Tutoriel / aide mémoire

MPLAB IDE 7.40

C30

CREMMEL Marcel

Lycée Louis Couffignal
STRASBOURG

MICROCHIP

Table des matières

Tutoriel / aide mémoire MPLAB C30 sur dsPIC

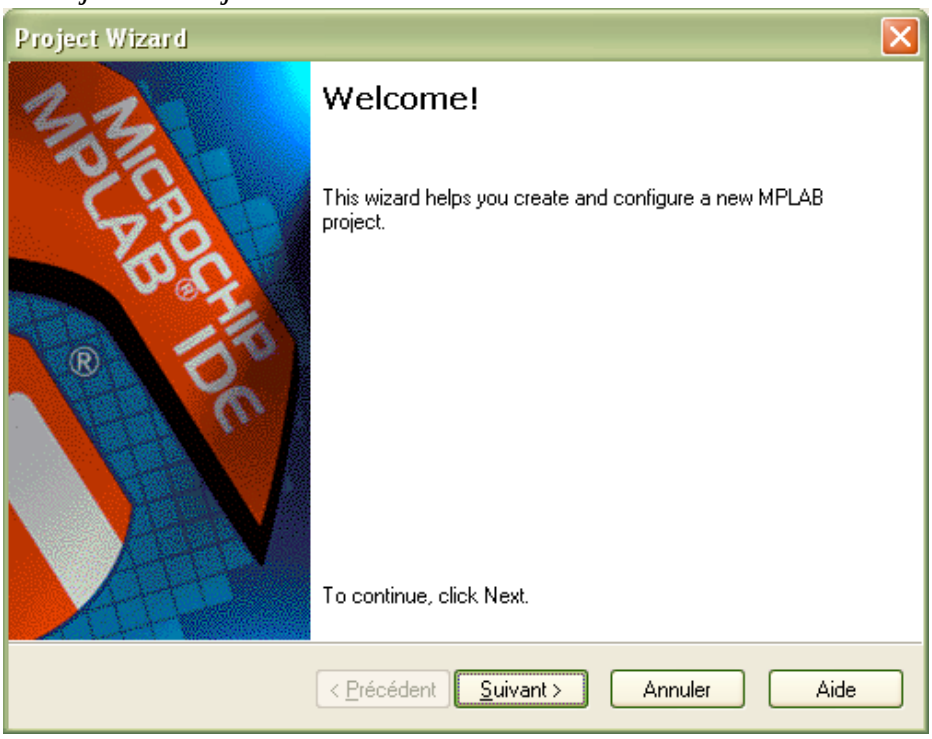
L'environnement de développement (IDE) MPLAB regroupe tous les outils nécessaires à la mise au point d'une application avec un cœur de microcontroleur dsPIC, entre autres, de MICROCHIP :

- éditeur de texte interactif
- compilateur C (et assembleur)
- simulateur
- debugger si on dispose de l'équipement nécessaire

*Ce document est un "tutorial" qui permet de **démarrer un nouveau projet**, le **compiler** et le **simuler** en présentant les principales commandes dans ces 3 phases de développement.*

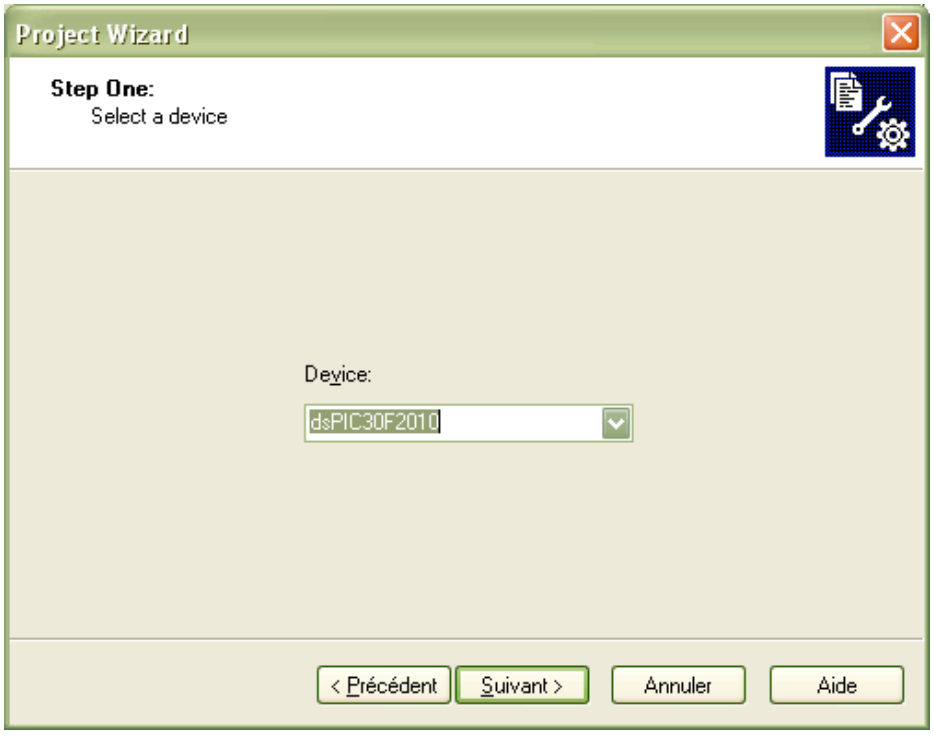
1. Démarrer un nouveau projet

1. Project → Project Wizard …



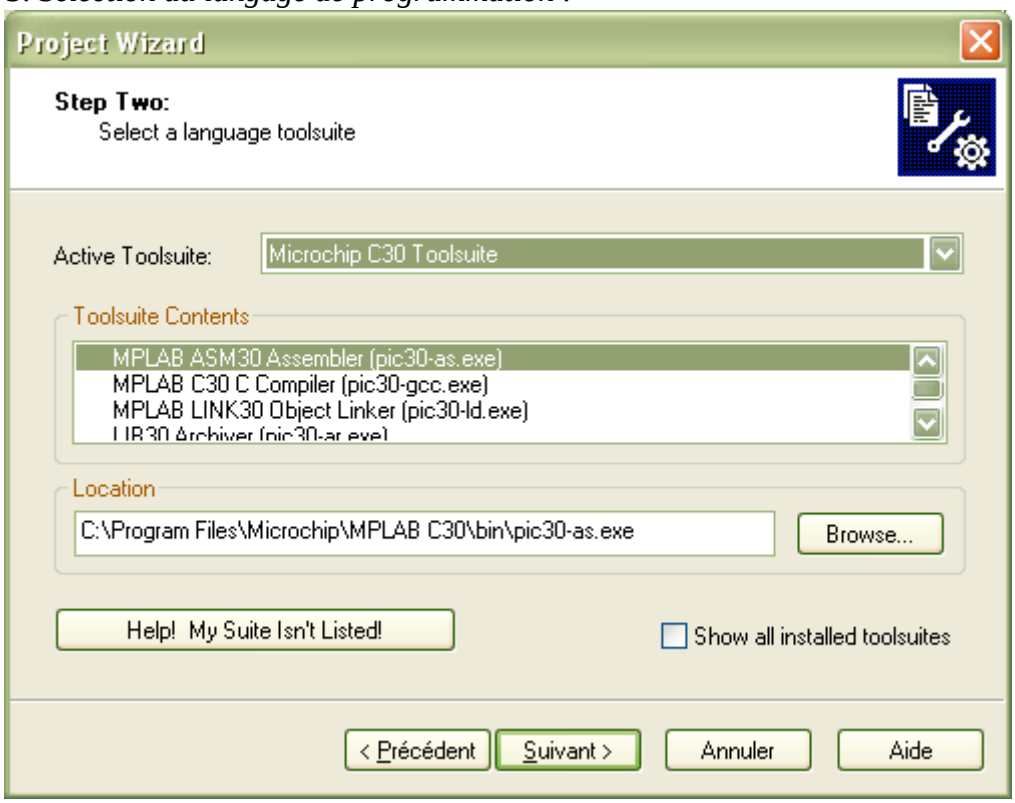
Rien de particulier dans cette phase : cliquer sur "Suivant"

2. Ecran suivant :



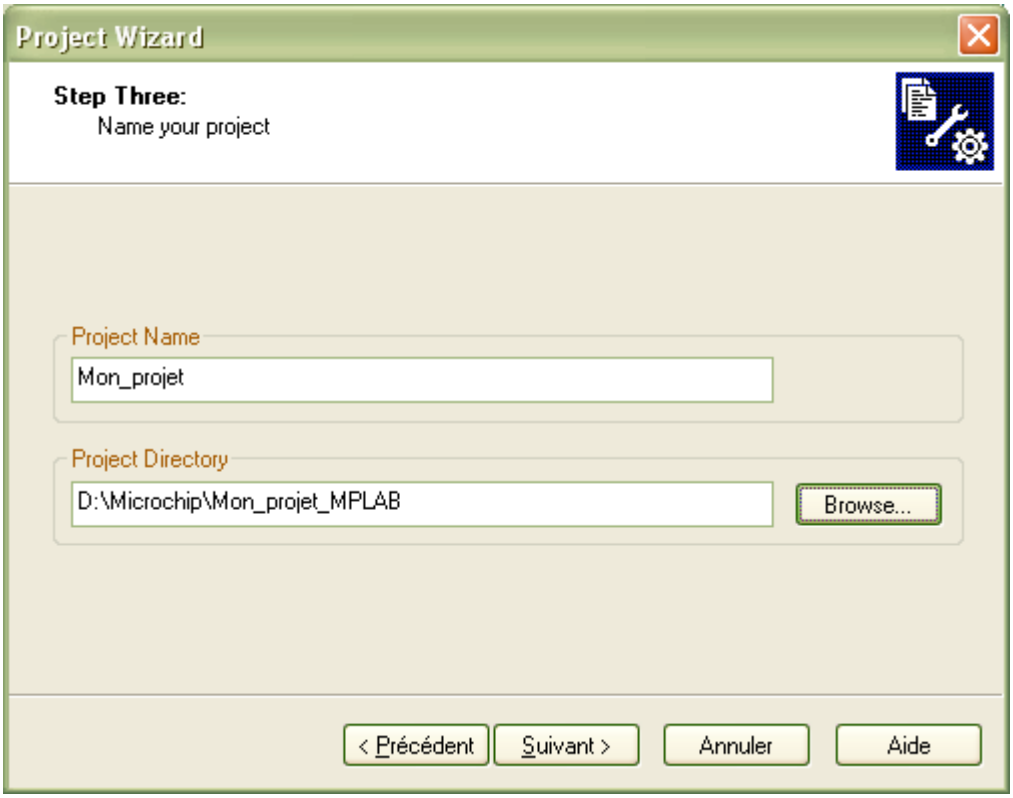
Choisir la cible dans la liste proposée.

3. Sélection du langage de programmation :



Choisir "Microchip C30 Toolsuite" dans "Active Toolsuite"
Vérifier que chaque outil est bien localisé sur le PC : cliquer chaque outil et vérifier son emplacement.

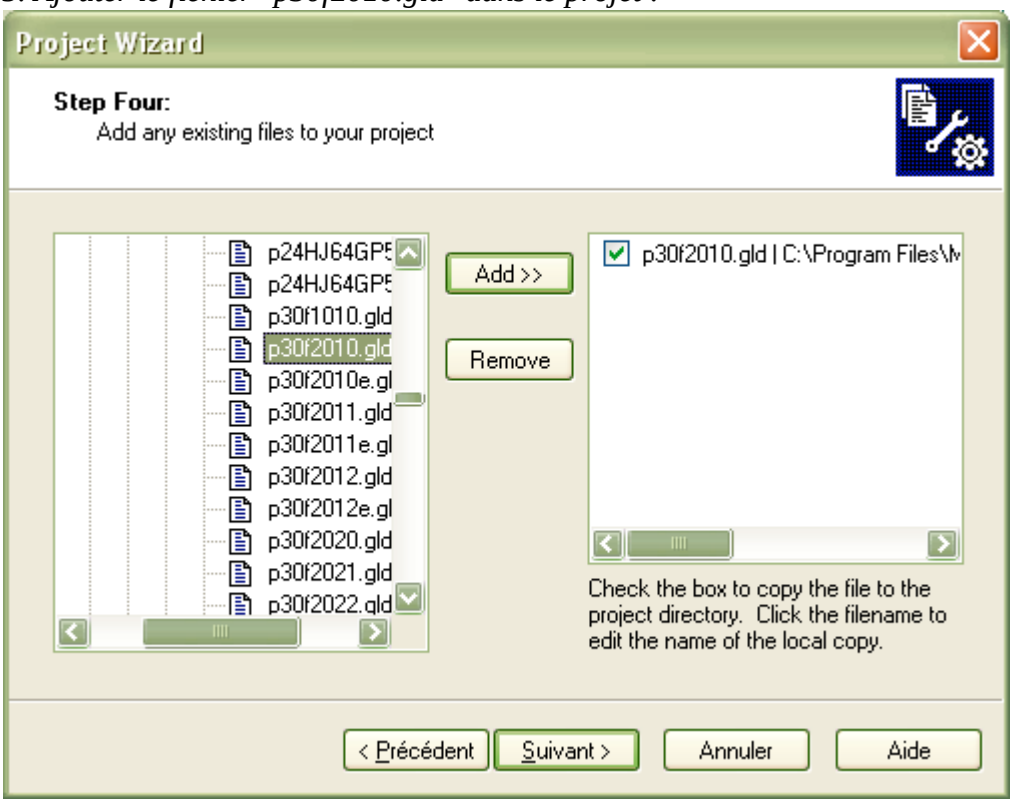
4. Nom et emplacement du projet :



Si le dossier n'existe pas encore, il peut être créé dans cette étape. Le placer dans le répertoire D:\Microchip

Le nom du projet peut être différent de celui du dossier.
Éviter de placer plusieurs projets dans un même dossier.

