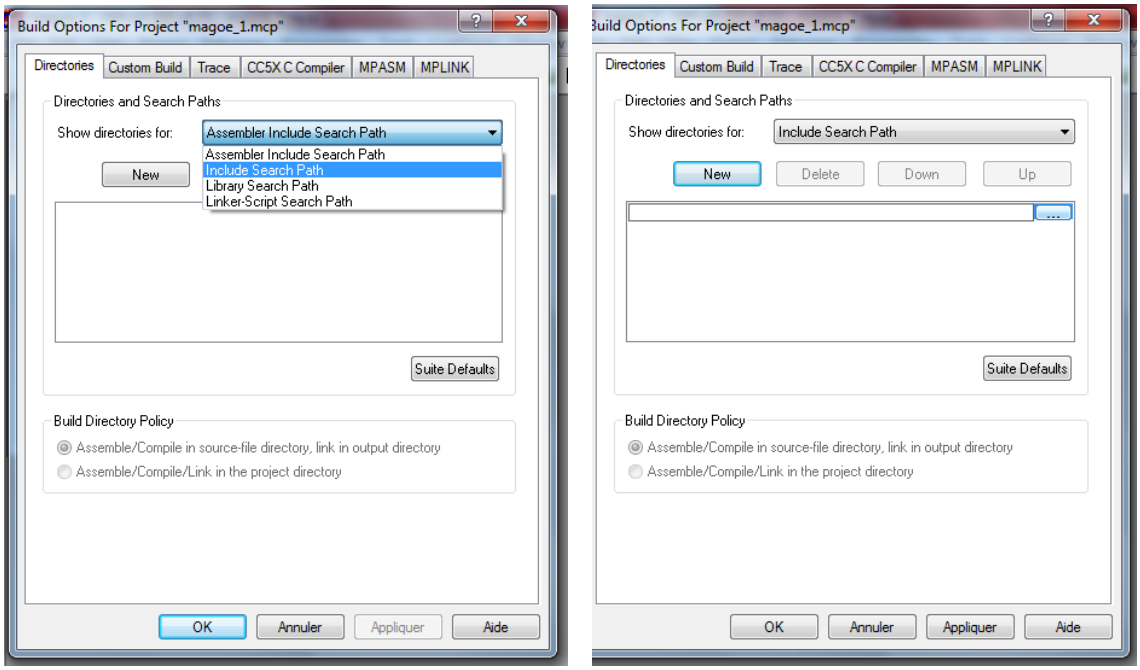


II.3.2. Les options.

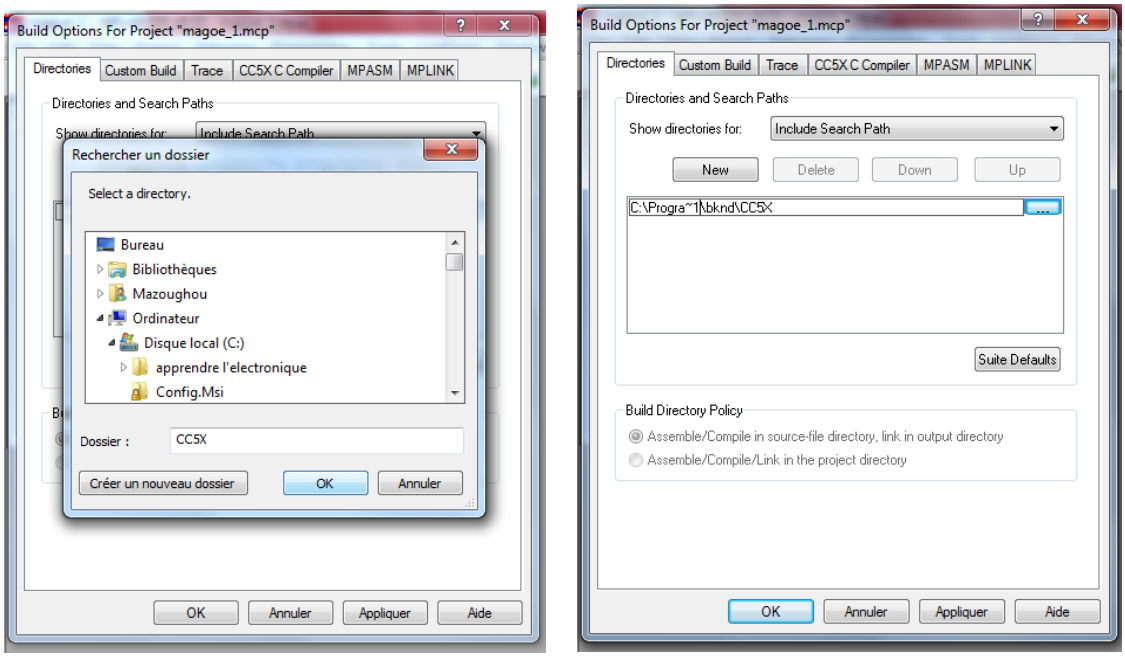
Pour fonctionner correctement, CC5X a besoin d'accéder aux données spécifiques du PIC sélectionné. Ces données sont définies dans des fichiers de définition (header.h) situés dans le répertoire où CC5X a été installé. Il convient de définir ce chemin dans une fenêtre ouverte par le menu Project/Build Options.



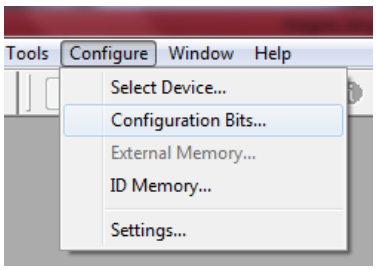
Dans la liste déroulante ***Show directories for***, sélectionner ***Include Search Parth***, cliquer sur ***New*** puis le bouton qui apparait juste à droite afin d'indiquer le chemin d'accès aux fichiers header.h.

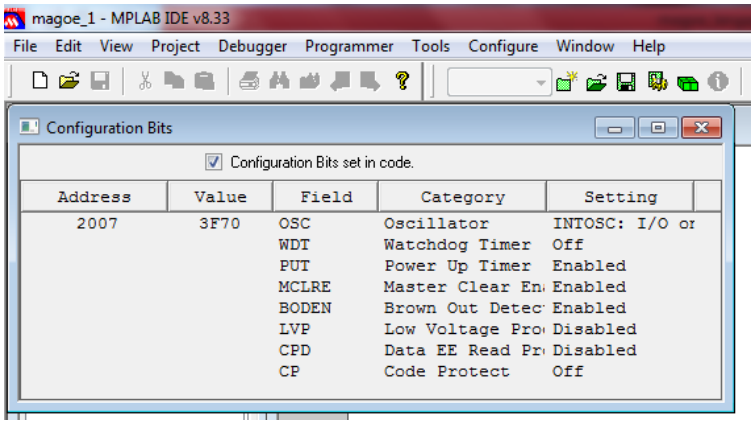


Cliquer sur ***OK***. Si nécessaire, remplacer ***Program File*** par ***Progra~1*** et cliquer sur ***Appliquer*** puis sur ***OK***.



Il faut ensuite définir les options propres au microcontrôleur : Menu ***Configure***, ***Configuration Bits***.





WDT.

En validant cette option, le "Watch Dog Timer" sera activé. C'est à dire qu'un oscillateur interne indépendant de l'oscillateur externe sera fonctionnel même si le microcontrôleur est en position sommeil. Cet oscillateur veillera sur le déroulement normal des instructions.

PUT.

En validant cette option, le "Power-Up Timer" sera activé. Le microcontrôleur effectuera à sa mise sous tension un Reset général d'une durée relativement courte, le temps que la tension d'alimentation se stabilise.

MCLRE.

En validant cette option, le "Memory Clear" sera activé. Il sera donc possible de faire une remise à zéro externe par la broche " RA5\MCLR\Vpp " du microcontrôleur. Cette borne sera reliée au +5V du pic à travers une résistance (2,2 kΩ par exemple).

CP.

En validant cette option, le "Code Protect" sera activé. Le programme intégré dans la mémoire du composant ne sera pas lisible si l'on fait une relecture de celui-ci. Cependant le composant reste effaçable pour être reprogrammé si celui -ci contient une mémoire Flash.

Attention si vous cochez cette case, le composant ne pourra pas être vérifié après programmation et un message d'erreur interviendra systématiquement lors de la vérification du composant après programmation. On évitera donc de cocher cette case.

BODEN.

En validant cette option, le "Brown Out Detect" est activé. Le microcontrôleur effectuera un reset général lorsque la tension d'alimentation devient faible.

LVP.

En validant cette option, le "Low Votage Programming" est activé. Il sera ainsi possible de programmer le microcontrôleur à partir d'une basse tension (5V).

CPD.

En validant cette option, le "Data EEPROM Read Protect" sera activé. Les données stockées dans la mémoire EEPROM de donnée ne seront pas lisible si l'on fait une relecture de celui-ci. Cependant le composant reste effaçable pour être reprogrammé si celui -ci contient une mémoire Flash.

OSC.

Cette option permet de sélectionner le type d'oscillateur utilisé dans le montage.

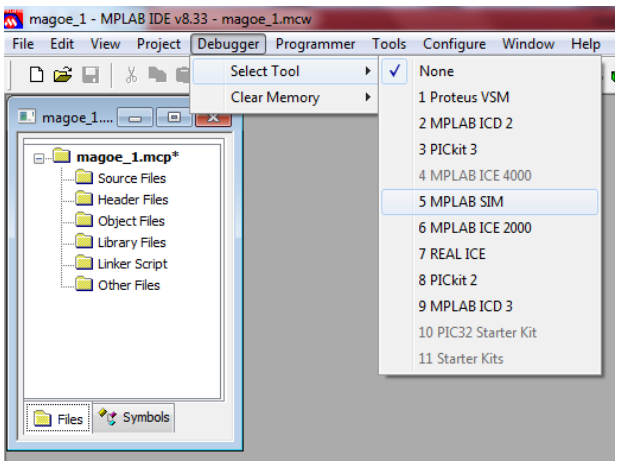
- ***LP :*** Low Power cristal (cristal faible cosommation).

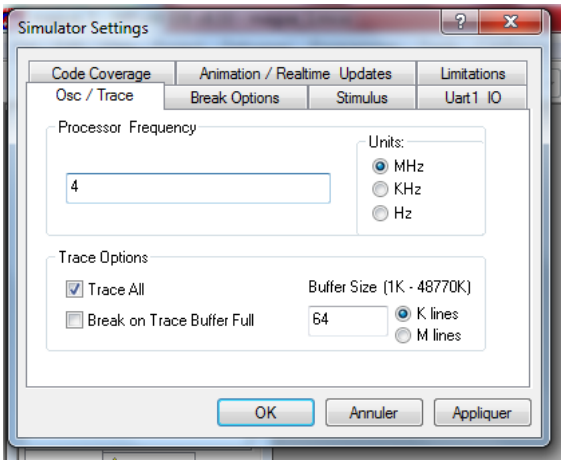
- ***XT :*** Cristal/Resonator (résonateur à quartz).
- ***HS :*** Higth Speed Cristal/Resonator (résonateur à quartz de haut fréquence).
- ***RC :*** Oscillateur externe RC.
- ***INTOSC :*** Oscillateur interne de précision.
- ***EC :*** Entrée d'horloge externe.

