



P 1

## Introduction

sommaire - organisation - contexte

P 9

## CPU

Architecture – mapping – variables – démarrage

P 21

## Outil MPLAB X

Description – 1<sup>er</sup> programme - debug

P 28

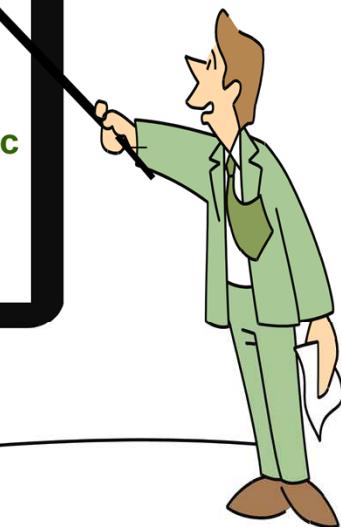
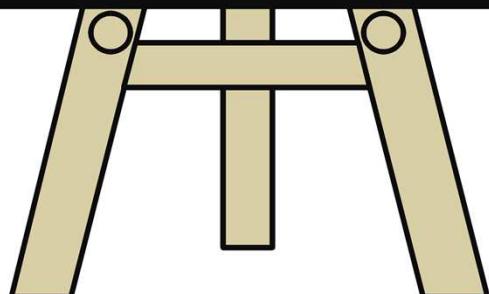
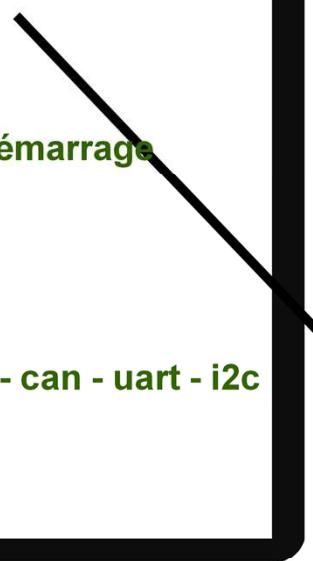
## Périphériques

horloge - ports - interruptions - timers - can - uart - i2c

P 78

## Robot

Contrat 0 ...



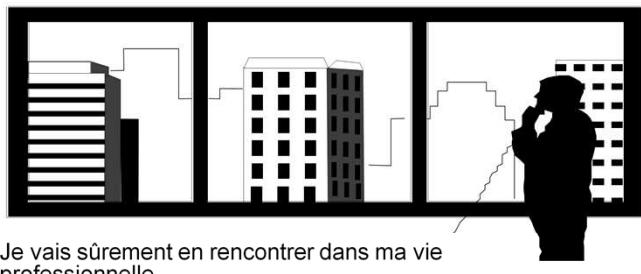
Ce cours s'intitule Projet Pilotage Robot, même si son contenu peut vous paraître éloigné de la réalisation pratique d'un robot, il vous donne les bases informatiques nécessaires pour concevoir votre robot. Dans la mesure du possible les TP auront été pensés « robot » surtout les TP avancés, les premiers TP étant plus pédagogiques. En clair si vous faites tous les TP avancés vous aurez le logiciel correspondant au contrat 0 du projet robot.

notes



## ○ Avantages

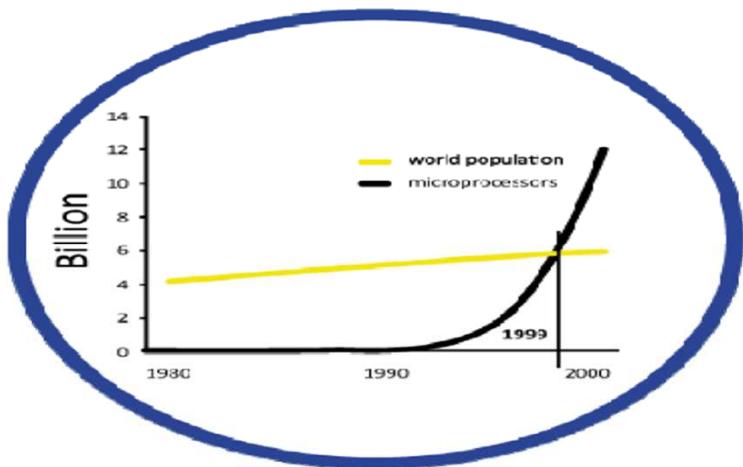
- Flexibilité
- Coût
- Simplicité du Design
- Encombrement



Je vais sûrement en rencontrer dans ma vie professionnelle

## ○ Un marché

- En explosion
- Omniprésent



Inutile d'insister sur le fait que les microcontrôleurs sont omniprésents dans les systèmes intelligents. Vous serez amenés à les côtoyer à divers niveaux: programmation, marketing, chef de projet... IL EST INDISPENSABLE COMPTE TENU DE VOTRE **CHOIX DE FORMATION** QUE VOUS SACHIEZ CE QU'EST UN MICROCONTRÔLEUR.

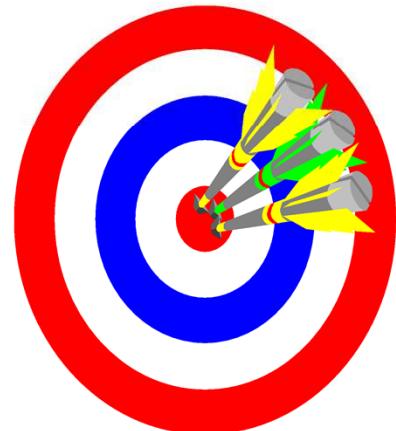
notes





## ○ Se familiariser avec

- Architecture PIC
- Le C embarqué
- Les algorigrammes
- Les périphériques utiles au robot
- Le Debug en simulation



## ○ Métriques

- Test écrit de 1H30
- Rattrapage écrit



Le cours sera orienté programmation en langage C embarqué notamment écriture des drivers (pilotes) des différents périphériques. Durant les TPs l'accent sur mis sur les algorigrammes (description de la solution sur papier) et le debug de vos applications en simulation sur ISIS.

Comme pour toute acquisition de connaissance il est bon d'avoir une note de pratique et une note de théorie. Comme la pratique, vous devez l'acquérir dans la réalisation du robot, il ne sera donné ici qu'une note de théorie via un test écrit de 1H30 avec bien sûr la possibilité d'un rattrapage. Je rappelle que lors du rattrapage la note est écrêtée à 10.





## ○ Livres et site

- **PICmicro 18C MCU Family Reference Manual ( DS39500A )**
- **Programming and Customizing PICmicro Microcontrollers**  
Mike Predko
- [www.microchip.com/compilers/MPLAB](http://www.microchip.com/compilers/MPLAB)

## ■ Microchip Developer's Help Center:

- <http://microchip.wikidot.com>

## ■ Microchip Forums

- <http://www.microchip.com/forums>

Vous trouverez sur internet de nombreux exemples de programmation de microcontrôleurs PIC qui sont très répandus dans l'industrie.





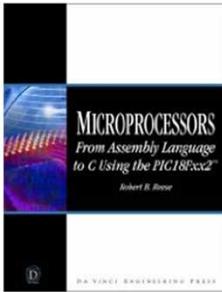
## Advanced PIC Microcontroller Projects in C

### From USB to RTOS with the PIC18F Series

by Dagan Ibrahim

ISBN-10: 0750686111

ISBN-13: 978-0750686112



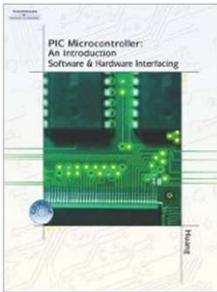
## Microprocessors

### From Assembly Language to C Using the PIC18Fxx2

by Robert B. Reese

ISBN-10: 1584503785

ISBN-13: 978-1584503781



## PIC Microcontroller

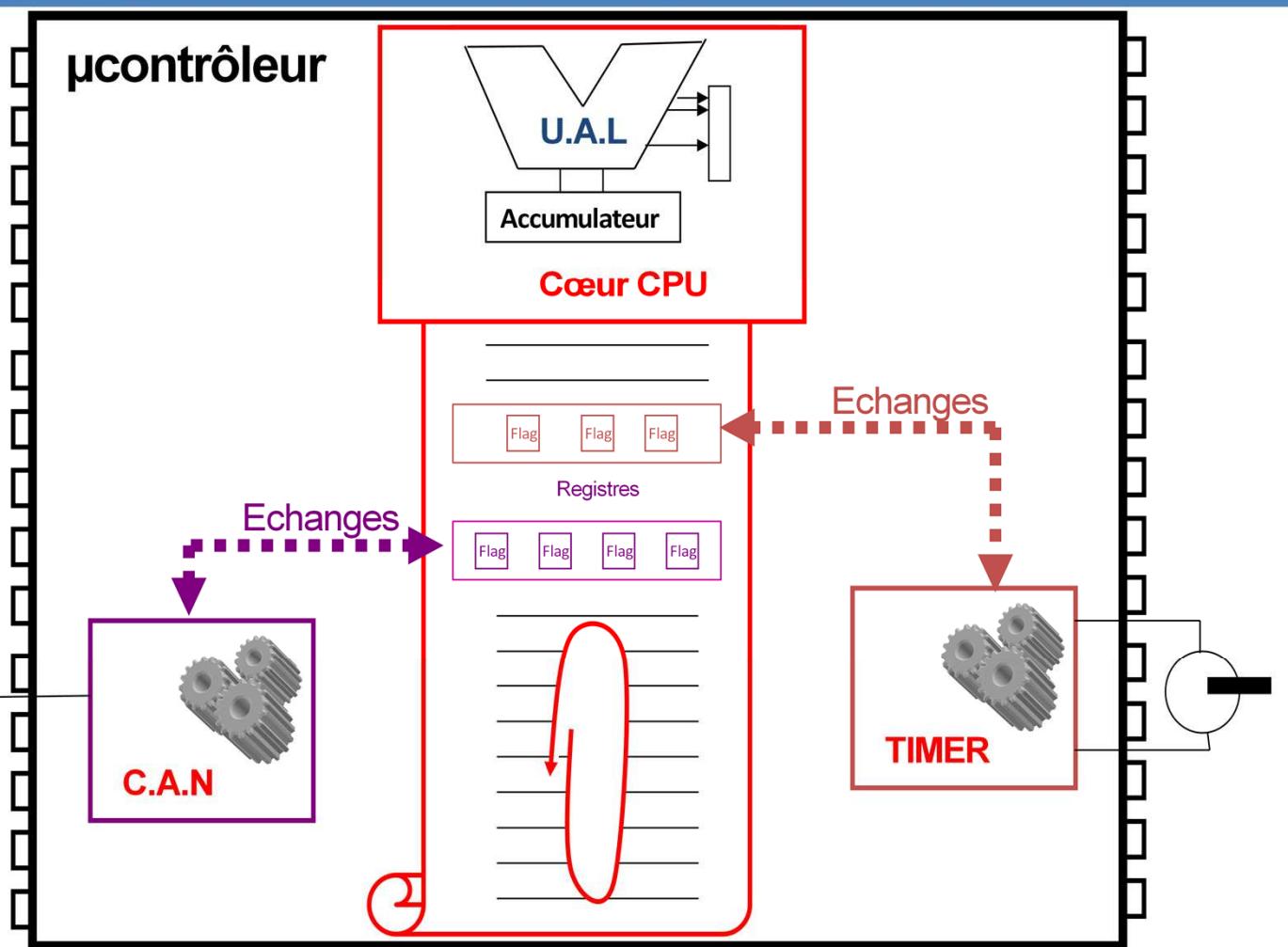
### An Introduction to Software & Hardware Interfacing

by Han-Way Huang

ISBN-10: 1401839673

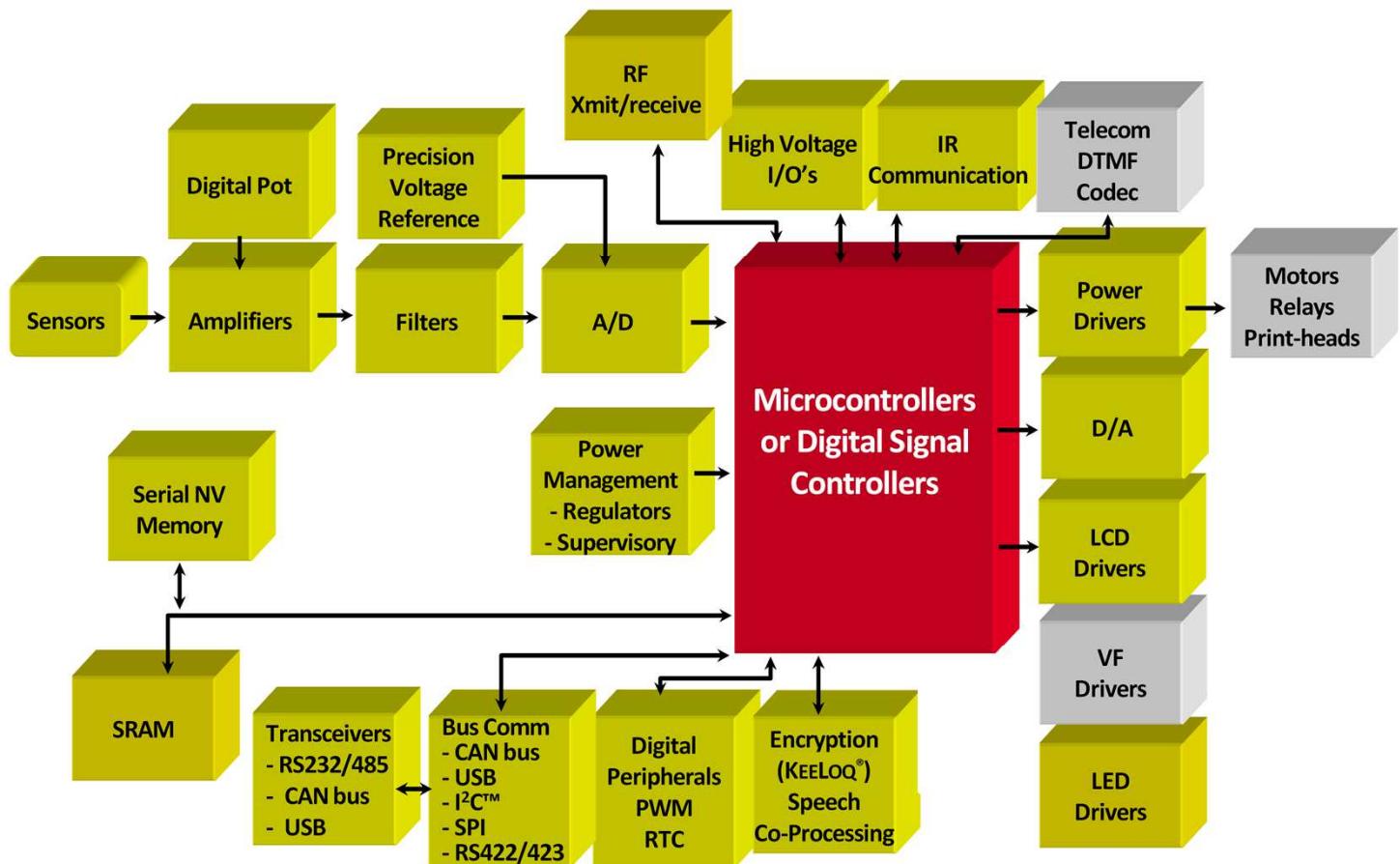
ISBN-13: 978-1401839673

notes



Bien comprendre la notion de cœur CPU et de périphériques. Le cœur ou CPU est uniquement une unité de calcul associée à une unité d'exécution du logiciel. Les périphériques sont des unités matérielles qui tournent en parallèle avec le logiciel exécuté par la CPU. Les périphériques permettent donc de diminuer énormément la charge CPU, ne pas hésiter à les utiliser. Les échanges entre ces unités matérielles et le logiciel (CPU) se font via des registres de contrôle (ou registres de fonction). Donc pour écrire un driver (pilote) de périphérique il faut réunir la liste des registres de contrôle le concernant et lire les data-sheets pour comprendre la signification de chaque flag de contrôle du registre. Les drivers s'écrivent en 2 étapes, d'abord une étape d'initialisation dans le mode de fonctionnement qui nous intéresse puis une phase d'utilisation, lecture ou écriture de résultats.