5. Ajouter le fichier "p30f2010.gld" dans le projet :

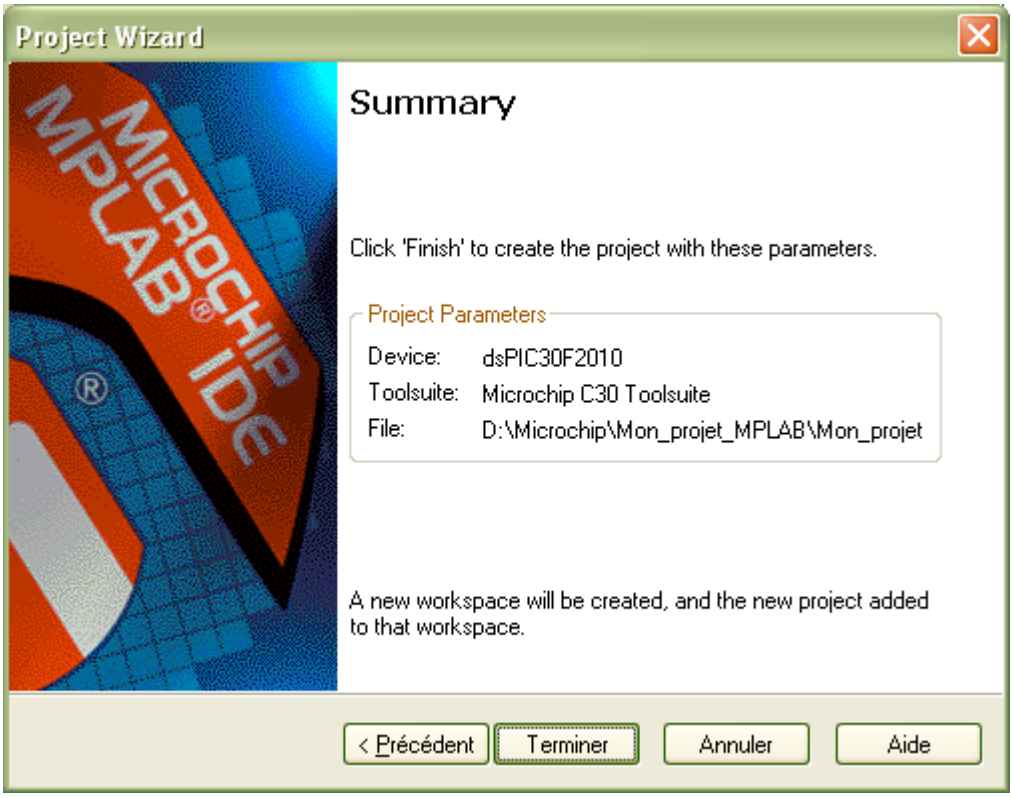


Ce fichier texte informe l'éditeur de lien (linker) du plan mémoire du microcontroleur.

Le fichier se trouve dans le dossier C:\Program Files\Microchip\MPLAB C30\support\gld

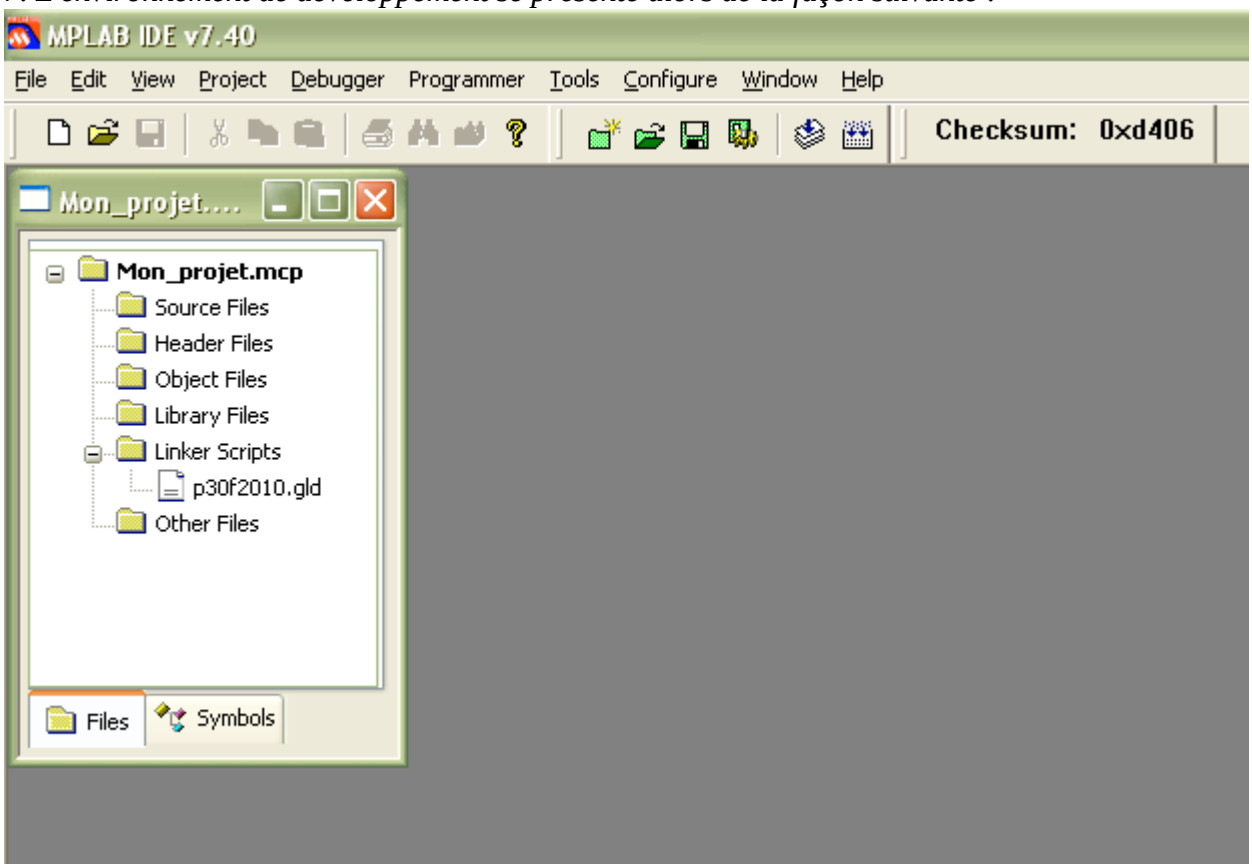
Cocher la case pour copier ce fichier dans le dossier.

6. Résumé des paramètres du projet :

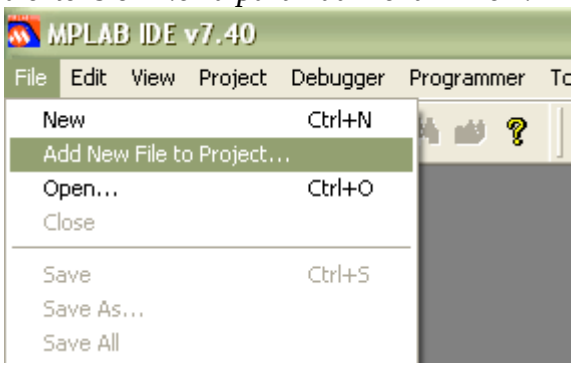


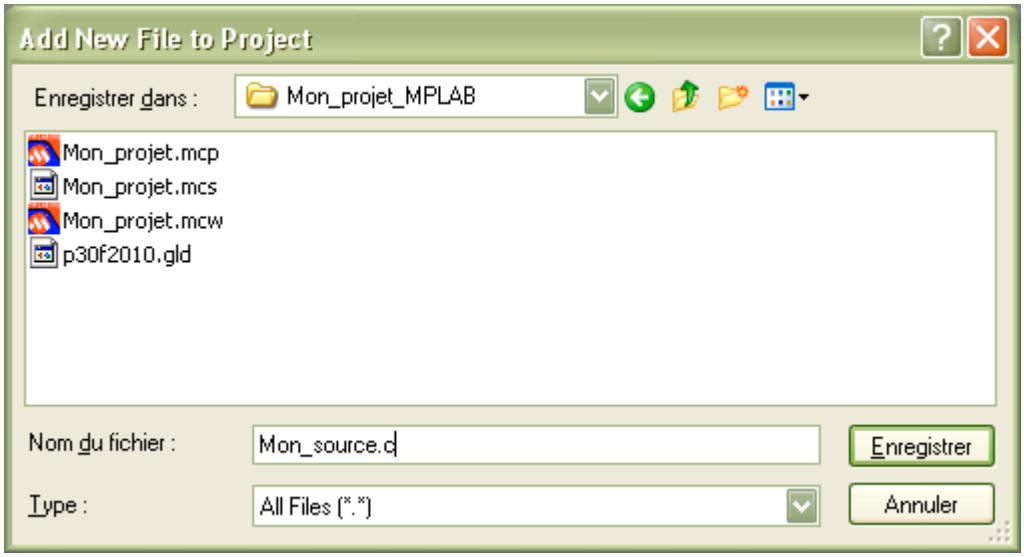
Cliquer sur "Terminer" pour créer le projet

7. L'environnement de développement se présente alors de la façon suivante :



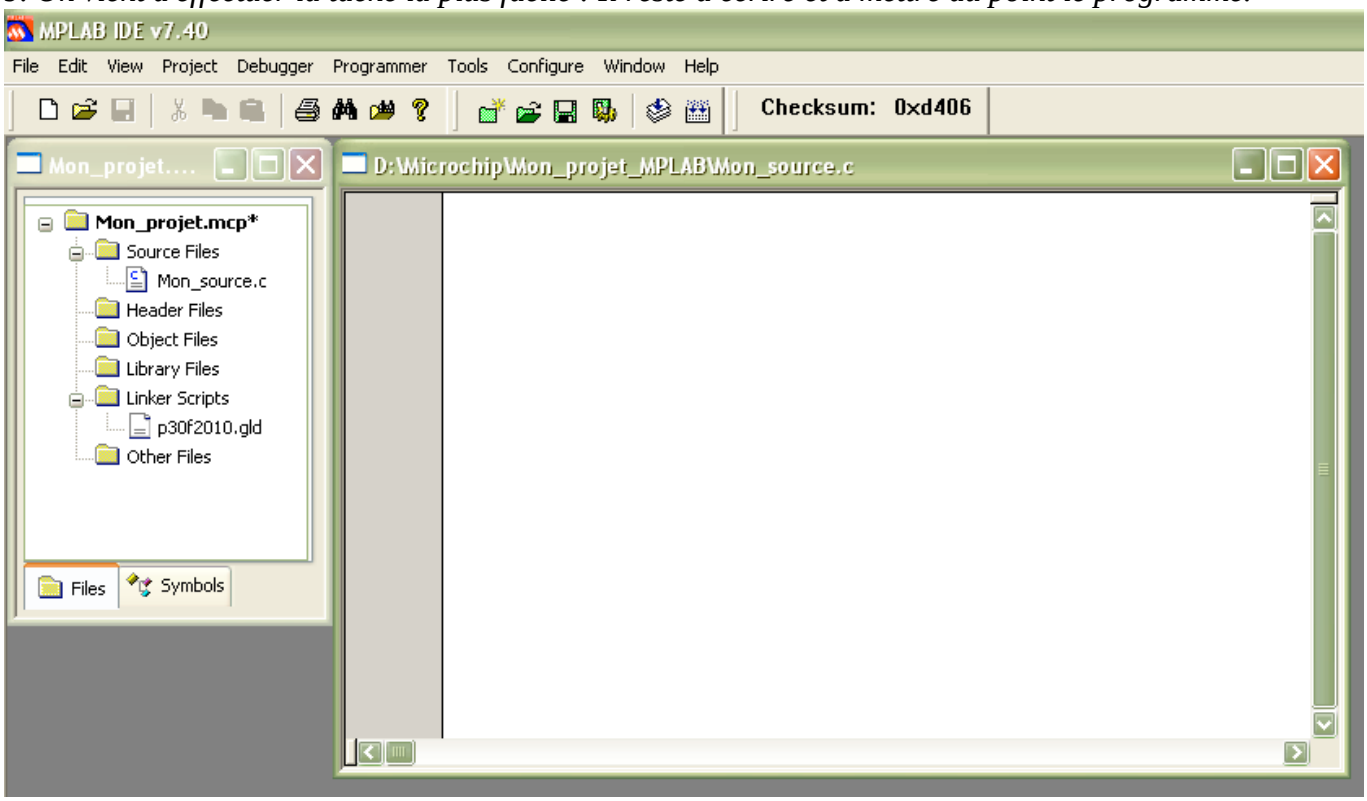
8. Mais il ne comporte encore aucun programme à traiter ! Pour ce faire, il faut créer un fichier source d'extension ".c" à partir du menu "File" :





Choisir un nom significatif en rapport avec le traitement effectué par le programme

9. On vient d'effectuer la tâche la plus facile ! Il reste à écrire et à mettre au point le programme.



Note : on peut changer les paramètres de l'éditeur de texte (par exemple : la taille des caractères ou la numérotation des lignes) en ouvrant le menu contextuel par un clic droit.