II.3.3. Debugger.

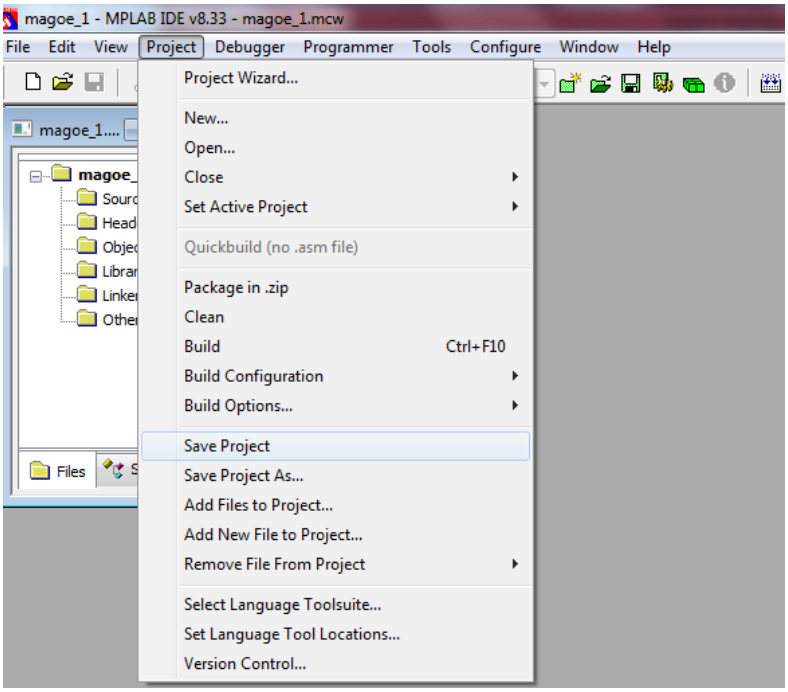
Pour pouvoir utiliser le Debugger afin de faire la simulation du programme, il faut sélectionner MPLAB SIM dans le menu Debugger.

Dans le menu Debugger, de nouvelles sélections apparaissent. Choisir settings pour définir quelques options pour la simulation, en particulier la fréquence de l'horloge dépendant du PIC choisi.





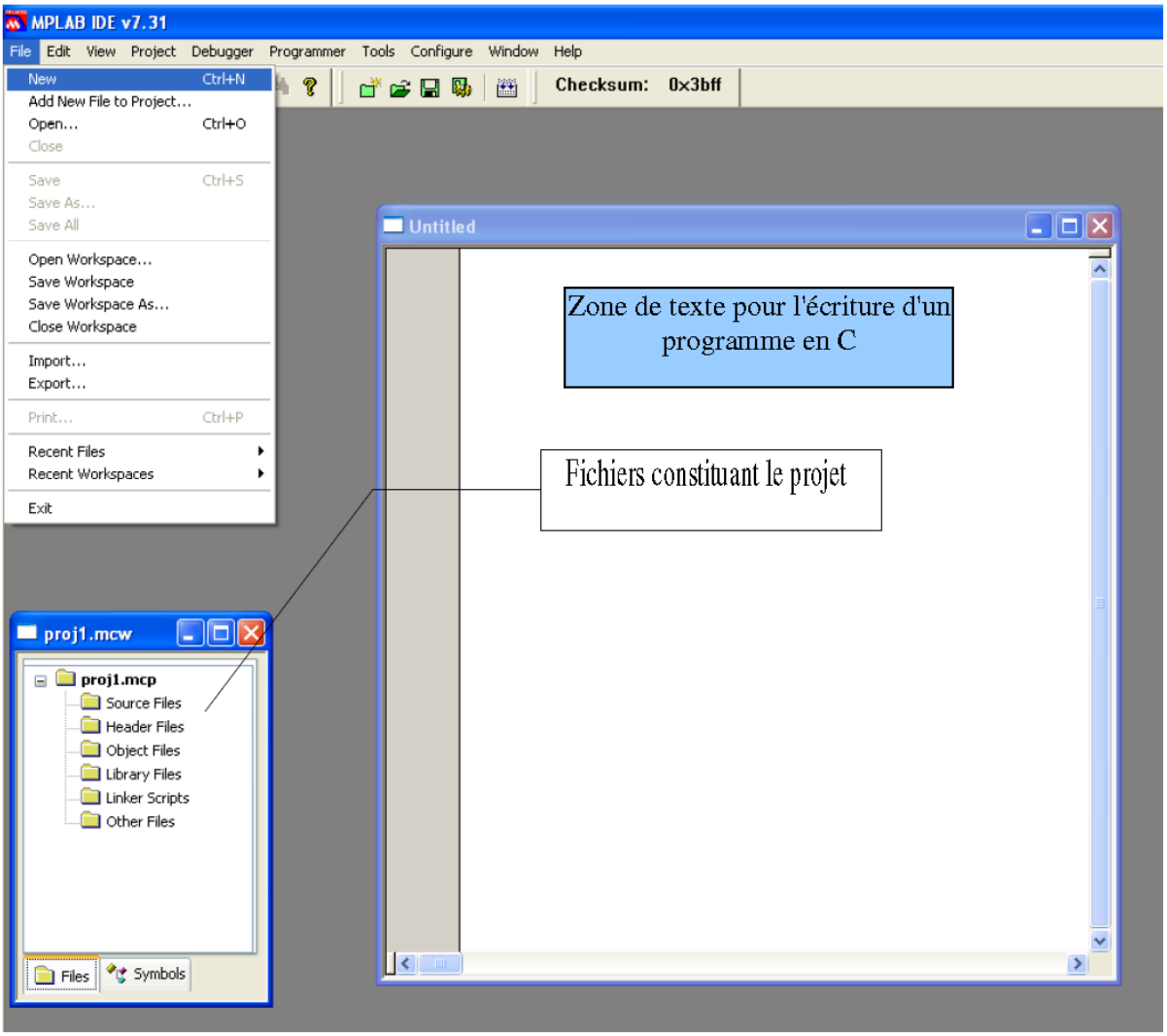
Après toutes ces modifications il convient d'enregistrer le projet.



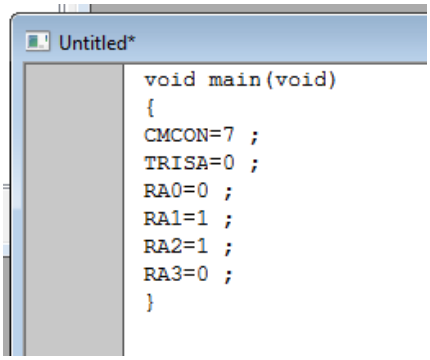
II.4. Ecriture de programme et premières simulations.

II.4.1. Le fichier texte.

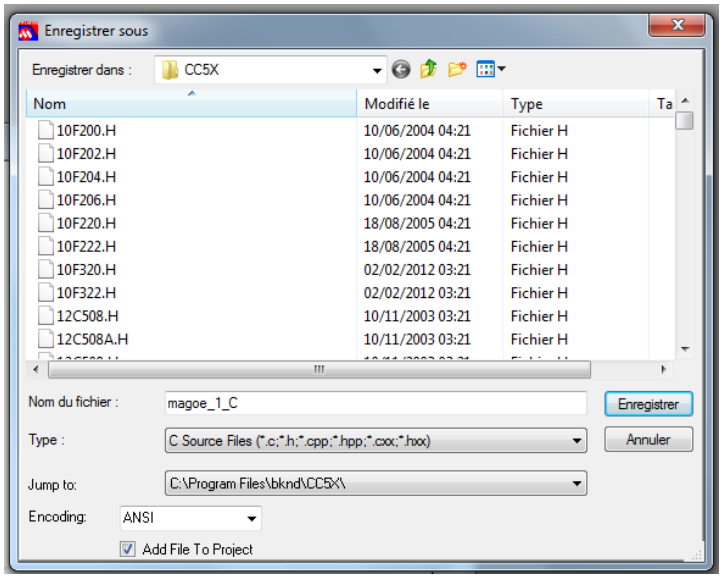
Ayant bouclé l'étape de la création du projet, on désire maintenant ouvrir une fenêtre pour l'écriture du programme en langage C. Pour cela, dans le menu fichier, sélectionner new.

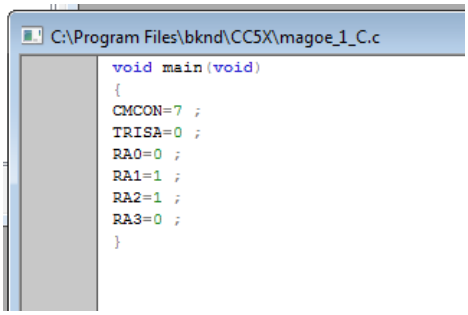


Taper dans la zone de texte, sans pour l'instant chercher à comprendre, le programme suivant :



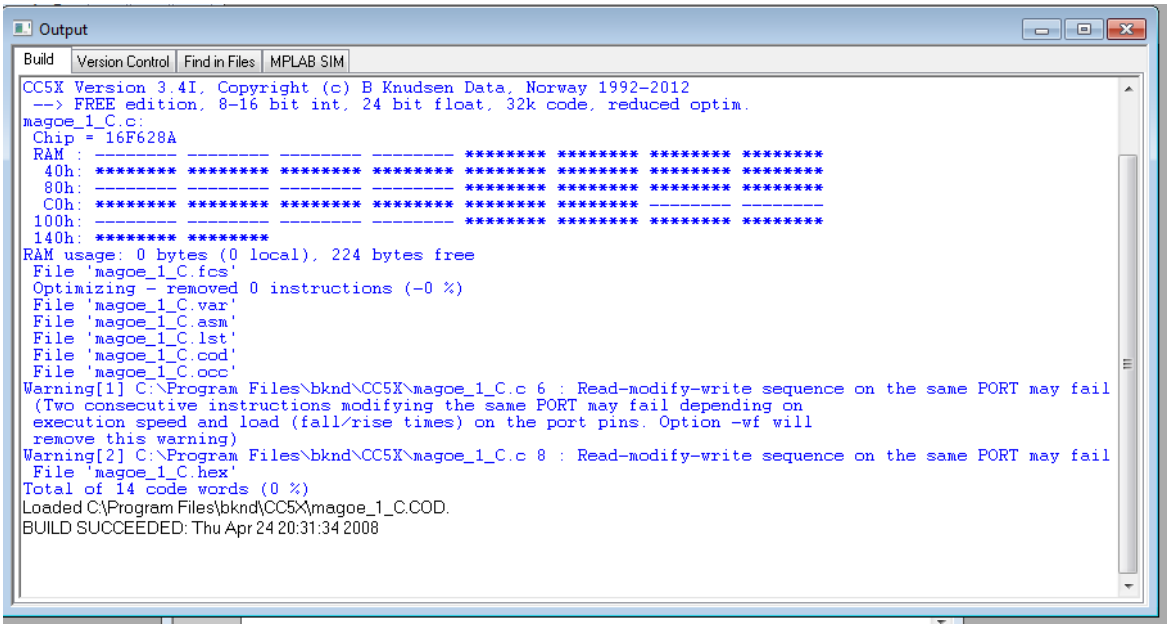
Sauvegarder ensuite le fichier que l'on nommera par exemple ***magoe_1_C*** dans le même répertoire que le projet. Indiquer le type de fichier (C Source File : fichier source C). Le fichier ainsi créé doit alors être ajouté comme fichier source dans le projet (Add File To Project) puis cliquer sur ***Enregistrer***. Le texte ainsi saisie change de forme.