notes

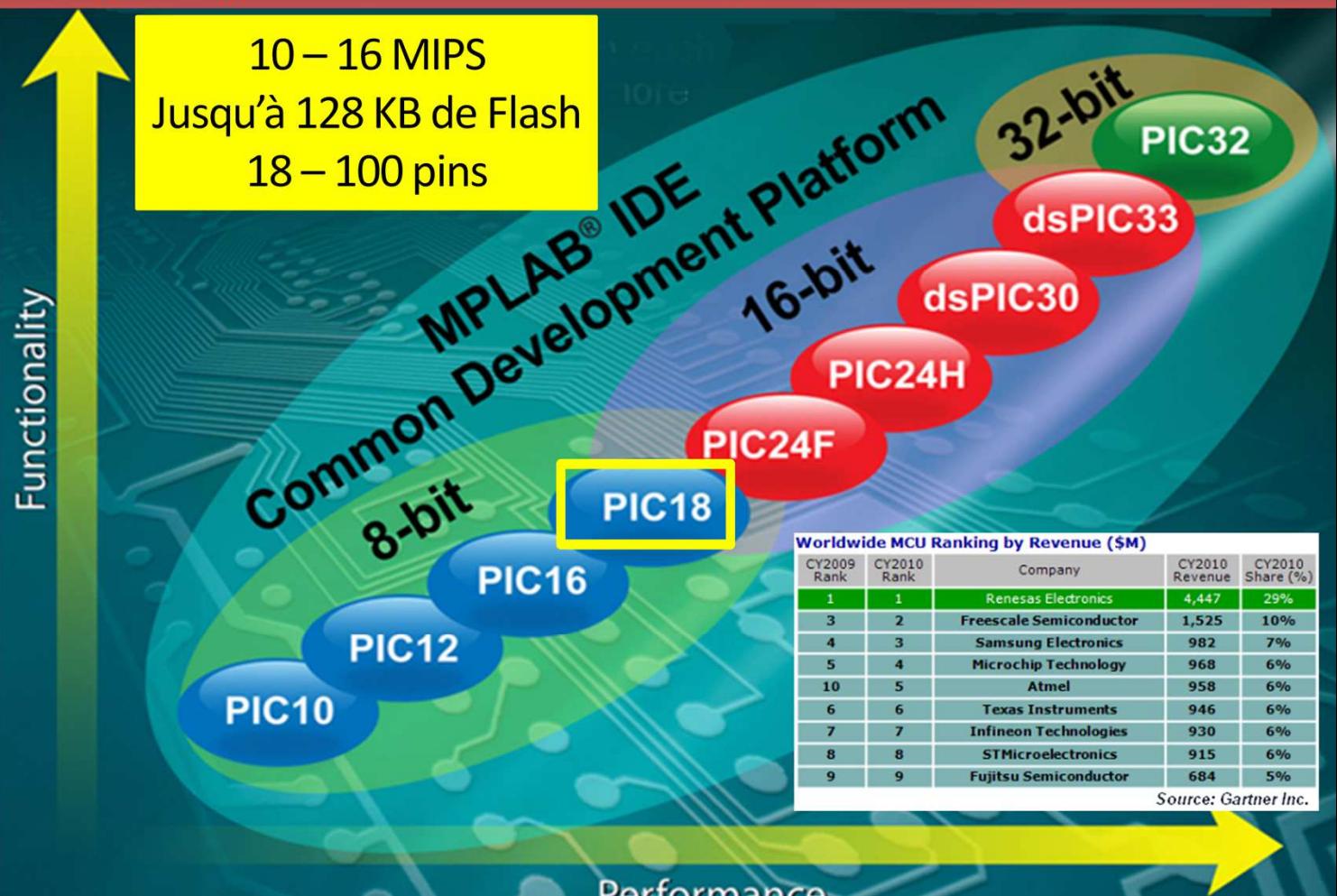


Ce transparent montre un système embarqué typique avec toute la chaîne de traitement du signal. Le monde réel commence par un capteur (température, pression, mouvement, etc ...). Ensuite, le signal est amplifié, filtré, converti en numérique pour être traité par le microcontrôleur. Un microcontrôleur ne peut traiter que des données numériques (0 ou 1) alors la plupart des capteurs, mesurant le monde réel, fournissent une information analogique d'où la nécessité d'une chaîne de traitement. Il y a des blocs de gestion d'alimentation, de circuits d'interfaces, de configuration de bus, de communication. Le microcontrôleur est entouré de toutes parts par des circuits analogiques et cela va de soi le monde réel est analogique. Il faut des circuits de gestion, de communication, de conversion pour passer du monde réel au monde virtuel binaire que contrôlent les microcontrôleurs

La société MicroChip s'est fait une spécialité de ces circuits d'interface avec le monde réel. Les blocs en jaunes montrent les fonctions fournies par MicroChip, fonctions intégrées à un microcontrôleur ou externes ou les 2.

Vous pouvez voir comment la stratégie de Microchip permet de couvrir tout l'environnement des microcontrôleurs, ces derniers compris. Cette société fournit des solutions complètes à vos problèmes de systèmes embarqués. Des solutions pas forcément les plus performantes, mais souvent les plus économiques, elle s'est spécialisée dans le bas coût.

notes



Worldwide MCU Ranking by Revenue (\$M)

CY2009 Rank	CY2010 Rank	Company	CY2010 Revenue	CY2010 Share (%)
1	1	Renesas Electronics	4,447	29%
3	2	Freescale Semiconductor	1,525	10%
4	3	Samsung Electronics	982	7%
5	4	Microchip Technology	968	6%
10	5	Atmel	958	6%
6	6	Texas Instruments	946	6%
7	7	Infineon Technologies	930	6%
8	8	STMicroelectronics	915	6%
9	9	Fujitsu Semiconductor	684	5%

Source: Gartner Inc.

Microchip Technology Corp. a introduit en 1989 son 1<sup>er</sup> PIC (Peripheral Interface Controller) 8-bit, possédant quelques pins IOs, 1 Timer, quelques 100s bytes ROM pour le programme, une petite RAM pour les données. Plus tard, Microchip devient le premier fabricant de microcontrôleurs 8-bit au monde.

Il a été décidé d'utiliser un microcontrôleur PIC 8 bits sur le robot. Ce sont des microcontrôleurs très répandus sur les petits systèmes comme les télécommandes avec une grande diversité de performances et fonctionnalités. Leurs coûts sont les plus bas du marché pour des performances supérieures ou équivalentes et leurs outils de développement sont téléchargeables gratuitement sur le web. La société MicroChip est aujourd'hui au 4<sup>eme</sup> rang mondial du marché des microcontrôleurs.

notes



- Pourquoi ce cours s'appelle Projet pilotage robot ?
- Qu'est ce le debug ?
- Qu'est ce qu'un périphérique ?
- Qu'est ce qu'un driver ou pilote ?
- Qu'est ce qu'une CPU ?
- Qu'est ce qu'un microcontrôleur 8 bits ?
- Rôle de la chaîne de traitement dans un système embarqué ?
- Pourquoi Microchip ?





P 1

## Introduction

sommaire - organisation - contexte

P 10

## CPU

Architecture – mapping – variables – démarrage

P 21

## Outil MPLAB

Description – 1<sup>er</sup> programme - debug

P 28

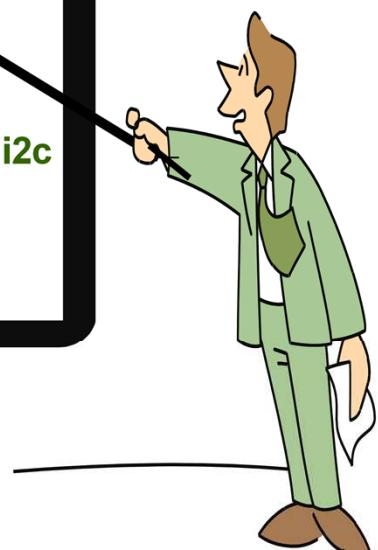
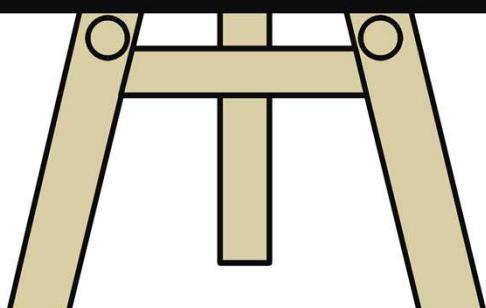
## Périphériques

horloge - ports - interruptions - timers - can - uart - i2c

P 78

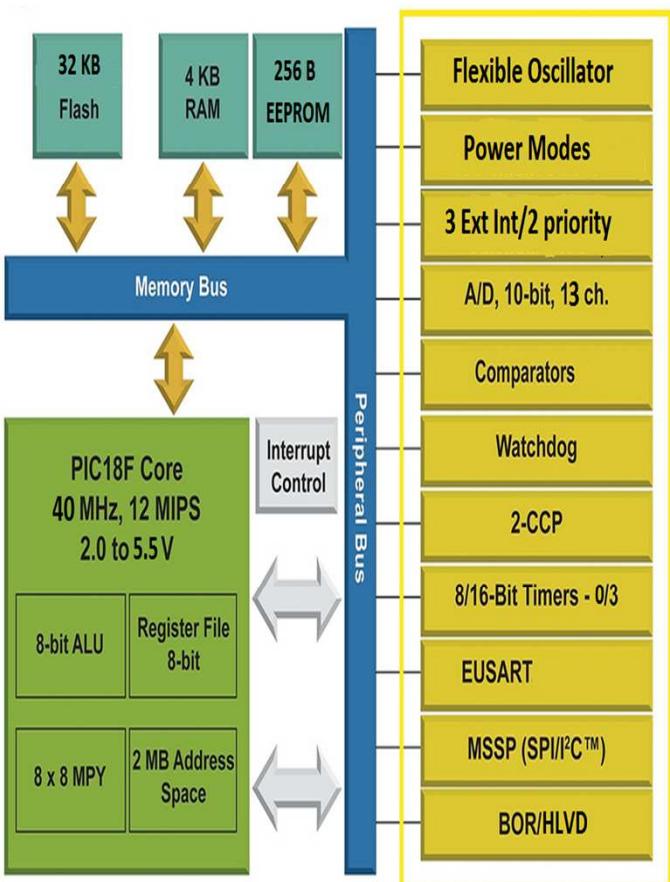
## Robot

Contrat 0 ...





## ○ Bloc diagramme



## ○ Caractéristiques

- Microcontrôleur RISC
- Architecture Harvard
- Instructions 16 bits
- Données 8 bits
- Pipeline 2 étages
- Pile matérielle 31 mots
- Programmation série ( ICSP )
- Broches multiplexées
- 40 Mhz max

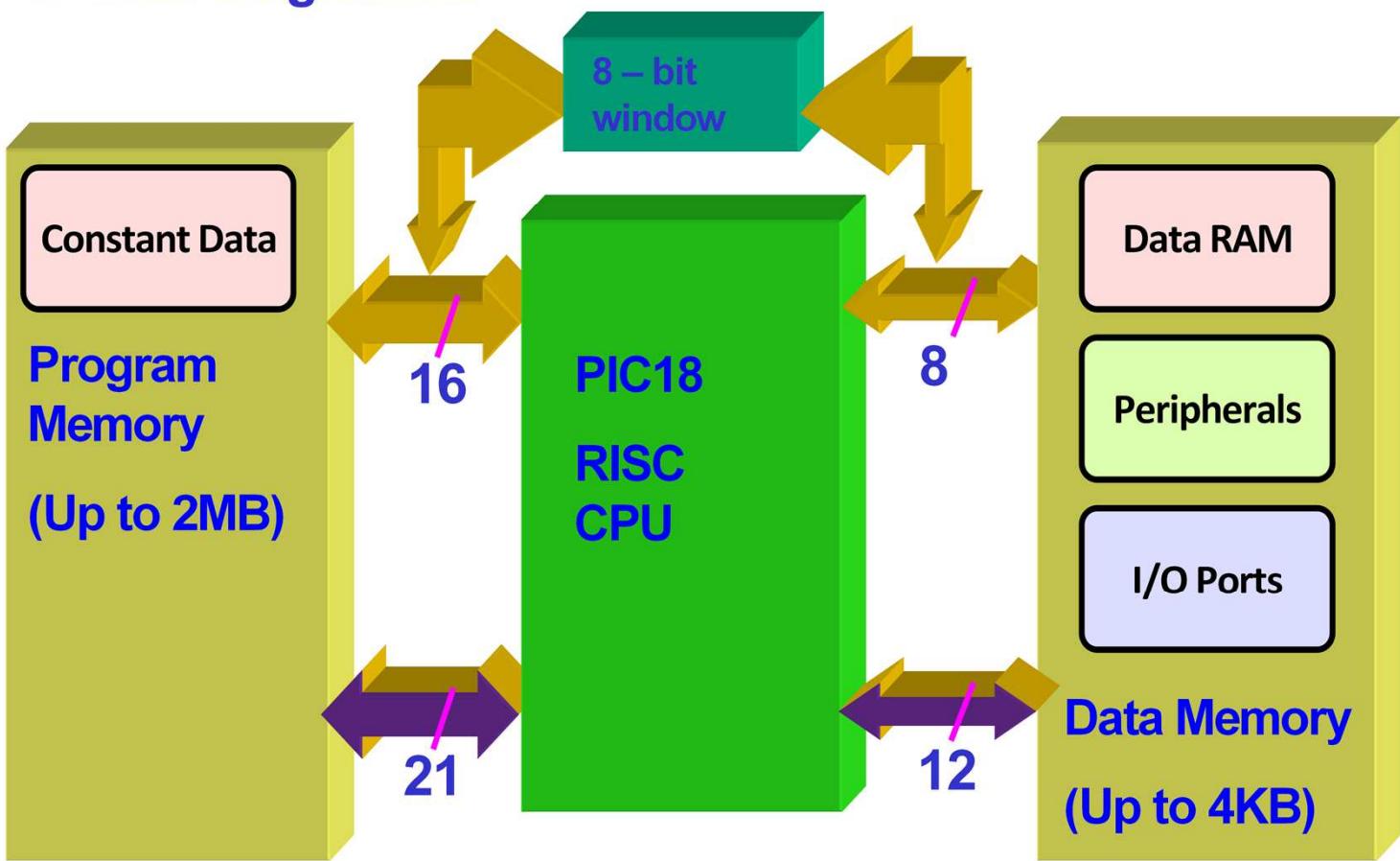
Il s'agit d'une architecture RISC c'est-à-dire avec un jeu d'instruction réduit: 77 instructions de 16 bits. L'architecture est Harvard, c'est-à-dire avec 2 espaces mémoire séparés programme/ donnée ce qui augmente les performances par rapport à l'architecture Von Neumann classique et autorise des largeurs de bus différentes entre programme et donnée: 8 bits données, 16 bits instructions. Un Pipeline de 2 étages (Fetch , Execute) permet une exécution des instructions en 1 cycle dans la plupart des cas, grosse différence par rapport aux architectures CISC qui nécessitent elles plusieurs cycles pour exécuter une instruction. La pile de 31 mots est dite matérielle, car non mappée en RAM. Le chargement du programme en mémoire et son debug se fera via une liaison série spécialisée ICSP (In-Circuit Serial Programming). Cette liaison série occupe 2 pattes du microcontrôleur RB6 et RB7, attention ces pattes ne sont plus disponibles pour l'application. Le circuit possède 28 broches, mais chaque patte a plusieurs fonctionnalités, jusqu'à 3 fonctions différentes. Une PLL intégrée permet une fréquence d'horloge maximale de 40 MHz. Le « F » dans la référence du composant désigne la mémoire Flash interne de 32 ko. Il y a 3 types de mémoire dans le PIC18: Flash, RAM, EEPROM. De nombreux périphériques viennent compléter ce tableau, on aura l'occasion de revenir sur ces périphériques dans les séances suivantes.

notes





## ○ Bloc diagramme

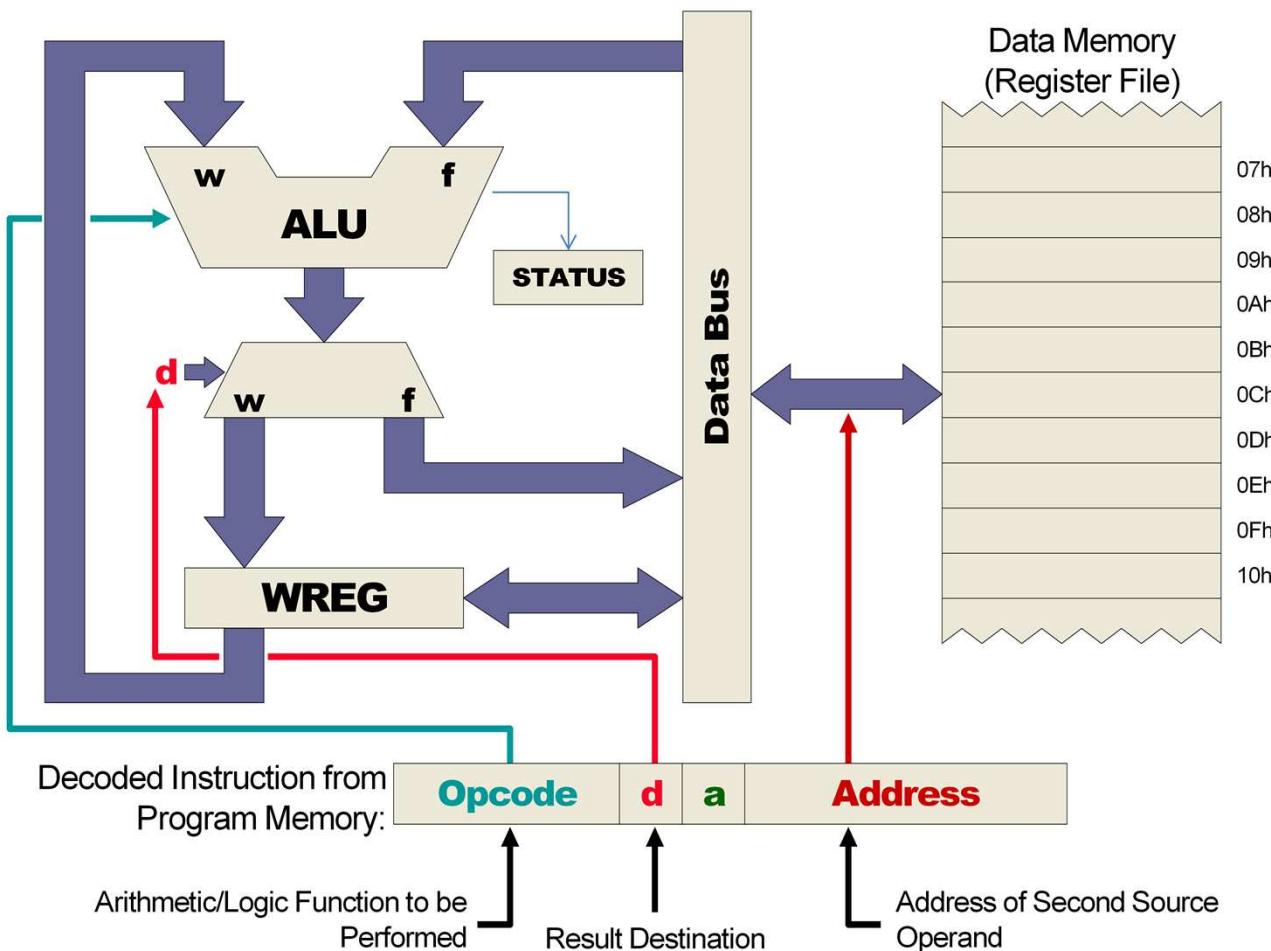


Ce transparent montre l'architecture Harvard avec les 2 espaces mémoires séparés et la taille des différents bus. En fait pour être précis il s'agit d'une architecture Harvard modifiée, car une passerelle existe entre la mémoire donnée et la mémoire programme. Cela permet par exemple de lire des constantes en mémoire programme et de les recopier en mémoire RAM pour initialiser des variables. Si vous déclarez une variable « rom const int x = 25; » elle sera localisée en flash, et sera lue via des instructions « table read »

Tous les périphériques, leurs registres de contrôle sont accessibles en RAM comme des variables de type char.

notes





Le PICmicro présente certains avantages distincts par rapport aux autres processeurs RISC, notamment dans sa mise en œuvre du fichier de registre. Dans un processeur RISC traditionnel, le fichier de registre et les données de RAM sont deux entités distinctes. Les seules opérations que vous pouvez effectuer sur les données de la RAM sont des « Load » et des « Store ». Toute l'arithmétique ou la manipulation logique des données doit être effectuée dans le fichier de registre. Cela signifie que les données doivent d'abord être chargées à partir de la RAM de données dans le fichier de registre, opéré, puis stockées en données RAM.