10. Bits de configuration *(menu : Configure → Configuration bits …)*

Chaque dsPIC comporte un certain nombre de bits de configuration : ils déterminent entre autres le type d'horloge, l'activation du chien de garde, la protection mémoire etc.

Ces bits sont situés en mémoire flash et sont donc programmables. Toutefois, il est préférable de reprendre la configuration reproduite ci-dessous qui est celle des µC utilisés en TP.

Configuration Bits

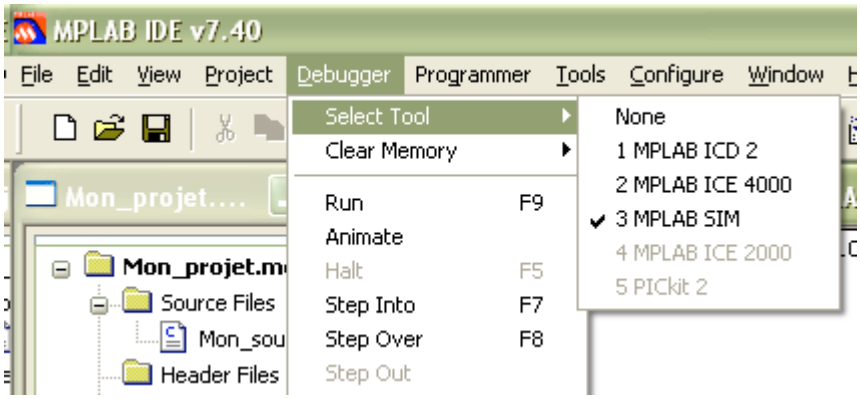
Ad...	Value	Category	Setting
F80000	C307	Clock Switching and Monitor	Sw Disabled, Mon Disabled
		Oscillator Source	Primary Oscillator
		Primary Oscillator Mode	XT w/PLL 16x
F80002	003F	Watchdog Timer	Disabled
		WDT Prescaler A	1:512
		WDT Prescaler B	1:16
F80004	87A2	Master Clear Enable	Enabled
		PWM Output Pin Reset	Control with PORT/TRIS regs
		High-side PWM Output Polarity	Active High
		Low-side PWM Output Polarity	Active High
		PBOR Enable	Enabled
		Brown Out Voltage	2.7V
		POR Timer Value	16ms
F8000A	0007	General Code Segment Code Protect	Disabled
		General Code Segment Write Protect	Disabled
F8000C	C003	Comm Channel Select	Use PGC/EMUC and PGD/EMUD

Résumé de la configuration :

Horloge : quartz XT; Fcy=F_Q*16/4 (soit 16MHz avec un quartz de 4 MHz)

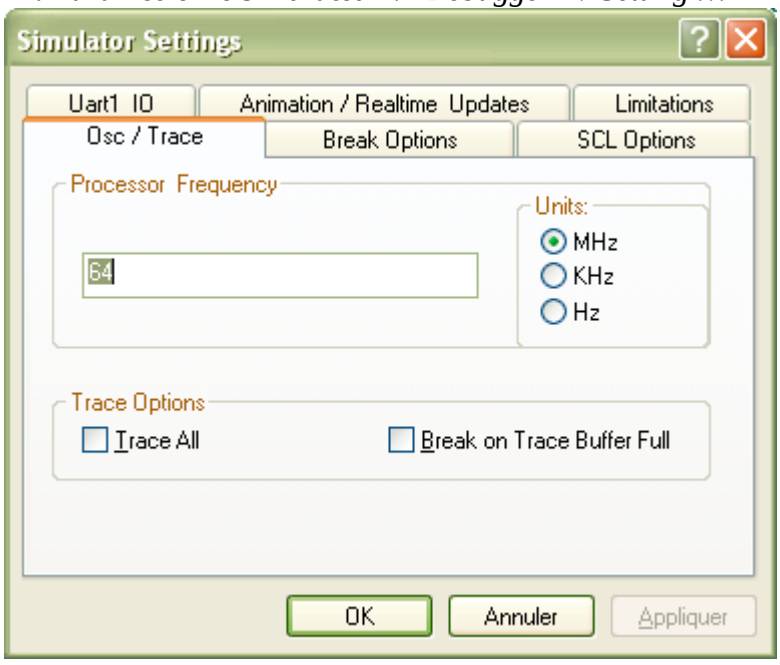
- *Chien de garde inhibé*
- *Mémoire non protégée*

11. Sélection du simulateur :

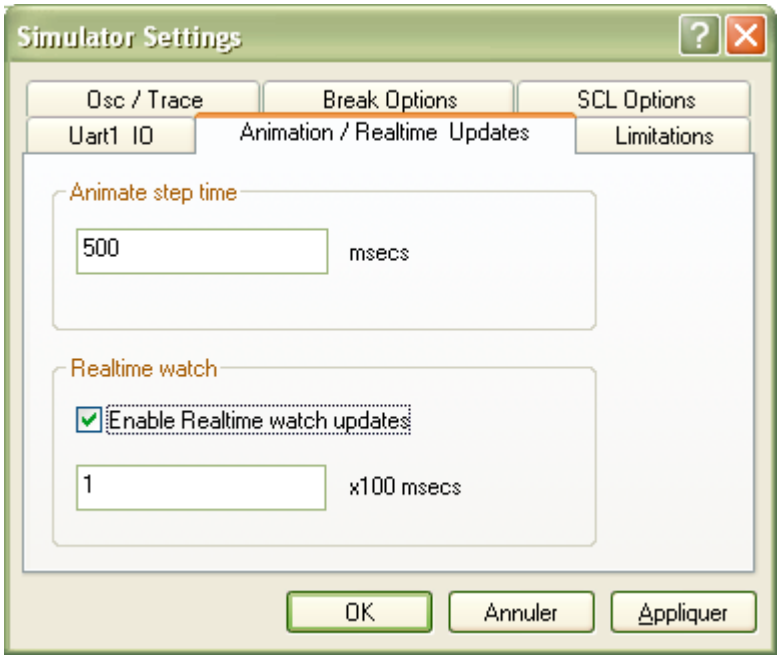


La première étape de la mise au point est toujours la simulation

***12. Paramétrer le simulateur** : "Debugger → Setting …"*



Choix de l'horloge : Fosc=64MHz

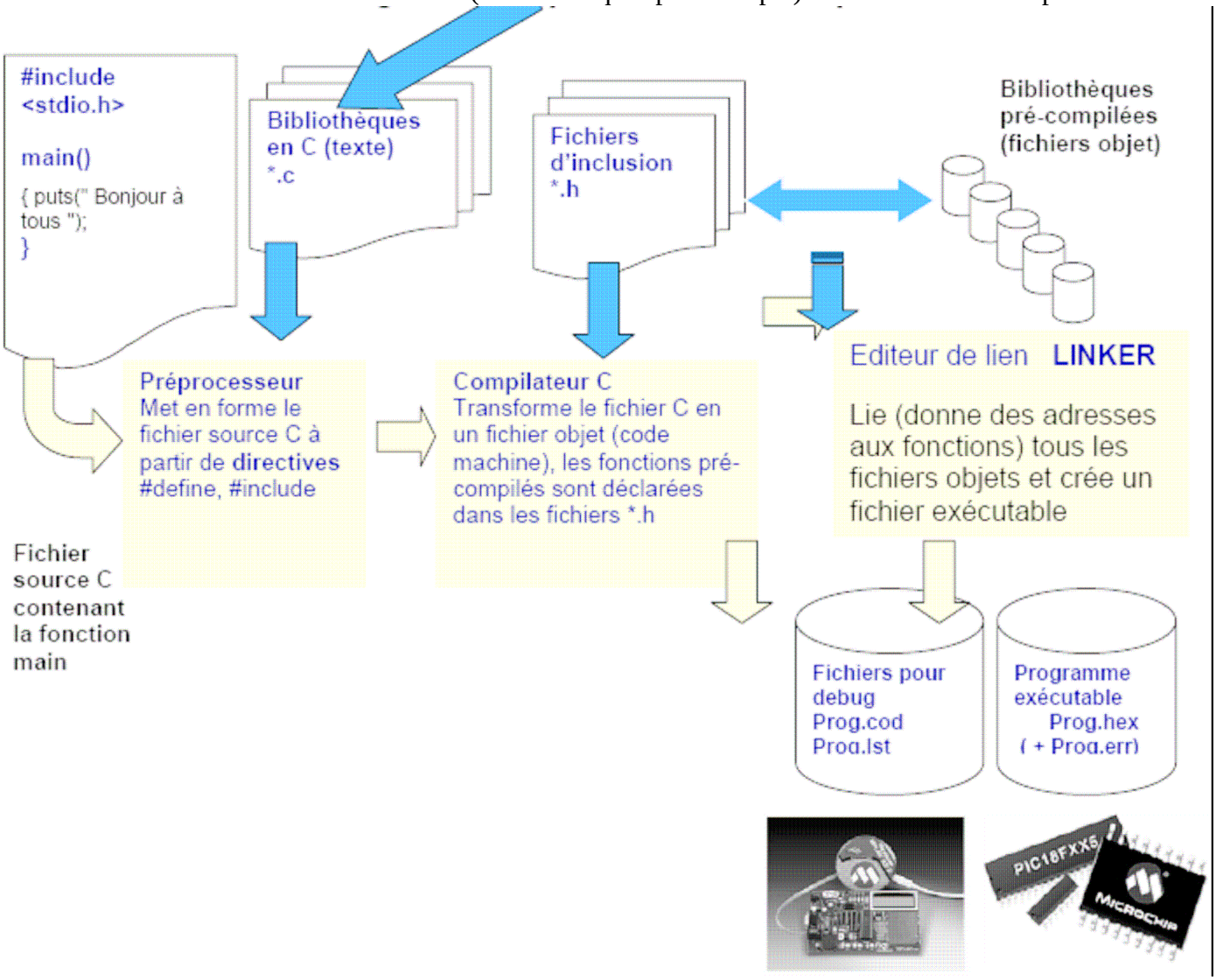


Cocher "Enable Realtime watch update" et régler le paramètre à 100mS

2. Compilation

2.1 Processus de compilation

Le programmeur écrit son programme dans un ou plusieurs fichiers texte en utilisant le langage normalisé "C" (les fichiers "sources"). Ces fichiers ne peuvent pas être programmés dans le µC cible tels quels : le CPU de celui-ci ne connaît que le code machine et ne comprend rien au texte des fichiers source.
La tâche du compilateur est de "traduire" ces fichiers source en code machine (fichier .hex), en utilisant éventuellement des librairies de fonctions (mathématiques par exemple) fournies avec le compilateur.