II.4.2. La compilation.

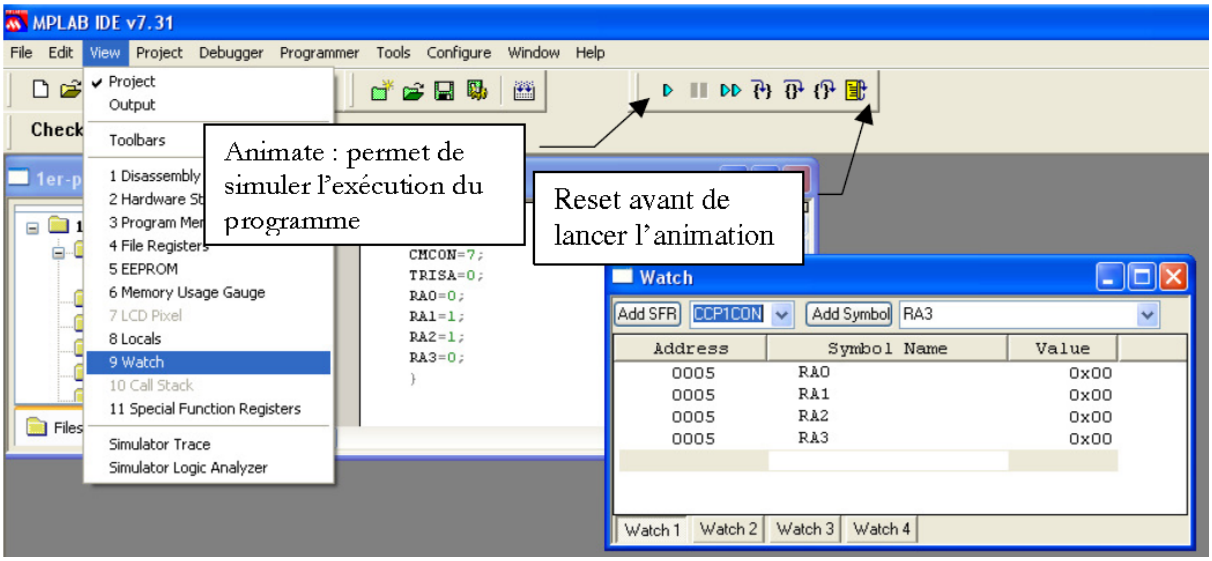
Le projet créé peut maintenant être compilé : Menu Projet/Build.



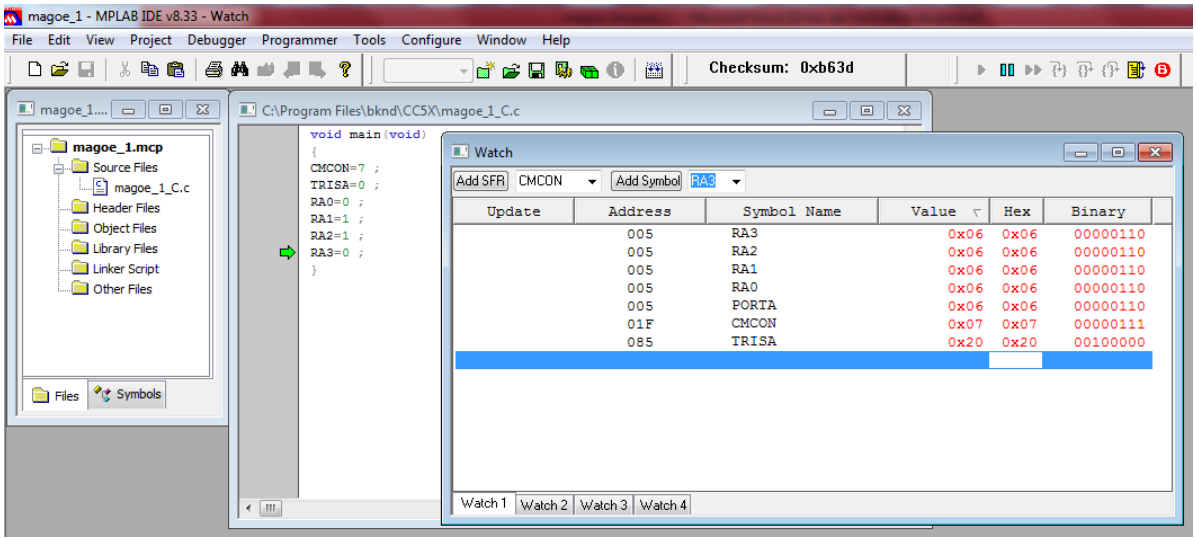
II.4.3. Simulation.

Avant de simuler le fonctionnement du programme, il faut définir ce qu'il convient d'observer. Pour cela sélectionner Watch dans le menu View.

La compilation ayant été réalisée auparavant, on peut sélectionner ***Add symbol*** : RA0 pour visualiser l'état de RA0 lors de la simulation du programme. Puis sélectionner dans la liste ***Add SFR*** : CMCON et TRISA pour visualiser l'état de ces registres. Sélectionner aussi PORTA pour voir le mot binaire disponible sur le port A du pic.

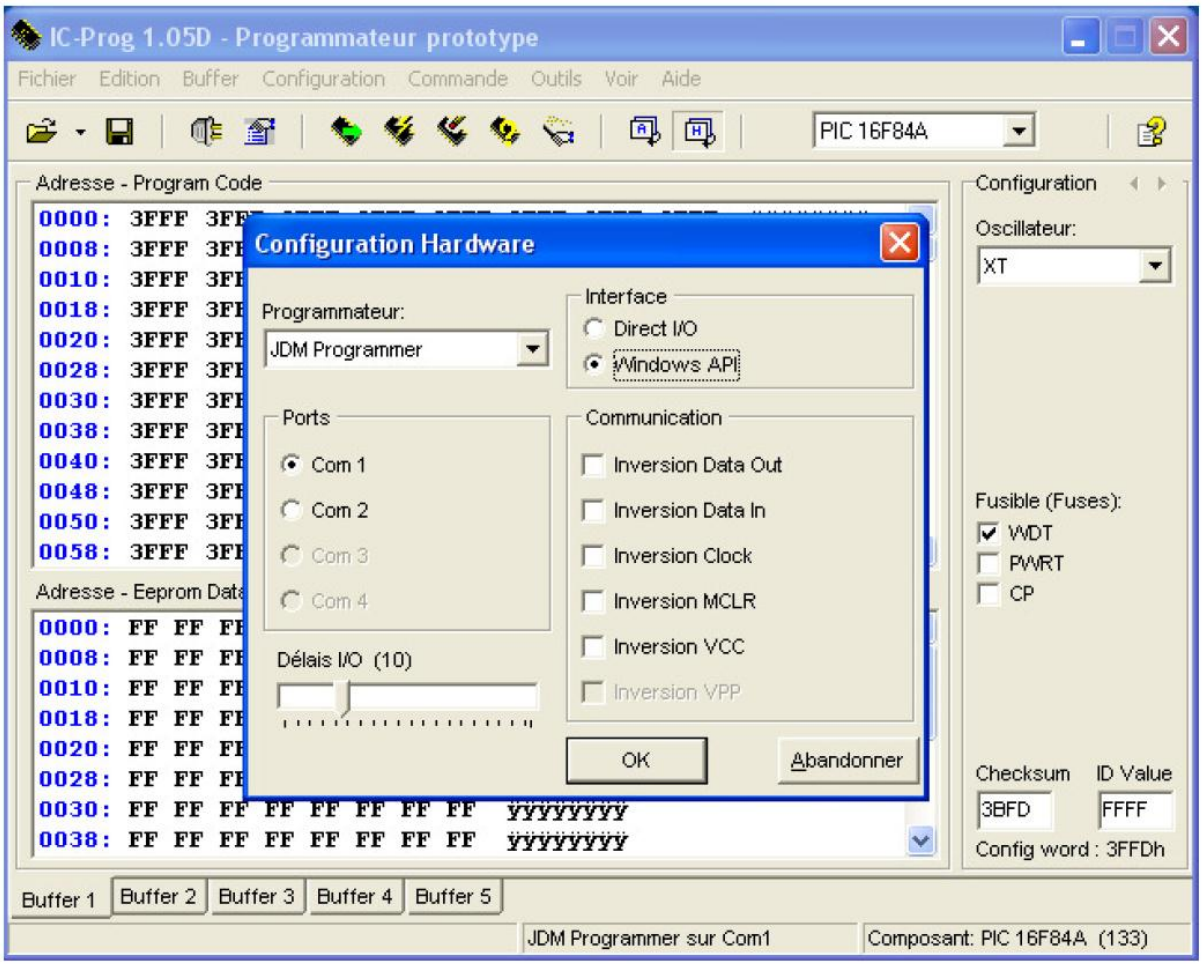


Lancer l'exécution de la simulation. On observe alors la modification des valeurs des registres et du port de sortie. Noter que PORTA et les bits RA0, RA1 etc. … affichent en réalité la même information qui est le mot binaire disponible sur le port de sortie, donc de chaque bit RA0 à RA7.



II.5. Programmation matériel du PIC.

Le transfert du fichier source dans le PIC se fera à l'aide d'un programmateur qui sera relié le port COM d'un PC. Le logiciel utilisé est ICPROG (figure ci-dessous).



II.5.1. Configuration Hardware.

Permet de configurer l'interface de programmation entre le logiciel et la carte de programmation.