À cause du nombre important de « load » et « store » requis par les programmes, un grand fichier de registres est souhaitable, afin de conserver les données pour des traitements ultérieurs sans avoir besoin de les sauvegarder sans cesse en RAM. Le PICmicro pousse ce concept à sa limite en ne faisant aucune différence entre les données de RAM et les registres. Strictement parlant, le PICmicro n'a pas de données de RAM, mais dispose d'un fichier de registre énorme, ce qui est l'idéal.

En raison de cet arrangement, toute instruction qui peut fonctionner sur un emplacement d'un registre peut fonctionner sur n'importe quel emplacement du fichier de registre. Cela signifie que le PICmicro fait rarement usage d'instructions à vocation unique, comme celle qui efface uniquement le bit de retenue dans le registre de statut. Au lieu de cela, il y a une instruction à usage général qui peut effacer n'importe quel bit à n'importe quel endroit du fichier de registre. La plupart des instructions suivent ce modèle de fonction à usage général.

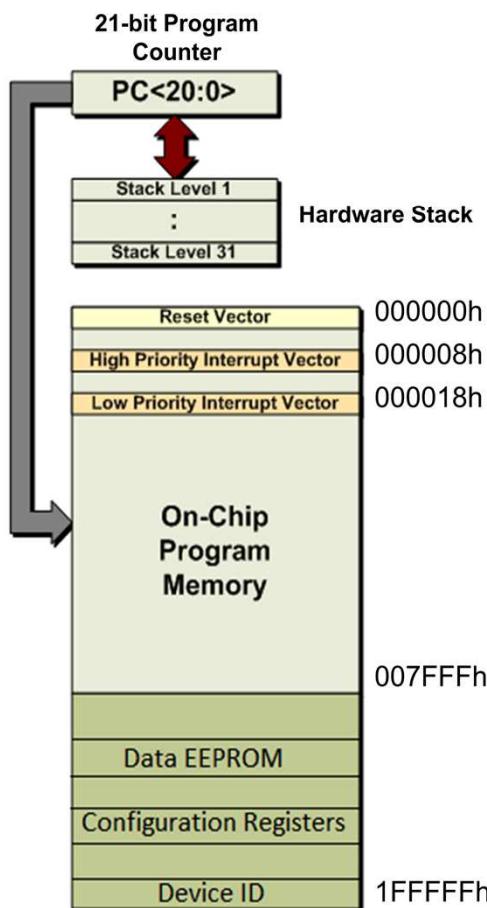
C'est aussi le moment de souligner la relation entre le registre W et le fichier de registre. La plupart des instructions telles que l'ajout de deux nombres ont besoin de deux opérandes. Dans ces cas, l'un des opérandes doit être dans le registre W, tandis que l'autre peut venir de n'importe où dans le fichier de registre. Lorsque l'opération est effectuée, le résultat peut être stocké dans l'un des deux endroits: le registre W ou le fichier registre, d'où l'autre opérande est venu. Ceci est déterminé par le «d» (destination) dans l'instruction.

Le registre STATUS reflète un certain nombre de propriétés du résultat issu du calcul précédent effectué par l'ALU: Négative, Overflow, Zero, Digit Carry, Carry.

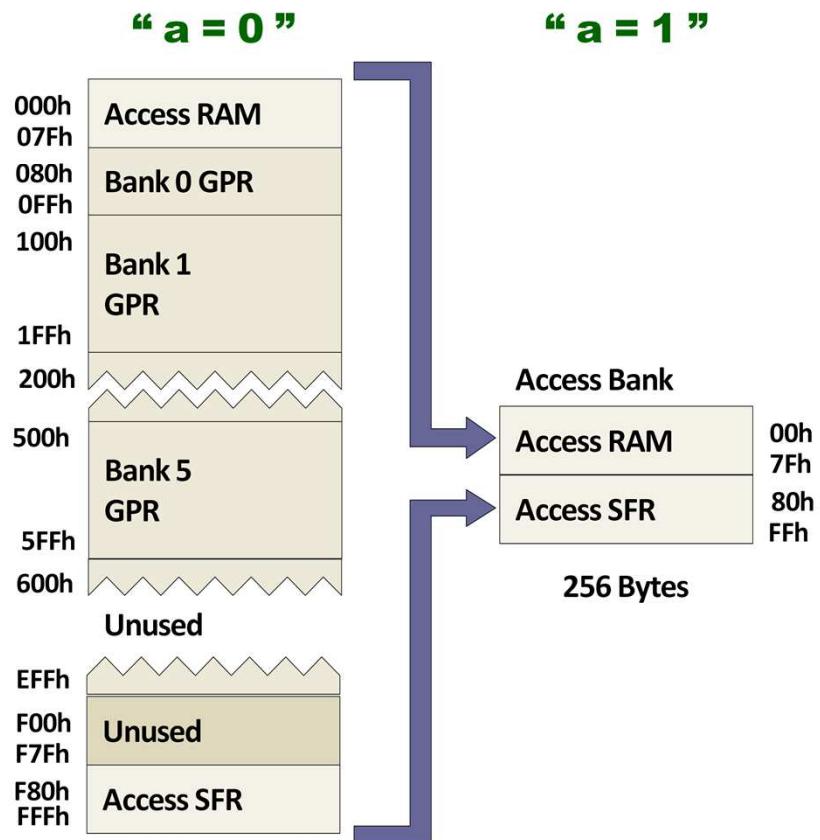
notes



## ○ Programme



## ○ Donnée PIC18F2520: Register File Map



La capacité d'adressage de la mémoire programme de la famille PIC18 est de 2 Mo , sur le PIC18F2520 seuls 32 ko de mémoire Flash sont utilisés. Notez le vecteur de reset à l'adresse 0 ainsi que les 2 vecteurs d'interruption 8h et 18h. Le vecteur de reset est la première instruction à s'exécuter après un reset. Généralement c'est une instruction « goto » vers le « main » ou vers un programme de « startup » se situant quelque part dans la mémoire. Notez que, quel que soit l'interruption on déarrera toujours l'exécution du sous-programme d'interruption à partir de l'adresse 8h ou 18h. Noter la pile de 31 mots , cette pile matérielle est utilisée uniquement pour stocker l'adresse de retour d'un appel de routine.

Une zone EEPROM existe, mais pas adressable directement, il faudra passer par des registres de contrôle. Les bits de configuration sont en dehors de l'espace de code exécutable, on les positionne via des `#pragma config` En début du main().

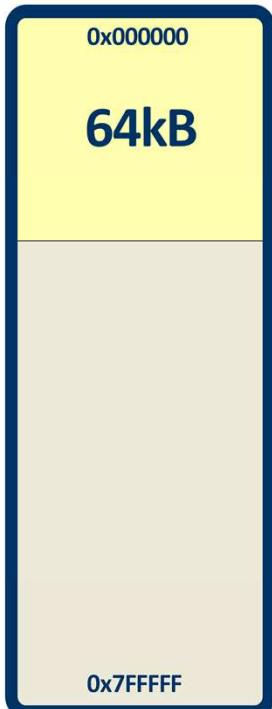
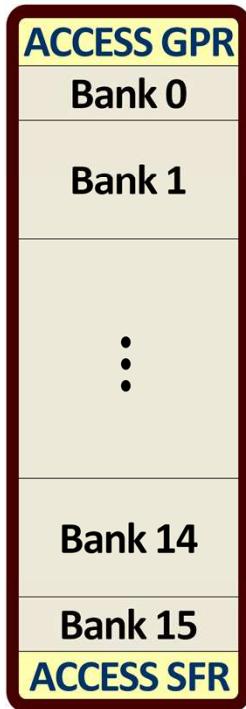
La capacité d'adressage de la mémoire donnée est de 4 ko maximum, mais sur le PIC18f2520 seul 1 Ko est utilisé. Cette mémoire est partitionnée en 4 pages de 256 o. Il existe un registre spécial appelé Bank Select Register (BSR pour faire court), qui est utilisé pour indiquer au processeur quelle page est active.

Il y a une fonctionnalité supplémentaire, et qui est la page d'accès, qui se compose de la moitié inférieure de la page 0 et de la moitié supérieure de la page 15. Ces régions peuvent être accessibles directement quelque soit BSR en positionnant le bit d'accès ( a ) dans les instructions (Mode Access Bank).

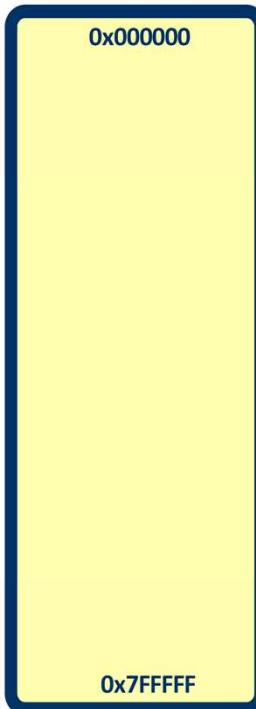
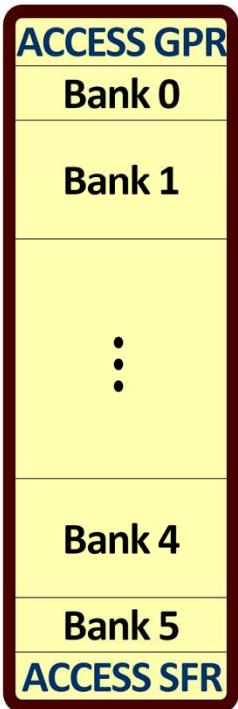




## ○ Modèle Small

Data Memory  
(RAM)Program Memory  
(Flash)

## ○ Modèle Large

Data Memory  
(RAM)Program Memory  
(Flash)

Dans l'outil de développement MPLAB, on peut choisir parmi 2 modèles de placement par défaut: modèle small ou large. Le modèle small place par défaut les variables dans la page ACCES GPR (128 o), la commutation entre les différentes pages est désactivée. Le programme doit tenir dans 64 Ko. C'est le modèle qui permet l'accès le plus rapide aux données. Si les données du programme dépassent 128 o ou que la taille du programme dépasse 64 Ko alors il faut passer en modèle mémoire Large. Attention les librairies utilisent le modèle mémoire Large donc vous aurez des warnings si vous utilisez des fonctions de la librairie en modèle small. Le modèle mémoire se sélectionne via le menu Project/Buid Options, sélectionner l'onglet MPLAB C18 puis la catégorie Memory Model. On peut outrepasser ce placement par défaut via les déclarations FAR et NEAR.



	rom	ram
near	Adresse mémoire programme < 64K	Dans la page Acces
far	Partout dans la mémoire programme	Partout dans la mémoire donnée

## C18 default : far ram

Un placement par défaut existe suivant le modèle mémoire choisi, mais on peut outrepasser ce placement en ajoutant **near** ou **far** dans les déclarations de variables

Far Ram => Données stockées dans la mémoire paginée, commutation de page nécessaire pour accéder aux données

Near Ram => Données dans la page Acces. Utile pour les variables globales.

Far ROM data => Données résidantes en mémoire programme à une adresse qui peut être supérieure à 64 K

Far ROM pointer => Pointeur pouvant accéder en mémoire programme en dessous et au-dessus de 64 K

Near ROM data => Donnée résidant en mémoire programme à une adresse inférieure à 64 K

Near ROM pointer => Pointeur pouvant accéder en mémoire programme uniquement aux adresses inférieures à 64 K

Le compilateur C18 ne stocke pas les variables déclarées avec le mot clé **const** en mémoire programme. Si vous voulez que vos constantes soient en mémoire programme, vous devez utiliser le qualificatif **rom**. Le mot clé **const** engendre un traitement de type variable initialisée. Cela signifie qu'elles seront stockées en mémoire programme et ensuite copiées durant le « runtime code » ( le morceau de code fourni par le compilateur qui se lance entre le reset et l'exécution du main, aussi appelé startup code ).

Je vous rappelle qu'on travaille dans les systèmes embarqués donc on cherche toujours à optimiser vitesse et taille du code. Il est donc recommandé de choisir le modèle small quitte à déplacer les variables les moins accédés avec le qualificatif **far**. Bien sûr si on utilise les fonctions de la librairie ou si la taille du code dépasse 64 Ko on est obligé de passer en modèle Large.

Exemples:

```
char a;                                //placement selon le modèle mémoire Small ou Large
rom char b=6;                            // b est en zone programme (FLASH) modifiable
ram near char c;                        // c est en ACCES RAM
const rom char d=7;                      // d est en zone programme, non modifiable
char * p;                                // p est en RAM et pointe en RAM
rom near char * q;                       // q est en RAM et pointe en ROM ( FLASH )
```

notes





## ■ Un-initialized Data (.udata) ( .stack )

- `int x;`

## ■ Initialized Data (.idata)

- `int y = 5;`

## ■ Data in program Memory (.romdata)

- `rom int z = 12;`

## ■ Executable content (.code)

- `void foo (void)  
{ ... }`

Le module qui place les variables en mémoire est le linker, pour cela il regroupe les variables par type à l'intérieur de sections prédéfinies.

Les sections sont des zones mémoires dans lesquelles sont regroupés les différents éléments de même type. Elles sont créées par le linker et ont un nom par défaut donné ci-dessus. En règle général, vous n'avez pas besoin d'utiliser le nom des sections dans votre programme à moins de chercher à optimiser où organiser votre code. Les noms précédés d'un point sont ceux utilisés par défaut par le linker quand il place les différents éléments de votre programme dans la mémoire de la cible. Vous pouvez créer vos propres noms de section pour regrouper ensemble des éléments dans la mémoire.

Les données initialisées sont stockées en mémoire programme et copiées en mémoire donnée au runtime ( exécution du fichier starup ).





## General Syntax

```
#pragma sectiontype [sectionname [=address ]]
```

Section Type	Memory Region	Usage
<b>udata</b>	RAM	Static Uninitialized Data (e.g. <b>char x;</b> )
<b>idata</b>	RAM	Static Initialized Data (e.g. <b>char x = 55;</b> )
<b>udata access</b>	ACCESS RAM	Static Uninitialized Data in Access RAM
<b>idata access</b>	ACCESS RAM	Static Initialized Data in Access RAM
<b>romdata</b>	Program/Flash	Variables and Constants
<b>code</b>	Program/Flash	Executable Code

## Example

```
#pragma romdata eedata_scn = 0xF00000
rom char eedata_values[4] = {0x01, 0x02, 0x03, 0x04};
rom int xee = 0x1234;
```

Bien que le linker soit parfaitement capable de placer les variables dans les sections appropriées, il est parfois avantageux en terme de performance de fournir vous-même cette information. Le **#pragma** rend possible de regrouper des variables dans une section spécifique ou de localiser une variable à une adresse spécifique.

Dans certaines circonstances, il est nécessaire de placer une variable ou une fonction à une adresse spécifique en mémoire. Il faut utiliser cette possibilité avec parcimonie, seulement lorsque cela est vraiment nécessaire, car se faisant on « lie les mains » du linker, rendant plus difficile l'optimisation du code.

notes





My Computer



Local Disk (C:)



Program Files



MPASM Suite



LKR

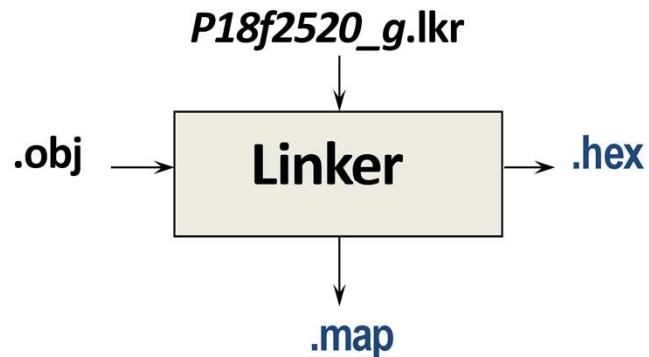
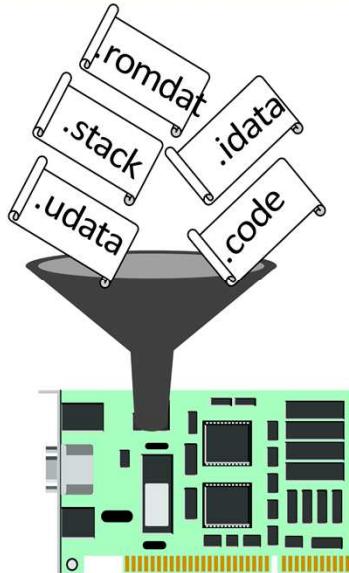


p18f4520\_g.lkr



p18f2520\_g.lkr

Universal Linker Script



Le linker fait appel pour placer les différentes sections en mémoire au linker script ( .lkr ). Ce linker script décrit le mapping mémoire de la cible

Il existe un linker script pour chaque composant PIC18 puisque leur mapping est différent. À l'origine, il y avait un grand nombre de linker script par composant pour tenir compte si on était en debug mode ou en release mode ( pas les mêmes zones mémoires utilisées ) si le jeux d'instruction était standard ou étendu, etc.. Le nouveau linker script universel sélectionne le script approprié à partir de la configuration de MPLAB. Quand on crée un projet, il n'est plus nécessaire d'inclure le fichier de script dans l'arborescence du projet. Il vaut mieux laisser MPLAB choisir automatiquement le linker script universel. La seule fois où vous devrez manuellement inclure le linker script dans l'arborescence du projet est lorsque vous décidez de créer un linker script personnalisé.