Rôle du pré-processeur :
Le pré-processeur ou pré-compilateur réalise des mises en forme et des aménagements du texte d'un fichier source, juste avant qu'il ne soit traité par le compilateur. Il existe un ensemble d'instructions spécifiques appelées **directives** pour indiquer les opérations à effectuer durant cette étape.
Les deux directives les plus courantes sont #define et #include.
#define correspond à une équivalence ex : #define pi 3.14 ou une définition de macro

Rôles des fichiers d'inclusion :
Les fichiers d'inclusion ou d'en tête *.h (header) contiennent pour l'essentiel cinq types d'informations :

- des définitions de nouveau type
- des définitions de structure
- des définitions de constantes

Le fichier d'inclusion p30f2010.h est particulièrement important lorsqu'on travaille en C sur ce micro-contrôleur : il définit tous les registres internes du micro-contrôleur dsPIC30F2010 de façon très détaillée et permet d'éviter de nombreuses erreurs dans l'affectation de ces registres.

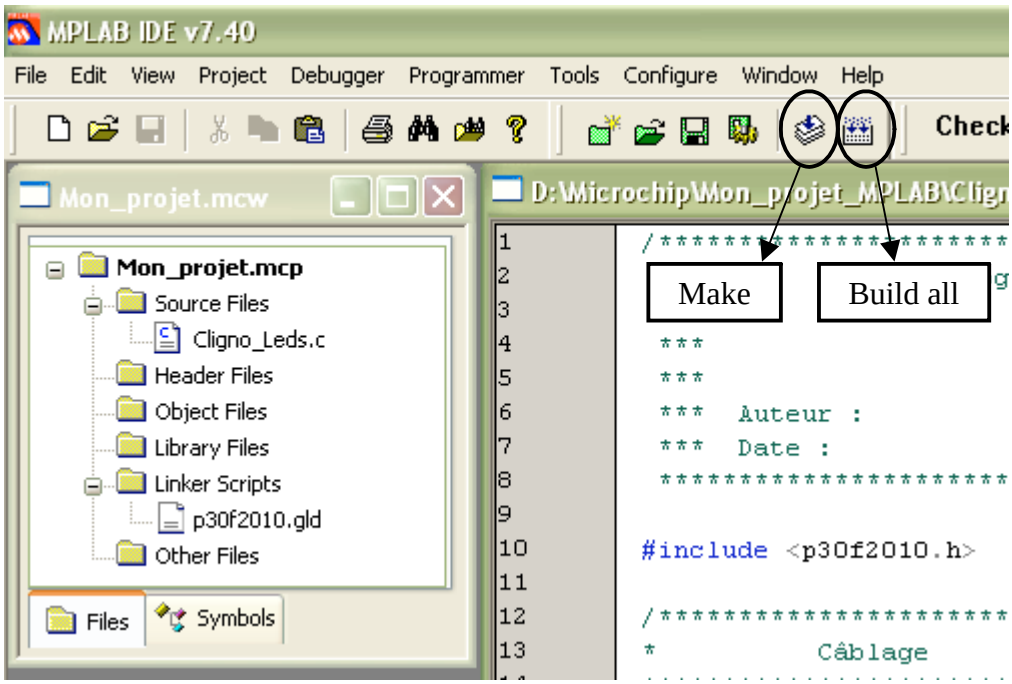
Par exemple, pour mettre à 1 le bit 2 du PORTB (noté RB2), on peut écrire :

- `PORT2 |= 0x0004;`
- `PORT2bits.RB2 = 1;` On utilise ici les déclarations de "p30f2010.h"

La 2° solution est bien plus sûre.

2.2 Lancer la compilation

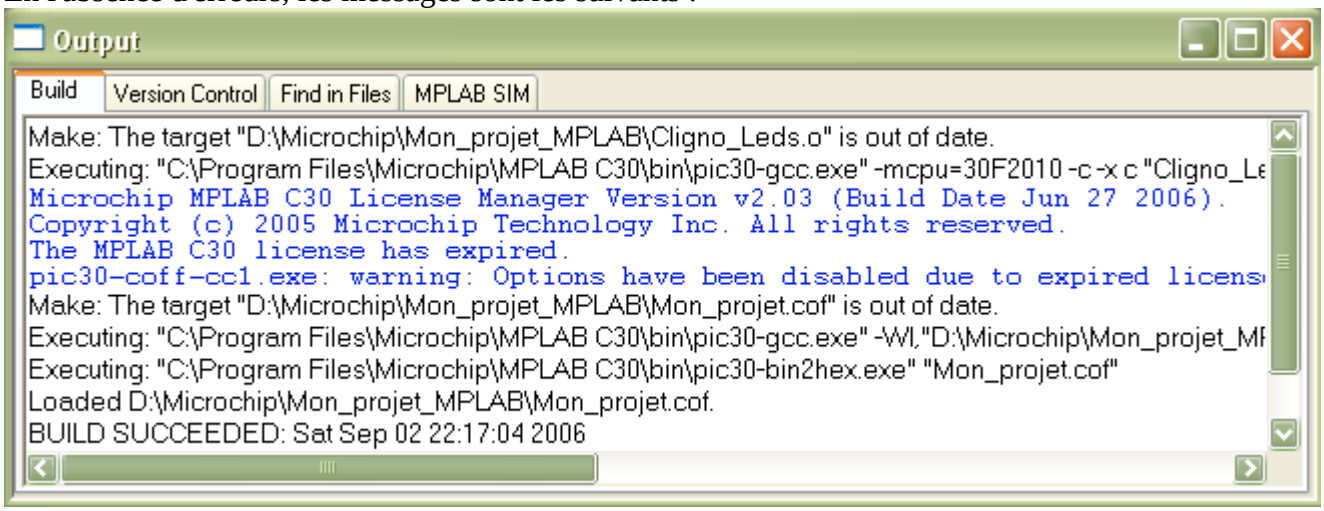
Vérifier avant tout la présence du fichier source (ici "Cligno_Leds.c") et du fichier "p30f2010.gld" dans la fenêtre du projet (ici "Mon_projet.mcw").



La procédure est très simple : il suffit de cliquer sur "Make" ou "Build All" !

Les 2 actions donnent les mêmes résultats. En fait "Make" tient compte de la date des fichiers, ce que ne fait pas "Build All".

En l'absence d'erreurs, les messages sont les suivants :

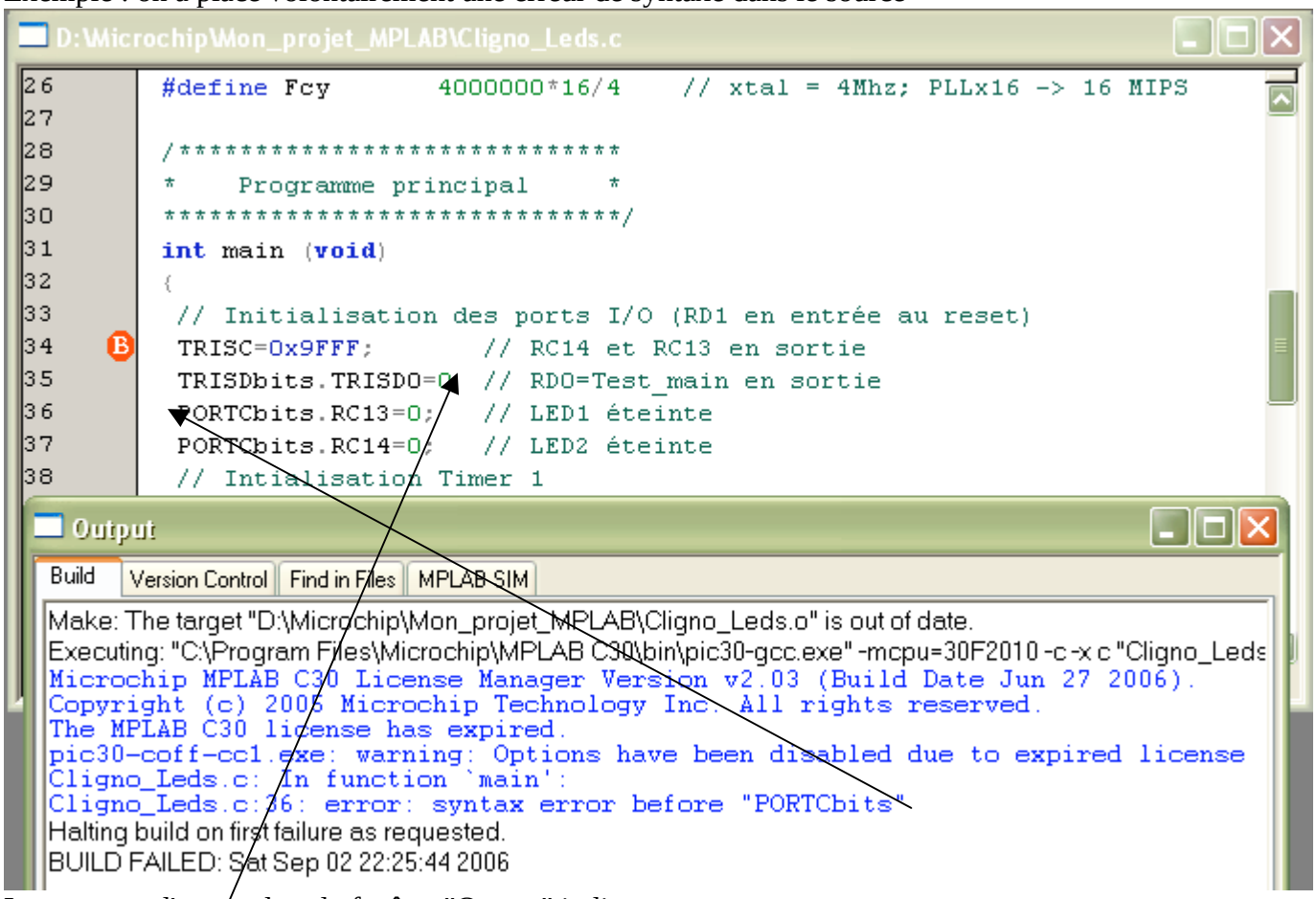


Le fichier de programmation est créé : il se nomme **<Nom du projet>.hex** .
Ce fichier est également utilisé par le simulateur (avec quelques autres fichiers qui définissent les symboles) pour permettre une simulation en suivant l'exécution du programme dans le source.

2.3 Corriger les erreurs de compilation

Personne n'a jamais compilé une première fois un programme sans erreur ! Mais l'environnement MPLAB facilite leur repérage et donc leur correction (la correction automatique n'existe pas encore !).

Exemple : on a placé volontairement une erreur de syntaxe dans le source



Le message d'erreur dans la fenêtre "Output" indique :

Cligno_Leds.c : In function `main':

Cligno_Leds.c : 36 : error : syntax error before "PORTCbits"

La position de l'erreur est repérée dans le source par un "double-clic" (voir flèche ci-dessus). En fait l'erreur est ici : le **;** manque.

La position indiquée est correcte car le compilateur ne tient pas compte des commentaires et sauts de ligne.

3. Simulation

Le simulateur de MPLAB est très complet et couvre la majorité des périphériques intégrés dans le µC. Ses principales spécifications sont :

