

TP microcontrôleur

mini projets sur PIC18



Sommaire

1 - Introduction	page 2
2 - Carte de prototypage	page 2
3 - Exercices	
3.1 Réalisation d'un compteur modulo 10	page 3
3.2 Somme des 20 1 ^{ers} entiers	page 3
3.3 Somme des 40 1 ^{ers} entiers	page 4
3.4 Appel d'un sous-programme	page 4
3.5 Utilisation d'un tableau stocké en mémoire RAM	page 5
3.6 Utilisation d'un tableau stocké en mémoire Flash	page 5
3.7 Utilisation des ports d'Entrées/Sorties	page 8
3.8 Réalisation d'une routine de temporisation software	page 8
3.9 Réalisation d'une routine de temporisation hardware	page 9
3.10 Utilisation des interruptions	page 10
3.11 Utilisation du convertisseur analogique/numérique	page 10
3.12 Génération d'un signal PWM	page 11

Annexes

1	Création d'un projet sous MPLABX	page 12
2	Ajout d'un fichier source à un projet	page 15
3	Utilisation du debugger - Simulation d'un programme	page 18
4	Appel d'un sous-programme - Exécution sur le composant ..	page 24

1 - Introduction

L'objectif de ce mini-projet est d'étudier le fonctionnement d'un microcontrôleur 8 bits, de la famille PIC18 de Microchip, ainsi que son environnement de développement.

Ce dernier est ce que l'on appelle un IDE (Integrated Development Environment): il intègre tous les outils de développement d'un logiciel embarqué:

- gestionnaire de projet,
- écriture d'un programme, que ce soit en assembleur ou en utilisant un langage évolué (langage C),
- compilation / édition de lien,
- simulation,
- chargement du fichier exécutable dans un composant,
- test du programme sur le composant cible.

Durant ce mini-projet, une série d'exercice permettra d'aborder les principales fonctionnalités d'un microcontrôleur.

Tous les programmes seront écrits en assembleur. Pour certains exercices, la vérification se fera en simulation uniquement, tandis que pour d'autres, le test se fera à l'aide d'une carte d'apprentissage comportant un microcontrôleur ainsi que quelques modules (bouton poussoir, LEDs, potentiomètre, etc...).

2 - Carte de prototypage

La carte de prototypage utilisée pour cette série d'exercices est basée sur un microcontrôleur **PIC18F45K20**.

Elle permet de mettre en œuvre les principales fonctionnalités du composant (écriture et simulation d'un programme, gestion des entrées/sorties, utilisation des périphériques, etc.).

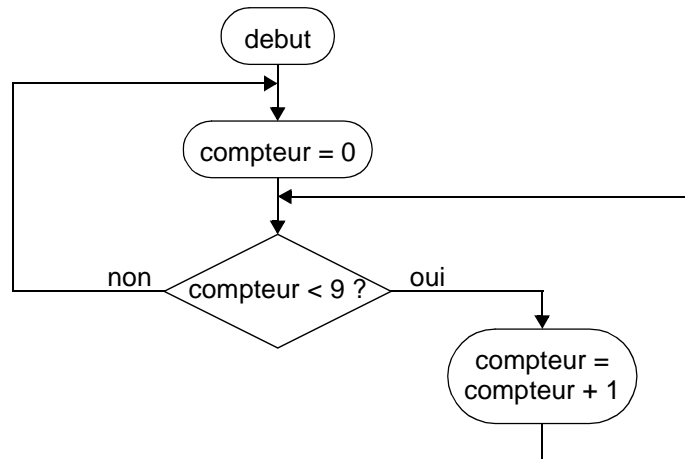
Cette carte sera utilisée via un module Debugger/Programmeur **PICKIT3**. Ce dernier permet de charger un programme dans un microcontrôleur afin de le tester et le mettre en point. Celui-ci peut être lancé en continu, ou en mode pas-à-pas (une instruction ou quelques instructions à la fois) afin de vérifier le comportement du système. Après validation du programme, il peut être chargé de façon permanente dans la mémoire non-volatile du composant.

3 - Exercices

3.1 Réalisation d'un compteur modulo 10

L'exercice consiste à écrire et tester un programme qui compte de 0 à 9, puis recommence à 0, en bouclant indéfiniment.

La valeur du compteur sera mémorisée dans une variable appelée (par exemple) «compteur», qu'il faudra définir dans la zone des variables.



Tester le programme par simulation en mode pas à pas.

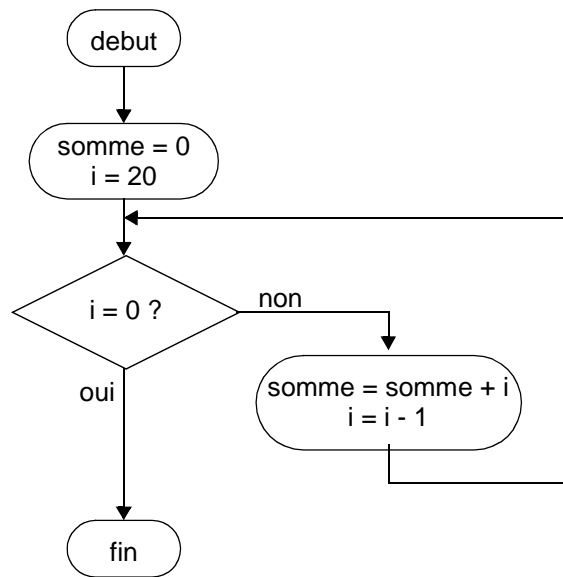
Visualiser l'évolution de la variable «compteur», du compteur ordinal (Program Counter), et du registre d'état.

3.2 Réalisation d'une boucle «for»: somme des 20 1^{ers} entiers

Cet exercice illustre comment réaliser en assembleur une boucle «for». Il consiste à calculer la somme des 20 premiers entiers.

Le résultat sera stocké dans une variable «somme» qu'il conviendra de déclarer dans la zone mémoire dédiée.

L'indice de boucle appelé par exemple «i» sera quant à lui stocké dans une autre variable.



Vérifier le bon fonctionnement du programme en le testant en mode pas-à-pas.

3.3 Manipulation de données codées sur plusieurs octets: somme des 40 1^{ers} entiers

Cette partie reprend l'exercice précédent, mais on cherche maintenant à sommer les 40 premiers entiers. Comme le résultat est supérieur à 255, il faudra le coder sur plus d'un octet. (Etant inférieur à $2^{16} - 1 = 65535$, 2 octets seront suffisants.)

En suivant la même démarche que précédemment, calculer la somme et stocker le résultat dans 2 variables «somme_High» et «somme_Low».

3.4 Appel d'un sous-programme

En assembleur comme en langage évolué, il est souvent utile de faire appel à des routines (sous-programmes).

Ceci permet:

- d'améliorer la lisibilité et la compréhension d'un programme,
- de n'écrire qu'une seule fois un ensemble d'instructions qui peuvent être exécutées de façon répétitive.

On repartira dans le cas présent sur la base de l'exercice précédent qui consiste à calculer la somme des premiers entiers, mais en faisant cette fois appel à un sous-programme «calcul_somme».

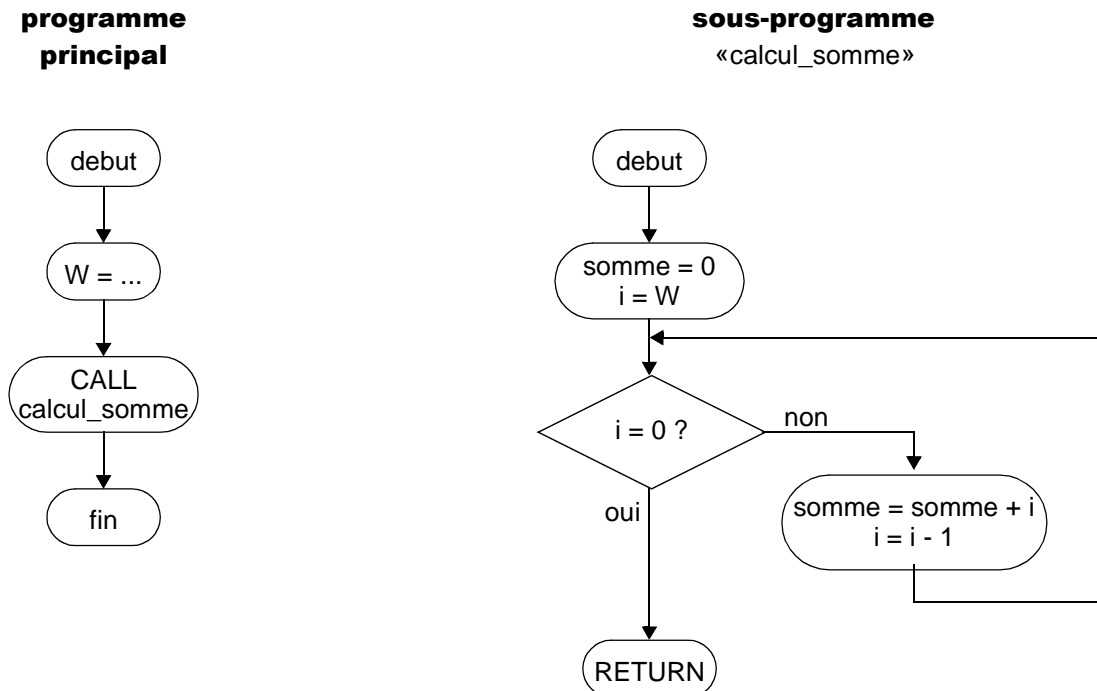
Ce sous-programme

- prendra en entrée le nombre d'entiers à additionner, stocké au préalable dans l'accumulateur,
- retournera le résultat dans les variables «somme».

rappel: le retour de sous-programme se fait avec l'instruction RETURN.

Le programme principal devra

- écrire dans l'accumulateur le nombre d'entiers à additionner,
- appeler le sous-programme «calcul_somme», grâce à l'instruction CALL.



Tester le programme en mode pas-à-pas. Visualiser le haut de la pile (Top Of Stack: registres TOSH et TOSL), ainsi que le pointeur de pile (Stack Pointer: registre STKPTR), notamment lors de l'exécution des instructions CALL et RETURN.

3.5 Utilisation d'un tableau (données stockées dans la mémoire RAM): recherche de min et de max

Dans certaines situations, il est utile d'organiser les données sous formes de tableaux. C'est notamment le cas lorsque l'on souhaite passer en revue (écrire ou lire) un groupe de données avec lesquelles on travaille.

Dans l'exemple qui suit, on cherchera le minimum et le max d'un ensemble de données organisées dans un tableau, et stockées dans la mémoire RAM. Cette dernière étant volatile (effacée lors d'une coupure d'alimentation), il faudra dans un premier temps remplir le tableau, pour ensuite le scruter afin d'en extraire les valeurs min et max.

- 1) Le programme principal devra d'abord appeler un sous-programme «ecrire_tableau» qui initialise un tableau avec les valeurs suivantes:

tableau[0]	25
tableau[1]	4
tableau[2]	2
tableau[3]	15
tableau[4]	16

tableau[5]	101
tableau[6]	33
tableau[7]	3

Les données seront écrites à partir de l'adresse 100h.

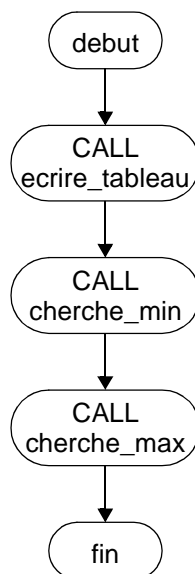
Utiliser pour cela l'**adressage indirect** (initialiser par exemple le registre FSR0 à 100h, puis travailler avec le registre INDF0, ou mieux encore, POSTINC0).

Ce sous-programme retournera également la taille du tableau dans une variable «taille_tableau».