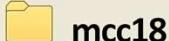
notes



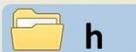
My Computer



Local Disk (C:)



mcc18



p18cxx.h



p18f4520.h

## Exemples d'utilisation

```

PORTAbits.RA0 = 1;
PORTA = 0x01;

val = PORTAbits.AN3;

PORTAbits.RA4 = 1;
PORTAbits.RA5 = 0;
PORTAbits.RA6 = 1;
PORTAbits.RA7 = 0;

PORTAbits.R4_7 = 5;

```

## Extrait PORTAbits déclaration dans p18f2520.h

```

extern volatile near unsigned char PORTA;
extern volatile near union {

```

struct {

```

    unsigned RA0:1;
    unsigned RA1:1;
    unsigned RA2:1;
    unsigned RA3:1;
    unsigned RA4:1;
    unsigned RA5:1;
    unsigned RA6:1;
    unsigned RA7:1;

```

};

struct {

```

    unsigned AN0:1;
    unsigned AN1:1;
    unsigned AN2:1;
    unsigned AN3:1;
    unsigned :1;
    unsigned AN4:1;
    unsigned OSC2:1;

```

};

} PORTAbits;

Definition  
structureNoms de bit  
primaires

Ajouts possible

```

struct {
    unsigned RA0_3:4;
    unsigned RA4_7:4;
};

```

Noms de bit  
secondairesDéclaration  
variable

Le fichier **p18F2520.h** déclare les registres et on peut les utiliser alors simplement dans notre programme, à condition d'y inclure le .h bien entendu, mais comment est défini l'adresse des registres puisque tous les registres ont une adresse en dur imposé par le constructeur. En fait cet adressage est fait dans la librairie **p18f2520.lib** en assembleur.

Extrait:

SFR_UNBANKED0	UDATA_ACS	H'F80'
PORTA		
PORTAbits	RES 1	// Adresse de PORTA = Adresse
de PORTAbits = 0xF80		

Conclusion: Vous ne pouvez pas modifier le nom des registres puisqu'ils sont définis dans la librairie, heureusement ce sont les mêmes que dans les data-sheets. Par contre rien ne vous empêche de modifier le nom des champs de bits à l'intérieur des registres comme le montre l'exemple. À noter que les registres sont octet-accessibles et bit-accessibles, c'est une des forces des microcontrôleurs PIC.

Une variable doit être déclarée volatile quand son contenu est susceptible de changer de façon aléatoire par rapport au déroulement général du programme. Toutes les variables qui peuvent être modifiées autrement que par logiciel (par le hardware ou les interruptions) doivent être volatiles. Volatile empêche les optimisations du compilateur sur les cases mémoires, car souvent l'accès ou l'ordre d'accès dans les registres est imposé, il ne faut pas laisser le compilateur modifier cela pour des raisons d'optimisation. **Conclusion tous les registres doivent être déclarés de type volatile**, puisque le hardware (les périphériques) peut les modifier à tout moment.

notes



Type	Bits	Min	Max
<code>char, signed char</code>	8	-128	127
<code>unsigned char</code>	8	0	255
<code>short, signed short</code>	16	-32768	32767
<code>unsigned short</code>	16	0	65535
<code>int, signed int</code>	16	-32768	32767
<code>unsigned int</code>	16	0	65535
<code>short long, signed short long</code>	24	-8,388,608	8,388,607
<code>unsigned short long</code>	24	0	16,777,215
<code>long, signed long</code>	32	$-2^{31}$	$2^{31} - 1$
<code>unsigned long</code>	32	0	$2^{32} - 1$

Type	Bits	E Min	E Max	N Min	N Max
<code>float</code>	32	-126	128	$2^{-126}$	$2^{128} \cdot (2 - 2^{-15})$
<code>double</code>	32	-126	128	$2^{-126}$	$2^{128} \cdot (2 - 2^{-15})$

Remarque: sur ce microcontrôleur 8 bits, un int est sur 16 bits, un long sur 32 bits. Le type bit n'existe pas.

notes





<code>++</code>	post ou pré-incrémentation
<code>--</code>	post ou pré-décrémentation
<code>&amp;</code>	adresse de la variable
<code>*</code>	adressage indirect (pointeur)
<code>-</code>	négation
<code>~</code>	complément à 1
<code>!</code>	inversion
<code>Sizeof</code>	taille en octets d'une variable
<code>(type)</code>	conversion de type

<code>*</code>	multiplication
<code>/</code>	division
<code>%</code>	reste de la division
<code>+</code>	addition
<code>-</code>	soustraction
<code>&gt;&gt;</code>	décalage à droite
<code>&lt;&lt;</code>	décalage à gauche
<code>&amp;</code>	ET bit à bit
<code> </code>	OU bit à bit
<code>^</code>	XOR bit à bit

<code>&lt;</code>	plus petit que
<code>&gt;</code>	plus grand que
<code>&lt;=</code>	plus petit ou égal
<code>&gt;=</code>	plus grand ou égal
<code>==</code>	identique
<code>!=</code>	différent
<code>&amp;&amp;</code>	ET logique
<code>  </code>	OU logique

Concernant les opérateurs, leur utilisation est plus diversifiée en C embarqué qu'en C standard.

**Exemple1:** lorsque les registres ne sont pas bit-adressable, pour accéder aux flags il faut faire des masques avec l'opérateur « et » pour faire une mise à zéro et l'opérateur « ou » pour faire une mise à « 1 »: registre = registre & masque ou registre = registre | masque.

Exemple:  
 $\text{Reg} = \text{Reg} \& 0xFE$ : mise à 0 du LSB  
 $\text{Reg} = \text{Reg} | 0x80$  : mise à 1 du MSB

**Exemple2:** int var1;  
 char var2;

$\text{var2} = (\text{char}) \text{var1}$  : var2 = poids faible de var1  
 $\text{var2} = (\text{char}) \text{var1}>>8$  : var2 = poids fort de var1

**Exemple3:** Attention aux formats des variables dans les calculs mathématiques, exemples:

int a,b  
 long p;  $p = a * b$ ; **NOK** résultat tronqué, sur 16 bits

Le format du résultat est le même que celui des opérandes:  
 $p = (\text{long}) a * (\text{long}) b$ ; **OK**

**Exemple4:** utiliser  $x = y >> n$  ( code 50, data 00 ) au lieu de  $x = y / 2^n$  ( code 500, data 150 )

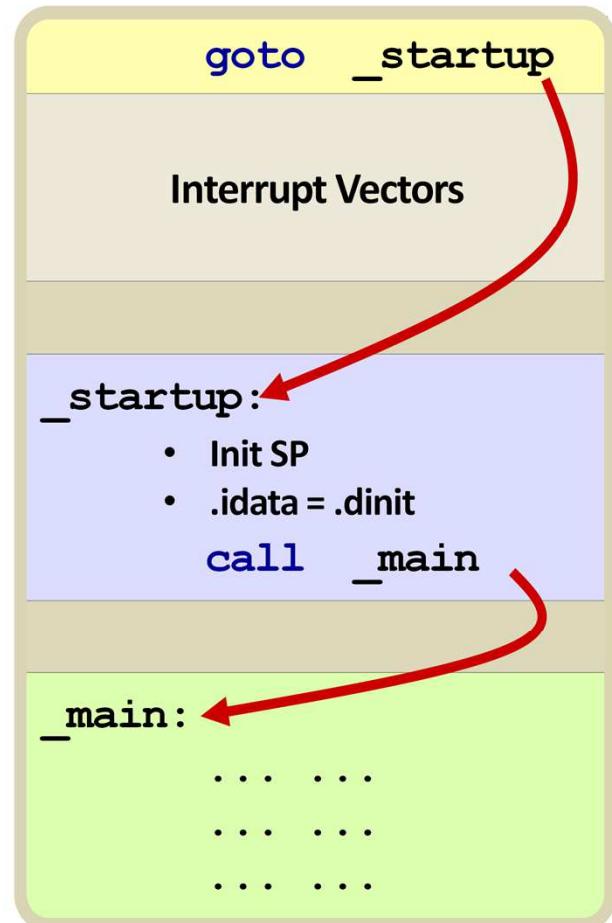
utiliser  $x = y << n$

au lieu de  $x = y . 2^n$

notes



## Program Memory

**Vecteur Reset (Address 0x000000)**

Comme déjà énoncé votre code n'est pas le premier programme à s'exécuter lorsque le composant est mis sous tension. Un branchement est effectué au C startup code.

Le programme démarre toujours à partir de l'adresse vecteur reset et automatiquement lance le fichier de startup **c018i.o** qui initialise la pile logicielle et les variables avant de lancer le **main**. Le fichier startup est contenu en librairie et est référencé dans le linker script (\*.lkr). Plusieurs fichiers de démarrage existent il suffit de changer le nom du fichier dans le linker script pour y avoir accès.

Remarque: la fonction **main** est appelée par le code de démarrage (Startup ou CRT - C Run Time) dans une boucle infinie, il faut donc impérativement placer une boucle infinie en fin de **main** sinon le **main** sera relancé, il ne faut pas sortir du **main**.

Le programme de démarrage par défaut est **c018i.o**. Il existe 2 autres alternatives: **c018i\_e.o** pour le jeu d'instructions étendues et **c018iz.o** qui fait la même chose que **c018i.o** avec en plus la mise à zéro des données de la section **.udata** ( uninitialized data). Donc attention le fichier de démarrage par défaut **c018i.o** n'initialise pas à zéro les variables déclarées dans votre programme. Si vous voulez être ANSI compatible, il faut utiliser **c018iz.o**

**c018.o** est un programme de démarrage qui peut être utilisé si vous n'utilisez pas de variables initialisées (e.g. **int x = 10**). L'étape d'initialisation des variables est sautée ce qui accélère le lancement de votre code.

*notes*



## Projet Pilotage Robot

## CPU: architecture – mapping – variables - démarrage

18f2520\_g.lkr

// File: 18f2520\_g.lkr  
// Generic linker script for the PIC18F2520 processor

```
#DEFINE _CODEEND_DEBUGCODESTART - 1
#DEFINE _CEND_CODEEND+_DEBUGCODELEN
#DEFINE _DATAEND_DEBUGDATASTART - 1
#DEFINE _DEND_DATAEND+_DEBUGDATALEN
```

LIBPATH.

```
#IFDEF _CRUNTIME
#IFDEF _EXTENDEDMODE
FILES c018i_e.o
FILES clib_e.lib
FILES p18f2520_e.lib
```

```
#ELSE
FILES c018i.o
FILES clib.lib
FILES p18f2520.lib
#FI
#FI
```

```
#IFDEF _DEBUGCODESTART
CODEPAGE NAME=page START=0x0 END=_CODEEND
CODEPAGE NAME=debug START=_DEBUGCODESTART END=_CEND PROTECTED
#else
CODEPAGE NAME=page START=0x0 END=0xFFFF
#FI

CODEPAGE NAME=idlocs START=0x200000 END=0x200007 PROTECTED
CODEPAGE NAME=config START=0x300000 END=0x30000D PROTECTED
CODEPAGE NAME=devid START=0x3FFFFE END=0x3FFFFF PROTECTED
CODEPAGE NAME=eedata START=0xF00000 END=0xF000FF PROTECTED
```

**Zone Debug**

**Fichier de démarrage**  
**Librairie C**  
**Librairie SFR**

```
#IFDEF _EXTENDEDMODE
DATABANK NAME=gpre START=0x0 END=0x5F
ACCESSBANK NAME=accessram START=0x60 END=0x7F
#else
ACCESSBANK NAME=accessram START=0x0 END=0x7F
#FI
```

```
DATABANK NAME=gpr0 START=0x80 END=0xFF
DATABANK NAME=gpr1 START=0x100 END=0x1FF
DATABANK NAME=gpr2 START=0x200 END=0x2FF
DATABANK NAME=gpr3 START=0x300 END=0x3FF
DATABANK NAME=gpr4 START=0x400 END=0x4FF
```

```
#IFDEF _DEBUGDATASTART
DATABANK NAME=gpr5 START=0x500 END=_DATAEND
DATABANK NAME=dbgspr START=_DEBUGDATASTART END=_DEND PROTECTED
#else //no debug
DATABANK NAME=gpr5 START=0x500 END=0x5FF
#FI
```

```
ACCESSBANK NAME=accesssfr START=0xF80 END=0xFFFF PROTECTED
```

**Définition mémoire Donnée**

```
#IFDEF _CRUNTIME
SECTION NAME=CONFIG ROM=config
#endif _DEBUGDATASTART
STACK SIZE=0x100 RAM=gpr4
#else
STACK SIZE=0x100 RAM=gpr5
#FI
#FI
```

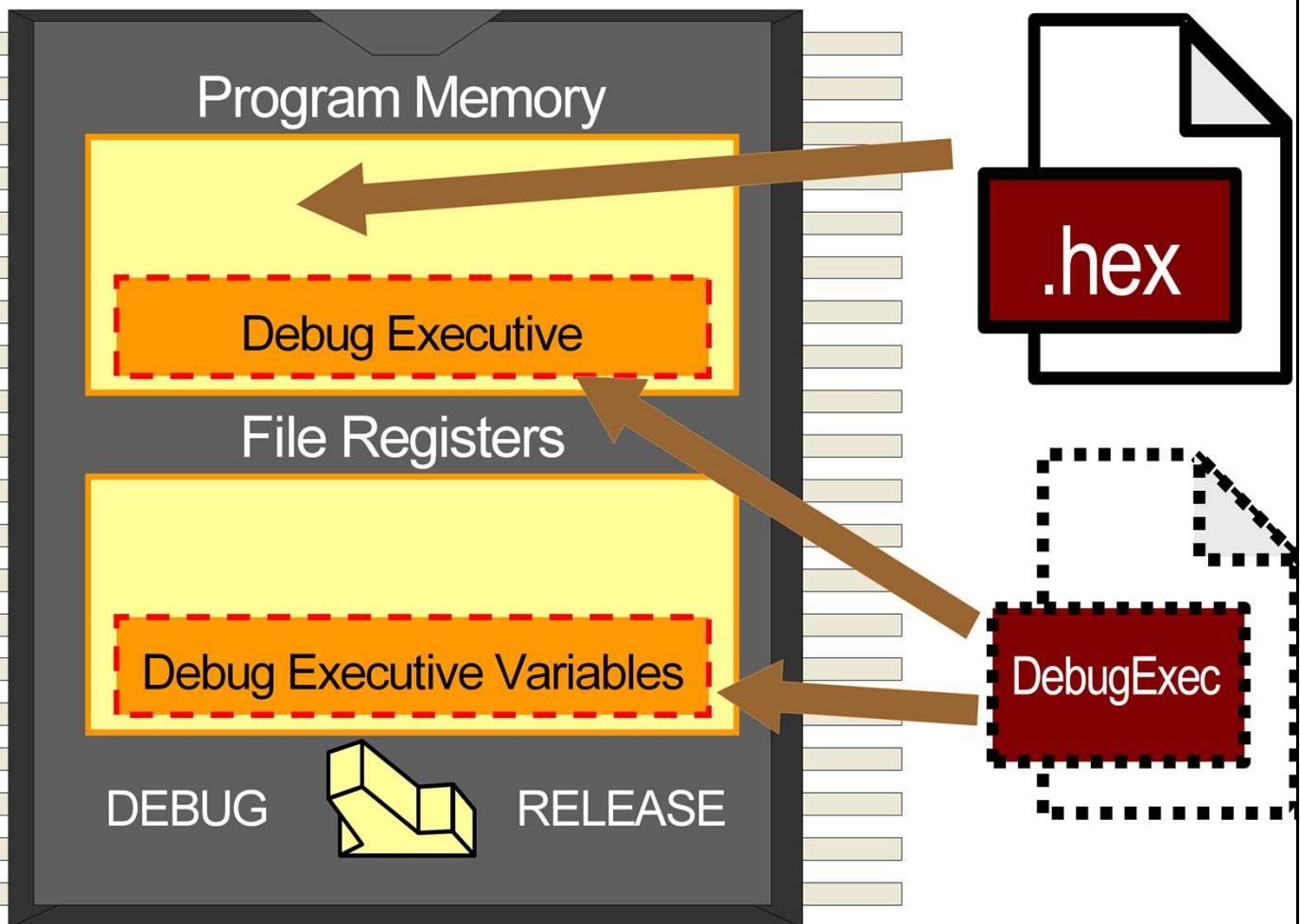
**Taille pile logiciel**

**Définition mémoire programme**

On constate qu'en mode debug une zone mémoire est allouée en zone programme et donnée.