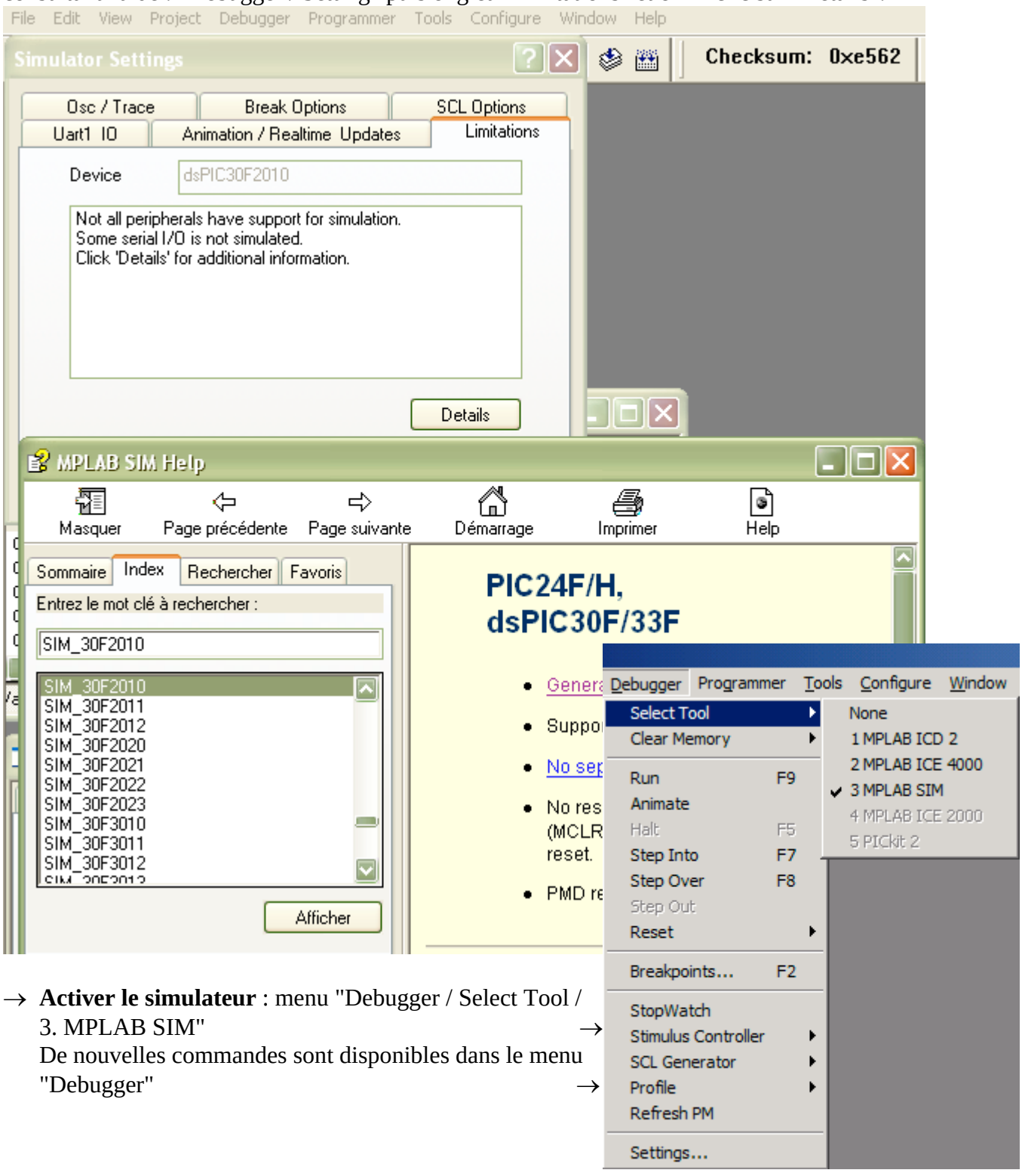
modes "run", "pas à pas" et "pas à pas" à vitesse réglable

points d'arrêts multiples

- fenêtre de suivi de variables et registres à taux de rafraîchissement réglable
- stimuli à affectation manuelle et programmée

simulation de l'UART via des fichiers "texte"

Il est prudent de vérifier que le simulateur supporte les périphériques utilisés dans le programme en consultant l'aide : "Debugger / Setting" puis onglet "Limitations" et enfin clic sur "Details".



→ **Activer le simulateur** : menu "Debugger / Select Tool / 3. MPLAB SIM" →

De nouvelles commandes sont disponibles dans le menu "Debugger" →

3.1 Exemple de programme

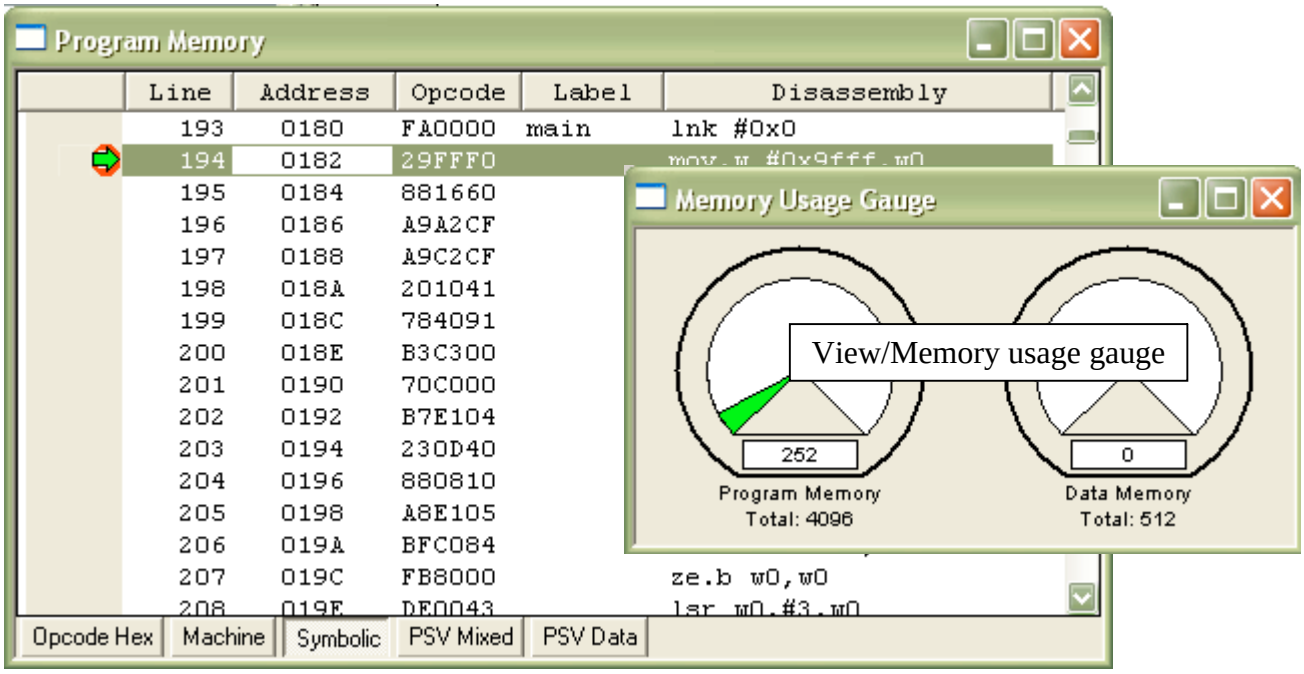
Les possibilités du simulateur sont présentées avec un petit programme qui fait clignoter 2 leds.
Voici le source :

```
/*****************************************************************
 ***                Programme d'apprentissage C30                ***
 ***                        dsPIC30F2010                         ***
 ***                    Clignotement de 2 leds                   ***
 ***  Auteur : CREMMEL Marcel                                    ***
 ***  Date : 04/09/2006                                          ***
 *****************************************************************/

#include <p30f2010.h>

/*****************************
*          Câblage           *
*****************************
Sorties :
  LED1 sur RC13
  LED2 sur RC14
Entrée :
  BP sur RD1
*/

/*****************************
* Constantes non mémorisées  *
*****************************/
#define Fcy       4000000*16/4    // xtal = 4Mhz; PLLx16 -> 16 MIPS

/*****************************
*    Programme principal     *
*****************************/
int main (void)
{
 // Initialisation des ports I/O (RD1 en entrée au reset)
 TRISC=0x9FFF;       // RC14 et RC13 en sortie
 PORTCbits.RC13=0;   // LED1 éteinte
 PORTCbits.RC14=0;   // LED2 éteinte
 // Intialisation Timer 1
 T1CONbits.TCKPS=3;  // Fcy pré-divisée par 256 (soit 62,5kHz)
 PR1=(2*Fcy/256)/10; // Période Timer 1 = 200mS
 TMR1=0;             // Raz du compteur du Timer 1
 T1CONbits.TON=1;    // Timer 1 "ON"
 while (1)           // Boucle sans fin. 1 signifie vrai
  {
   while (!IFS0bits.T1IF) {} // Attendre chargement Timer 1
   IFS0bits.T1IF=0;   // Raz indicateur pour période suivante
   if (PORTDbits.RD1) // Test du BP sur RD1
    {
     LATCbits.LATC13^=1; // Inverser LED1
     LATCbits.LATC14 =0; // Eteindre LED2
    }
   else
    {
     LATCbits.LATC13 =0; // Eteindre LED1
     LATCbits.LATC14^=1; // Inverser LED2
    }
  }
}
```

3.2 Premiers essais

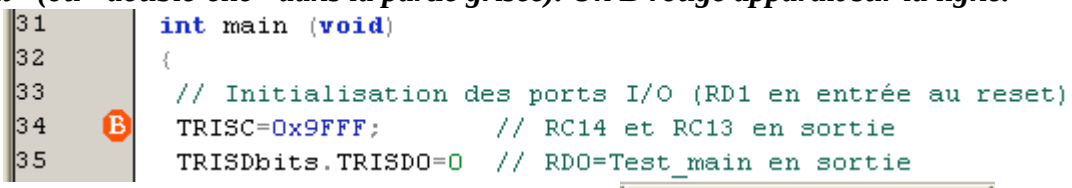
*Compiler **le programme (raccourci** F10)*

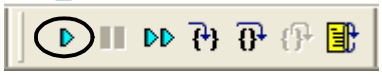
On peut observer le résultat en ouvrant la fenêtre "Program Memory" ("View / 3 Program Memory") :

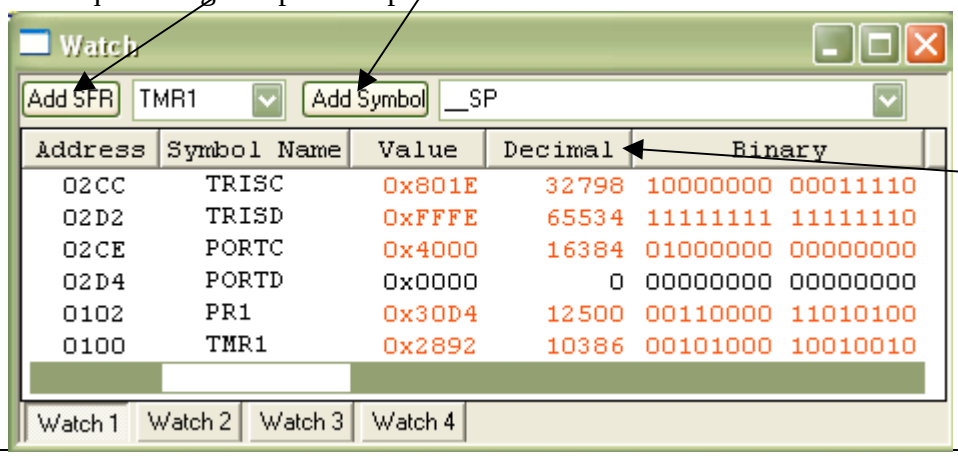


On constate que la première instruction de la fonction "main" est placée à l'adresse 0180h dans la mémoire programme. Un "reset" fait démarrer le CPU à l'adresse 0000h; le compilateur y place une instruction de saut vers des fonctions d'initialisation du pointeur de pile et des variables (à partir de l'adresse 0100h et jusqu'à 017Fh). Le CPU exécute la fonction "main" seulement après ces opérations.

Placer un point d'arrêt sur la première ligne d'instructions de la fonction "main" : clic droit puis** "Set Breakpoint" **(ou "double-clic" dans la partie grisée). Un B rouge apparaît sur la ligne.



- Lancer le programme : clic sur "**Run**" (raccourci **F9**) : 
- Le programme s'arrête au point d'arrêt, sur la 1° instruction du programme. En fait, avant d'en arriver à ce point, le µC a déjà exécuté un certain nombre de fonctions : initialisation du pointeur de pile, raz des variables déclarées dans le programme (aucune ici), etc.
- Fenêtre de suivi de variables et registres : **View / Watch**
  Pour placer un registre ou une variable : on peut la choisir dans les menus déroulants ou utiliser la technique "Drag and paste" à partir du source.

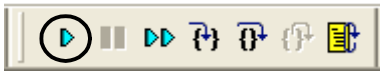
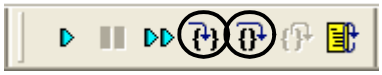
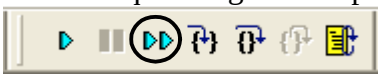
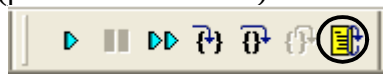


On peut choisir tous les codes d'affichage possible : hexa, binaire, ascii, etc. en cliquant avec le bouton droit sur cette ligne.

La couleur rouge des valeurs indique un changement d'état depuis le dernier rafraîchissement.