- 2) Le programme principal appellera ensuite le sous-programme «cherche_min» qui lit successivement toutes les valeurs du tableau, en partant de l'adresse de départ (ici, 100h), jusqu'à atteindre la dernière donnée (que l'on peut identifier grâce à la variable «taille_tableau» mise à jour précédemment).

Ce sous-programme retournera la plus petite valeur qu'il a trouvée, mise dans la variable «min».

- 3) Le programme principal appeler enfin le sous-programme «cherche_max» qui, de la même manière que précédemment, retournera la plus grande valeur du tableau, mise dans la variable «max».



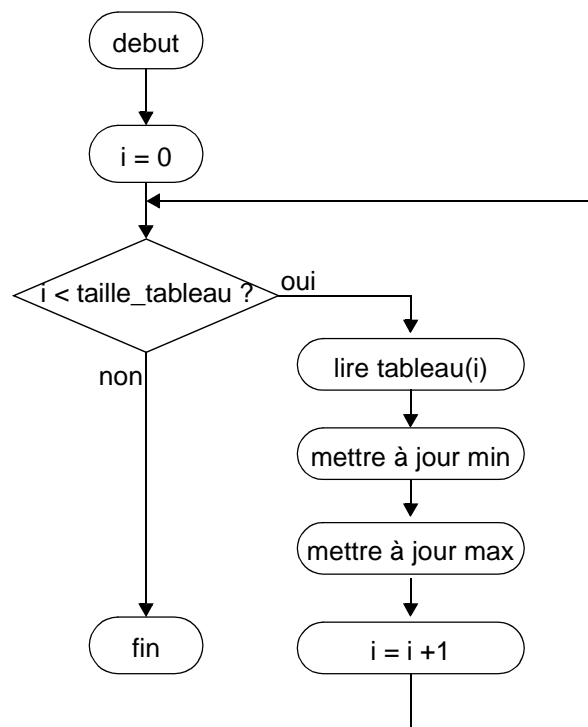
Tester le programme par simulation.

3.6 Utilisation d'un tableau (données stockées dans la mémoire Flash)

Lorsque les données sont écrites dans la mémoire RAM (volatile), il faut au préalable les initialiser, comme c'est le cas dans l'exemple précédent. Cette étape est par contre inutile si l'on travaille avec des constantes déjà écrites dans la mémoire Flash (ou mémoire programme, donc non volatile). Pour lire ces informations, il faudra donc les récupérer de la mémoire programme, afin de les placer dans la mémoire données et pouvoir ainsi travailler avec.

Il est demandé ici de chercher comme précédemment le minimum et le maximum dans une table de valeurs, mais comme ces dernières seront des constantes, il ne faudra pas initialiser le tableau: il suffira de le lire.

L'algorithme du programme sera le suivant:



Deux méthodes sont possibles. Vous utiliserez celle de votre choix.

3.6.1 *Computed GOTO* (cf. page 68 de la datasheet du PIC18F45K20)

Avec cette méthode, la constante est associée à l'instruction «RETLW nn», qui a le même effet que l'instruction RETURN, mais retourne en plus la valeur 'nn' dans le registre W.

Il faut pour cela appeler une routine, par exemple grâce à l'instruction «CALL table». W devra au préalable être chargé avec la valeur du décalage souhaité dans le tableau. Au début de la routine «table», la valeur du Program Counter sera décalée (instruction ADDWF PCL), ce qui permettra d'exécuter ensuite l'une des instructions «RETLW nn» de la liste.³

Pour récupérer un octet de donnée dans la mémoire programme, cette méthode utilise une instruction, soit 2 octets dans la mémoire programme. De plus, la technique «CALL ... / RETURN» occupe une place dans la pile pour stocker l'adresse de retour. Elle n'est donc pas optimale.

Dans le cas où l'on souhaite placer les données à une adresse particulière dans la mémoire Flash (par exemple en 0x0100), la syntaxe est la suivante:

```

...
CALL Table
...
label CODE 0x0100
Table
...
; modification du Program Counter pour
...
; aller pointer vers l'un des instructions RETLW qui suit
...
RETLW .....

```

3.6.2 *Table read* (cf. page 89 de la datasheet du PIC18F45K20)

Avec cette méthode, il est possible de transférer un octet de la mémoire programme dans un registre (TABLAT). L'octet dont l'adresse (sur 21 bits) est indiquée dans un pointeur dédié (registres TBLPTRL, TBLPTRH et TBLPTRU) est ainsi transféré dans TABLAT par une instruction spécifique (TBLRD*, TBLRD*+, etc.).

Le registre TABLAT peut ensuite être lu comme n'importe quel registre de la mémoire RAM.

Avec cette méthode, pour chaque donnée stockée dans la mémoire programme, un seul octet est utilisé dans la mémoire Flash (contre 2 avec la méthode «Computed GOTO»).

Dans le cas où l'on souhaite placer les données à une adresse particulière dans la mémoire Flash (par exemple en 0x0100), la syntaxe est la suivante:

```
label    CODE    0x0100
autre_label    DB    0x19, b'00000100', .....
```

3.7 Utilisation des ports d'Entrées/Sorties

La carte de prototypage comporte 8 LEDS connectées sur le port D (pattes RD0 à RD7).

L'objectif de cet exercice est d'allumer alternativement une LED sur deux.

Il faudra donc utiliser un PICkit3 et travailler en mode debug avec ce matériel (cf. annexe 4).

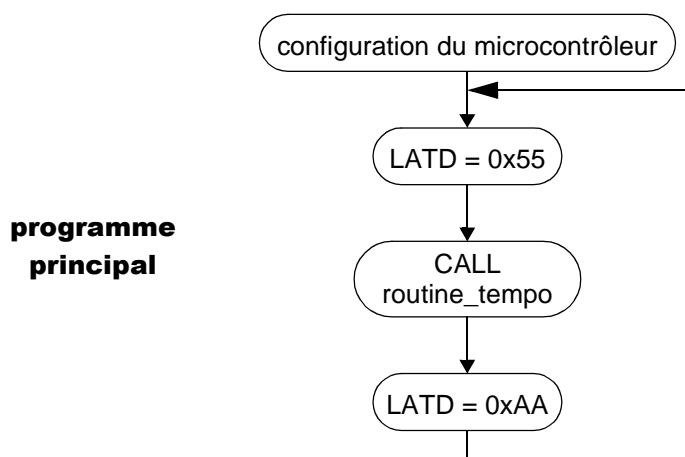
Le programme devra configurer le port D en sortie (registre TRISD), puis écrire alternativement b'01010101' et son complément sur ce port.

L'oscillateur interne sera utilisé (la carte de prototypage ne comprend pas de quartz), et on fera en sorte que le temps d'exécution d'une instruction soit de 1 μ s (= (1 MHz)⁻¹). Il faudra donc configurer la fréquence d'oscillateur à 4 MHz (registre OSCCON, cf. pages 28-29 de la datasheet du PIC18F45K20).

Tester sur la carte le programme en mode pas-à-pas.

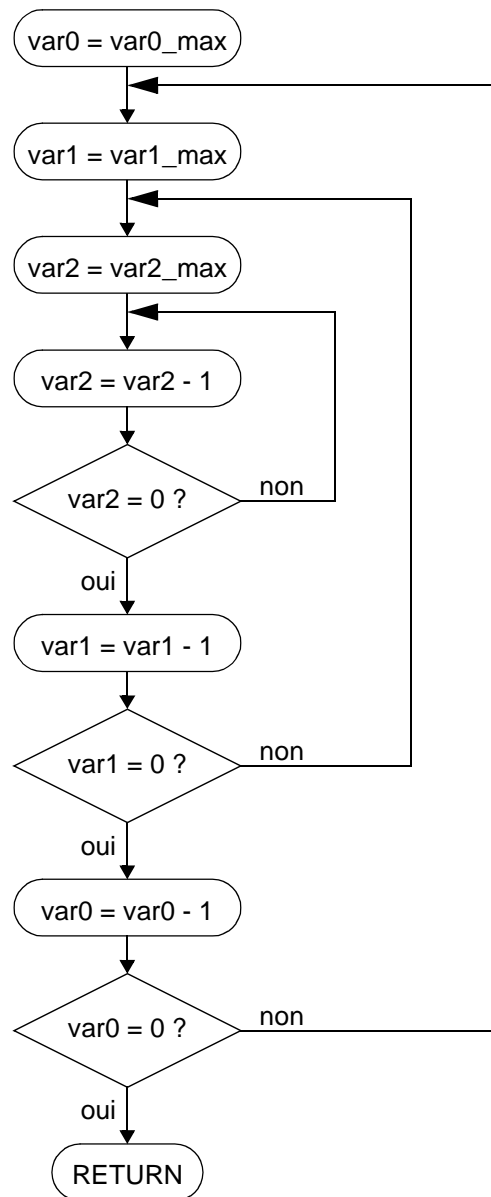
3.8 Réalisation d'une routine de temporisation software

L'exécution en continu du programme précédent ne permet pas de voir clignoter les LEDs, car la fréquence du microcontrôleur est trop grande. Pour cet exercice, il faudra donc écrire une routine de temporisation qui sera appelée avant de changer l'état du port de sortie.



La routine de temporisation ne servira qu'à «perdre du temps». Elle sera constituée de boucles imbriquées qui décrémenteront des variables (var0, var1, var2), à partir de leur valeur maximale, et jusqu'à ce qu'elles arrivent à 0.

sous-programme
«routine_tempo»



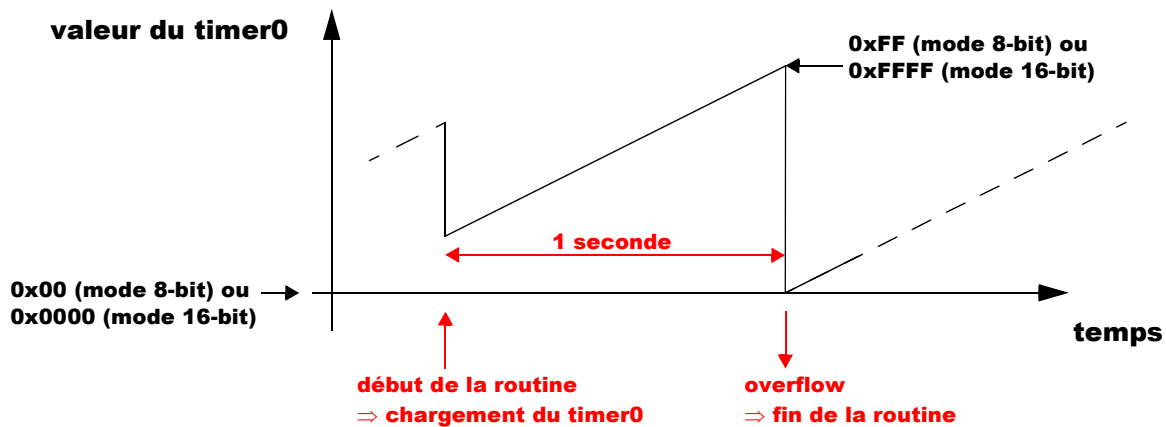
3.2 Réalisation d'une routine de temporisation hardware (utilisation d'un timer)

Pour cet exercice, l'objectif principal est le même que précédemment, et le programme principal gardera la même structure, mais la routine de temporisation se basera sur l'utilisation d'un timer pour fixer la routine de temporisation à **exactement 1 seconde** (à quelques µs près).

C'est le timer0 qui sera utilisé (cf. page 155 de la datasheet du PIC18F45K20). Il sera activé et configuré de manière à ce que sa période totale soit au moins égale à 1 seconde (registre T0CON).

Au début de la routine de temporisation, il faudra charger le timer à une valeur adéquate, puis attendre qu'il ait atteint sa valeur maximale (ou overflow). Cet événement peut être détecté en

testant le flag d'overflow TMR0IF dans le registre INTCON.



3.10 Utilisation des interruptions

La carte de prototypage comporte un bouton poussoir qui permet de forcer une entrée du microcontrôleur (RB0/INT0) à l'état bas lors d'un appui (cf. schéma électrique page 13/18 de la documentation de la carte).