Remarque: le mode étendu fait référence à un jeu d'instruction étendu ( 8 instructions supplémentaires ). L'avantage de ce mode, et donc de ces nouvelles instructions, est de permettre une réduction de la taille du code et de la vitesse d'exécution de 10 % environ.

notes



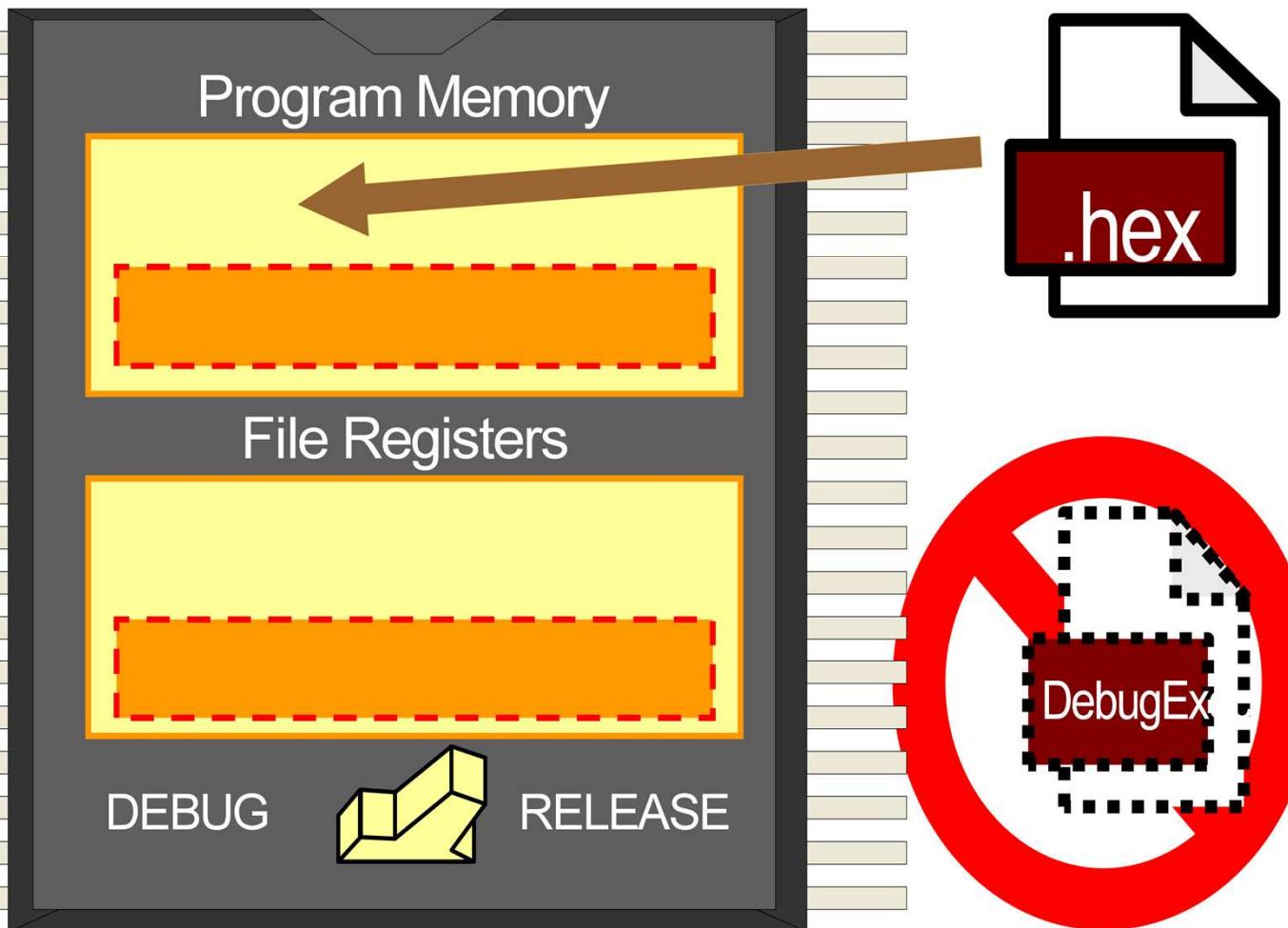
Lors de la mise au point de votre application, vous configurez l'outil de développement en **mode debug**. Cela a pour effet d'allouer un espace supplémentaire en Flash et en RAM pour l'exécutable de debug, nécessaire pour le bon fonctionnement des points d'arrêt, pour visualiser les registres ou les variables via l'outil de développement.

Ce mode debug fonctionne aussi sur la cible. Vous pouvez télécharger votre programme sur la cible et continuer à profiter des points d'arrêt et autres outils de debug. En contrepartie une partie de la mémoire de votre cible est occupée par le logiciel de debug et votre programme a besoin d'une commande externe issue d'un outil de debug pour se lancer.

Si vous voulez que votre programme se lance directement une fois charger sur la cible il faut passer en mode release au niveau de l'outil de développement.

notes





Si vous voulez que votre code s'exécute immédiatement une fois l'alimentation appliquée, vous devez recompiler votre code en mode release.

L'exécutif de debug n'est plus nécessaire, cependant la mémoire réservée pour le programme de debug reste inutilisée. Vous pourriez modifier votre code pour utiliser cet espace, mais ces zones mémoires ne seront alors plus disponibles pour le mode debug et vous empêcheront de debugger ce nouveau code modifié.

notes





## Projet Pilotage Robot

- Faut-il utiliser le périphérique liaison série pour télécharger votre programme ?
- Quels sont les caractéristiques d'une architecture RISC ?
- Quelle est la capacité mémoire du PIC18f2520 ?
- Quel est le modèle mémoire préconisé ? Pourquoi ?
- Quel est le modèle mémoire utilisé dans les librairies du PIC ?
- Le format d'un int est toujours le même quel que soit le microcontrôleur ?
- Je peux modifier le nom des flags SFR ?
- Je peux modifier le nom des SFR ? Pourquoi ?
- A quoi sert l'attribut volatile ? Quand faut-il l'utiliser ?
- Pourquoi faut-il une boucle infinie en fin de main ?
- Il existe toujours un fichier startup en C ?
- L'EEPROM apparaît-elle dans l'espace d'adressage ?
- Différence entre mode debug et release ?
- Quels informations contient le linker script ?





P 1

## Introduction

sommaire - organisation - contexte

P 9

## CPU

Architecture – mapping – variables – démarrage

P 28

## Outil MPLAB

Description – 1<sup>er</sup> programme - debug

P 28

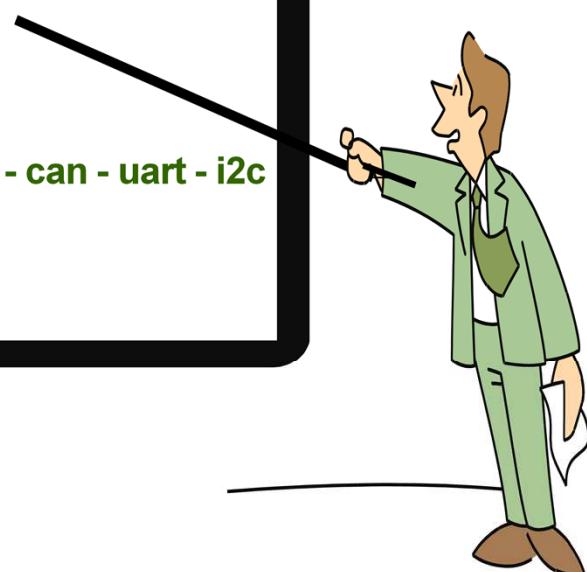
## Périphériques

horloge - ports - interruptions - timers - can - uart - i2c

P 78

## Robot

Contrat 0 ...



notes





# MPLAB X

## Integrated Development Environment

Editor

Project Manager

Language Tools

Source Level Debuggers

Simulators

Emulators

MPASM / ASM  
AssemblersMPLAB® SIM  
Simulator

PICkit™ 3

Microchip Supplied

MPLAB® XC  
C Compilers3<sup>rd</sup> Party

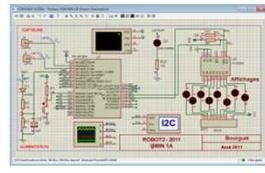
MPLAB® ICD 3

3<sup>rd</sup> Party

PROTEUS

MPLAB® REAL ICE™

User Supplied

3<sup>rd</sup> Party

Starter Kits

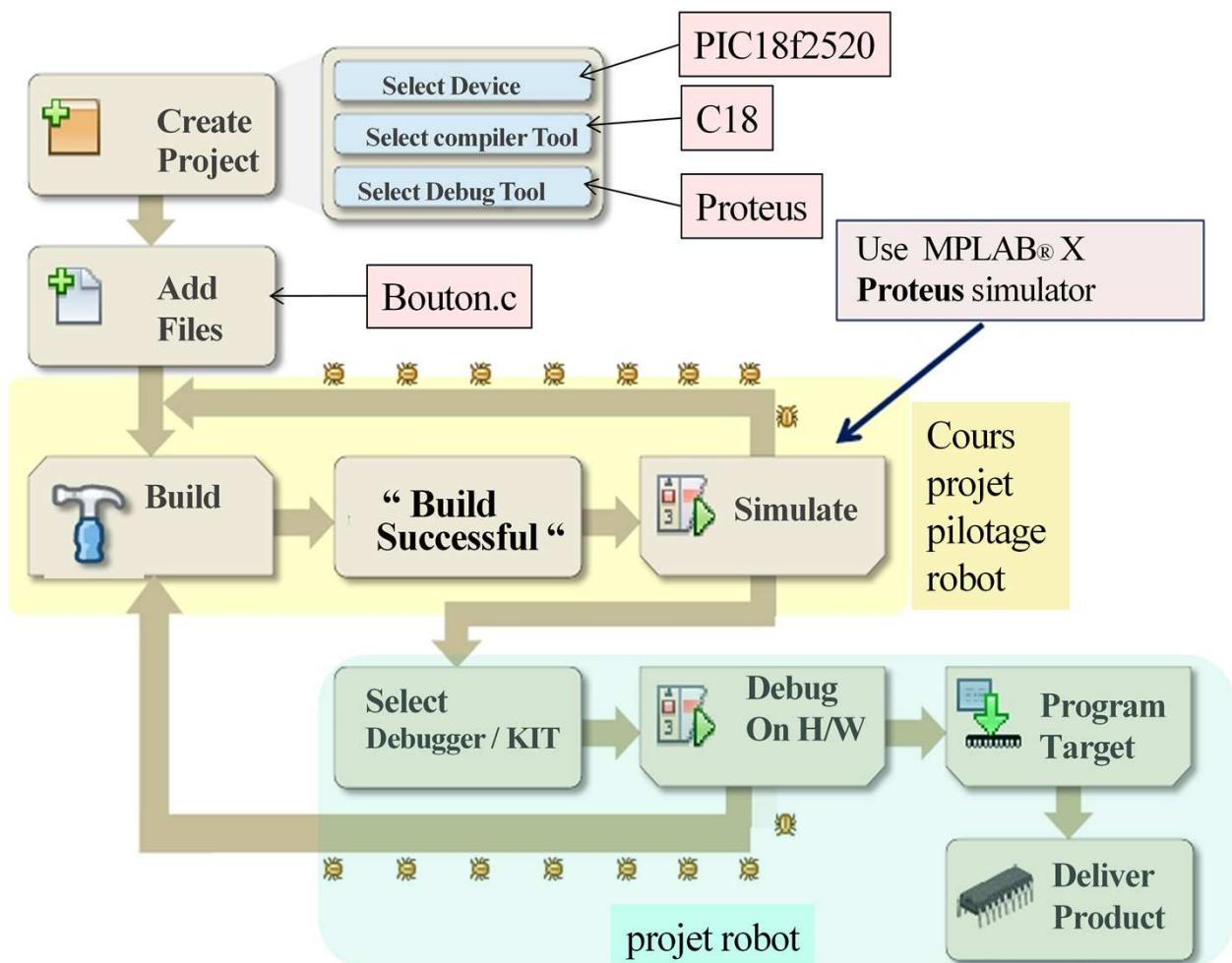
3<sup>rd</sup> Party and Legacy Instruments

L'outil MPLABX peut-être téléchargé gratuitement sur le site de Microchip, une version lite du compilateur « xc8 » est disponible également avec quelques limitations peu gênantes pour se former. Notamment la taille du code généré n'est pas optimisée.

MPLABX offre une passerelle avec le logiciel de saisie et de simulation PROTEUS, c'est ce moyen de debug que nous utiliserons pour ce cours. Le schéma ISIS utilisé ressemble à celui du robot, il vous permettra donc d'avancer dans le développement logiciel de votre robot. Le projet robot vous permettra ensuite d'utiliser les émulateurs tels que le pickit 3 pour télécharger votre programme sur votre cible matérielle et débugger in situ.

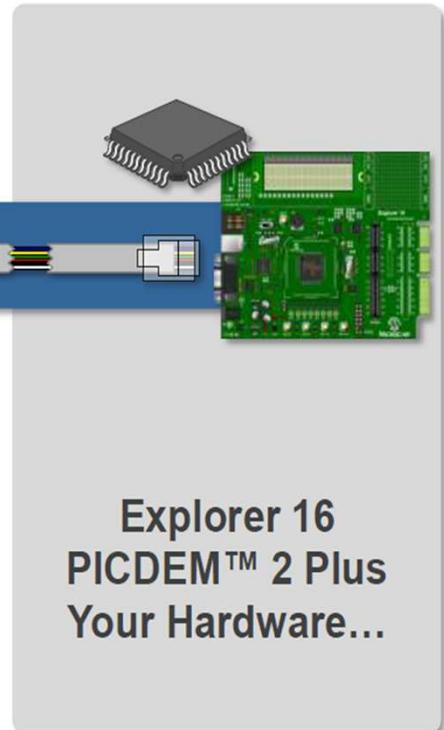
notes





Dans ce cours, nous travaillerons uniquement en simulation via le simulateur VSM de proteus. Cette première étape de debug permet généralement de résoudre 80 % des problèmes logiciels. Seuls les problèmes liés aux temps réels ne sont pas pris en compte par les simulateurs et pour cause. La deuxième étape consiste à charger et à débugger le programme sur la cible physique. Vous aurez l'occasion d'appliquer cette deuxième étape lors du projet robot. Cette étape se passera d'autant mieux que vous serez relativement confiant dans votre logiciel, l'une des difficultés des systèmes embarqués étant de déterminer l'origine des pannes logicielles ou matérielles.

notes

Integrated  
Development  
EnvironmentProgrammer  
DebuggerTarget  
Hardware

Lorsque vous aurez à travailler avec une cible réelle, il ne faudra pas perdre de vue qu'il vous faudra un module programmer debugger pour charger et debugger votre programme du PC vers la cible. Il en existe 3 plus ou moins performants voir transparent suivant.

notes



USB Speed	Full	Full / High	Full / High
Power to Target	✓	✓	
HW Breakpoints	✓	✓	✓
SW Breakpoints & Stopwatch		✓	✓
Trace			✓
Data Capture			✓
Logic Probe / Trigger			✓

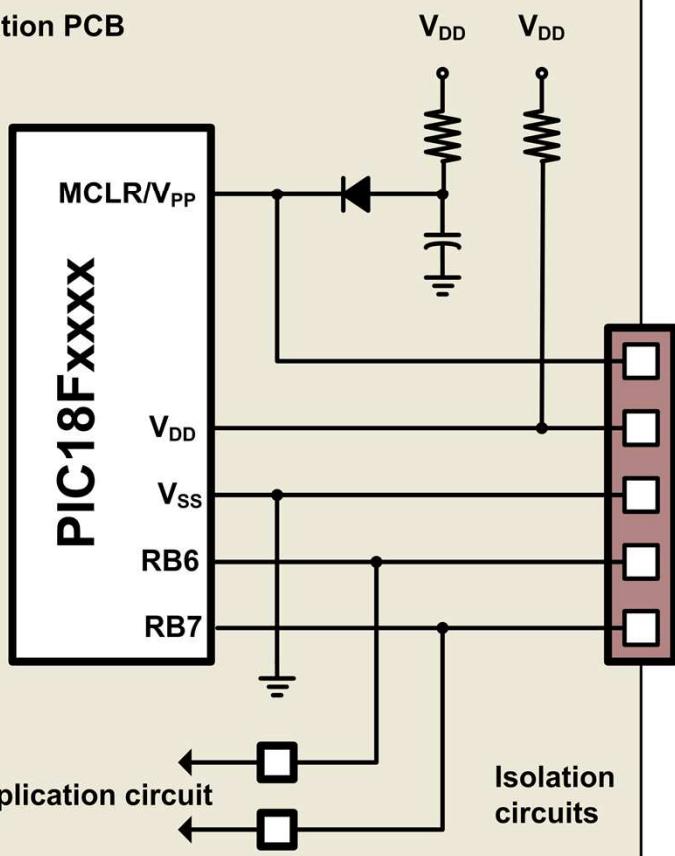
Respectivement 50 €, 150 €, 450 €

Les 3 programmeurs Debugger se connectent en USB côté PC, mais du côté de la cible il faut un brochage spécifique. En effet les PIC disposent d'une liaison série spécialisée pour charger le programme et debugger: liaison ICSP ( In Circuit Serial Programming )

notes

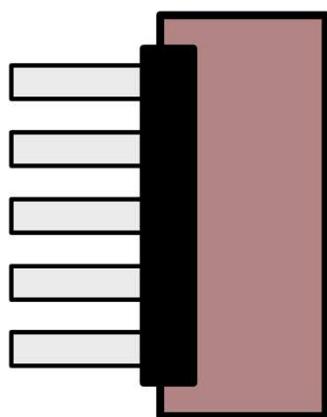


## Application PCB



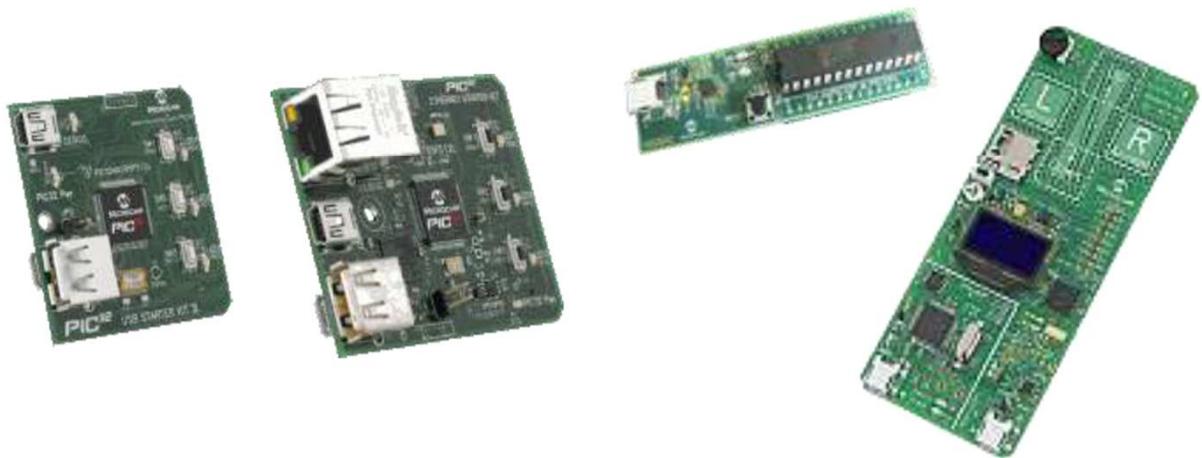
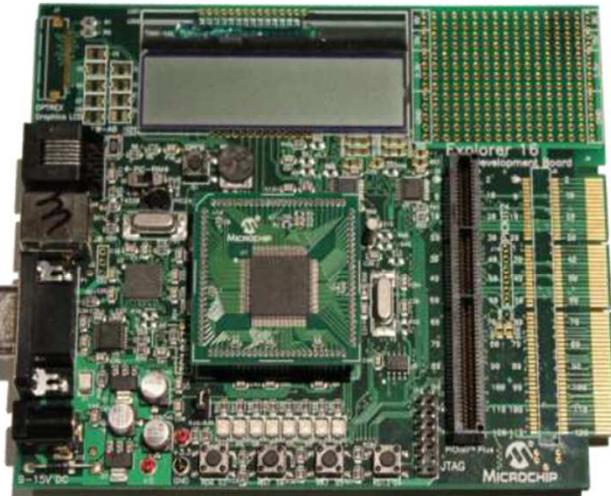
Pin	Function
V <sub>PP</sub>	Programming Voltage = 13V
V <sub>DD</sub>	Supply Voltage
V <sub>SS</sub>	Ground
RB6	Clock Input
RB7	Data I/O & Command Input