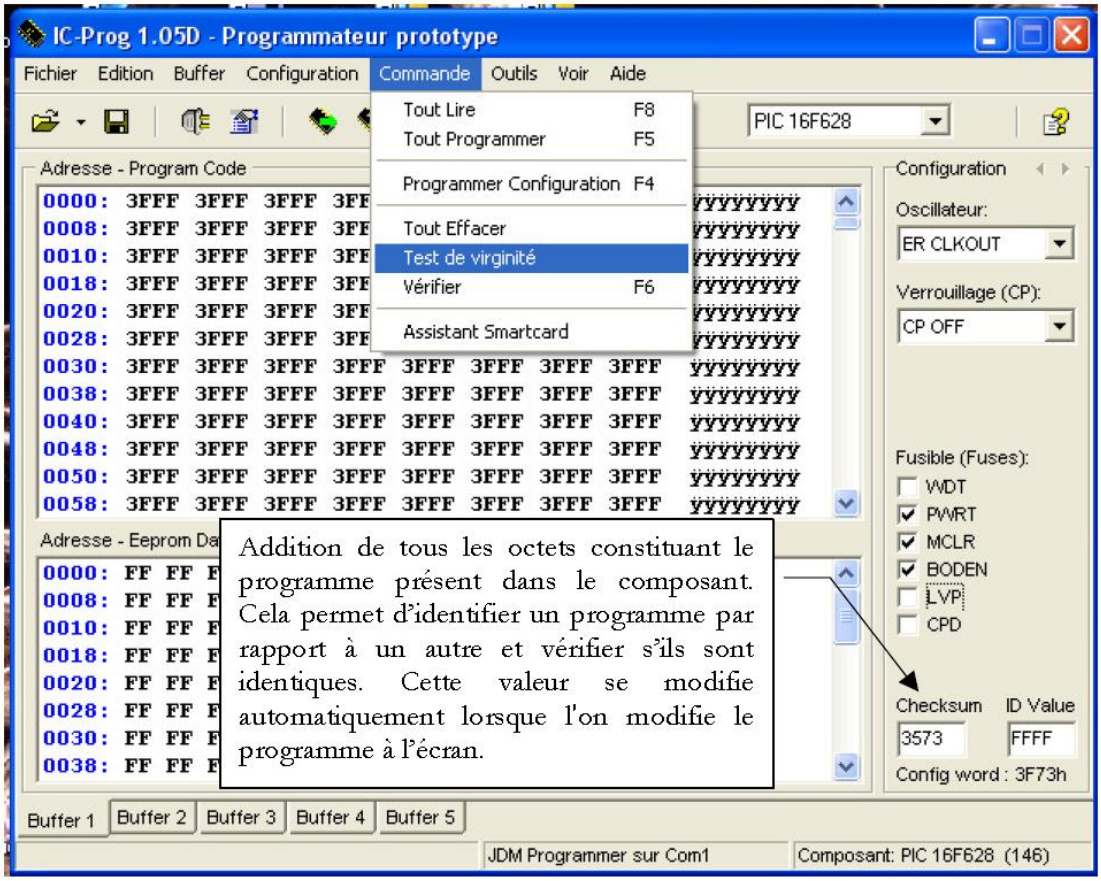
- **Programmateur.** Sélectionner ***JDM programmer***.
- **Ports.** COM1 ou COM2. Dans tous les cas la LED du votre programmateur doit s'allumer lorsque vous effectuez une opération de lecture ou d'écriture. Si ce n'est pas le cas changez de port sélectionné.
- **Délais I/O.** Ce réglage dépend du PC utilisé, essayez sur 1 ou sur 20 en cas de problème de programmation.
- **Interface.** Sélectionner toujours Windows API.
- **Communication.** Permet d'inverser les signaux envoyés ou reçus sur le port série. En général aucune case n'est cochée.

II.5.2. Test de virginité.

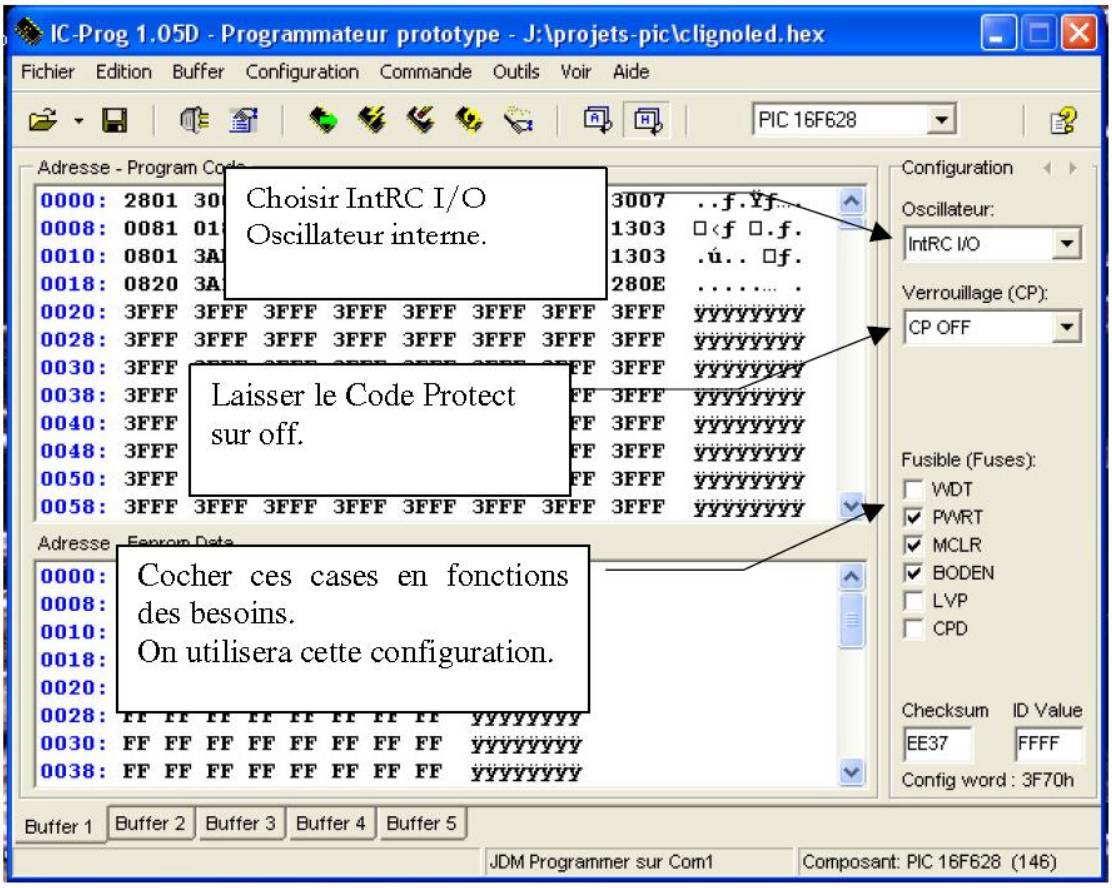
Relier le programmateur au port COM du PC par l'intermédiaire du câble. Placer un PIC dans le bon sens sur le support adéquat. Lancer le test de virginité (Menu Commande/Test de virginité). Ce test permet de vérifier si la mémoire programme du composant est vide.

Si le composant est vierge ou effacé, tous les bits de la mémoire seront au niveau logique 1 (FF). Cette fonction est à utiliser avant toute programmation car il n'est pas possible de programmer un composant correctement si celui-ci n'est pas vierge ou n'a pas été effacé

préalablement. Si ce n'est pas le cas, il faut effacer le composant : menu "Commande\Tout Effacer".



Un choix entre plusieurs oscillateurs peut être réalisé. Cette sélection dépend du type d'oscillateur connecté sur les entrées OSC1/CLKIN et OSC2/CLKOUT lors de l'utilisation du microcontrôleur sur son circuit final après la programmation. Pour les modes XT, un oscillateur à quartz ou un oscillateur TTL/CMOS est connecté sur les entrées OSC1/CLKIN et OSC2/CLKOUT. Pour le mode RC, un pont RC est connecté sur l'entrée OSC1/CLKIN, (fréquence moins précise).

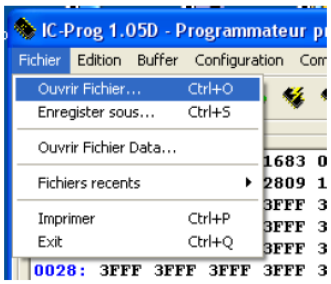


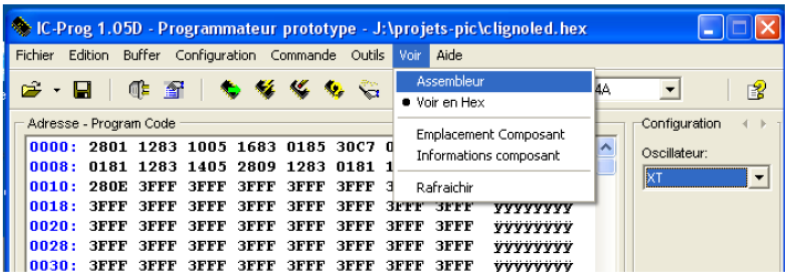
5.3. Transfert du fichier source dans le PIC.

Le logiciel du programmateur utilise un buffer, c'est à dire une mémoire intermédiaire entre les fichiers sur disques et les mémoires programmables des composants, tableau hexadécimal visualisé à l'écran.

Pour programmer un composant à partir d'un fichier il faut d'abord charger le contenu du fichier dans le buffer à l'aide de la commande « Fichier\Ouvrir fichier », puis transférer le contenu du buffer vers le composant avec le menu «Commande\Tout programmer».