Ce bouton sera utilisé pour mettre en marche ou arrêter le timer0 à chaque appui, en modifiant le bit TMR0ON dans le registre T0CON.

Comme cet événement peut arriver à n'importe quel moment, il sera traité comme une interruption.

La routine d'interruption devra à chaque appel modifier la valeur d'une variable «marche_arret», et selon la valeur de celle-ci arrêter ou mettre en marche le timer0.

Le programme principal devra configurer le microcontrôleur de manière à ce qu'une interruption soit déclenchée à chaque front descendant détecté sur la patte RB0 (interruption INT0, configurée par les bits INT0IE dans INTCON, et INTEDG0 dans INTCON2).

Remarque importante: pour que l'entrée RB0 soit lue comme une entrée logique, il faut mettre PBADEN à OFF dans les bits de configuration.

PBADEN OFF	PORTB A/D Enable bit	PORTB<4:0> pins are configured as digital I/O on Reset
---------------	----------------------	--

3.11 Utilisation du convertisseur analogique/numérique

Le but de cet exercice est d'utiliser la barrette de LEDs comme un «bar graph», pour allumer plus ou moins de LEDs selon la position du potentiomètre monté sur la carte de prototypage.

Ce dernier permet d'appliquer sur l'entrée RA0/AN0 du microcontrôleur une tension comprise entre 0 et VDD. Il faudra donc utiliser ici le convertisseur analogique-numérique pour mesurer la tension d'entrée, puis écrire en fonction de celle-ci une valeur particulière sur le port de sortie D (b'00000001', ou b'00000011', ou b'00000111', ..., ou b'01111111', ou b'11111111').

- Les sorties RD<7:0> et l'entrée RA0 seront respectivement configurées en sorties et en entrée (registres TRISD et TRISA).
- Le bit n° 0 du registre ANSEL devra être mis à 1 pour désactiver le buffer d'entrée

numérique de la patte RA0, car elle sera utilisée comme entrée analogique.

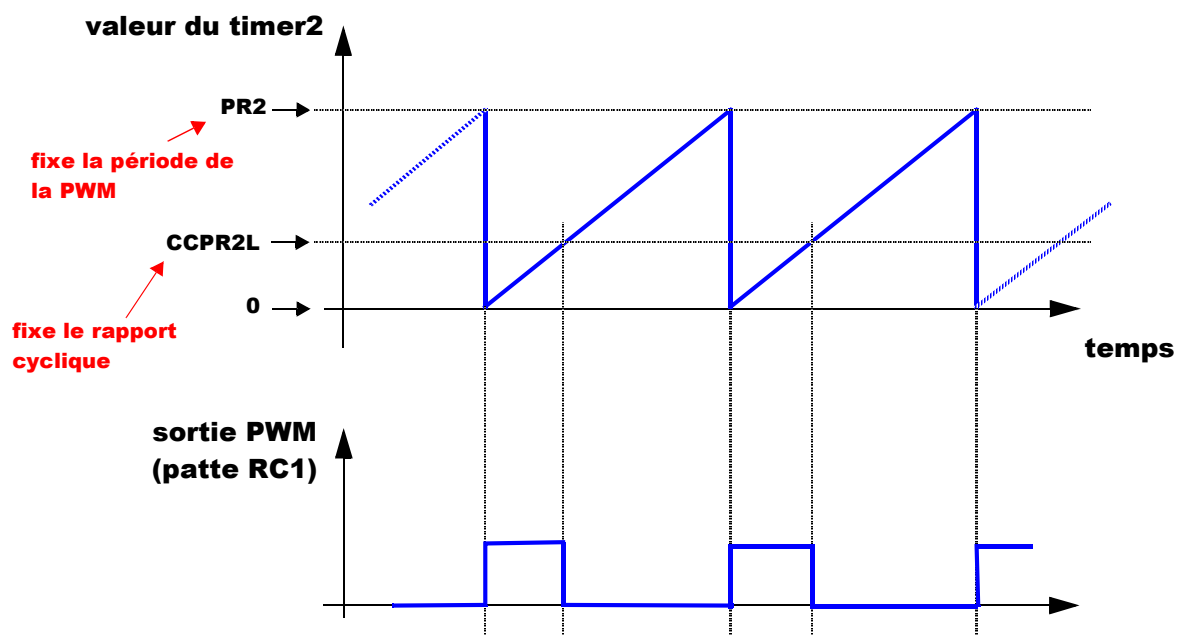
- Le registre ADCON1 sera configuré de manière à ce que les tensions de référence de l'ADC soient V_{ss} (0V) et V_{DD} ($\approx 3V$).
- Le registre ADCON2 sera configuré de manière à ce que le résultat de la conversion soit «left-justified», que la fréquence de fonctionnement du convertisseur f_{AD} soit égale à $f_{OSC}/4$, soit 1 MHz, et que le temps d'acquisition soit égal à 8 TAD.
- Le registre ADCON0 sera utilisé pour sélectionner l'entrée analogique RA0/AN0, mettre en route le convertisseur, lancer une nouvelle conversion, et tester la fin de la conversion en scrutant (méthode dite de «polling») le bit $\overline{GO/DONE}$.

3.12 Génération d'un signal PWM

Ce dernier exercice a pour but de générer un signal de rapport cyclique variable (PWM), compris entre 0 et 100 %, selon la position du potentiomètre.

On utilise pour cela la fonction PWM du timer2 (cf. page 149 de la datasheet du PIC18F45K20).

Le fonctionnement simplifié est le suivant:



Le programme principal devra:

- configurer le convertisseur analogique/numérique comme précédemment,
- configurer le timer2 pour le mode PWM (registres T2CON, CCP2CON, PR2),
- convertir en boucle l'entrée analogique RA0/AN0 et modifier en conséquence le registre CCPR2L afin que le rapport cyclique varie de 0 à 100 % lorsque le potentiomètre est tourné.

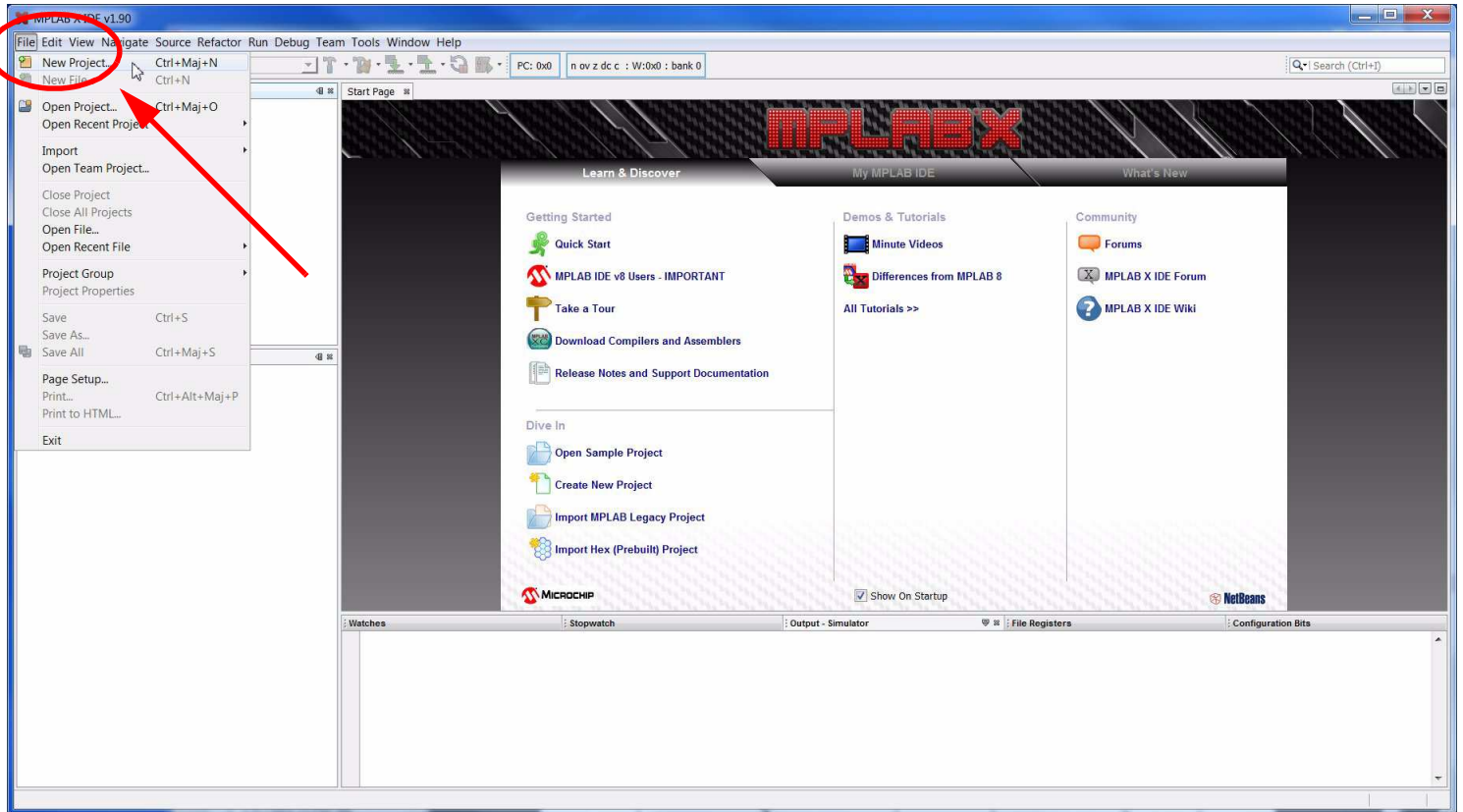
La sortie PWM sera observée à l'oscilloscope, sur la patte RC1.

ANNEXES

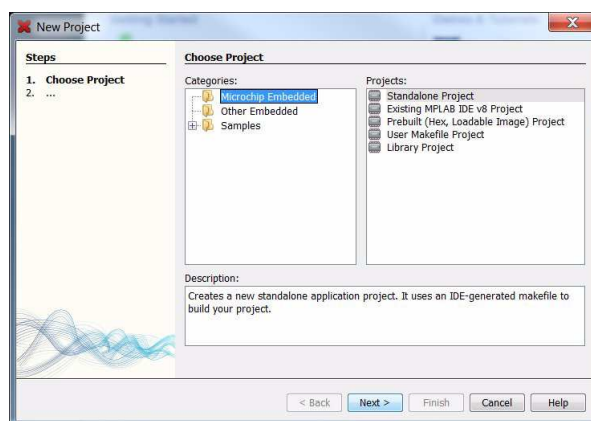
1 - Création d'un projet sous MPLABX

En démarrant MPLABX, commencer par créer un nouveau projet.

File → New Project...

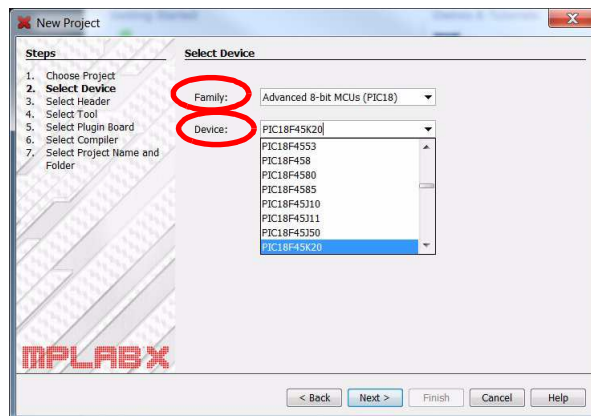


Dans l'étape **Choose Project**, garder les choix par défaut
(Categories: Microchip Embedded Projects: Standalone Project)
puis sélectionner **Next >**.