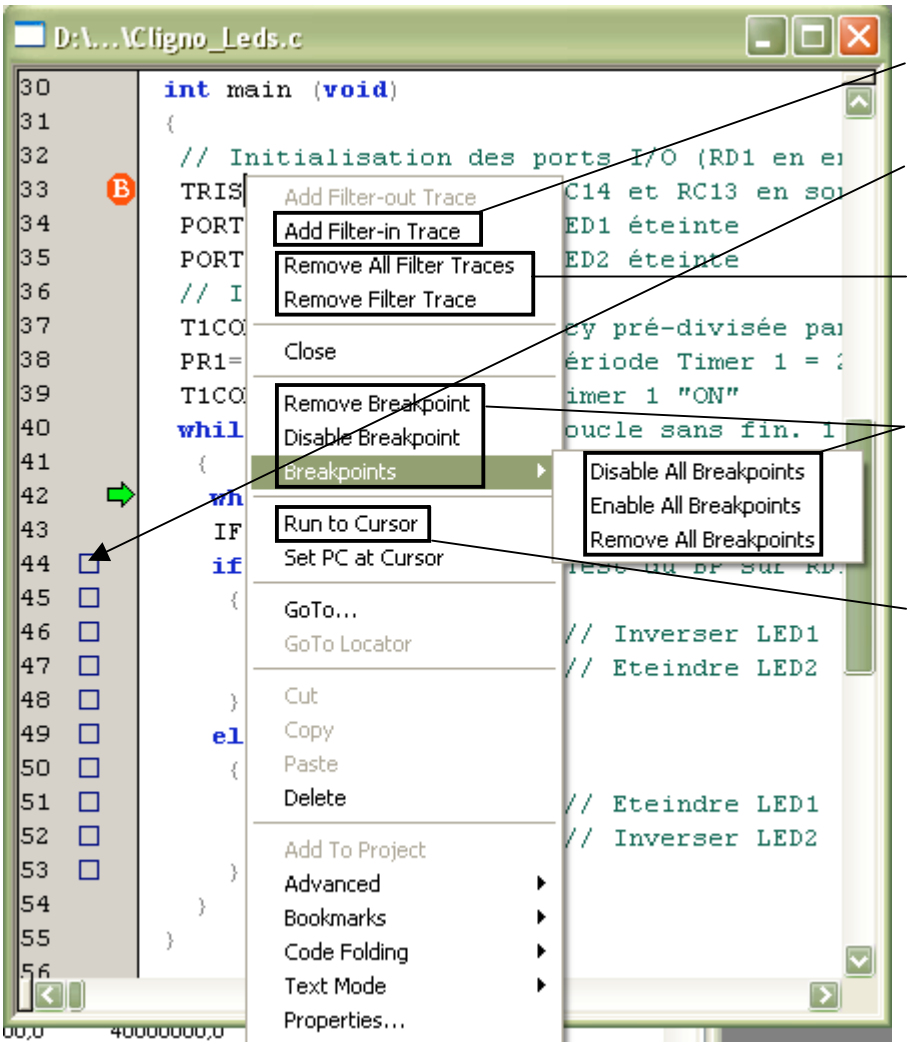
Lancer le programme ("Run"): on observe l'évolution des registres pendant l'exécution du programme.
Note : les valeurs peuvent être modifiées par un double-clic.

3.3 Commandes de base

- "**Run**" (raccourci **F9**) :  Le programme se lance à vitesse max à partir de la position actuelle du pointeur vert.
- **Pas à pas** (raccourcis : **F7** et **F8**) :  Il n'y a pas de différence entre les 2 choix proposés avec ce programme simple (il n'y a pas d'appel de fonction).
  Le pointeur vert indique la ligne de la prochaine instruction à exécuter.
- **Animate** :  Il s'agit d'un "pas à pas" automatique. Le rythme peut être réglé dans le menu : "Debugger / Settings" sous l'onglet "Animation/realtime Updates" : paramètre "Animate step time" (par défaut à 500mS)
- **Reset** (raccouci **F6**) :  Le CPU est mis à zéro. Il est à l'arrêt et prêt à exécuter l'instruction à l'adresse 0000h. Pour aller au début de la fonction "main", on recommande de placer un point d'arrêt sur la première ligne d'instructions, puis "Run".
- **Halt** (raccourci F5) :  . Arrêt asynchrone du CPU à la fin de l'instruction en cours.
- **Point d'arrêt :** Un "double clic" dans la partie grise à gauche de la ligne place ou retire un point d'arrêt

3.4 Commandes et fonctions avancées

- **Menu contextuel :** Clic droit dans le fichier source



Add Filter-in Trace
Balayer les lignes dont on veut analyser l'exécution.
Les lignes sélectionnées sont repérées par un petit carré.

Remove Filter Trace
Pour retirer des lignes de la sélection "Filter Trace"

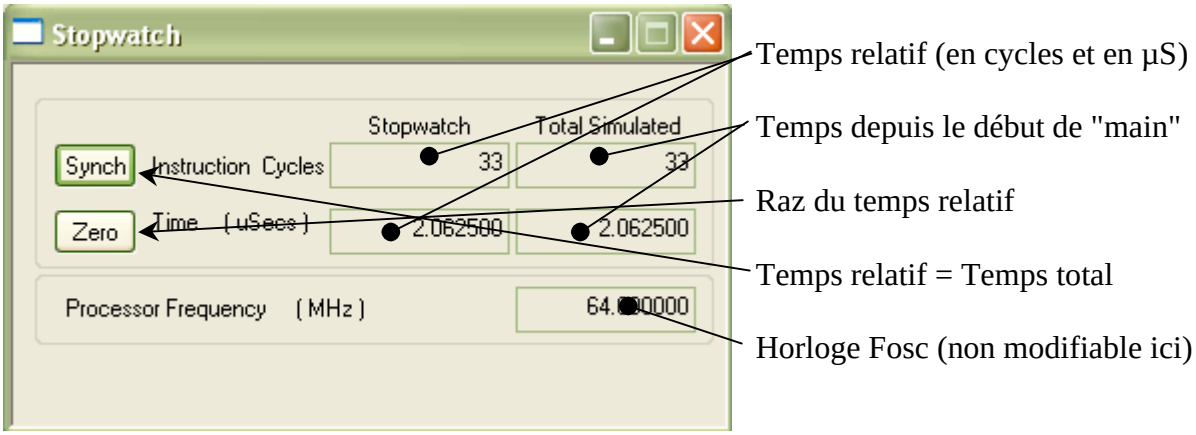
Gestion des points d'arrêt
Un "double clic" dans la partie grise à gauche de la ligne place ou retire un point d'arrêt

"Run" et arrêt au curseur

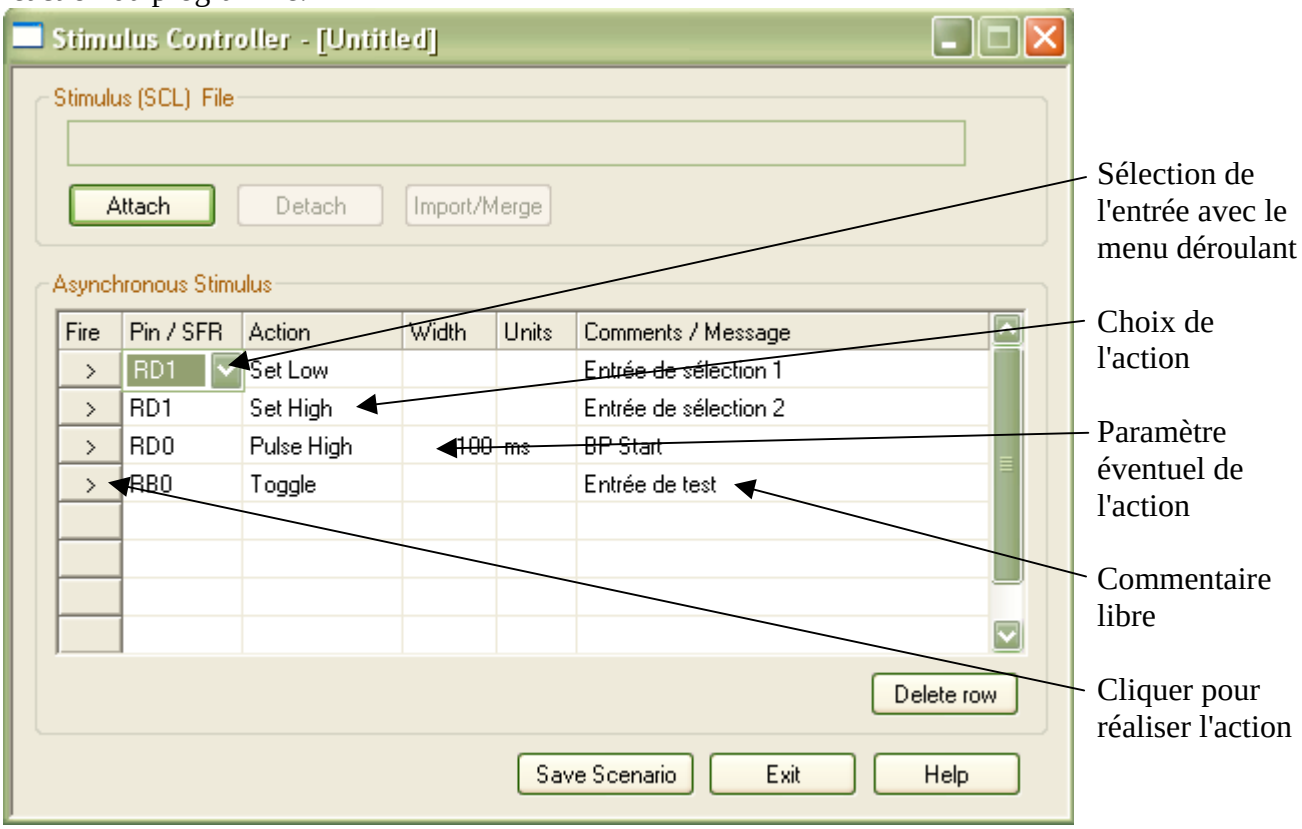
Les autres commandes sont peu ou pas utilisées en simulation.

La fonction "**Filter Trace**" est intéressante si on utilise l'analyseur logique : "View / Simulator Logic Analyser". On choisit les lignes qui affectent les signaux à observer : l'analyseur mémorise alors les instants pour lesquels ces sorties changent d'état et on obtient un chronogramme réaliste (voir l'exemple plus loin).

- **StopWatch** : "Debugger / StopWatch
  Cette fenêtre permet de mesurer le temps d'exécution du programme.

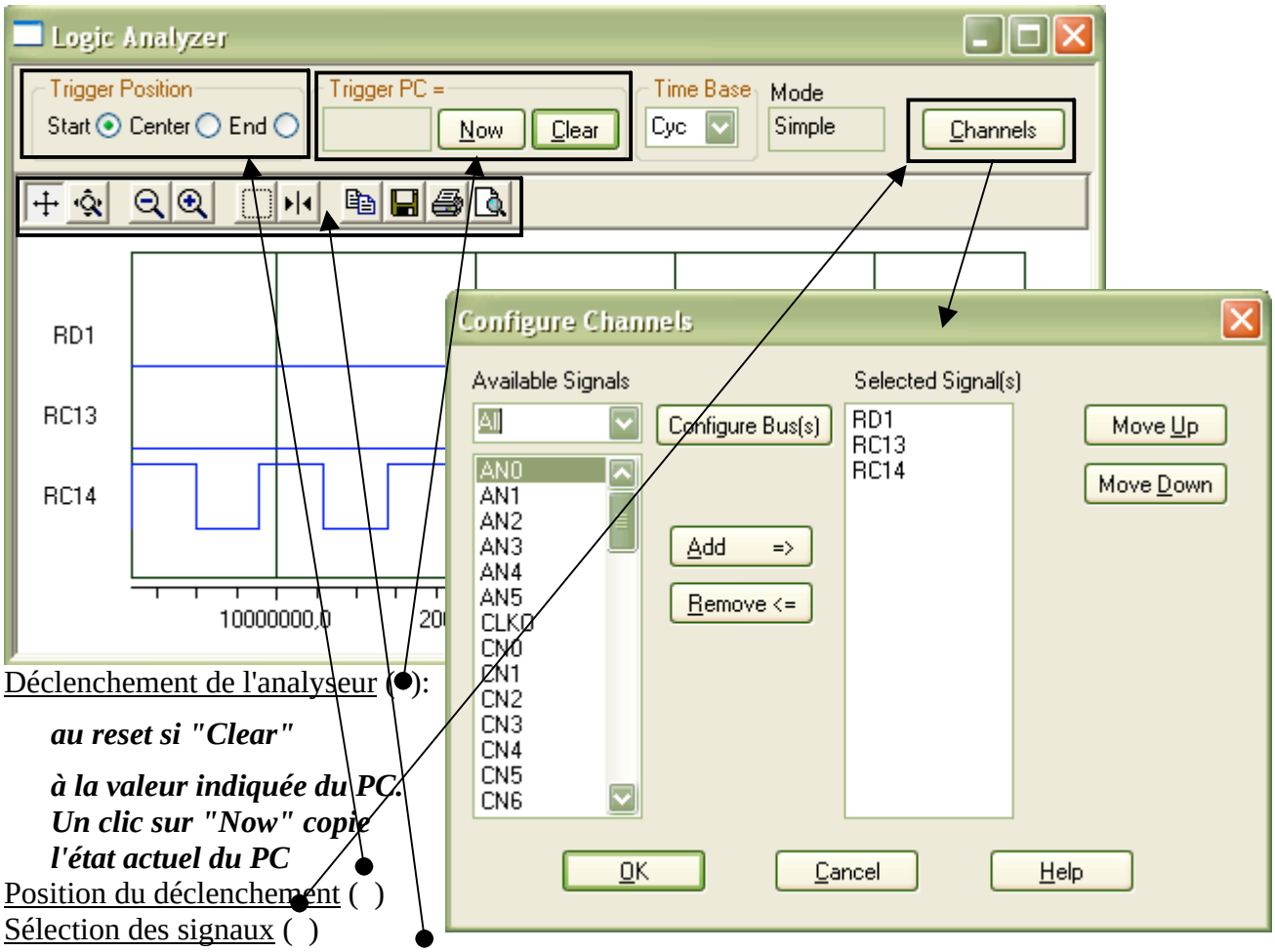


- **Stimulus Controller :** contrôleur de stimuli : "Debugger / Stimulus Controller / New Scenario"
  Cette fenêtre permet d'affecter manuellement toutes les entrées du µC dans le but d'observer la réaction du programme.