ICSP Connector



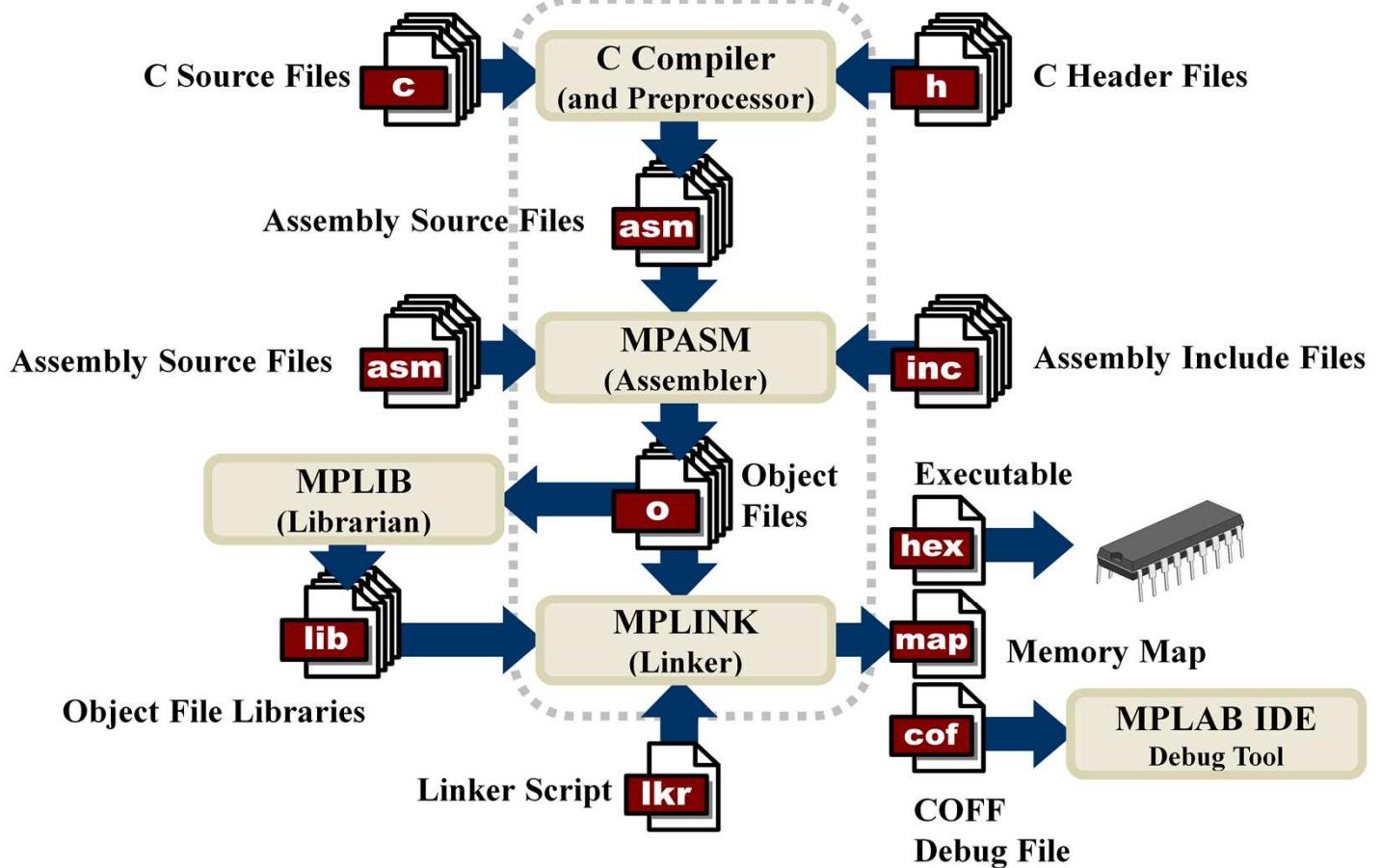
Seulement 2 broches utilisées sur le microcontrôleur RB6 et RB7.

notes



Différentes cartes d'évaluation ou de démarrage existent ( starter kit ) permettant de rapidement débuter une application.

notes

**C Compiler**

Compiles C source code into assembly language code

**Assembler**

Assembles assembly language code into object files

**Librarian**

Groups precompiled object files into a library (archive on GCC compilers) for convenient organization of reusable code

**Linker**

Takes relocatable code and arranges it into the microcontroller's memory map according to the guidelines provided in the linker script

**Compiler Driver Program**

This command line driver program allows application programs to be compiled, assembled and linked in a single step.

notes



Copier sur votre compte le répertoire : C:\MPLAB\_configs\TP PIC18 X

Lancer l'outil MPLAB X

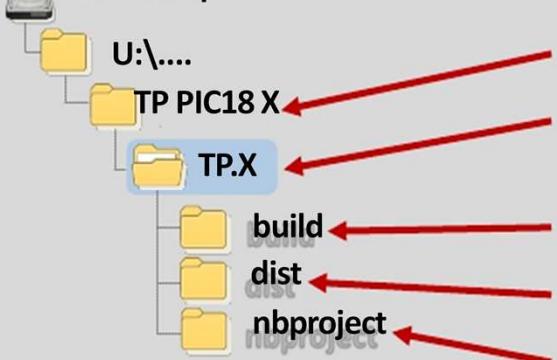


MPLAB X IDE

Créer un nouveau projet

: Voir diapo suivante

### Votre compte



Project Location = ..\TP PIC18 X

Project Directory = Project Name (TP)

Intermediates Directory (\*.o files)

Output Directory (\*.hex and \*.cof files)

Project Settings Directory

Un projet est défini par une collection de fichiers dans une structure de répertoires spécifiques, utilisés par MPLAB X pour garder une trace de tous les fichiers, de toutes les configurations, les paramétrages de votre application.

notes

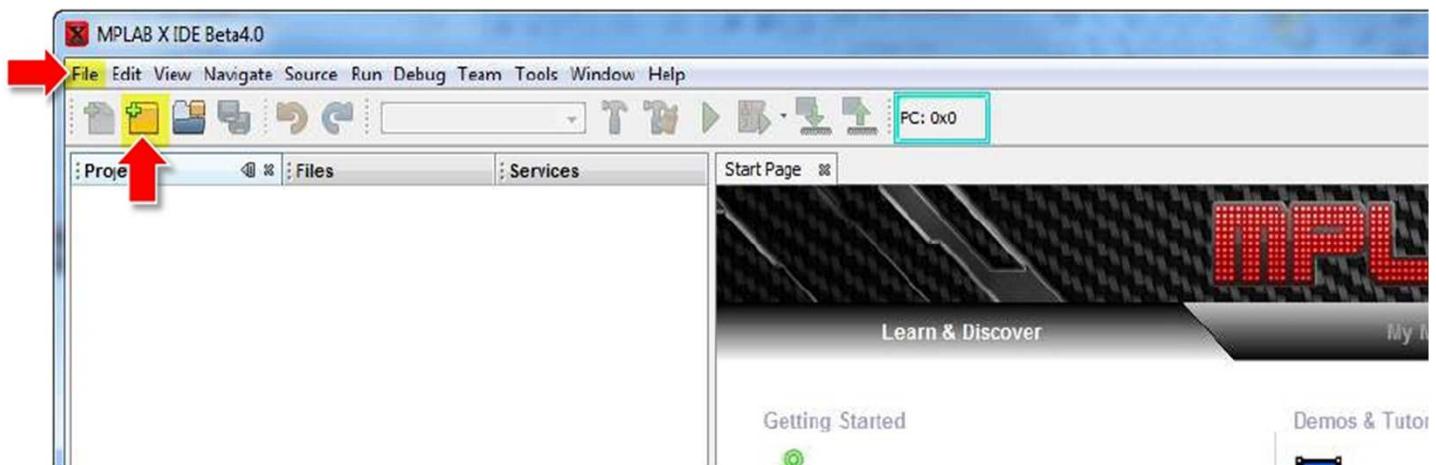


## 1 Launch the Project Wizard

From the menu, select: **File ▶ New Project**

OR

Click on the  icon

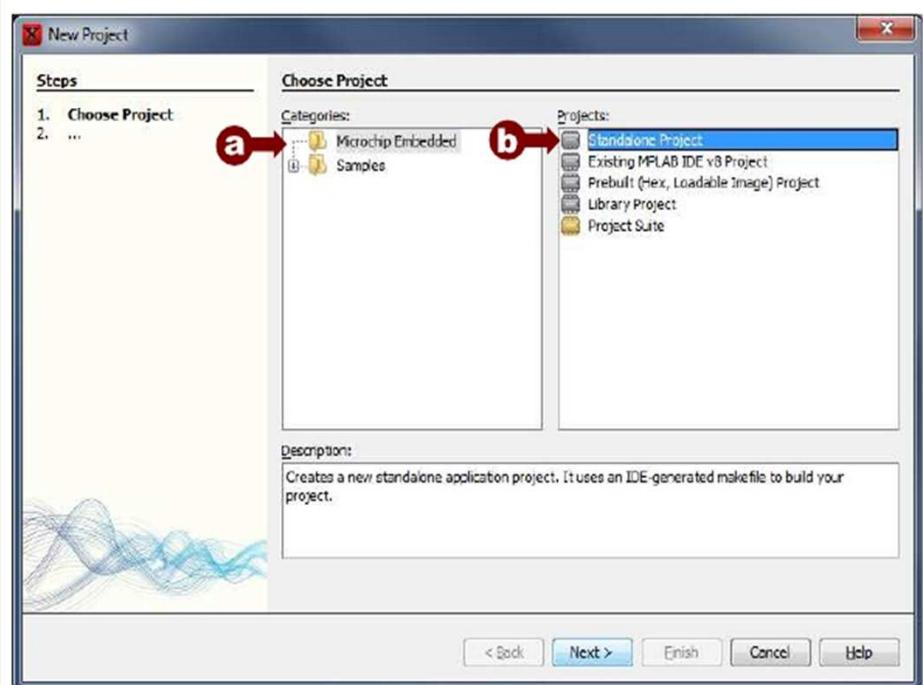




## 2 Select the project type

Select:

- a Microchip Embedded  
in the *Categories* box and:
- b Standalone Project  
in the *Projects* box



Click **Next >**

notes



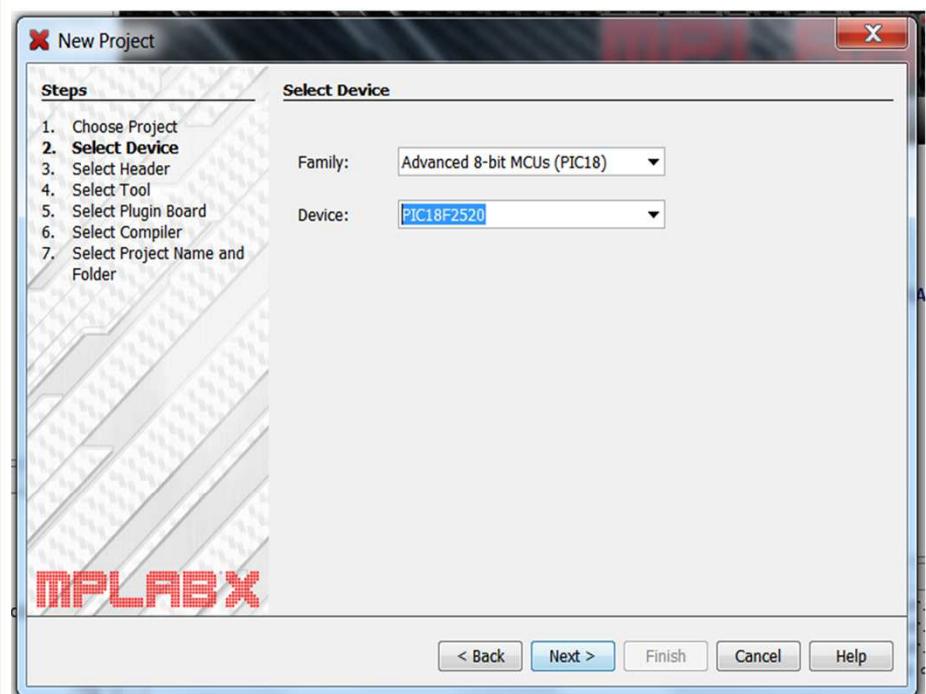
### 3 Select the product family and device

Choose based on the hardware you are using:

8-bit



**Robot**  
**PIC18**  
**PIC18f2520**



Click **Next >**

Nous travaillerons avec le PIC18f2520 qui est utilisé dans le robot.

notes



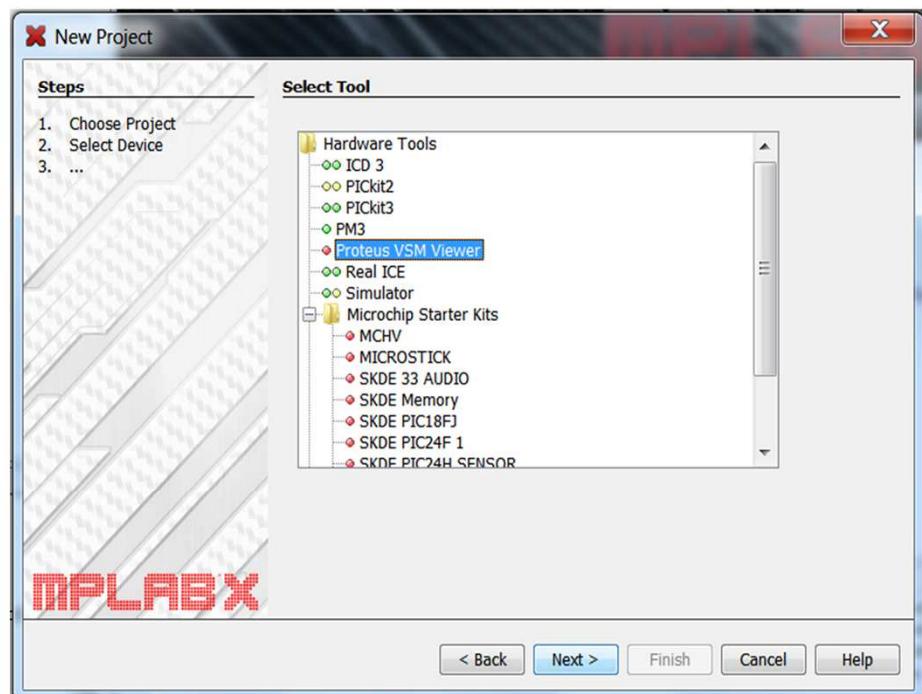
## 5 Select Tool

**Click on the serial number below the name of the tool you are using.**

**Select the “Simulator” if no Debugger/programmer connected**



If you don't see a serial number under a PICkit™, ICD or REAL ICE™ then it either isn't connected to the system or there is a problem with the driver.



**Click Next >**

Le simulateur proteus sera utilisé pour tester nos applications, il faut donc le sélectionner **Proteus VSM Viewer**. Plus tard dans le projet robot, le pickit 3 sera utilisé pour télécharger votre programme sur votre cible matérielle, il faudra alors sélectionner le numéro de série du PICkit3 dans Select Tool. Attention ce numéro de série n'apparaît que si un kit est connecté.