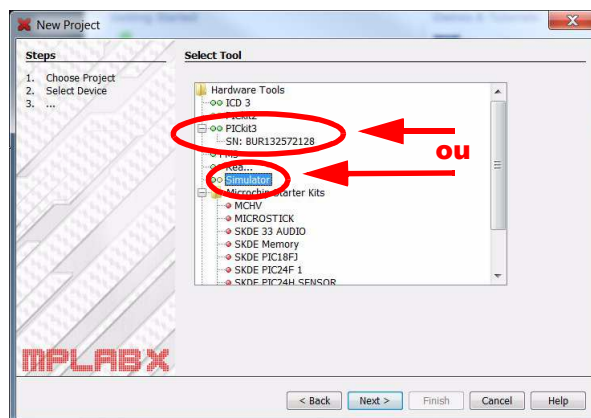


Dans l'étape **Select Device**, sélectionner
Family: **Advanced 8-bit MCUs (PIC18)**

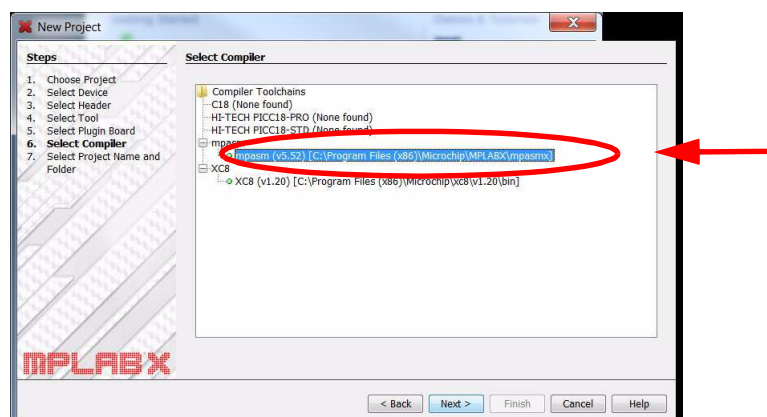
Device: **PIC18F45K20** (ou toute autre référence, en fonction du composant qui sera utilisé)
puis sélectionner **Next >**.



Dans l'étape **Select Tool**, sélectionner
Simulator pour débbugger un programme uniquement en simulation
ou
PICKit3 pour débbugger et faire tourner un programme sur un microcontrôleur réel
(une carte de prototypage doit dans ce cas être connectée au PC)
puis sélectionner **Next >**.

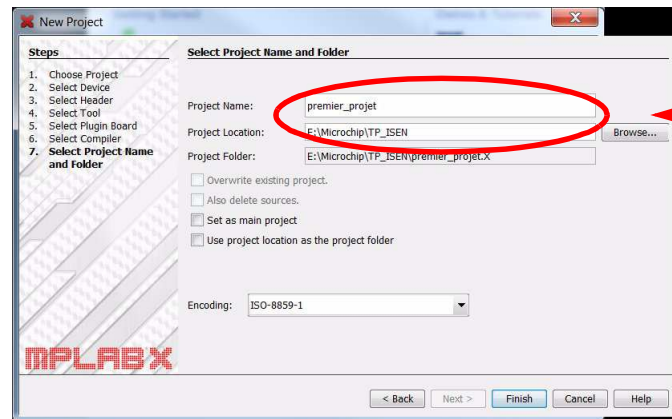


Dans l'étape **Select Compiler**, sélectionner
mpasm (v5.52)
(ce qui est la seule possibilité si aucun compilateur C n'a été installé pour MPLABX)
puis sélectionner **Next >**.



Dans l'étape **Select Project Name and Folder**, indiquer le nom du projet ainsi que son emplacement (un sous-répertoire xxxxxx.X sera automatiquement créé) puis sélectionner **Finish**.

Remarque importante: pour les noms de projets, répertoires ou fichiers, n'utiliser que les caractères alpha-numériques (pas d'espace, d'accent, de &, #, etc.).

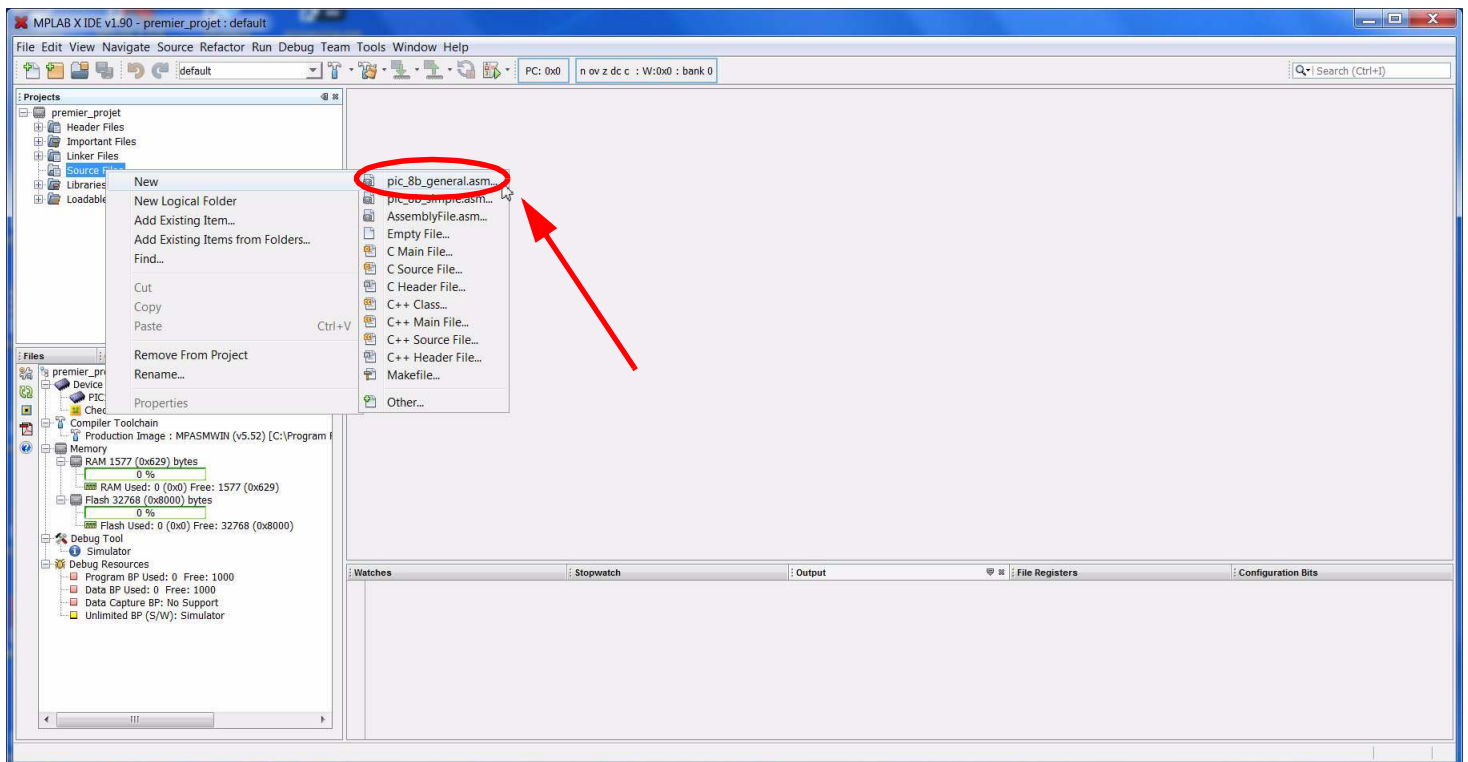


2 - Ajout d'un fichier source (programme assembleur) à un projet

L'onglet **Projects** de la fenêtre principale indique le projet actuellement ouvert: il faut maintenant lui associer un fichier qui permette d'écrire le programme à tester (fichier source).

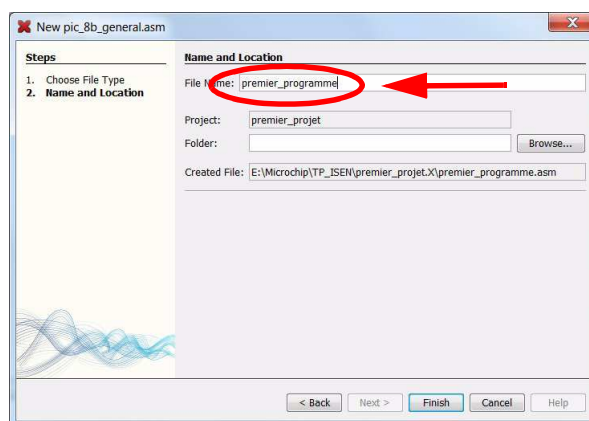
Par un click-droit sur **Source Files**, sélectionner.

New → pic_8b_general.asm...



Dans la fenêtre suivante, spécifier un nom de fichier, puis sélectionner **Finish**.

Remarque: il est théoriquement possible de garder le nom par défaut, mais dans ce cas tous les fichiers source auront le même nom *pic_8b_general.asm*, ce qui est une mauvaise pratique pour le suivi ultérieur des projets.



Le fichier créé par défaut contient un programme basique qui écrit la valeur 0x55 dans le registre de travail W (*MOVLW 0x55*), puis boucle indéfiniment sur la même instruction (*GOTO \$*).

Il «ne reste plus» qu'à éditer le fichier pour écrire le programme souhaité.

Remarques importantes:

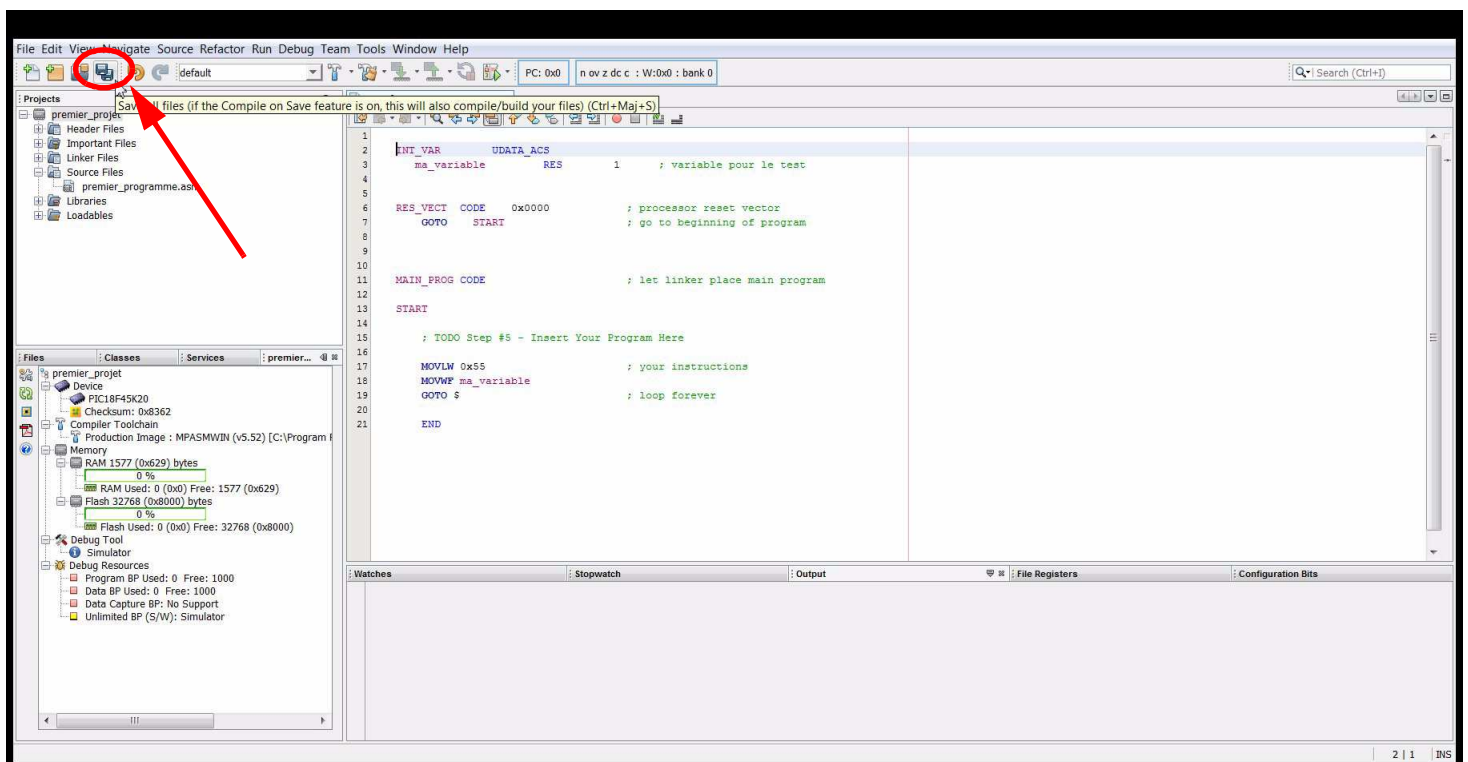
- Au début du programme, inclure la ligne

```
#include "p18F45K20.inc"
```