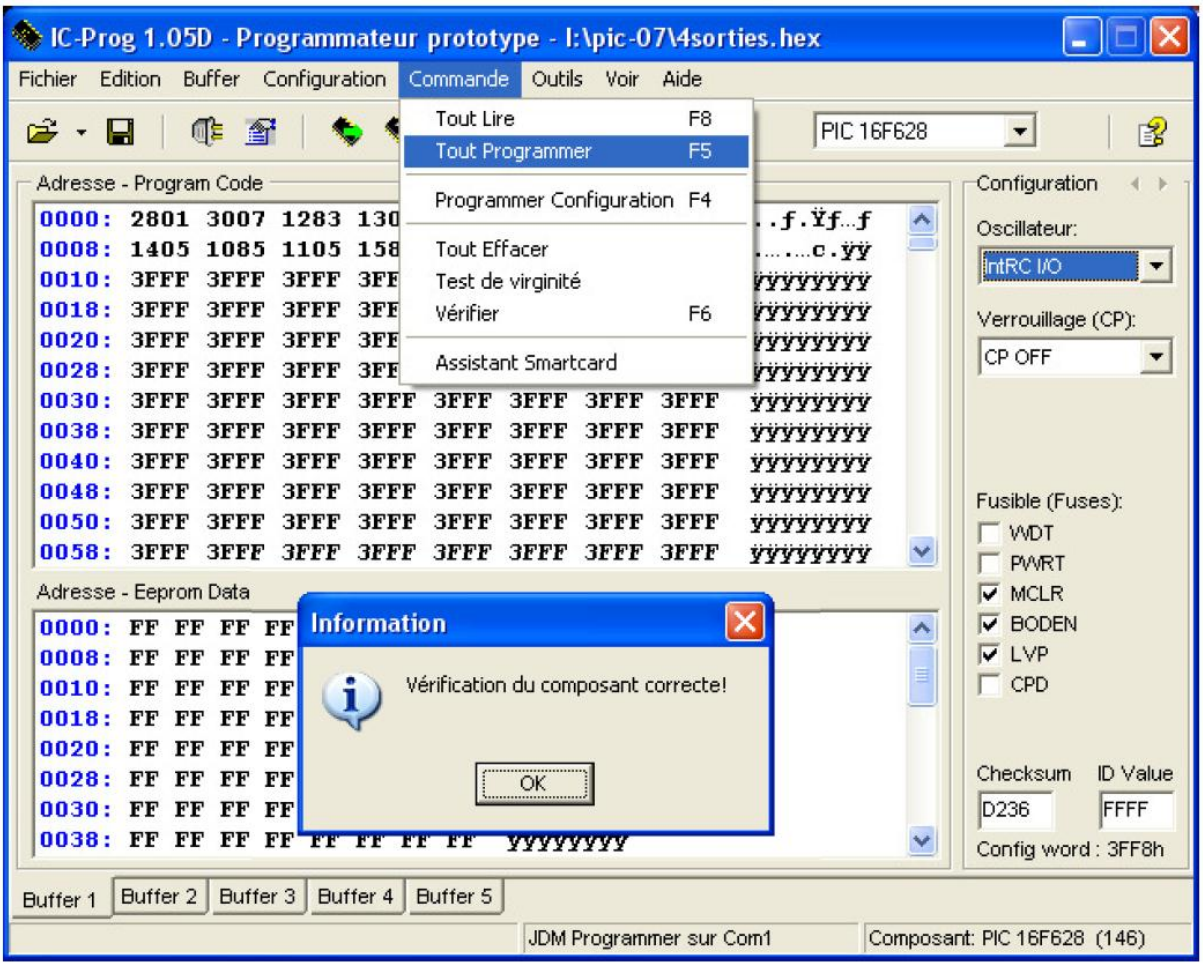
Dans ICprog, ouvrir le fichier sorties.hex créé par le compilateur précédemment. Le programme peut être affiché en hexadécimal ou en assembleur dans la fenêtre Adresse-Program Code.





On constate que le Checksum a changé de valeur. Vérifier que la configuration des fusibles correspond à celle de la compilation du programme dans MPLab, puis choisir Commande/Tout programmer.

Lorsque le transfert du programme dans le pic est réalisé, le logiciel procède à une vérification. Si un message d'erreur apparaît, il peut s'agir d'une mauvaise connexion du programmateur (erreur de port série) ou d'une mauvaise alimentation du programmateur.



III. PROGRAMMATION DES PIC EN C.

III.1. Les composants élémentaires du C.

Un programme en langage C est constitué des six groupes de composants élémentaires suivants :

- les identificateurs,
- les mots-clefs,
- les constantes,
- les chaînes de caractères,
- les opérateurs,
- les signes de ponctuation.

On peut ajouter à ces six groupes les commentaires, qui sont enlevés par le préprocesseur.

III.1.1. Les identificateurs.

Le rôle d'un identificateur est de donner un nom à une entité du programme. C'est une suite de caractères parmi :

- les lettres (minuscules ou majuscules, mais non accentuées),
- les chiffres,
- le "blanc souligné" ( ).

Le premier caractère d'un identificateur ne peut pas être un chiffre. Les majuscules et minuscules sont différenciées.

III.1.2. Les mots-clefs.

Un certain nombre de mots, appelés *mots-clefs*, sont réservés pour le langage lui-même et ne peuvent pas être utilisés comme identificateurs. L'ANSI C compte 32 mots clefs : ***(auto; const; double; float; int; short; struct; unsigned; break; continue; else; for; long; signed, switch; void;case; default; enum; goto; register; sizeof; typedef; volatile;char; do; extern; if; return; static; union; while)*** que l'on peut ranger en catégories.

- **Les spécificateurs de stockage :** auto ; register ; static ; extern ; typedef.
- **Les spécificateurs de type :** char ; double ; enum ; float ; int ; long ; short ; signed ; struct ; union ; unsigned ; void.
- **Les qualificateurs de type :** const ; volatile.
- **Les instructions de contrôle :** break; case; continue; default; do; else; for; goto; if; switch; while.
- **Divers:** return; sizeof.

III.1.3. Les commentaires.

Un commentaire débute par /* et se termine par */. Ils permettent de rendre lisible un programme.

III.2. Structure d'un programme C.

Un programme en C se présente de la façon suivante :

[*directives au préprocesseur*]
[*déclarations de variables externes*]
[*fonctions secondaires*]

```
main()
{
        d´eclarations de variables internes
        instructions
}
```

Une *instruction* est une expression suivie d'un point-virgule. Le point-virgule signifie en quelque sorte "évaluer cette expression". Plusieurs instructions peuvent être rassemblées par des accolades *{* et *}* pour former une *instruction composée* ou *bloc* qui est syntaxiquement équivalent à une instruction. Par exemple :

```
for (;;)
{
do                      // boucle de memorisation d'actions sur les boutons poussoirs 1 2 et 3
        {
        if (inter1) etat_inters.0=1;
        if (inter2) etat_inters.1=1;
        if (inter3) etat_inters.2=1;
        }
while (!inter4);
led1=etat_inters.0;             // mise a jour des leds
led2=etat_inters.1;
led3=etat_inters.2;
etat_inters=0;
while (inter4) ;                // attente que l'on relache inter4
}
```

Le préprocesseur est un programme exécuté lors de la première phase de la compilation. Il effectue des modifications textuelles sur le fichier source à partir de *directives*. Les différentes directives au préprocesseur sont introduites par le caractère #. Ils ont pour but :

- L'incorporation de fichiers source (#include),
- La définition de constantes symboliques (#define),
- La compilation conditionnelle (#if, #ifdef,. . .).

En C, toute variable doit faire l'objet d'une déclaration avant d'être utilisée. Une instruction composée d'un spécificateur de type et d'une liste d'identificateurs séparés par une virgule est une *déclaration*. Par exemple :

```
char a @ 0x11;          // reservation d'un octet nomme a à l'adresse 0x11
bit b @ 0x12.2;         // reservation d'un bit nomme b à la place 2 de l'octet 0x012
char c, d;              // reservation de deux octets nommes c et d
bit e;                  // reservation d'un bit nomme e
uns8 f;                 // reservation d'un octet contenant le nombre 9 et nomme f
int16 g;                // reservation de 2 octets pour un nombre signe nomme g
float h;                // reservation de 3 octets pour un nombre a virgule flottante
```

main est la fonction principale, c'est elle qui est toujours exécutée en première position. Les fonctions secondaires peuvent être placées indifféremment avant ou après la fonction principale.

III.2. Premières règles d'écriture en C.

1. Les instructions propres au langage C doivent être écrites en minuscule.

2. Les instructions particulières au microcontrôleur doivent être écrites en majuscule (PORTA, TRISB, RA6, etc.).
3. Les retours à la ligne et les espaces servent uniquement à aérer le code. Ils peuvent ne pas être utilisés.
4. Les accolades délimitent un groupe d'instruction.
5. Le signe // désigne un commentaire.
6. Le point-virgule indique la fin d'une instruction.
7. Le mot clé **0b** indique que la suite est écrite en binaire.
8. Le mot clé **0x** indique que la suite est écrite en hexadécimal.
9. L'absence de mot clé une écriture en décimal.

III.4. Les types prédéfinis.

Le C est un langage *typé*. Cela signifie en particulier que toute variable, constante ou fonction est d'un type précis. Le type d'un objet définit la façon dont il est représenté en mémoire. La mémoire de l'ordinateur se décompose en une suite continue d'octets. Chaque octet de la mémoire est caractérisé par son *adresse*, qui est un entier. Quand une variable est définie, il lui est attribué une adresse. Cette variable correspondra à une zone mémoire dont la longueur (le nombre d'octets) est fixée par le type.

Chaque compilateur utilise des déclarations différentes pour chaque variable. Voici les types les plus utilisés par CC5X.

- **Le type bit.**
  - ✓ Le type bit occupe un bit en mémoire.
- **Le type entier non signé (entier naturel).**
  - ✓ Le type char ou uns8 occupe un octet. Nous pouvons déposer dans cette variable un nombre compris entre 0 et 255 (0 à 2^8-1).
  - ✓ Le type uns16 occupe deux octets en mémoire (0 à 65535 : 0 à 2^{16}-1).
  - ✓ Il existe aussi les types uns24 et uns32 qui ne sont pas disponibles dans la version d'évaluation du CC5X.
- **Le type entier signé (entier relatif).**
  - ✓ Le type int ou int8 occupe un octet en mémoire (-128 à 127).
  - ✓ Le type int16 utilise deux octets en mémoire (-32768 à 32767).
  - ✓ Il existe aussi les types int24 et in32 qui ne sont pas disponibles avec la version d'évaluation du CC5X.
- **Les types virgule fixe (décimal).** Dans ce type un nombre d'octet est réservé pour la partie entière et un autre pour la partie décimale. Ce type n'existe pas dans la version d'évaluation du CC5X.
  - ✓ Le type fixedU8_8 occupe deux octets en mémoire (0 et 255,996). La résolution pour ce type est de 0.00390625.
  - ✓ Le type fixed8_8 occupe deux octets en mémoire (-128 et 127,996). La résolution pour ce type est de 0.00390625.
  - ✓ Il existe d'autres types à virgule fixe (une vingtaine). Vous trouverez des informations sur ces variables dans la documentation du compilateur.
- **Types à virgule flottante (décimal).** Dans ce type, les nombres sont mis sous une forme assez complexe, comportant une mantisse (chiffres) et un exposant. Dans la version d'évaluation, seul le type float est disponible. Il est codé sous trois octets, et peut aller de +/-1,1e-38 à +/-3,4e38. L'inconvénient de ce type est la longueur du code pour réaliser des opérations et le temps de traitement.

III.5. Les variables.

Une variable est une portion réservée de la mémoire RAM à laquelle on donne un nom. Elle est utilisée afin de garder en mémoire une entité acquise dans le temps, pour la réutiliser plus tard.

La déclaration d'une variable se fait de deux manières :

- ***'type' 'nom' @'adresse_de_la_portion_réservée' ;***
- ***'type' 'nom'*** (le compilateur réserve une portion encore libre).

La place de la déclaration des variables dans le programme est importante et détermine la durée de vie de la variable.