Remarque: la première fois, peut-être que le choix proteus VSM viewer n'apparaîtra pas, il faut alors installer le plugin:

### Tools/Plugins

Onglet **Available Plugins**, sélectionner **Proteus VSM Viewer**

notes



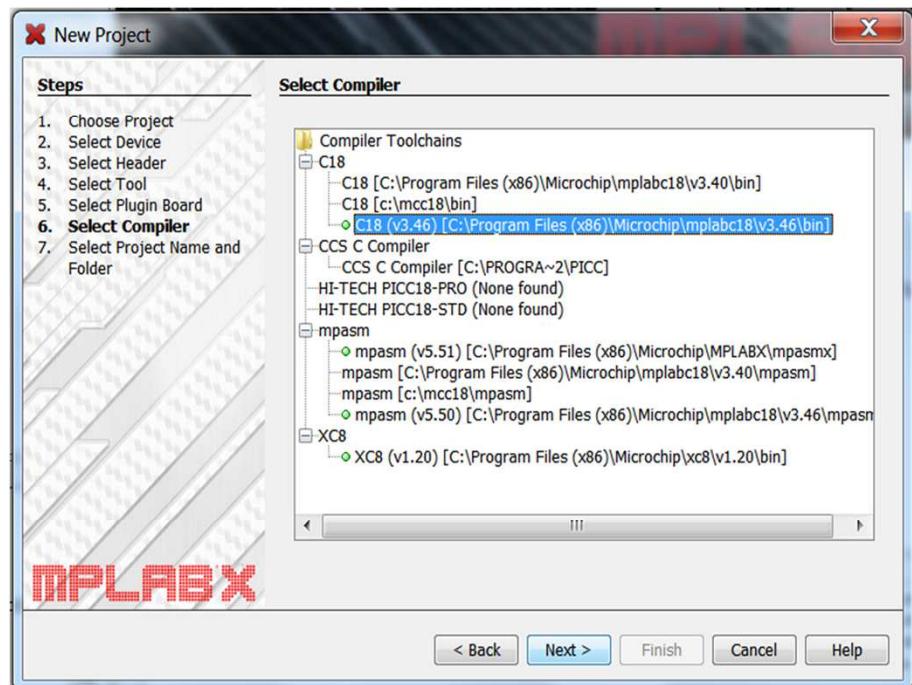
## 6 Select Compiler

Click on the compiler with a version number under the compiler suite heading.

Only valid compilers for selected device are shown.



If you don't see a version number under a compiler name then it either isn't installed or the IDE cannot find it.



Click **Next >**

Le compilateur s'installe indépendamment de MPLAB X. Plusieurs compilateurs peuvent être installé, à vous de sélectionner le bon. On utilisera le **compilateur C18 de Microchip** pour des raisons historiques: le projet robot utilise ce compilateur, mais le compilateur le plus performant et conseillé avec MPLAB X est XC8 ( pour les microcontrôleurs 8 bits ).

Remarque: les chemins et la version du compilateur peuvent être différents sur vos postes.

notes



## 7 Select Name and location

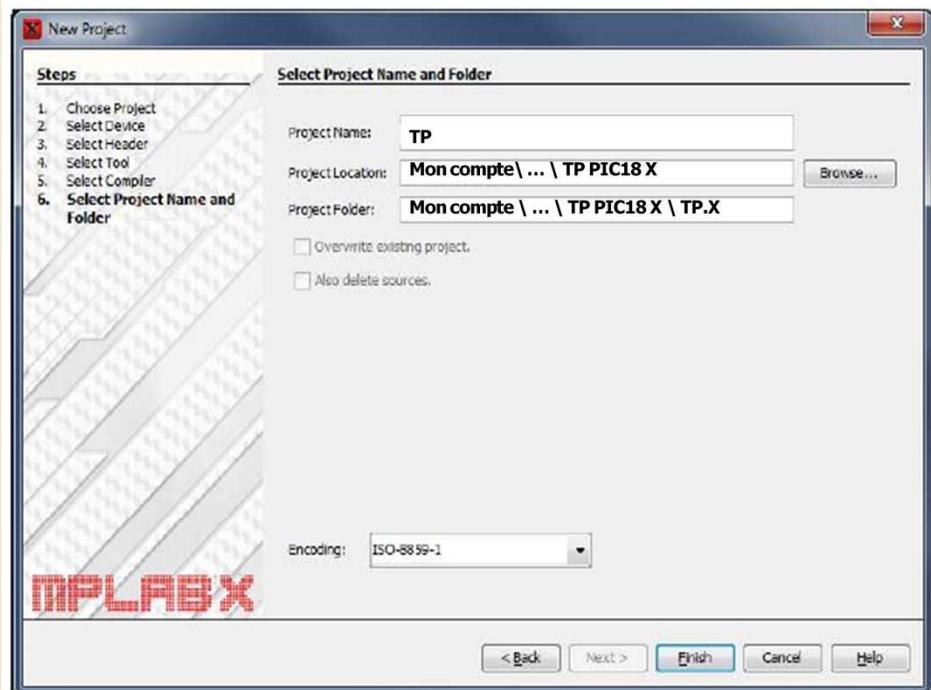
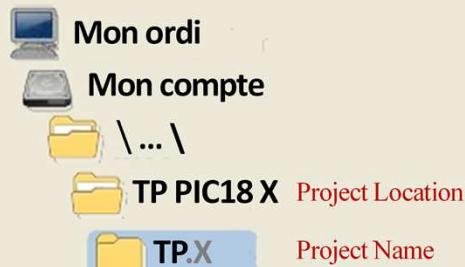
Enter a *Project Name*:

TP

Enter a *Project Location*:

U:\... \TP PIC18 X

A folder with the project name will be created in the project location.



Click **Finish**

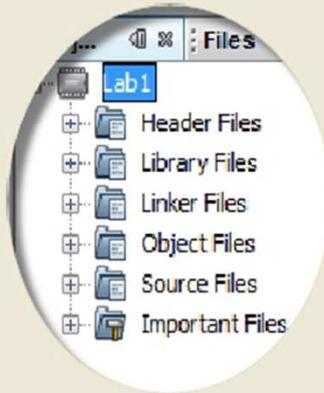
Je vous conseille de travailler sur votre compte perso, pour retrouver facilement vos fichiers. Dans le cas contraire, pensez à sauvegarder votre répertoire de travail à la fin du cours. Utiliser le répertoire TP PIC18 X ( à recopier ) pour travailler et y créer vos projets, il contient déjà de la documentation et des fichiers sources utiles pour vos TP.

notes



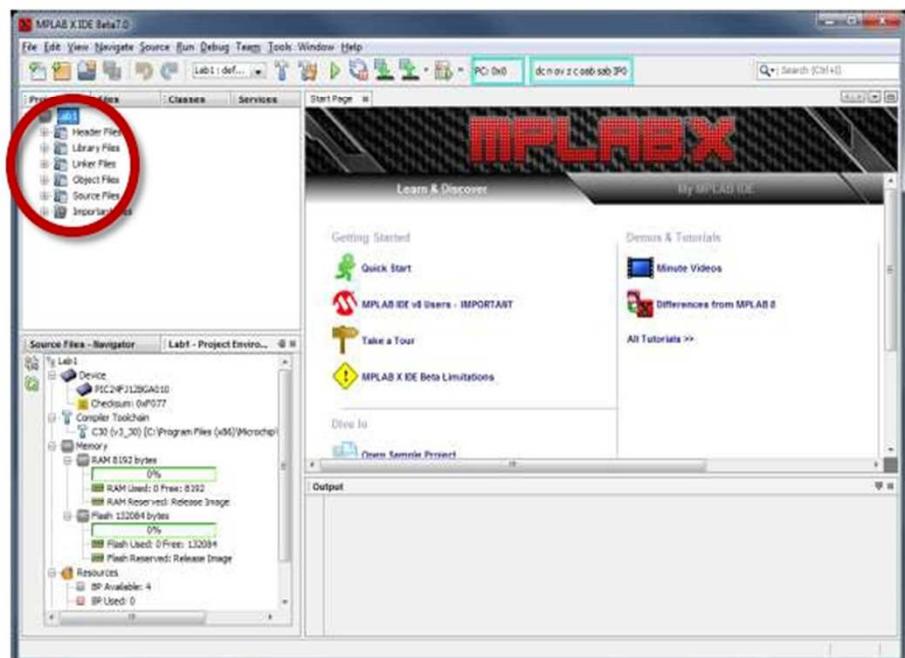


## Empty Project Tree



You should now see a project tree under the Projects tab.

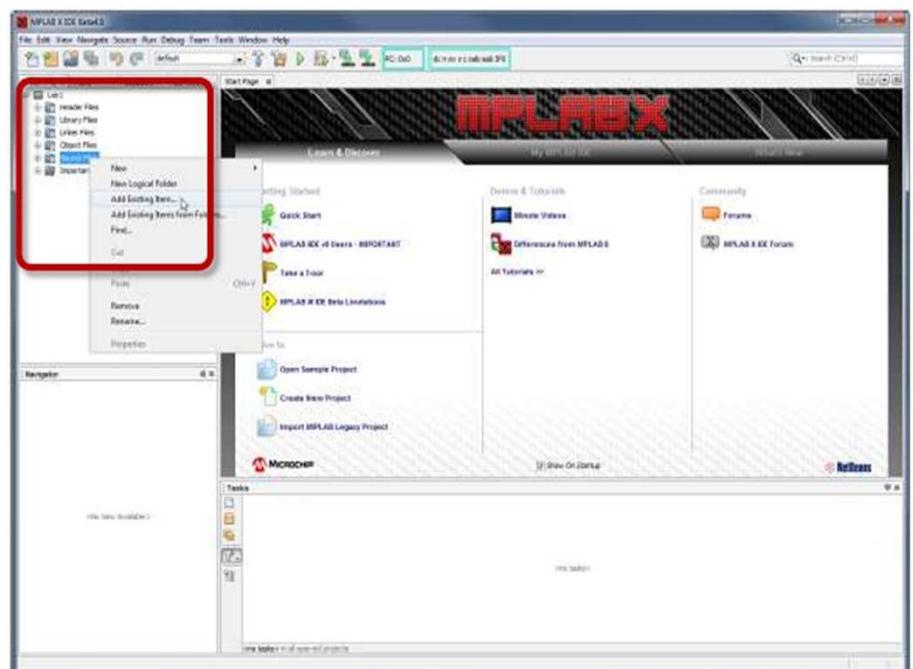
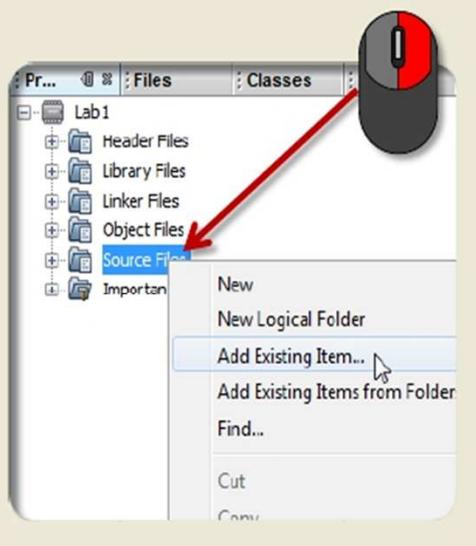
Next, we'll begin adding files to the project.





## 8 Add C Source Files

Right click on the Source Files folder and select **Add Existing Item...** from the popup menu



Ajouter le fichier bouton.c se trouvant dans le répertoire TP PIC18 X.

notes

## Projet Pilotage Robot

## OUTIL MPLAB: description - 1er programme - debug

MPLAB X IDE v1.85 - PIC18F : default

File Edit View Navigate Source Refactor Run Debug Team Tools Window Help

Screenshot of the MPLAB X IDE interface showing a project named "PIC18F". The code editor displays a C program named "bouton.c". The code includes configuration pragmas for oscillator and power management, and a main loop that toggles an LED connected to port B0. The Navigator pane shows memory usage details, and the Output pane shows a successful build and load message.

```

3 * Author: marques
4 *
5 * Created on 7 août 2013, 16:26
6 */
7
8 #include <p18f2520.h>
9
10 #pragma config OSC = INTIO67
11 #pragma config PBADEN = OFF, WDT = OFF, LVP = OFF, DEBUG = ON
12
13
14 unsigned char led_test;
15
16 void main(void) {
17
18     TRISBbits.RB0=1;
19     TRISBbits.RB5=0;
20     while(1) {
21         if (PORTBbits.RB0 ) {
22             PORTBbits.RB5=1;
23             led_test = 1;
24         }
25         else {
26             PORTBbits.RB5=0;
27             led_test = 0;
28         }
29     }
30 }
31
32

```



Il est recommandé de visualiser le fichier de déclaration des registres p18f2520.h et le fichier de link 18f2520\_g.lkr. Le premier est à consulter dès que vous avez un doute sur le nom des registres ou de ses flags, le deuxième vous permet d'observer le mapping utilisé, le fichier de startup (c018i.o) et les librairies appelées (clib.lib et p18f2520.lib). Remarque c'est dans p18f2520.lib qu'est définie l'adresse des registres, clib.lib étant la librairie standard du C.

Dans votre programme « bouton.c », on remarque des **#pragma** qui initialisent des bits de configuration du microcontrôleur. Pour avoir la signification de ces flags cliquer sur **Window / Pic Memory Views / Configuration Bits**.

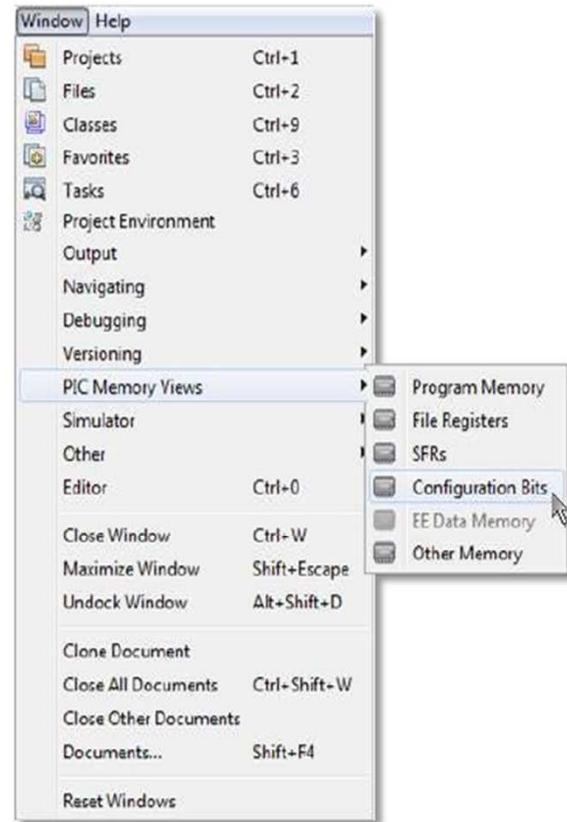
notes



## Open Configuration Bits window

From the main menu select:

Window ►  
PIC Memory Views ►  
Configuration Bits



Dans votre programme « bouton.c », on remarque des #pragma qui initialisent des bits de configuration du microcontrôleur. Pour avoir la signification de ces flags cliquer sur **Window / Pic Memory Views / Configuration Bits**.

notes



Click on the **Build** button  
on the toolbar



After if

**BUILD SUCCESSFUL**



**Debug Project**

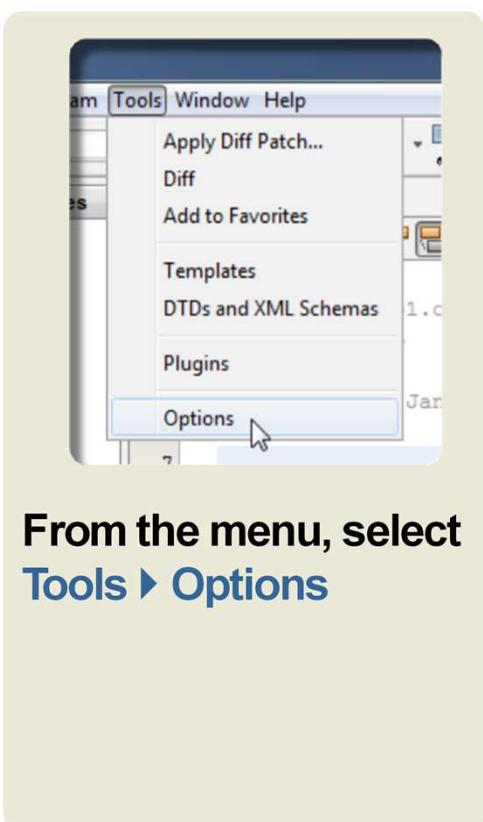
Function	MPLAB® X
End Debug Session	
Halt / Pause	
Run / Continue	
Reset	
Run To Cursor	
Step Into	
Step Over	
Step Out	
Reset	
Focus Cursor at PC	

Remarque: en cliquant sur l'onglet Debug, le projet est rebuilder en mode debug même si le build précédent s'est terminé « successful ». En effet en mode debug il faut rajouter du code dans le soft pour l'instrumenter. Un certain nombre de ressources sont réservées pour le débug. Le debug se lance avec l'outil debug sélectionné lors de la création du projet, dans notre cas il ouvre proteus et nous donne accès aux fonctions citées ci-dessus.