Attention : les affectations ne sont effectives qu'en cours de simulation.

- **Logic Analyser** : "View / Simulator Logic Analyser"
  Attention : il faut au préalable sélectionner les lignes "Filter-in Trace" dans le programme car l'analyseur ne mémorise les états des signaux qu'aux instants d'exécution des instructions correspondantes.



Déclenchement de l'analyseur (●):

au reset si "Clear"

à la valeur indiquée du PC.
Un clic sur "Now" copie l'état actuel du PC

Position du déclenchement ( )
Sélection des signaux ( )
Manipulation du chronogramme ()

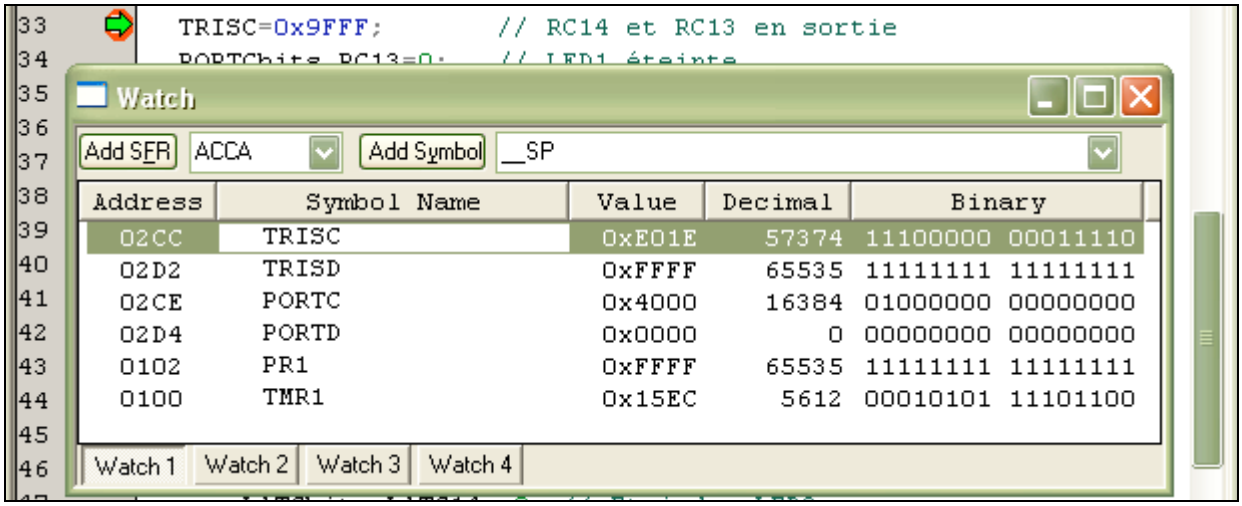
3.5 Exemple

On va traiter l'exemple de programme fourni au § 3.1 pour se familiariser avec toutes ces fonctions.
Le programme a été compilé au début du §3.2 : tout est prêt pour la simulation.

Point d'arrêt au début de "main" ***: double clic dans la partie grise sur la ligne 33***

```
32
33  B   // Initialisation des ports I/O (RD1 en entrée au reset)
34      TRISC=0x9FFF;       // RC14 et RC13 en sortie
35      PORTCbits.RC13=0;   // LED1 éteinte
        PORTCbits.RC14=0;   // LED2 éteinte
```

"Reset" puis "Run" ***: le programme s'arrête au point d'arrêt (voir §3.2 pour la fenêtre "Watch")***



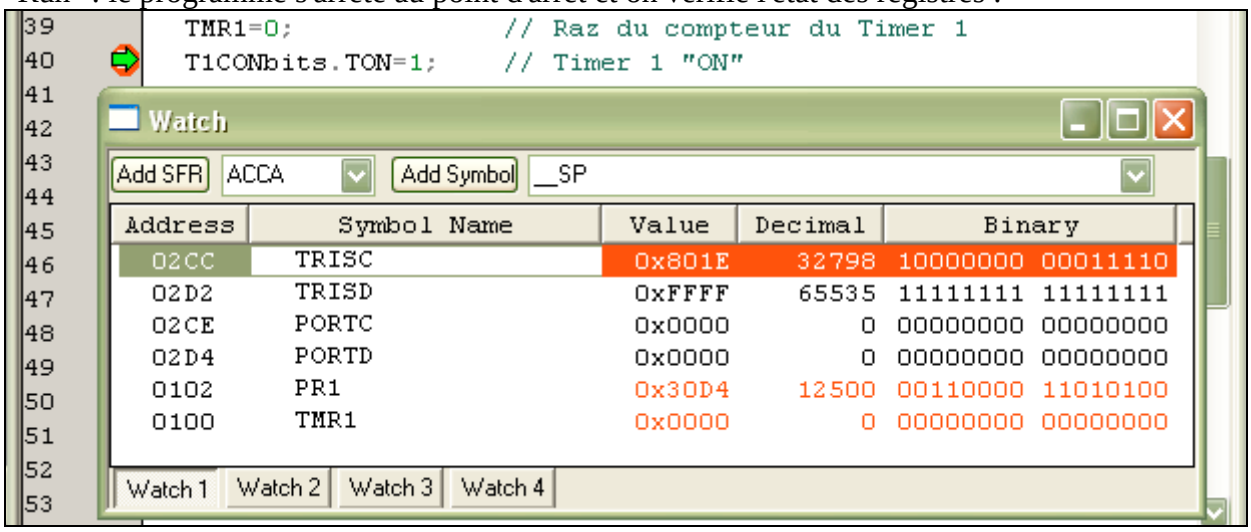
On faisant un pas (F7 : la flèche verte descend d'une ligne), on constate que le registre TRISC ne prend pas la valeur 0x9FFF. Ceci est dû au fait que seuls les ports RC13, RC14 et RC15 sont utilisés sur un dsPIC30F2010.

1. **Vérification de la configuration des registres**

On place un point d'arrêt juste avant la boucle sans fin :

```
39   TMR1=0;              // Raz du compteur du Timer 1
40   T1CONbits.TON=1;     // Timer 1 "ON"
41   while (1)            // Boucle sans fin. 1 signifie vrai
```

"Run" : le programme s'arrête au point d'arrêt et on vérifie l'état des registres :



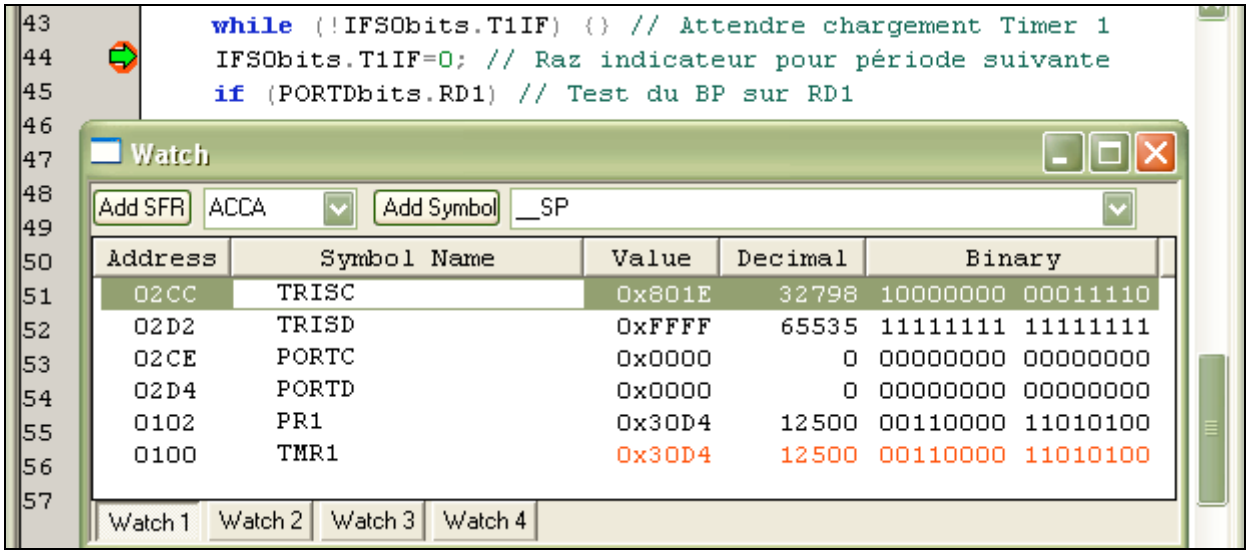
Constatations : les affections prévues sont réalisées et on constate que RD1="0"

2. **Vérification du positionnement de l'indicateur T1IF du registre IFS0**

Pour cela on place un point d'arrêt juste après le test de cet indicateur :

```
43   while (!IFS0bits.T1IF) {} // Attendre chargement Timer 1
44   IFS0bits.T1IF=0; // Raz indicateur pour période suivante
45   if (PORTDbits.RD1) // Test du BP sur RD1
```

On relance le programme. On constate que le compteur TMR1 s'incrémente dans la fenêtre "Watch". L'indicateur T1IF passe à "1" quand TMR1 atteint la valeur du registre PR1. Quand c'est le cas, le programme passe à la ligne du point d'arrêt et il s'arrête. On obtient :



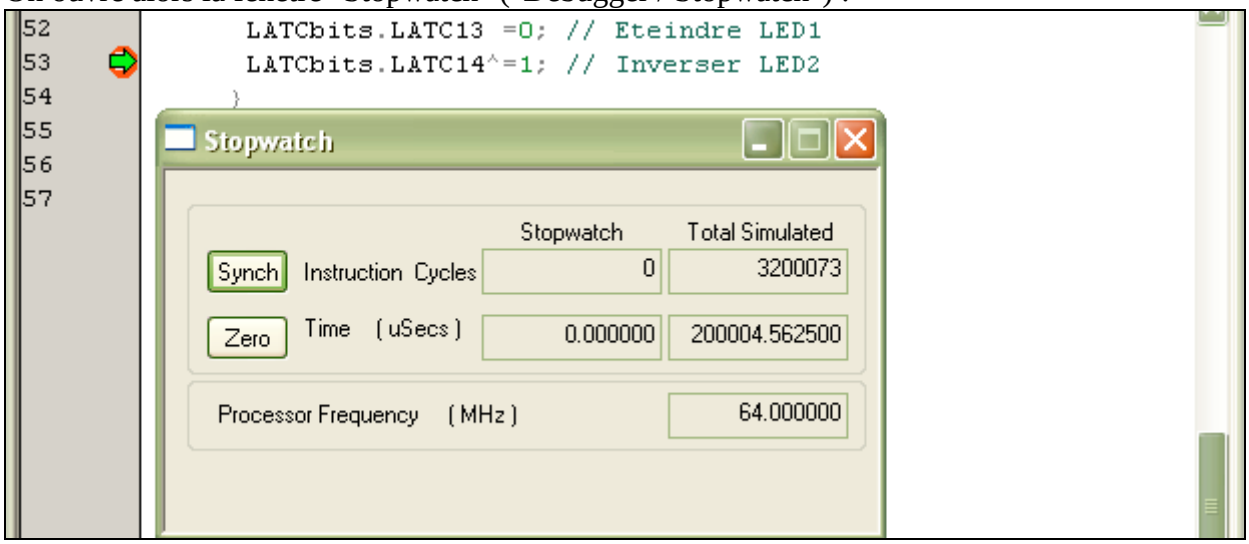
Tout est conforme : TMR1=PR1. On peut supprimer le dernier "breakpoint".
La prochaine instruction du programme met l'indicateur T1IF à "0". Il sera remis à "1" par le "Timer 1" 200mS plus tard (temps CPU).