- **Variable globale**: La variable **existe durant tout le programme**. N'importe quelle fonction peut y avoir accès en écriture ou en lecture. On la déclare au début du programme avant toute fonction.
- **Variable locale**: La variable **existe uniquement dans la fonction ou elle a été créée.** L'intérêt d'une telle variable réside dans l'optimisation de la mémoire. Un même emplacement mémoire peut alors être utilisé par plusieurs fonctions si celles-ci ne sont pas imbriquées

Il est important de connaître la place des variables dans la mémoire afin de les visualiser lors de la simulation. Il est aussi indispensable de ne pas déclarer plus de variable que de mémoire disponible. Pour cela, lors de la compilation, une table récapitulative de l'état de la RAM est affichée. Voici ainsi un petit programme qui ne fait rien mais sert uniquement à comprendre le processus.

```
// Attention de respecter les majuscules et minuscules

//------------------------declaration des variables--------------------------

char a @ 0x11;              // reservation d'un octet nomme a à l'adresse 0x11
bit b @ 0x12.2;             // reservation d'un bit nomme b à la place 2 de l'octet 0x012
char c, d;                  // reservation de deux octets nommes c et d
bit e;                      // reservation d'un bit nomme e
uns8 f;                     // reservation d'un octet contenant le nombre 9 et nomme f
int16 g;                    // reservation de 2 octets pour un nombre signe nomme g
float h;                    // reservation de 3 octets pour un nombre a virgule flottante

//-----------------------------Fonction a----------------------------

void aa(void)
{
char loca;
        nop();              // instruction qui ne fait rien
}

//-----------------------------Fonction b----------------------------

void bb(void)
{
char locb;
        nop();              // instruction qui ne fait rien
}


//-----------------------------Fonction principale----------------------------

void main(void)

{
        aa();
        bb();
}
```

Taper ce programme, le compiler. Les messages d'information de la compilation nous informent de l'état de la RAM, nous allons étudier ces messages.

```
RAM : -------- -------- -------- -------- =..7.... ..****** ******** ********
 40h: ******** ******** ******** ******** ******** ******** ******** ********
 80h: -------- -------- -------- -------- ******** ******** ******** ********
 C0h: ******** ******** ******** ******** ******** ******** -------- --------
100h: -------- -------- -------- -------- ******** ******** ******** ********
140h: ******** ********
RAM usage: 10 bytes (1 local), 214 bytes free
```

Explication des lignes de 1 à 6 : Chaque symbole représente l'état d'un octet. La première ligne, par exemple, récapitule l'état des octets 00h à 39h (le h signifie que l'expression est écrite en hexadécimal).

- \- Octet réservé par le compilateur pour le fonctionnement du microcontrôleur (registre PORTA, TRISB, …) ou par l'utilisateur à une place précise (0X11).

- . Variable globale réservé par l'utilisateur, mais dont la place n'a pas d'importance.
- = Variable locale.
- 7 Octet où 7 bits sont disponibles.
- * Octet libre.

La ligne 7 récapitule l'état de la RAM. Remarquez qu'il n'y a qu'un seul octet de réservé pour deux variables locales.

III.6. Les constantes.

Une constante est une valeur qui apparaît littéralement dans le code source d'un programme. Ces constantes vont être utilisées, par exemple, pour initialiser une variable.

III.6.1. Utilisation implicite des constantes.

Le fait d'écrire un nombre dans le programme utilise une constante. Par exemple mettons le registre OPTION à 55.

```
OPTION = 55 ;
```

Le mot clé ***define*** permet également de définir une constante :

#define nom_de_la_constante valeur_de_la_constante.

Le compilateur remplace tous les ***nom_de_la_constante*** qu'il rencontre dans le programme par ***valeur_de_la_constante***.
Exemple : Mettons les registres OPTION et PORTB à 55.

```
#define reglage 55 ;
OPTION = reglage ;
PORTB = reglage ;
```

III.6.2. Utilisation de constantes en EEPROM de programme.

Il est possible de stocker des données dans l'EEPROM de programme au moment de la programmation. Evidement ces données ne sont pas modifiables, ce sont aussi des constantes.
La déclaration d'une telle constante se fait comme suit :

const type nom_de_la_constante = valeur_de_la_constante.

Le compilateur gère le lieu de stockage des données.

III.7. Les expressions.

Les expressions nous permettent de réaliser une opération entre plusieurs entités. Une expression est composée d'un ***opérateur*** et d'un ou de plusieurs ***opérandes***. Nous étudierons 5 types d'opérateurs (les affectations, les opérateurs unaires, les opérateurs binaires, les auto affectations, et les comparaisons).

I / Les affectations

- **" = " Affectation** :

  Place la valeur du 2e opérande dans le 1er.
  Le premier opérande ne peut pas être une constante.
  Exemple

```
a = 5;          // place la valeur 5 dans a
```

II / Les opérations unaires

Elles n'admettent qu'un seul opérande.

- **" . " Sélection d'un bit** (le point) :

  Fait référence à un bit de l'opérande.
  Exemple :

```
PORTA.2 = 1;    //  met RA2 à 1
```

- **" - " Négation** :

  Change le signe de l'opérande.
  Exemple :

```
a = 10;
b = - a;        // b = - 10
```

- **" ! " Complémentation logique** :

  Complémente une variable de type bit.
  Exemple :

```
bit a,b;
a = 1;
b = !a;         // b = 0
```

- **" ~ " Complémentation à un** :

  Inverse tous les bits de l'opérande.
  Exemple :

```
char a, b ;
a  =  5;    //  a = 0000 0101  soit  5
b  = ~a;    //  b = 1111 1010  soit 255 - 5
```

" ++ " Incrémentation :

Ajoute 1 à l'opérande.
Exemples :

```
a = 14;
a++;            // après exécution a=15
c = ++a;        // pré-incrémentation : a = (a+1) puis c = a
                // après exécution c = 16
```

Attention : l'instruction ci-dessous est impossible avec le compilateur CC5X.

```
d = b++;        // post-incrémentation : d = b  puis b = (b+1)
```

- **" -- " Décrémentation** :

  Retranche 1 à l'opérande.
  Exemples :

```
a = 14;
a--;
a = --a;        // pré-décrémentation : a = (a-1) puis c = a
                // après exécution c = 12
```

Attention : l'instruction ci-dessous est impossible avec le compilateur CC5X.

```
d = b--;        // post-décrémentation : d = b  puis b = (b-1)
```

III / Les opérations binaires.

Elles admettent plusieurs opérandes.

- **" + " Addition** :
  Effectue la somme algébrique des opérandes.
  Exemple : a = b + 3; // a contient la somme de b et de 3

- **" - " Soustraction** :
  Effectue la différence des opérandes.
  Exemple a = b - 3; // a contient la différence de b et de 3

- **" * " Multiplication** :
  Effectue le produit des opérandes.
  Exemple a = b * 3; // a contient 3 fois la valeur de b

- **" / " Division** :
  Effectue le quotient des opérandes.
  Exemple a = b / 3; // a contient le tiers de b
  En général, si a est un entier, (3 * a) ne redonnera pas b.

- **" % " Modulo** :
  Renvoie le reste de la division entière des opérandes.
  Exemple a = 7 % 3; // a = 7 - (3 * 2) soit 1

- **" && " ET logique** :
  Renvoie vrai si les 2 opérandes sont vrais et faux dans les 3 autres cas.
  Exemple if (a && b) c=1; // c=1 si a et b sont vrais sinon c=0
  // l'instruction if sera abordée plus loin
  Attention: sous CC5x, il n'est pas possible d'utiliser le ET logique dans une affectation.

- **"&" ET binaire** :
  Effectue un ET logique bit à bit entre les opérandes.
  Exemple a = 5; // a = 0000 0101
  b = 6; // b = 0000 0110
  c = a & b; // c = 0000 0100 soit c = 4

- **" || " OU logique** :
  Renvoie "1" si 1 ou 2 opérandes sont vrais et "0" dans l'autre cas.
  Exemple if (a || b) c=1; // c = 1 si a ou b sont non nuls , sinon c = 0
  // l'instruction if sera abordée plus loin
  Attention: sous CC5x, il n'est pas possible d'utiliser le OU logique dans une affectation.

- **" | " OU binaire** :
  Effectue un OU logique bit à bit entre les opérandes.
  Exemple a = 5; // a = 0000 0101
  b = 6; // b = 0000 0110
  c = a | b; // c = 0000 0111 soit c = 7

- **" ^ " OU exclusif binaire** :
  Effectue un OU exclusif bit à bit entre les opérandes.
  Exemple : a = 5; // a = 0000 0101
  b = 6; // b = 0000 0110
  c = a ^ b; // c = 0000 0011 soit c = 3

- **" << " Décalage de bits à gauche** :

  Décale vers la gauche tous les bits du 1er opérande. Le nombre de décalages est donné par le 2e opérande.

  Exemple : a = 11; // a = 0000 1011 soit a = 0xb
  b = a << 3; // b = 0101 1000 soit b = 0x58

- **" >> " Décalage de bits à droite** :

  Décale vers la droite tous les bits du 1er opérande. Le nombre de décalages est donné par le 2e opérande.

  Exemple a = 11 ; // a = 0000 1011
  b = a >> 3 ; // b = 0000 0001 soit b = 1

IV / Les auto affectations

C'est une écriture condensée liée à une opération binaire et une affectation.
Si l'on écrit a = a + b ; l'écriture de "a" intervient 2 fois, ce qui est inutile.
On pourra écrire : a += b ; // d'abord a+b , ensuite a = (a+b)

Exemples	a += 1;	// a = a + 1	Affectation-somme
	b -= 2;	// b = b - 2	Affectation-différence
	c *= 3;	// c = c * 2	Affectation-produit
	d /= 4;	// d = d / 4	Affectation/division
	e %= 5;	// e = e % 5	Affectation-modulo
	f &= 6;	// f = f & 6	Affectation-et
	g |= 7;	// g = g | 7	Affectation-ou
	h ^= 8;	// h = h ^ 8	Affectation-ou exclusif
	i <<= 9;	// i = i << 9	Affectation-décalage à gauche
	j >>= 10;	// j = j >> 10	Affectation-décalage à droite

V / Les comparaisons

- **" = = " Egalité** :

  Renvoie vrai si les opérandes sont égaux et faux s'ils sont différents.

  Exemple if(a == 2) b=1; // b vaut 1 si a = 2
  // l'instruction if sera abordée plus loin

- **" ! = " Non égalité** :

  Renvoie vrai si les opérandes sont différents et faux s'ils sont égaux.

  Exemple if(a != 2) b=0; // b vaut 0 si a différent de 2
  // l'instruction if sera abordée plus loin

- **" > " Plus grand que** :

  Renvoie vrai si le 1er opérande est supérieur au 2e sinon faux.

  Exemple a = (10 > 20); // a = 0

- **" < " Plus petit que** :

  Renvoie vrai si le 1er opérande est inférieur au 2e sinon faux.