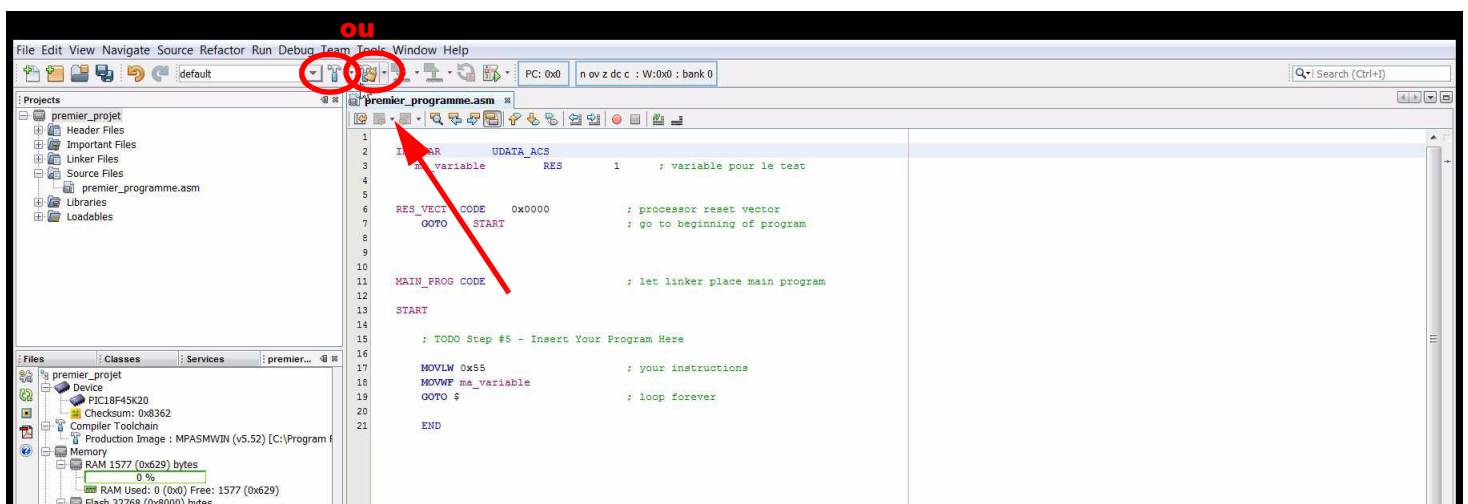
(Ceci permet au compilateur d'interpréter les noms de registres du microcontrôleur utilisé, tels que WREG, STATUS, PORTA, etc.)

- Changer le nom START utilisé par défaut comme label du programme principal (car ce nom est déjà employé pour une autre raison dans le fichier p18F45K20.inc). Vous pouvez utiliser n'importe quel autre nom valide (START2, debut, toto, etc.)

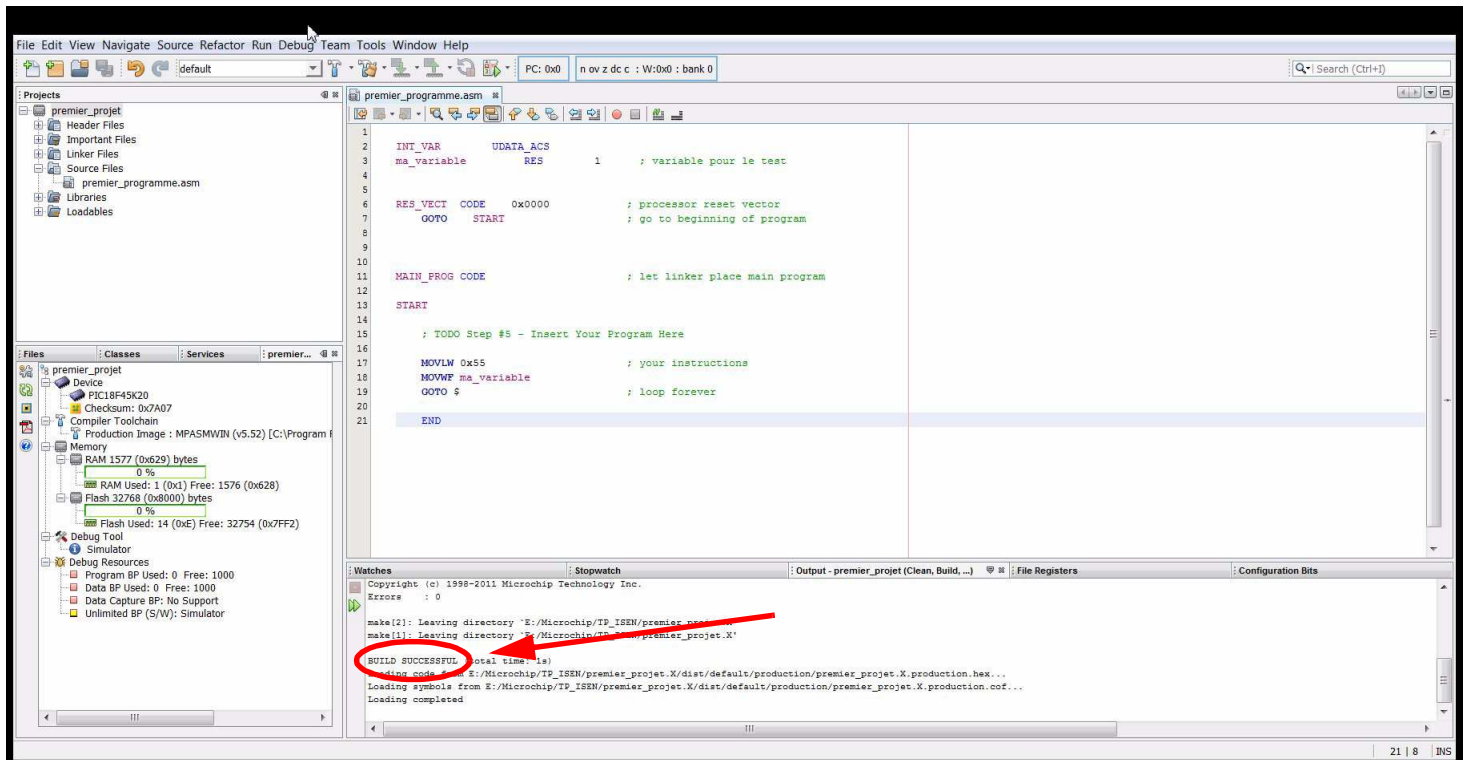
Après édition du fichier, le sauvegarder (File → Save ou Save All)



puis le compiler pour traduire le programme en langage machine (Run → Build Project ou Clean and Build Project).



S'il n'y a pas d'erreur de syntaxe dans le programme, le message BUILD SUCCESSFUL doit apparaître dans la fenêtre «Output».



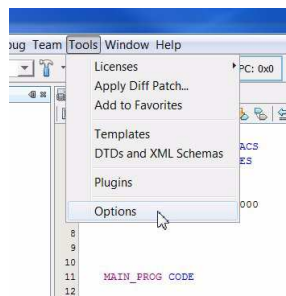
3 - Utilisation du «debugger» - Simulation d'un programme

3.1: lancement du debugger

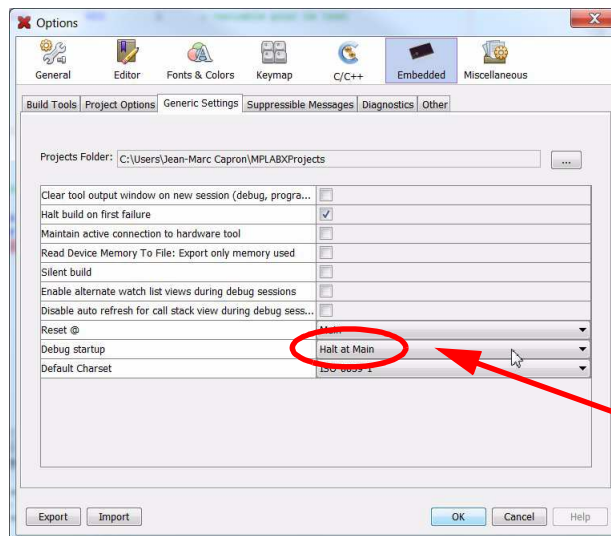
Après compilation du programme, il est possible de le simuler et de visualiser le contenu des divers registres du microcontrôleur en lançant le debugger.

Dans un premier temps, si l'on souhaite lancer la simulation en mode pas-à-pas pour étudier le comportement du programme, il faut éviter que le simulateur n'exécute tout le programme au démarrage. Une option de MPALB doit pour cela être modifiée.

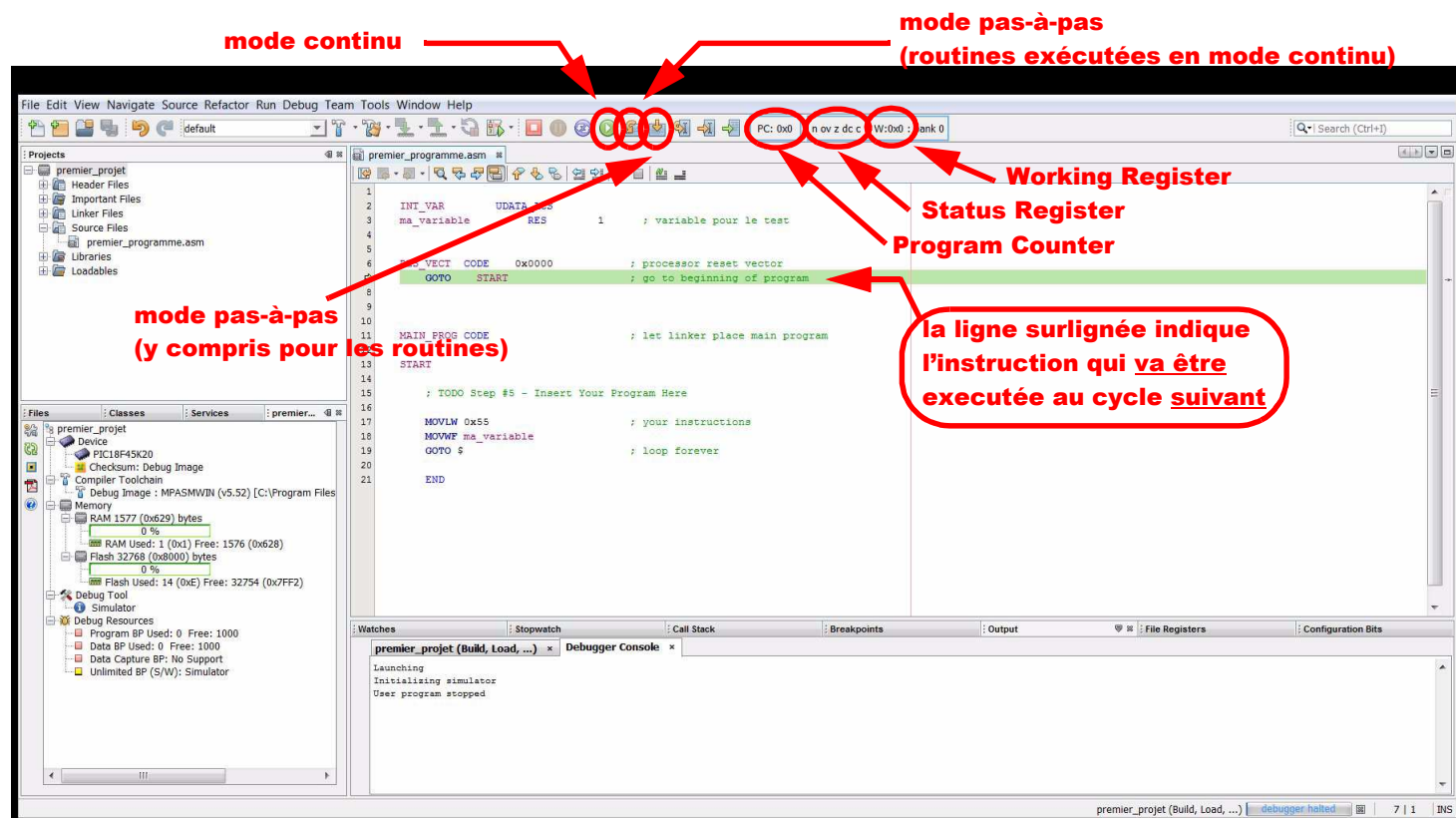
Tools → Options



Embedded → Generic Settings → Debug Startup: Halt at Main.