3. Mesure de la période d'allumage / extinction d'une LED

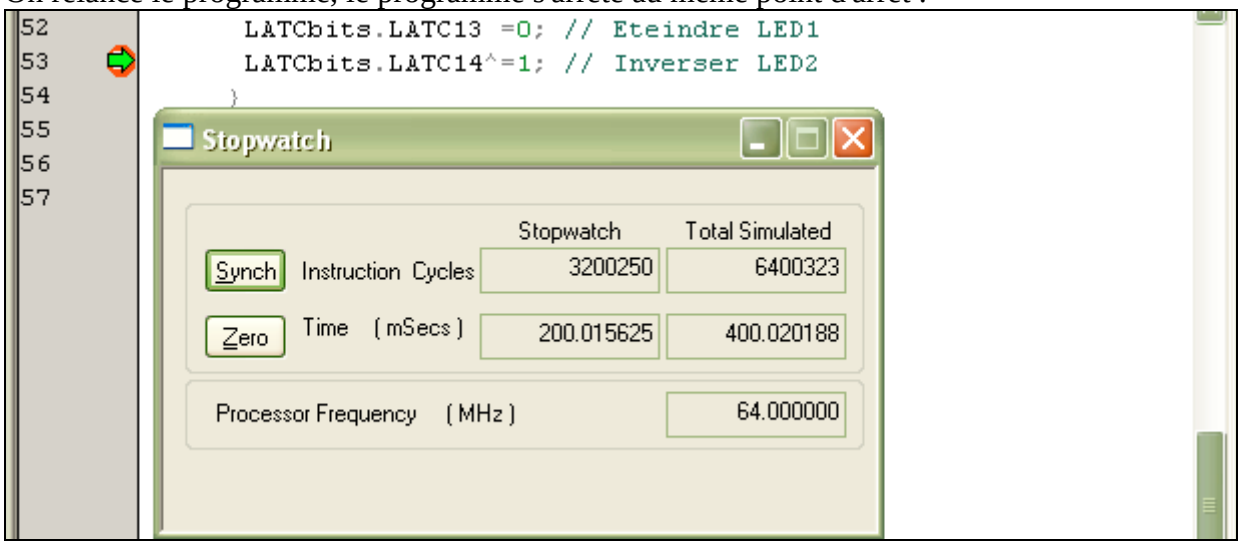
Pour mesurer cette période, on place un nouveau point d'arrêt sur la ligne qui inverse l'état de la LED2 :

```
50      else
51       {
52        LATCbits.LATC13 =0; // Eteindre LED1
53        LATCbits.LATC14^=1; // Inverser LED2
54       }
```

On relance le programme, le programme s'arrête au "breakpoint".
On ouvre alors la fenêtre "Stopwatch" ("Debugger / Stopwatch") :



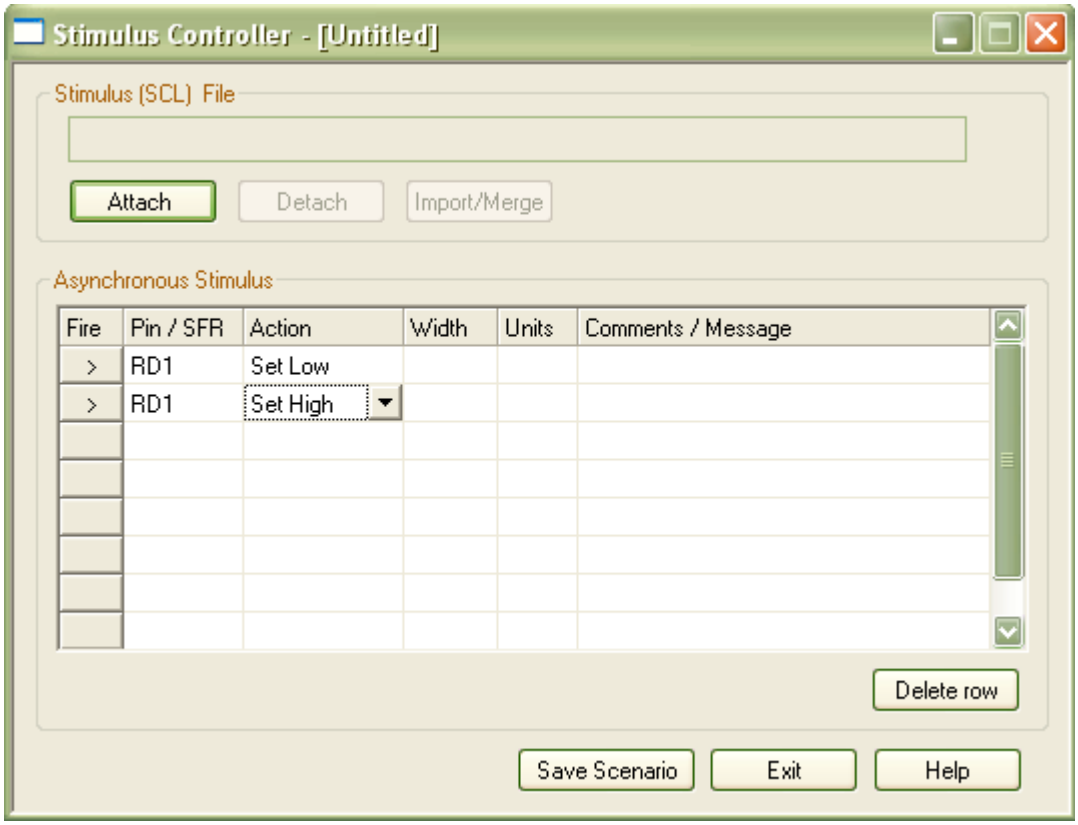
Le compteur "Stopwatch" doit être normalement à 0. Si ce n'est pas le cas : clic sur "Zero"
On relance le programme, le programme s'arrête au même point d'arrêt :



Le temps réellement écoulé est un peu plus grand que 200mS : c'est conforme pour un clignotement de LED.
En relançant le programme, on mesure de la même façon la durée d'extinction de la LED.

4. Simuler l'entrée RD1

L'entrée RD1 agit sur le déroulement du programme. Cela va être vérifié dans cette étape.
Pour agir sur RD1 pendant le déroulement du programme, on ouvre la fenêtre "Stimulus Controller" : "Debugger / Stimulus Controller / New Scenario".
On place 2 lignes RD1 comme décrit au §3.4 : une qui affecte RD1 à "0", l'autre à "1" :

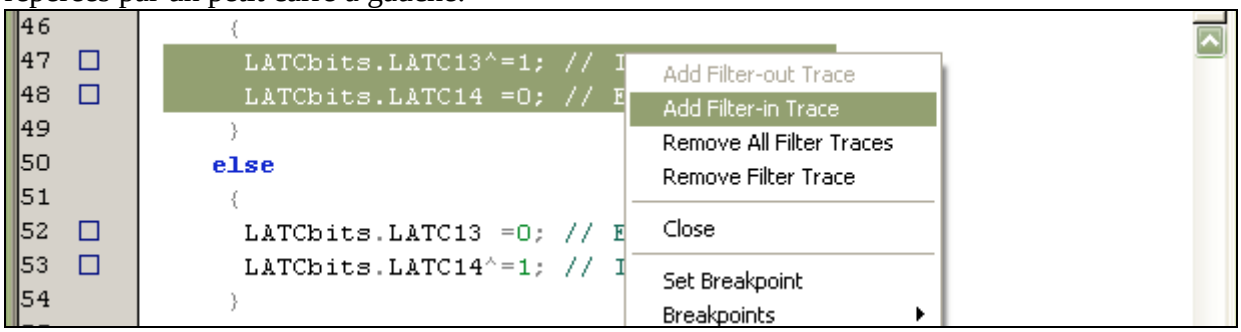


Retirer tous les points d'arrêt et relancer le programme.
On observe alors le PORTC dans la fenêtre "Watch" :

- si RD1 = 0 (clic sur "Fire"), sa valeur doit basculer entre les valeurs 0x0000 (RC14=LED2="0") et 0x4000 (RC14=LED2="1")
- si RD1 = 1 (clic sur "Fire"), sa valeur doit basculer entre les valeurs 0x0000 (RC13=LED1="0") et 0x2000 (RC13=LED1="1")

5. Chronogramme complet :

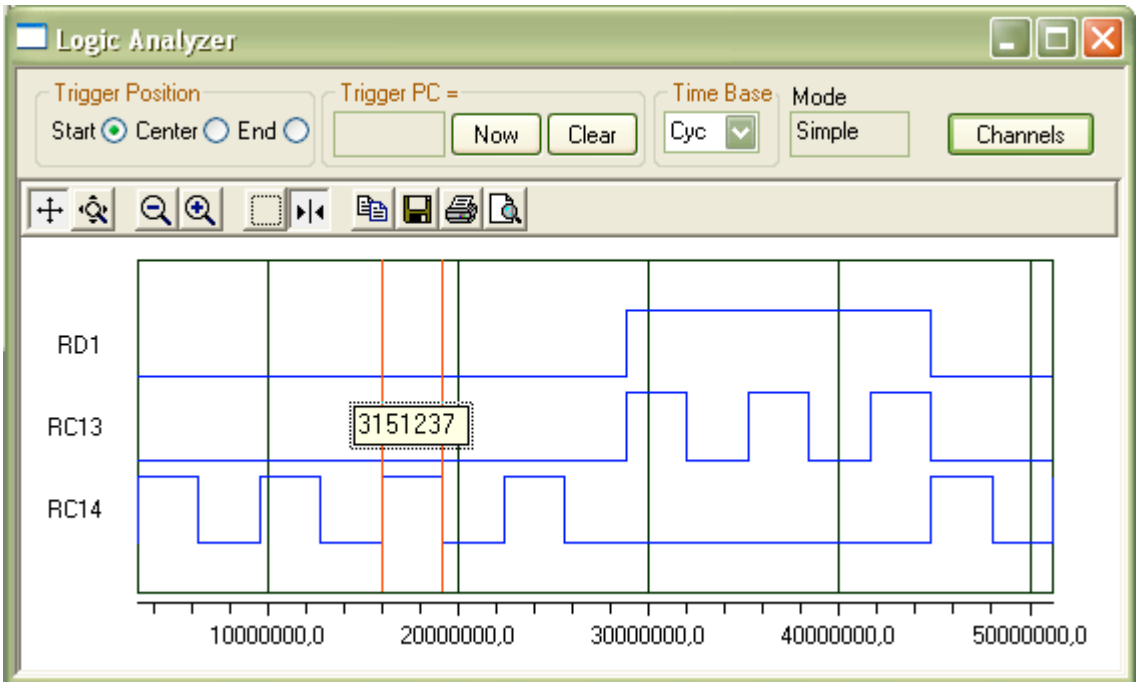
L'analyseur logique permet d'obtenir par simulation un chronogramme de fonctionnement typique. Pour cela il faut commencer par sélectionner les lignes "Filter-in Trace" (voir menu contextuel au §3.4). L'analyseur logique ne prend en compte que les lignes ayant l'attribut "Filter-in Trace", repérées par un petit carré à gauche.



Ces lignes ont été choisies car ce sont les seules qui affectent les sorties RC13 et RC14.
On ouvre alors l'analyseur logique : "View / Simulator Logic Analyser" et on le paramètre comme décrit au §3.4

On retire alors tous les points d'arrêt, on fait un "reset" et on lance le programme. On agit sur RD1 via le "Stimulus Controller" (plusieurs mises à "1" et à "0"). On arrête le programme au bout de quelques secondes.

On obtient alors le résultat suivant :



Le résultat est conforme.
Les curseurs permettent d'évaluer le nombre de cycles entre 2 changements d'état : 3151237 cycles. Avec une horloge Fcy de 16MHz, cela correspond à une durée de 197mS : CQFD.
Note : l'entrée RD1 semble changer d'état en même temps que RC13. Ce n'est pas vrai dans le cas réel ! En fait, l'état de l'entrée RD1 n'est prise en compte par l'analyseur qu'aux moments des exécutions des lignes de programme d'attribut "Filter-in Trace" (celles qui ont un "carré"), c'est à dire aux moments des affectations de RC13 et RC14.

Raccourcis clavier

Debugger Menu	Toolbar Buttons	Hot Key
Run		F9
Halt		F5
Animate		
Step Into		F7
Step Over		F8
Step Out Of		
Reset		F6