  Exemple a = (10 < 20); // a = 1

- **" >= " Plus grand que ou égal à** :
  Renvoie vrai si le 1er opérande est supérieur ou égal au 2e sinon faux.
  Exemple if(b >= 2) a=1; // a vaut 1 si b est supérieur ou égal à 2
  // l'instruction if sera abordée plus loin

- **" <= " Plus petit que ou égal à** :
  Renvoie vrai si le 1er opérande est inférieur ou égal au 2e sinon faux.
  Exemple if(a <= 2) b=1; // b vaut 1 si a est inférieur ou égal à 2
  // l'instruction if sera abordée plus loin

Attention. Il est très **difficile** avec CC5X **d'utiliser une comparaison dans une affectation**.
L'instruction suivante est correcte mais ne fonctionne pas sous CC5X :
b = (a != 2);
L'instruction suivante fonctionne sous CC5X:
a = (10 < 20); //a=1

III.8. Les tableaux.

Un tableau est un ensemble fini d'éléments de même type, stockés en mémoire à des adresses contigües. La déclaration d'un tableau à une dimension se fait de la façon suivante :

type nom-du-tableau[nombre-éléments];

Où ***nombre-éléments*** est une expression constante entière positive. Par exemple, la déclaration int t[6]; indique que t est un tableau de 6 éléments de type int. Pour différencier les éléments nous avons besoin d'un index. Les éléments prennent ainsi le nom de t[index].

Index	0	1	2	3	4	5
Variable	t[0]	t[1]	t[2]	t[3]	t[4]	t[5]

Déposons par exemple -24 dans t[3] : t[3]=-24 ;

Pour plus de clarté, il est recommandé de donner un nom à la constante ***nombre-éléments*** par une directive au préprocesseur, par exemple :

#define *nombre-éléments 6.*

On accède à un élément du tableau en lui appliquant l'opérateur []. Les éléments d'un tableau sont toujours numérotés de 0 à ***nombre-éléments -1***.

On peut initialiser un tableau lors de sa déclaration par une liste de constantes de la façon suivante :

type nom-du-tableau[N] = {constante-1,constante-2,...,constante-N};

Par exemple, on peut écrire :
#define N 4 ;
int tab[N] = {1, 10, 3, 9};

Si le nombre de données dans la liste d'initialisation est inférieur à la dimension du tableau, seuls les premiers éléments seront initialisés. Les autres éléments seront mis à zéro.

Lors d'une initialisation, il est également possible de ne pas spécifier le nombre d'éléments du tableau. Par défaut, il correspondra au nombre de constantes de la liste d'initialisation.

III.9. Les fonctions.

III.9.1. Généralités.

La plupart des langages de programmation nous permettent de subdiviser nos programmes en sous-programmes plus simples et plus compacts : les fonctions. A l'aide de ces structures nous pouvons **modulariser** nos programmes pour obtenir des solutions plus élégantes et plus efficaces.

Une fonction peut en appeler une autre grâce à des instructions particulières. A la suite d'un appel, **la fonction appelante s'arrête** et **la fonction appelée se met en exécution** jusqu'à ce qu'elle appelle une autre fonction ou se termine. Lors de la fin de son exécution, la fonction appelante reprend son fonctionnement depuis son arrêt. **Une fonction peut envoyer les données à la fonction qu'elle appelle** au moment de son appel. **Une fonction peut envoyer des données à la fonction qui l'a appelée** au moment du retour. **Une fonction appelée doit toujours être définie avant la fonction qui l'appelle**. Elle est soit écrite, soit déclarée avant la fonction qui l'appelle. Chaque fonction peut être appelée plusieurs fois si nécessaire. **Lors du lancement du programme, la fonction "main" s'exécute** en premier.

III.9.2. Fonction sans paramètres.

Une fonction sans paramètres **réalise une suite d'opérations sans échanger aucune donnée** avec le reste du programme.

Structure d'une telle fonction :

```
void nom_de_la_fonction(void)
    {
        // Corps de la fonction (ensemble des instructions qui constituent la fonction)
    }
```

Déclaration d'une telle fonction :

```
void nom_de_la_fonction(void);
```

Utilisation de cette fonction :

```
instruction x;
nom_de_fonction();
instruction y;
nom_de_fonction();
```

L'instruction x se réalise, puis les instructions du corps de la fonction, ensuite l'instruction y, et pour finir une nouvelle fois les instructions du corps de la fonction. Ainsi la fonction s'est réalisée deux fois, mais nous ne l'avons écrite qu'une fois. Elle ne figurera en mémoire qu'une seule fois aussi.

Exercice d'application: Un compteur d'actions.

Nous allons réaliser un programme qui comptera le nombre de fois que l'on a appuyé sur le bouton poussoir 1. A chaque appui une nouvelle LED s'allume. Nous aurons donc au maximum 4 actions.

La difficulté vient du fait que les interrupteurs ont du rebond. C'est à dire que chaque fois que l'on appuie ou que l'on relâche un bouton poussoir, l'interrupteur rebondit et le microcontrôleur comptabilise plusieurs actions. Nous allons donc écrire une fonction "anti-rebond" qui sera tout simplement une temporisation afin d'attendre que le contact

s'immobilise. Nous réaliserons aussi une fonction affichage qui mettra à jour les LED. Dans le programme ci-dessous, la fonction "anti-rebond" est écrite après la fonction qui l'appelle, elle doit donc être déclarée avant. Ce n'est pas le cas de la fonction "affichage".

III.9.3. Fonction avec paramètres d'entrée.

Dans une fonction avec paramètre d'entrée, **la fonction appelée reçoit une ou plusieurs données** appelées "paramètres". Ces paramètres seront stockés au moment de l'appel de la fonction dans des variables dites temporaires qui seront détruite à la fin de la réalisation de la fonction.

Structure d'une telle fonction avec deux paramètres.

```
void nom_de_la_fonction(type1 nom1; type2 nom2)
{
        // Corps de la fonction (ensemble des instructions qui constituent la fonction)
        pouvant utiliser nom1 et nom2.
}
```

Le void qui signifiait "pas de paramètres" est maintenant remplacé par les paramètres d'entrée et leurs types.

Déclaration d'une telle fonction :

```
void nom_de_la_fonction(type1 nom1; type2 nom2);
```

Utilisation de cette fonction :

```
instruction x;
nom_de_fonction(p,q);
instruction y;
nom_de_fonction(r,s);
```

p, q, r, s peuvent être un nombre, une variable. Il est indispensable que p et r soient compatibles avec type1, et que q et s soient compatibles avec type2. Lors du premier appel, la fonction crée les variables nom1 et nom2. Elle copie ensuite p dans nom1 et q dans nom2. Le corps de la fonction peut se réaliser avec les variables nom1 et nom2 qui contiennent p et q. A la fin de la réalisation de la fonction, les variables nom1 et nom2 sont détruites. Lors du deuxième appel, la fonction crée à nouveau les variables nom1 et nom2, place r dans nom1 et s dans nom2. Le corps de la fonction peut se réaliser à nouveau avec les variables nom1 et nom2 qui contiennent cette fois r et s. A la fin de la réalisation de la fonction, les variables nom1 et nom2 sont détruites. La fonction n'a été écrite qu'une fois en mémoire, mais a été exécutée deux fois et avec des données différentes.

Exercice d'application: Un compteur de points.

Quatre équipes s'affrontent et doivent chacune effectuer 5 actions. Chaque fois qu'une équipe a réalisé une action, elle appuie sur son bouton poussoir pour incrémenter son compteur. La valeur du compteur est alors affichée sur les leds (1 led par action). Lorsqu'une équipe a réalisé la 5e action, la led associée à l'équipe s'allume, et le jeu est terminé.

III.9.4. Fonction avec paramètres de sortie.

Dans une fonction avec paramètres de sortie, **la fonction appelée retourne une donnée** au programme. Cette donnée est appelée paramètre de sortie.

Structure d'une telle fonction :

```
type nom_de_la_fonction(void)
{
        // Corps de la fonction
        return donnée ;
}
```

La donnée à renvoyer doit bien évidemment être compatible avec le type déclaré. Pour renvoyer le paramètre et mettre fin à la fonction, il suffit de noter "return" suivi de la donnée que l'on veut renvoyer. Cette ligne doit obligatoirement être la dernière de la fonction. Il est aussi possible d'avoir plusieurs lignes "return donnée" dans une même fonction. La première ligne rencontrée met alors fin à cette fonction. La donnée peut être un nombre, une variable, un calcul.

Déclaration d'une telle fonction :

```
type nom_de_la_fonction(void);
```

Utilisation de cette fonction :

```
instruction x;
a = nom_de_fonction();
instruction y;
```

Instruction x se réalise en premier, puis les instructions de la fonction. A la fin de la réalisation de la fonction, la donnée qui constitue le paramètre de sortie est déposée dans a. Nous en sommes à la troisième ligne, instruction y se réalise.

Exercice d'application: Un dé électronique.

Le bouton poussoir 1 permet d'afficher de façon aléatoire une LED. Nous allons alors écrire une fonction qui fournira un nombre entre 1 et 4. Cette fonction sera appelée chaque fois que le bouton poussoir1 sera enclenché.

III.10. Les interruptions.

III.10.1. Généralités.

Jusqu'ici, les fonctions étaient appelées par le programme lui-même. **Une interruption est une fonction qui se réalise lorsque un évènement se produit**, et non lorsque le programme le décide. Les évènements conduisant à une interruption dépendent du microcontrôleur. Au moment de l'appel de l'interruption, le programme s'arrête. Le microcontrôleur sauvegarde l'adresse de la dernière instruction exécutée ainsi que les registres importants. L'exécution de l'interruption commence. A la fin de sa réalisation, les registres importants reprennent les états qu'ils avaient avant l'appel, le programme reprend à l'endroit où il s'était arrêté.

III.10.2. Le PIC 16F628A.

Chaque microcontrôleur fonctionne différemment lors d'une interruption. De plus, chaque langage C est différent face à ces mêmes interruptions. Nous allons donc assez rapidement analyser le comportement du PIC16F628A, et voir comment gérer cette ressource avec le C que nous utilisons depuis le début. Le PIC16F628A possède 10 sources d'interruption :

1. Changement d'état des pattes RB4 à RB7;
2. Débordement du registre TMR0 (passage de 0XFF à 00);
3. Débordement du registre TMR1 (passage de 0XFF à 00);

4. Front sur la patte INT (Cette patte est aussi la patte RB0). Le sens du front est déterminé par certains bits du registre option;
5. Fin d'écriture dans l'EEPROM du PIC (partie de la mémoire qui ne s'efface pas en cas de coupure d'alimentation).
6. Changement de valeur de sortie d'un comparateur ;
7. Fin de transfert de donnée du registre TXREG au registre TSR ;
8. Fin de transfert de donnée du registre RSR au registre RCREG ;
9. Capture de la valeur du registre TMR1.
10. Egalité entre la valeur du registre TMR2 et celle de PR2.