La fenêtre principale permet ensuite de tester le programme en mode pas-à-pas (Step Into) ou de lancer son exécution (Continue), tout en visualisant le contenu des divers registres.

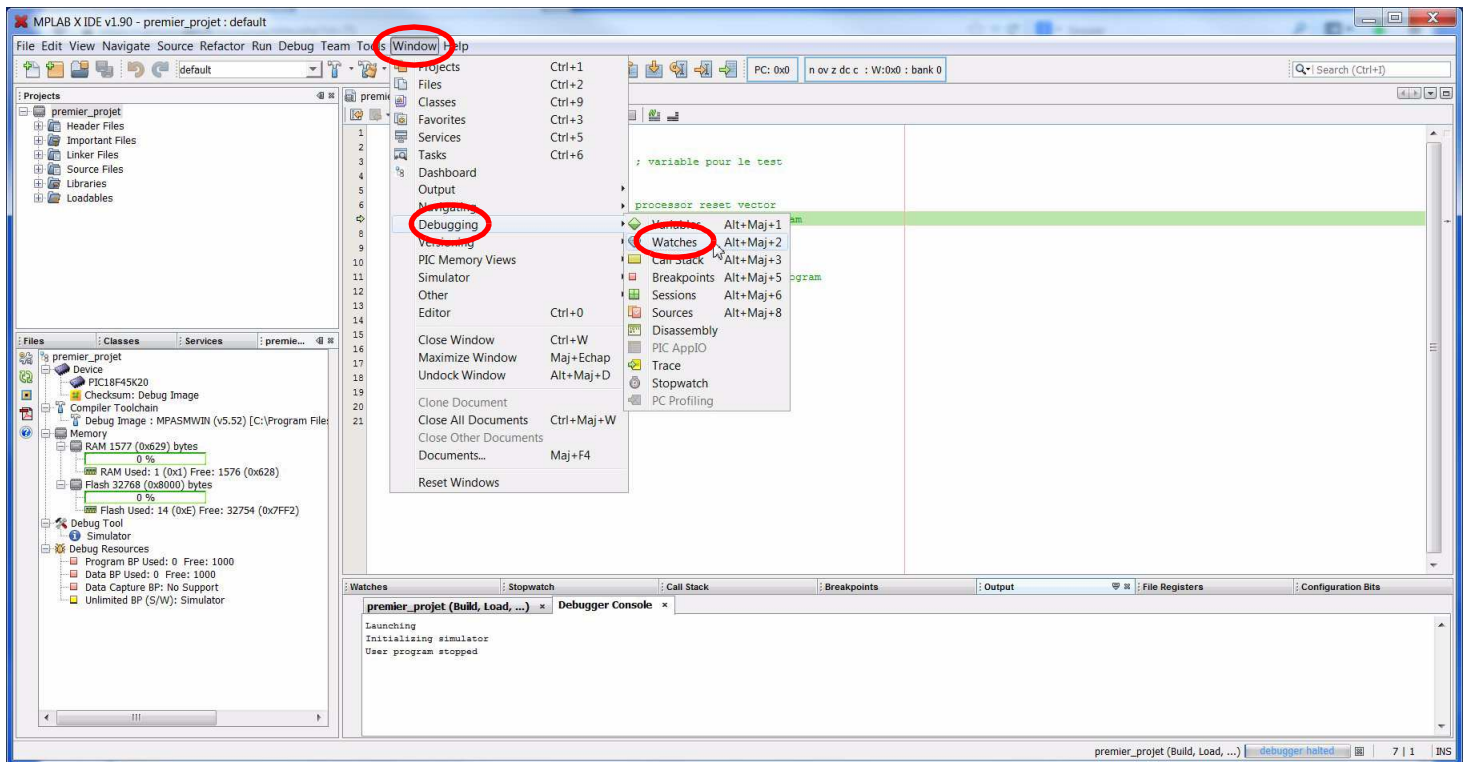


La barre d'outils en haut de la fenêtre principale permet de visualiser:

- le compteur ordinal ou Program Counter (adresse de l'instruction qui sera exécutée au cycle CPU suivant),
- le registre d'état ou Status Register (contient les flags qui renseignent sur le résultat de la dernière opération - une lettre minuscule signifie un flag à 0, une lettre majuscule signifie un flag à 1),
- l'accumulateur ou Working Register,
- le numéro de banque courante.

3.2 Visualisation du contenu des registres (contenu de la RAM)

Ouvrir la fenêtre «Watches» par la commande **Window** → **Debugging** → **Watches**



Un click-droit dans la fenêtre «Watches» permet d'ouvrir la fenêtre «New Watch».

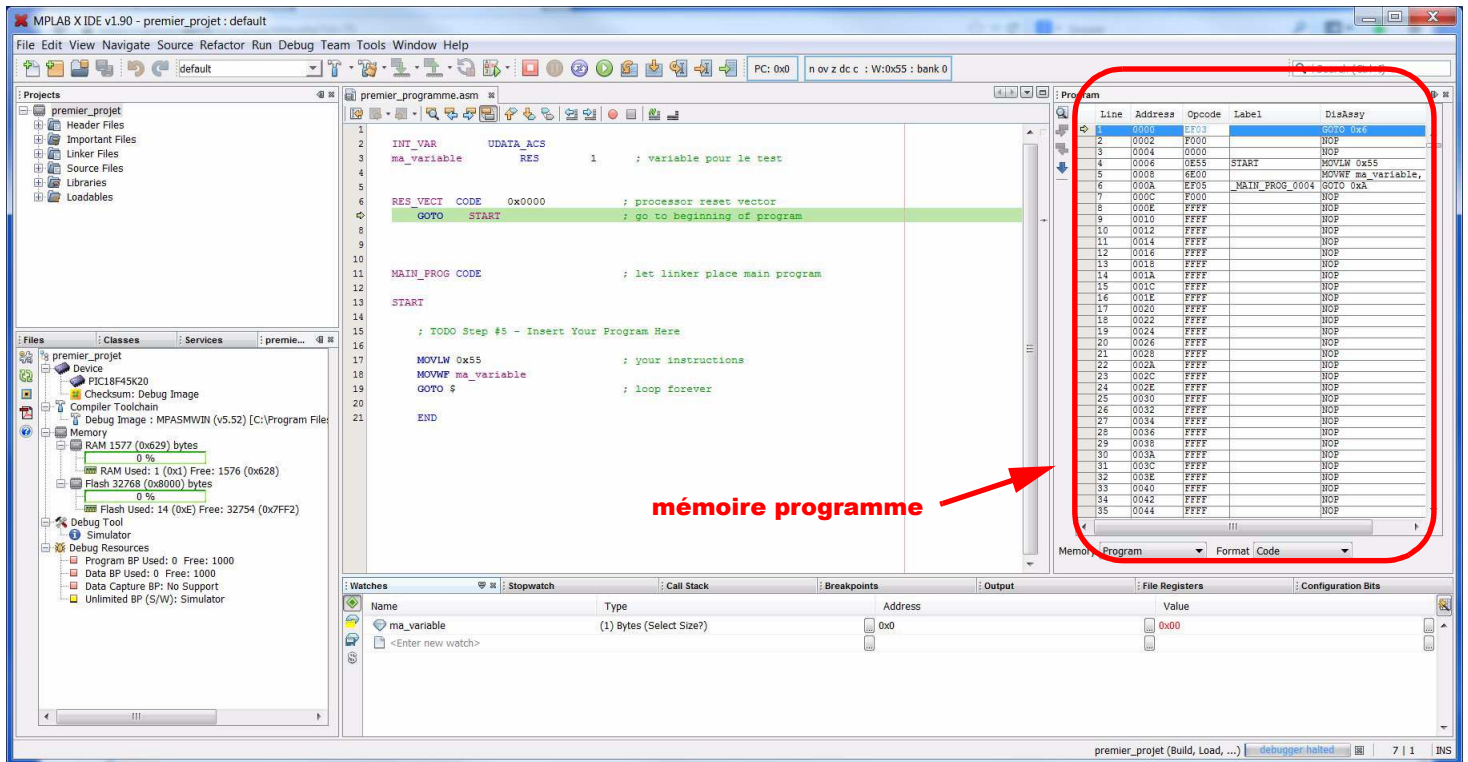


«Global Symbols» permet de visualiser les variables définies par l'utilisateur.

«SFR» (Special Function Register) permet de visualiser les registres spécifiques au microcontrôleur, qui sont liés aux périphériques et au matériel (ports d'entrée/sortie, timers, ADC, etc.). (Se reporter à la datasheet du composant pour connaître la fonction de ces registres.)

3.3 Visualisation de la mémoire programme (contenu de la Flash)

Ouvrir la fenêtre «Program» par la commande
Window → PIC Memory Views → Program Memory



La fenêtre donne les indications suivantes:

- Line: numéro d'instruction,
- Address: emplacement de l'instruction dans la mémoire programme,
- Opcode: instruction en langage machine (sur 16 bits),
- Label: label éventuel donné à une instruction par l'utilisateur,
- DisAssy (Disassembly): instruction en langage assembleur.

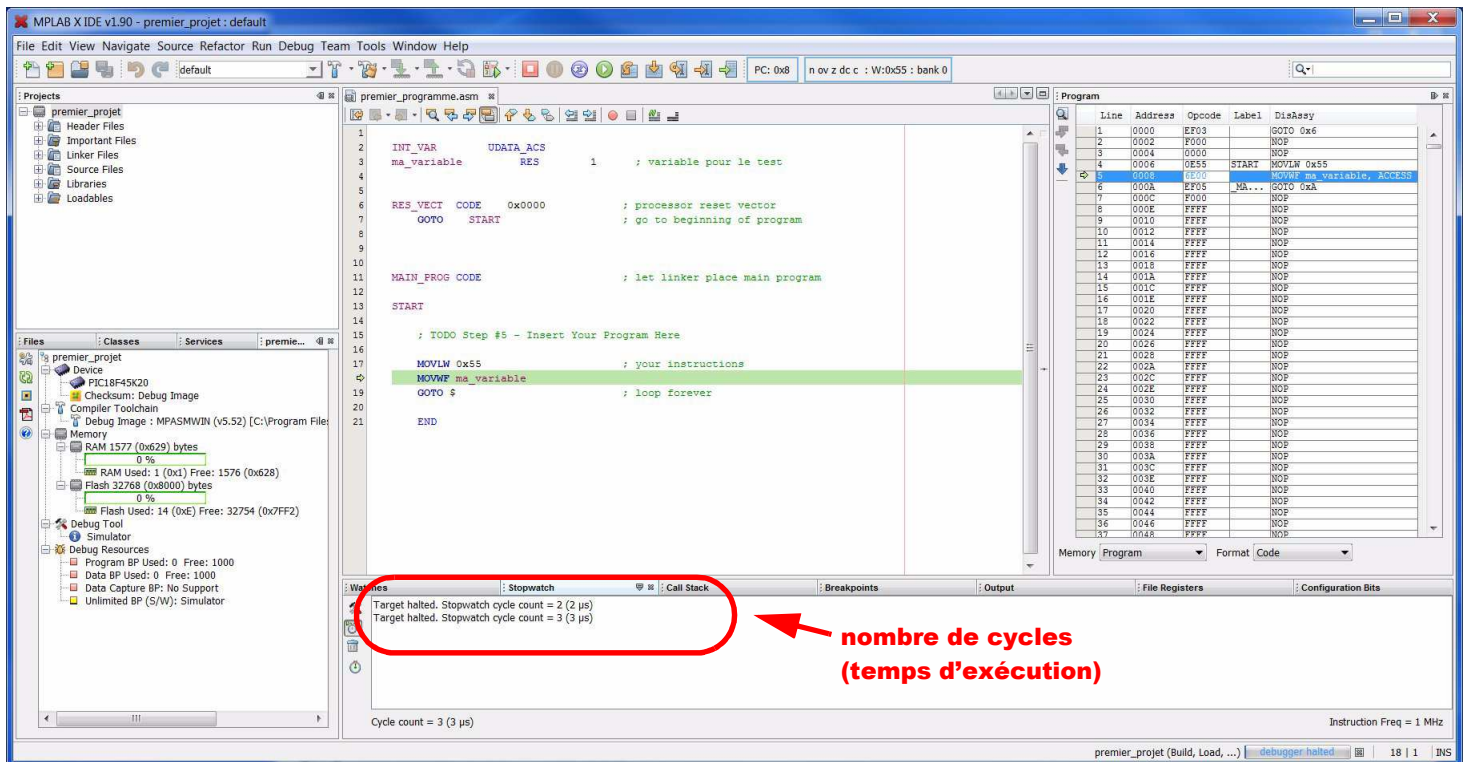
3.4 Visualisation du nombre de cycles (temps d'exécution)


Il est possible de visualiser le nombre de cycles nécessaire pour exécuter une instruction, une routine, ou une partie de programme.

Ouvrir dans un premier temps la fenêtre «Stopwatch»:

Window → Debugging → Stopwatch

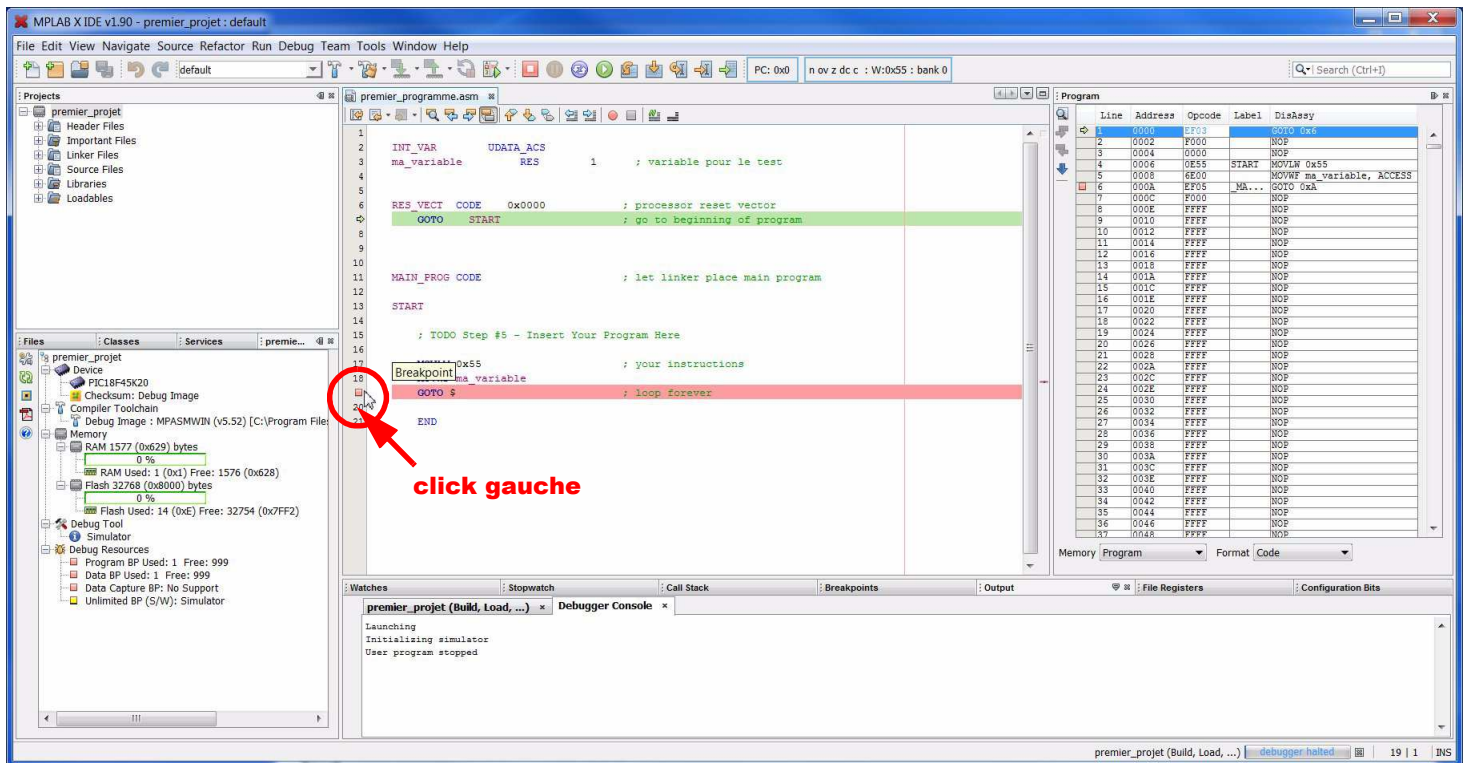
En lançant la simulation, que ce soit en mode pas-à-pas ou en mode continu, le nombre de cycles s'affiche dans la fenêtre «Stopwatch».



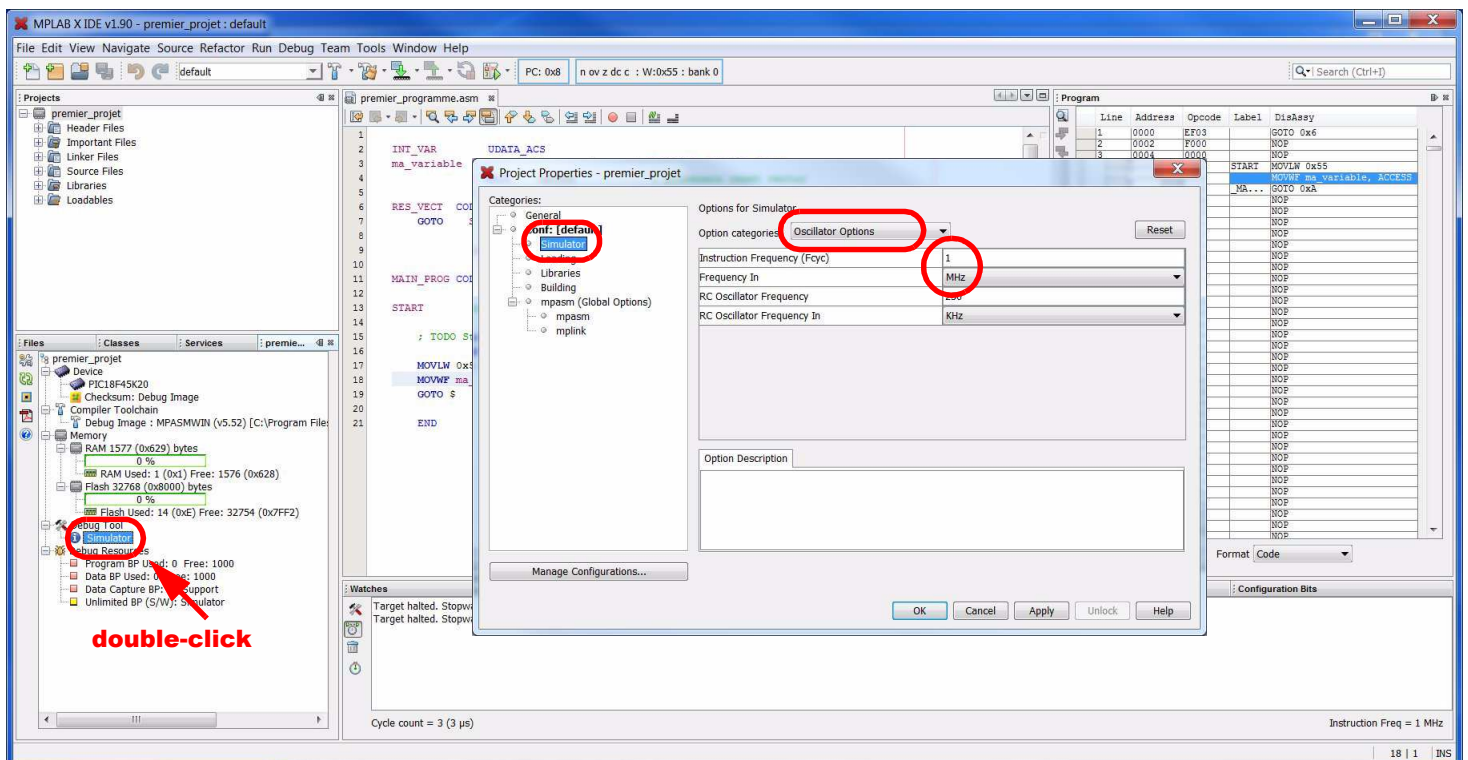
Remarque: la simulation en mode pas-à-pas peut rapidement s'avérer fastidieuse lorsque les programmes sont conséquents. Il peut dans ce cas être utile de simuler directement toute une partie de programme, jusqu'à une instruction donnée. Il suffit pour cela de placer un point d'arrêt (**breakpoint**) sur l'instruction concernée, en cliquant dans la marge, puis de lancer la simulation en mode continu. 

L'instruction où est placé le point d'arrêt est surlignée en rouge.

Un nouveau click dans la marge supprime de la même manière le point d'arrêt.