La gestion des interruptions passe par 3 opérations:

1. Déclaration du fichier utile à la gestion des interruptions;
2. Configuration des interruptions;
3. Ecriture de l'interruption;

La déclaration du fichier utile à la gestion des interruptions est indispensable pour utiliser les instructions de sauvegarde des registres important au moment de l'interruption. Elle passe par l'écriture de la ligne ci-dessous à placer en début de fichier :

```
#include "int16CXX.h".
```

La configuration des 10 interruptions se fait à l'aide de bits des registres INTCON (INTerrupt CONfiguration), PIE1 (Peripheral Interrupt Enable) et OPTION ; le plus souvent au début de la fonction main.

- RBIE (RB Interrupt Enable) : mis à 1, il autorise les interruptions dues au changement d'état des pattes RB4 à RB7.
- T0IE (Tmr0 Interrupt Enable) : mis à 1, il autorise les interruptions dues au TMR0.
- INTE : (INTerrupt Enable) : mis à 1, il autorise les interruptions dues à un front sur la patte RB0/INT.
- INTEDG : (INTerrupt EDGe) : un 1 valide les fronts montants comme source d'interruption sur RB0/INT, un 0 valide les fronts descendants.
- EEIE (EEprom Interrupt Enable) : un 1 valide la fin d'écriture dans l'EEPROM comme source d'interruption.
- GIE (General Interrupt Enable) : mis à 1, il permet à toutes les interruptions de s'exécuter selon leurs configurations. Au début de l'interruption, ce bit est mis à 0 automatiquement. A la fin de l'interruption, ce bit est mis à 1 automatiquement.
- PEIE (PEripheral Interrupt Enable) : mis à 1 il permet aux interruptions périphériques de s'exécuter selon leurs configurations.
- EEIE (EEprom write Interrupt Enable) : mis à 1, il autorise les interruptions dues à la fin d'écriture dans l'EEPROM de donnée.
- CMIE (CoMparator Interrupt Enable) : mis à 1, il autorise les interruptions dues aux changements de valeur de sortie du comparateur.
- RCIE (usart ReCeive Interrupt Enable) : mis à 1, il autorise les interruptions dues à la fin des transferts de donnée entre RSR et RCREG.
- TXIE (usart Transmit Interrupt Enable) : mis à 1, il autorise les interruptions dues à la fin des transferts de donnée entre TXREG et TSR.
- CCP1IE (CCP1 Interrupt Enable) : mis à 1, il autorise les interruptions dues à la capture de la valeur du TMR1.
- TMR2IE (TMR2 Interrupt Enable) : mis à 1, il autorise les interruptions dues à l'égalité entre les valeurs de TMR2 et PR2 ;
- TMR1IE (TMR1 Interrupt Enable) : mis à 1, il autorise les interruptions dues au TMR1.

REGISTER 4-3: INTCON – INTERRUPT CONTROL REGISTER (ADDRESS: 0Bh, 8Bh, 10Bh, 18Bh)

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE	PEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF
bit 7							bit 0

bit 7 **GIE:** Global Interrupt Enable bit
1 = Enables all un-masked interrupts
0 = Disables all interrupts

bit 6 **PEIE**: Peripheral Interrupt Enable bit
1 = Enables all un-masked peripheral interrupts
0 = Disables all peripheral interrupts

bit 5 **T0IE**: TMR0 Overflow Interrupt Enable bit
1 = Enables the TMR0 interrupt
0 = Disables the TMR0 interrupt

bit 4 **INTE**: RB0/INT External Interrupt Enable bit
1 = Enables the RB0/INT external interrupt
0 = Disables the RB0/INT external interrupt

bit 3 **RBIE**: RB Port Change Interrupt Enable bit
1 = Enables the RB port change interrupt
0 = Disables the RB port change interrupt

bit 2 **T0IF**: TMR0 Overflow Interrupt Flag bit
1 = TMR0 register has overflowed (must be cleared in software)
0 = TMR0 register did not overflow

bit 1 **INTF**: RB0/INT External Interrupt Flag bit
1 = The RB0/INT external interrupt occurred (must be cleared in software)
0 = The RB0/INT external interrupt did not occur

bit 0 **RBIF**: RB Port Change Interrupt Flag bit
1 = When at least one of the RB<7:4> pins changes state (must be cleared in software)
0 = None of the RB<7:4> pins have changed state

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

REGISTER 4-4: PIE1 – PERIPHERAL INTERRUPT ENABLE REGISTER 1 (ADDRESS: 8Ch)

R/W-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0	R/W-0
EEIE	CMIE	RCIE	TXIE	—	CCP1IE	TMR2IE	TMR1IE
bit 7							bit 0

bit 7 **EEIE:** EE Write Complete Interrupt Enable Bit
1 = Enables the EE write complete interrupt
0 = Disables the EE write complete interrupt

bit 6 **CMIE:** Comparator Interrupt Enable bit
1 = Enables the comparator interrupt
0 = Disables the comparator interrupt

bit 5 **RCIE:** USART Receive Interrupt Enable bit
1 = Enables the USART receive interrupt
0 = Disables the USART receive interrupt

bit 4 **TXIE:** USART Transmit Interrupt Enable bit
1 = Enables the USART transmit interrupt
0 = Disables the USART transmit interrupt

bit 3 **Unimplemented:** Read as '0'

bit 2 **CCP1IE:** CCP1 Interrupt Enable bit
1 = Enables the CCP1 interrupt
0 = Disables the CCP1 interrupt

bit 1 **TMR2IE:** TMR2 to PR2 Match Interrupt Enable bit
1 = Enables the TMR2 to PR2 match interrupt
0 = Disables the TMR2 to PR2 match interrupt

bit 0 **TMR1IE:** TMR1 Overflow Interrupt Enable bit
1 = Enables the TMR1 overflow interrupt
0 = Disables the TMR1 overflow interrupt

REGISTER 4-2: OPTION_REG – OPTION REGISTER (ADDRESS: 81h, 181h)

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
$\overline{\text{RBPU}}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
bit 7							bit 0

bit 7 **$\overline{\text{RBPU}}$**: PORTB Pull-up Enable bit
1 = PORTB pull-ups are disabled
0 = PORTB pull-ups are enabled by individual port latch values

bit 6 **INTEDG**: Interrupt Edge Select bit
1 = Interrupt on rising edge of RB0/INT pin
0 = Interrupt on falling edge of RB0/INT pin

bit 5 **T0CS**: TMR0 Clock Source Select bit
1 = Transition on RA4/T0CKI/CMP2 pin
0 = Internal instruction cycle clock (CLKOUT)

bit 4 **T0SE**: TMR0 Source Edge Select bit
1 = Increment on high-to-low transition on RA4/T0CKI/CMP2 pin
0 = Increment on low-to-high transition on RA4/T0CKI/CMP2 pin

bit 3 **PSA**: Prescaler Assignment bit
1 = Prescaler is assigned to the WDT
0 = Prescaler is assigned to the Timer0 module

bit 2-0 **PS<2:0>**: 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

REGISTER 4-5: PIR1 – PERIPHERAL INTERRUPT REGISTER 1 (ADDRESS: 0Ch)

R/W-0	R/W-0	R-0	R-0	U-0	R/W-0	R/W-0	R/W-0
EEIF	CMIF	RCIF	TXIF	—	CCP1IF	TMR2IF	TMR1IF
bit 7							bit 0

bit 7 **EEIF:** EEPROM Write Operation Interrupt Flag bit
1 = The write operation completed (must be cleared in software)
0 = The write operation has not completed or has not been started

bit 6 **CMIF**: Comparator Interrupt Flag bit
1 = Comparator output has changed
0 = Comparator output has not changed

bit 5 **RCIF**: USART Receive Interrupt Flag bit
1 = The USART receive buffer is full
0 = The USART receive buffer is empty

bit 4 **TXIF**: USART Transmit Interrupt Flag bit
1 = The USART transmit buffer is empty
0 = The USART transmit buffer is full

bit 3 **Unimplemented**: Read as '0'

bit 2 **CCP1IF**: CCP1 Interrupt Flag bit
Capture Mode
1 = A TMR1 register capture occurred (must be cleared in software)
0 = No TMR1 register capture occurred
Compare Mode
1 = A TMR1 register compare match occurred (must be cleared in software)
0 = No TMR1 register compare match occurred
PWM Mode
Unused in this mode

bit 1 **TMR2IF**: TMR2 to PR2 Match Interrupt Flag bit
1 = TMR2 to PR2 match occurred (must be cleared in software)
0 = No TMR2 to PR2 match occurred

bit 0 **TMR1IF**: TMR1 Overflow Interrupt Flag bit
1 = TMR1 register overflowed (must be cleared in software)
0 = TMR1 register did not overflow

L'écriture de l'interruption est plus subtile qu'il n'y paraît. Lors de la mise sous tension du microcontrôleur, le programme exécute en premier l'instruction à l'adresse 0 du programme. Le compilateur place à cette adresse l'instruction en assembleur "goto main". Ce qui signifie aller à la fonction "main". Lorsqu'une interruption se produit, le microcontrôleur sauvegarde son environnement et saute à l'adresse 0X4 du programme. Il est donc indispensable d'écrire l'interruption à partir de cette adresse, puis d'écrire les autres fonctions. Ainsi les adresses 0X1, 0X2, 0X3 restent vides. La structure prend la forme suivante :

```
#pragma origin 4
interrupt nom_de_l'interruption(void)
{
        int_save_registers
        Corps de l'interruption
        int_restore_registers
}
```

- **#progma origin 4** indique au compilateur d'écrire à partir de l'adresse 4.
- **interrupt nom_de_l'interruption(void)** indique que cette fonction est une interruption et lui donne un nom.

- **int_save_register** est une instruction qui sauvegarde l'état des registres importants.
- int_restore_registers est une instruction qui restitue l'état des registres importants.
- **Le corps de l'interruption** passe en premier par la détection de la source d'interruption. Pour cela, nous utiliserons certains bits du registre INTCON.

Exercice d'application :

1) *Notre premier exemple consiste à faire clignoter une LED et mettre en fonction une autre LED si un bouton poussoir est appuié.*
2) *Nous allons maintenant programmer une serrure codée. Le code se compose grâce aux 3 boutons poussoirs 1, 2 et 3 afin de verrouiller la porte. Le code est 1322. Le bouton poussoir 4 permet d'ouvrir la porte lorsqu'elle n'est pas verrouillée. Une LED vertes s'allume pendant que le bouton poussoir 4 est enclenché. Une LED jaune clignote lorsque la porte n'est pas verrouillée, une autre LED rouge s'allume lorsque la porte est verrouillée.*