Remarque: le simulateur prend par défaut 1 μ s comme temps d'exécution d'une instruction (soit une fréquence d'oscillateur de 4 MHz).
Il est possible de modifier ce paramètre de la manière suivante:



4 - Utilisation du «debugger» - Exécution d'un programme sur le composant

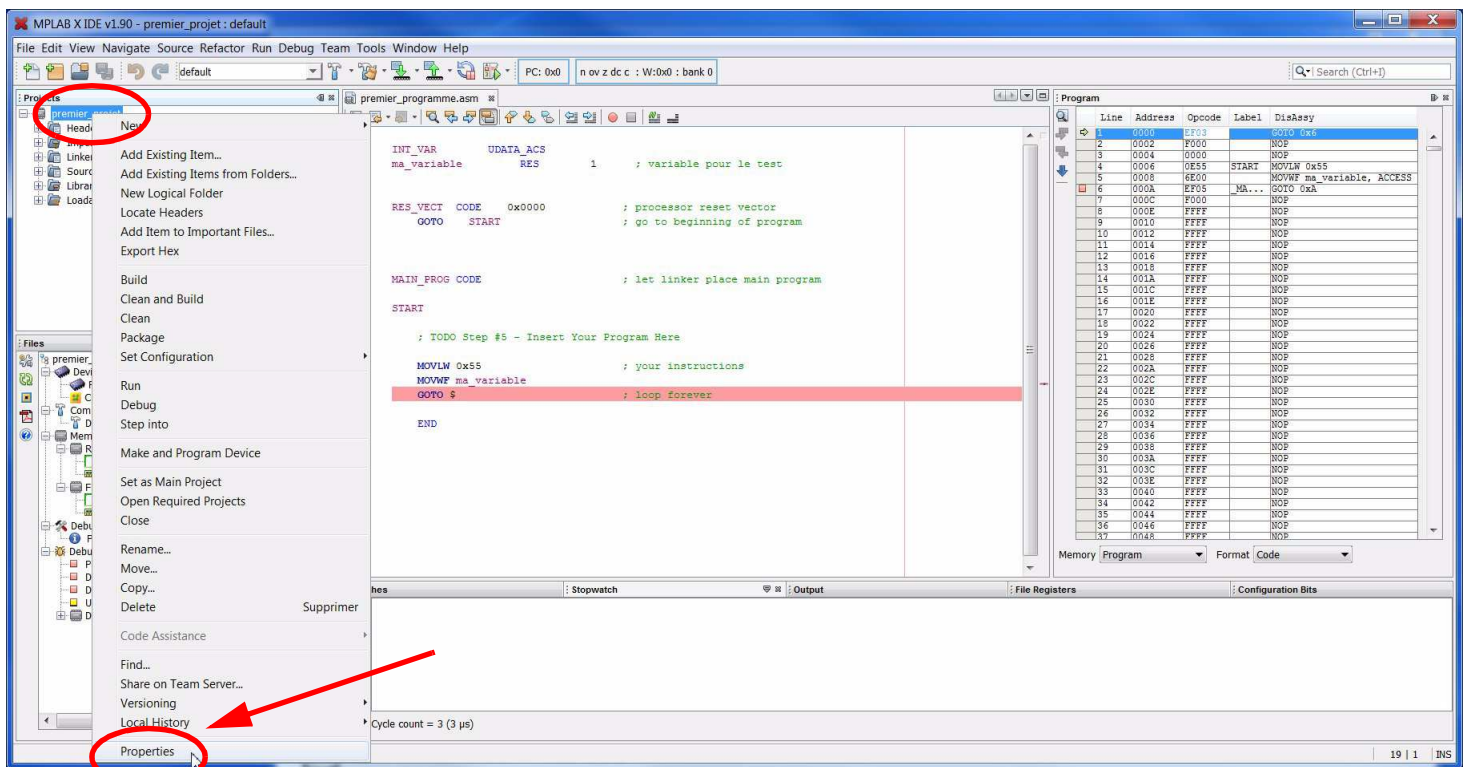
4.1: Utilisation du PICKit3

Pour exécuter le programme sur le composant réel, et non pas en simulation pure, un outil de debug (par exemple PICKit3) doit être connecté entre le PC et le microcontrôleur.

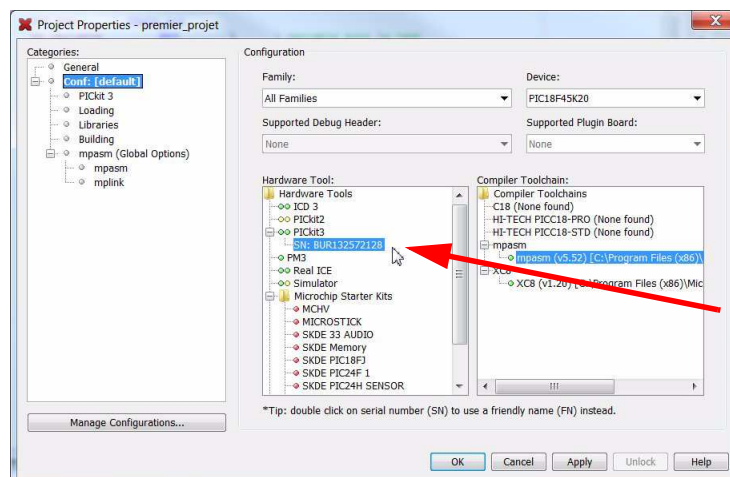
Si c'est le cas, lors de la création du projet (cf. annexe 1) il faut choisir comme outil de débog PICKit3 au lieu de Simulator.

Remarque: ceci peut toujours être modifié par la suite dans les propriétés du projet.

Click-droit sur le nom du projet, dans la fenêtre Projects → **Properties**



puis **Conf:[default] → Hardware Tool**

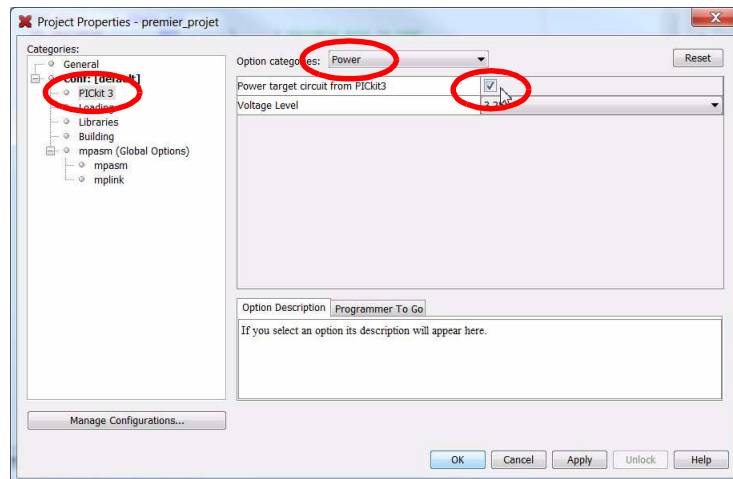


3.2: Alimentation de la carte de prototypage par le PICkit3

Lorsque la carte de prototypage ne dispose pas d'une source d'alimentation dédiée, il est possible de l'alimenter à partir du PICkit3, **à condition qu'elle consomme moins de 30 mA**.

Pour utiliser cette option, éditer les propriétés du projet (cf. page précédente).

PICkit3 → Power → Power target circuit from PICkit3

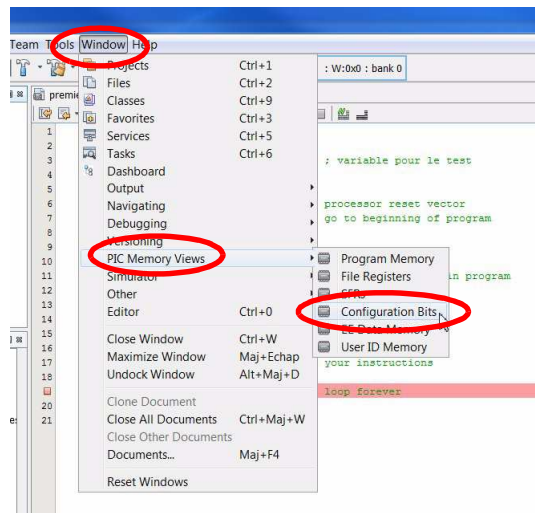


3.3: Edition des bits de configuration (Configuration Bits)

Avant de lancer le debugger sur la carte de prototypage, il est nécessaire d'éditer les bits de configuration qui vont spécifier les options matérielles du microcontrôleur.

Ouvrir la fenêtre «Configuration Bits».

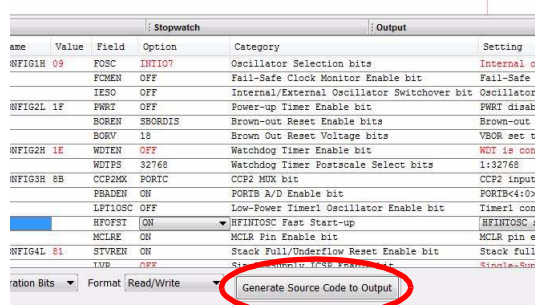
Window → PIC Memory Views → Configuration Bits



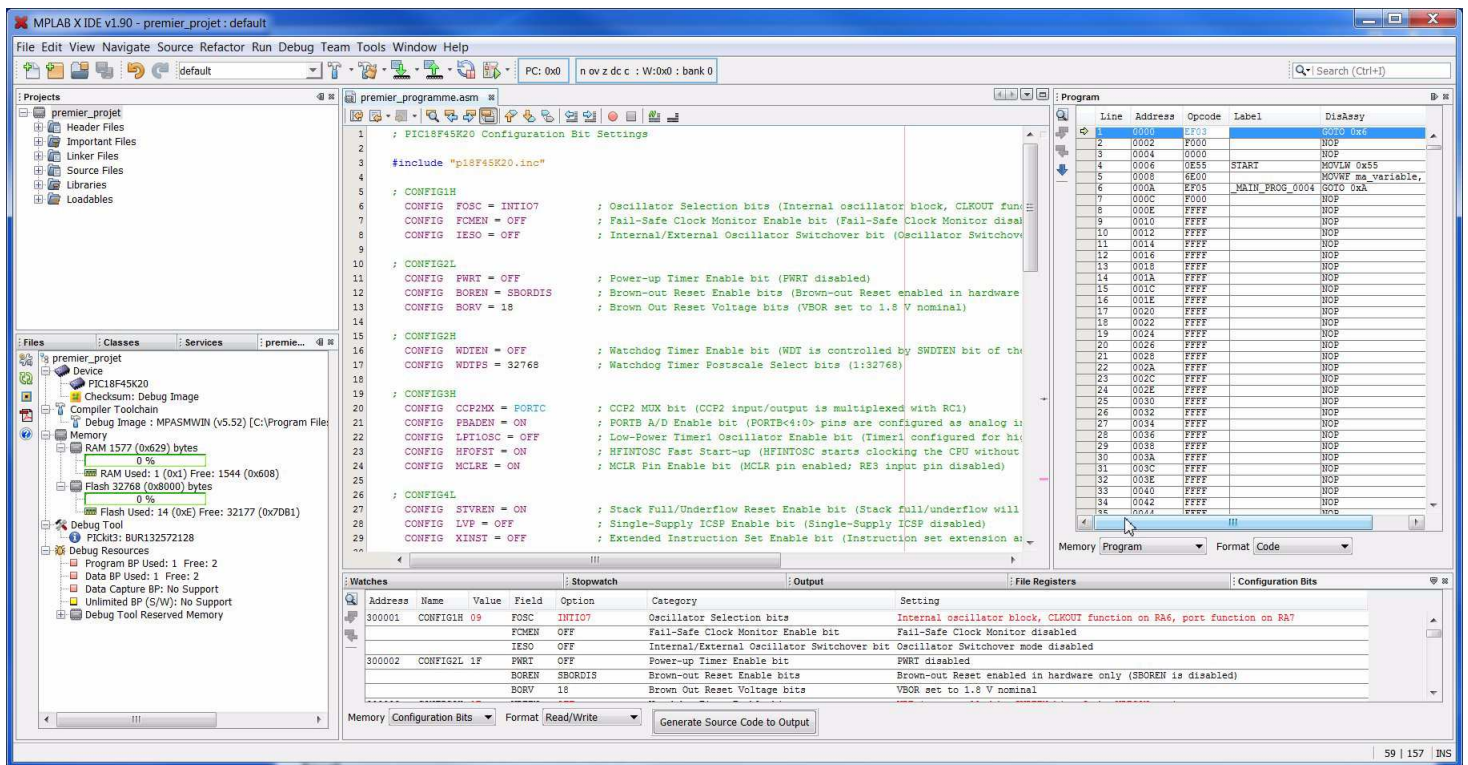
Dans la fenêtre «Configuration Bits», un certain nombre de bits de configuration doivent être modifiés:

FOSC	INTIO7	Oscillator Selection bits	Internal oscillator block, CLKOUT function on RA6, port funct
WDTEN	OFF	Watchdog Timer Enable bit	WDT is controlled by SWDTEN bit of the WDTCN register
LVP	OFF	Single-Supply ICSP Enable bit	Single-Supply ICSP disabled

Sélectionner ensuite «Generate Source Code to Output»



Copier-coller le code ainsi généré au début du programme.



Le debugger peut ensuite être lancé comme précédemment, mais le programme sera réellement exécuté sur le microcontrôleur, et tous les registres observés dans l'IDE (données ou programme) refléteront le contenu réel du microcontrôleur.