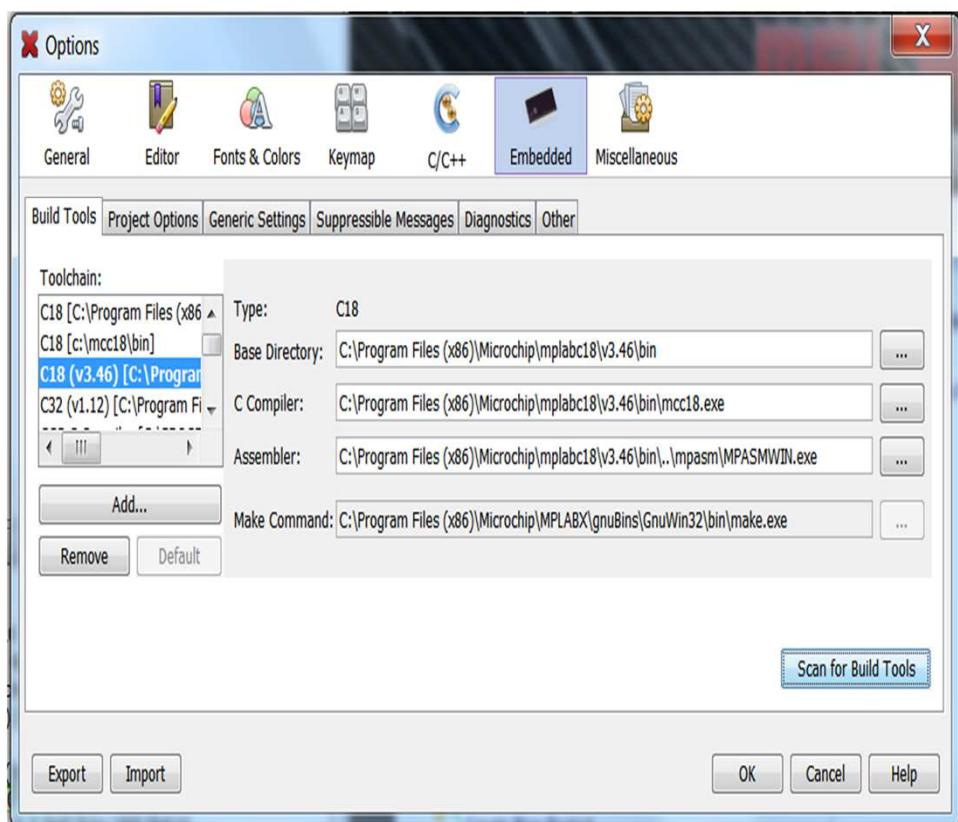
Analyser le fichier bouton.c, commenter les lignes de code, et tester-le avec ISIS. Vous pouvez utiliser le mode pas à pas pour détailler le fonctionnement

notes





From the menu, select  
Tools ▶ Options

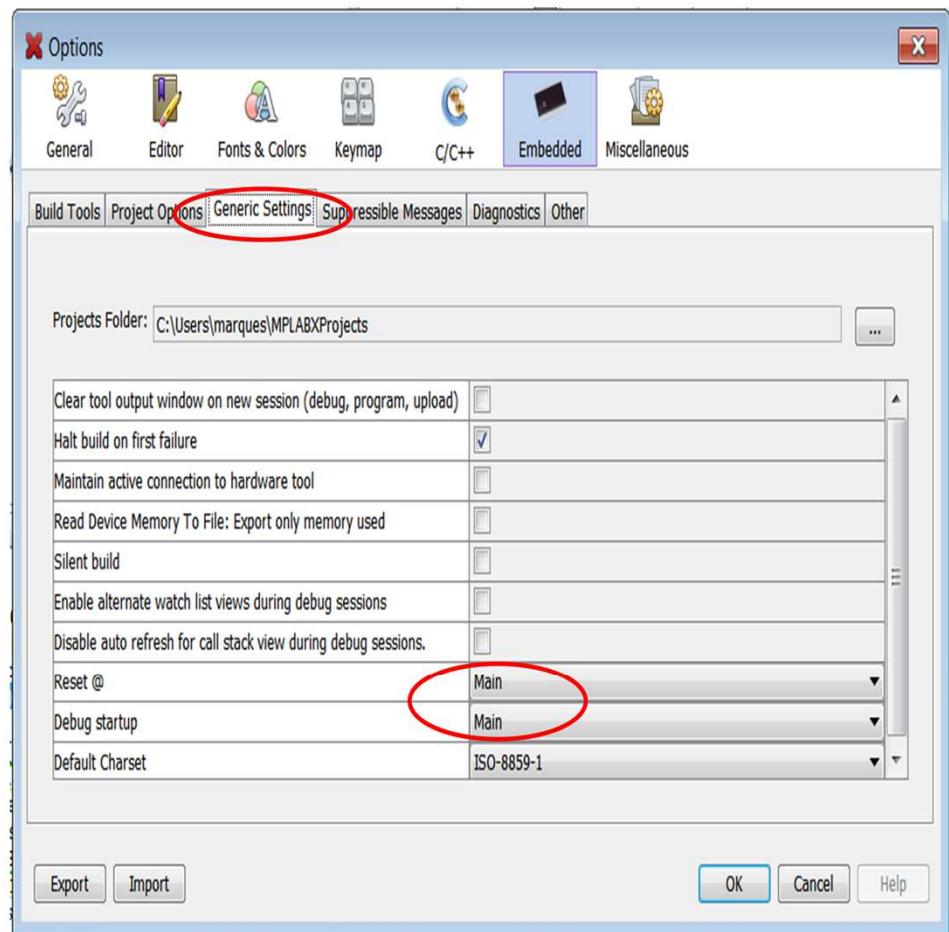


Si le build ne se lance pas convenablement vérifiez que le bon compilateur est sélectionné ainsi que les chemins via l'onglet Tools / Options

notes



**Vérifier que le reset et le démarrage du debugger est le main**

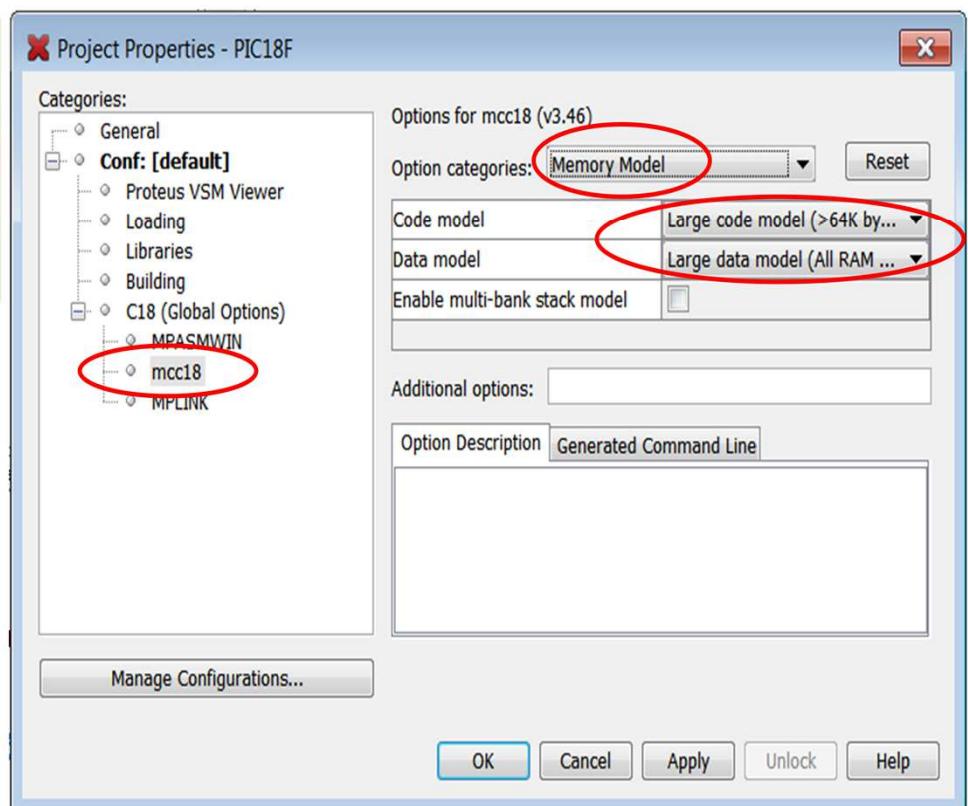


Vérifier que le debugger vous donnera la main au début du main et que le reset également vous ramènera en début de main. C'est plus pratique que de se retrouver au début du fichier startup runtime ( reset vector ).

notes



## Vérifier les options du compilateur C18

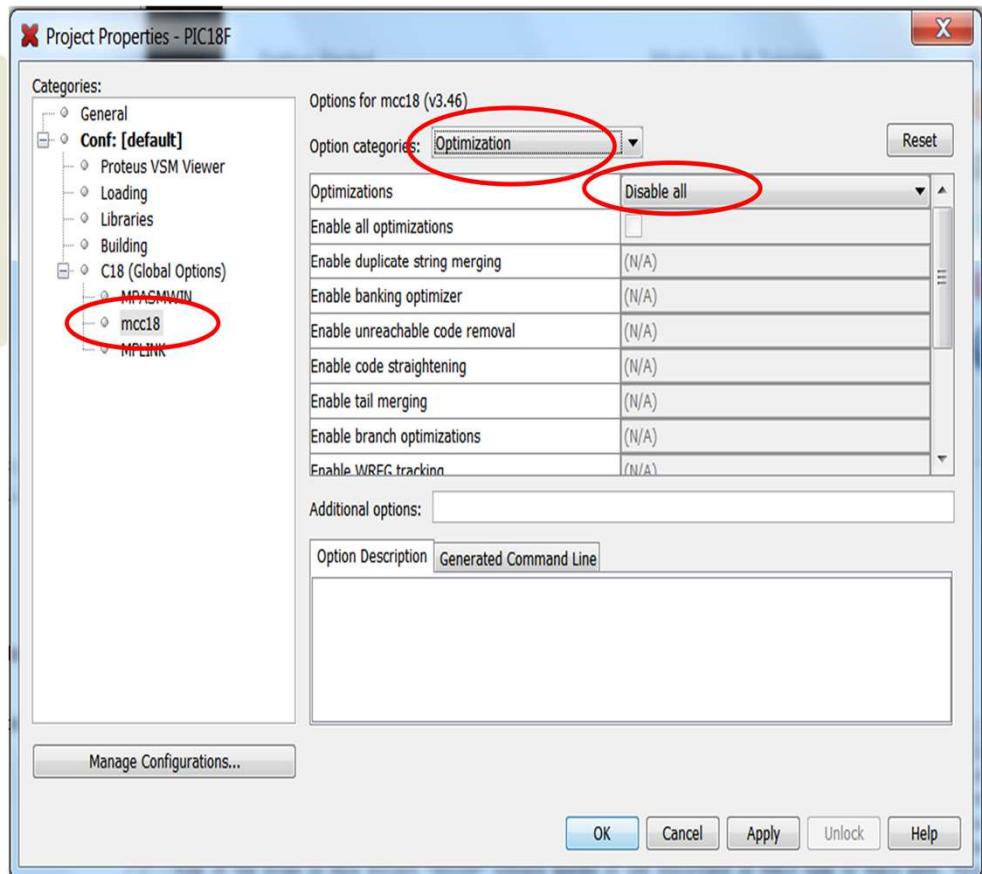


Vérifier le modèle mémoire utilisé par le compilateur C18. En choisissant Large model vous évitez les problèmes avec les bibliothèques.

notes



## Vérifier les options du compilateur C18

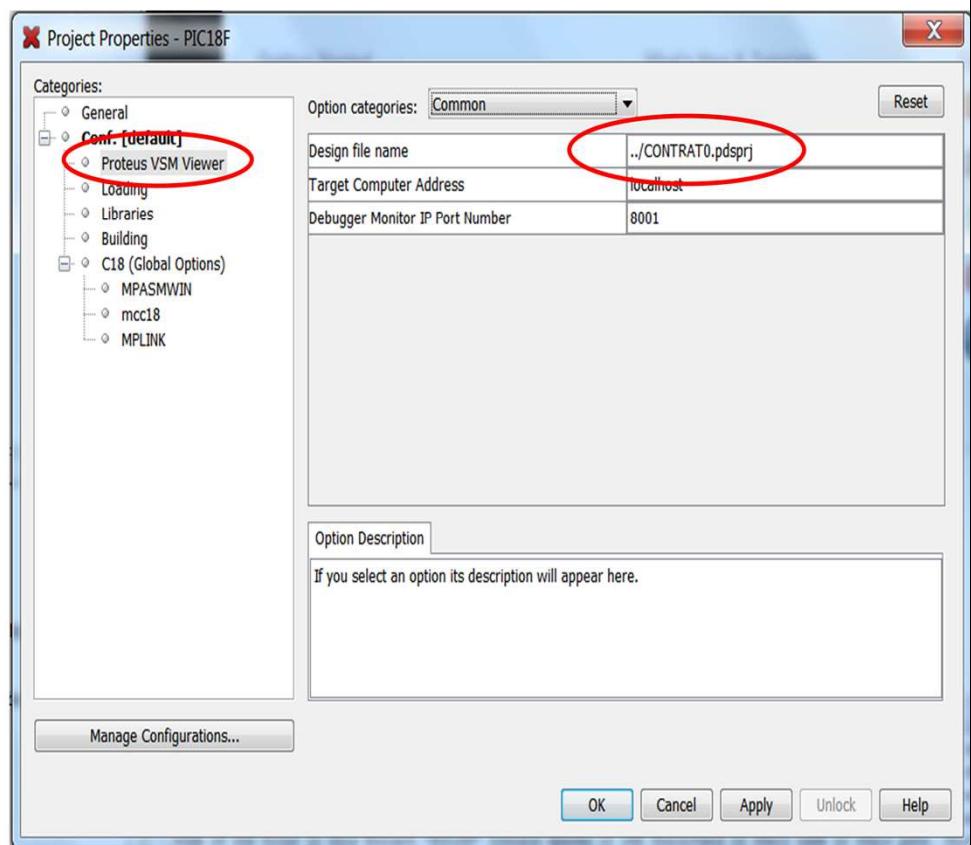


Attention il faut désactiver les optimisations du compilateur car la version Lite du compilateur ne permet pas ces optimisations et vous pouvez alors avoir un fonctionnement erratique du programme.

notes



## Vérifier le chemin du Design Proteus



Vérifier que le design file name de Proteus est bien le bon, dans le cas contraire sélectionner le dans votre répertoire de travail : **TP PIC18 X / CONTRATO.pdsprj**

notes



## Line Breakpoints

Click on a line number in the glyph margin to toggle a breakpoint on that line.

A red square indicates that a breakpoint is set (■). The line of code will also be highlighted in red.

```

1 #include <p24FJ120GA010.h>
2 #include "libLCD24.h"
3
4 _CONFIG1(FWDIEN_OFF & JTAGEN_OFF)
5
6 int main(void)
7 {
8     lcdInit();
9     Output
10    TRISA = 0;
11    ■ LATA = 0x55;
12    PORTB = 0;
13    while(1);
14    //TODO: Translate to Spanish
15
16 }
17
18

```

**Click on line number to toggle breakpoint**

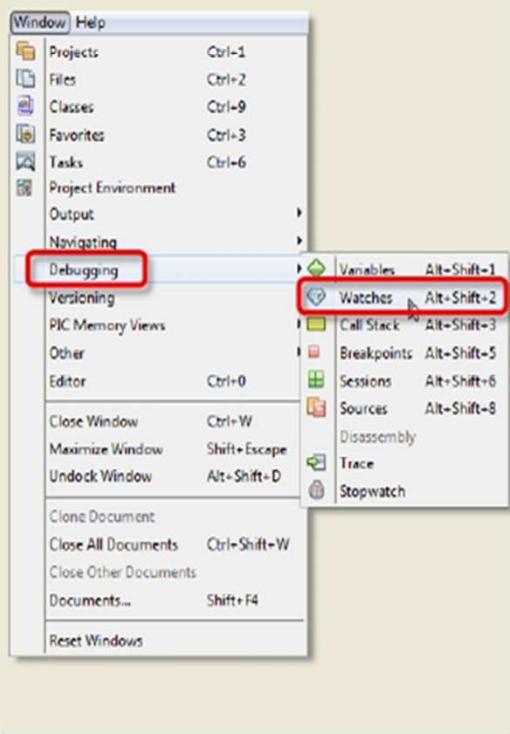
Vous avez à votre disposition un certain nombre d'outils de debug, parmi les plus utilisés citons: les points d'arrêt, la fenêtre watch qui permet de suivre l'évolution de vos données et la fenêtre des registres de contrôle qui permet de vérifier la bonne initialisation des registres.

En utilisant ces outils placer un point d'arrêt, visualiser dans la fenêtre watch la variable led\_test et vérifier la bonne initialisation des registres TRISB et PORTB.

notes



From the main menu  
select: Window ▶ Debugging ▶ Watches



Name	Address	Type	Value
LATA0	0x2C4	SFR	0x0055
LATA1	0x2C4	LATA<0>	0x01
LATA2	0x2C4	LATA<1>	0x00
LATA3	0x2C4	LATA<2>	0x01
LATA4	0x2C4	LATA<3>	0x00
LATA5	0x2C4	LATA<4>	0x01
LATA6	0x2C4	LATA<5>	0x00
LATA7	0x2C4	LATA<6>	0x00
LATA8	0x2C4	LATA<7>	0x00
LATA9	0x2C4	LATA<8>	0x00
LATA10	0x2C4	LATA<9>	0x00
LATA11	0x2C4	LATA<10>	0x00
LATA12	0x2C4	LATA<11>	0x00
LATA13	0x2C4	LATA<12>	0x00
LATA14	0x2C4	LATA<13>	0x00
LATA15	0x2C4	LATA<14>	0x00
PORTB	0x2C0	SFR	0x0000
<Enter new watch>			

Vous avez à votre disposition un certain nombre d'outils de debug, parmi les plus utilisés citons: les points d'arrêt, la fenêtre watch qui permet de suivre l'évolution de vos données et la fenêtre des registres de contrôle qui permet de vérifier la bonne initialisation des registres.

En utilisant ces outils placer un point d'arrêt, visualiser dans la fenêtre watch la variable led\_test et vérifier la bonne initialisation des registres TRISB et PORTB.

notes

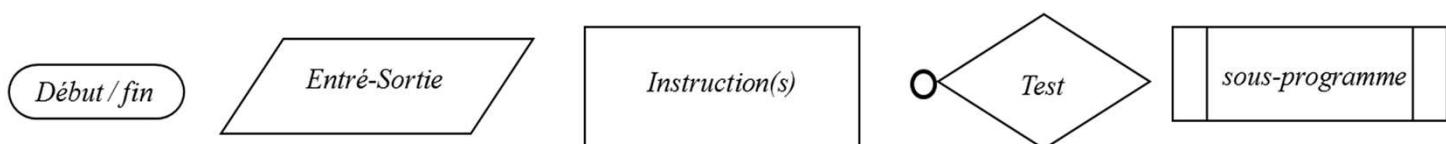


## Projet Pilotage Robot

## OUTIL MPLAB: description – 1er programme – debug

Désigne le diagramme d'une séquence d'actions d'une application.

Un synonyme est algorithme



Séquence linéaire	Séquence alternative "si...alors...sinon"	Séquence répétitive "tant que...faire..."	Séquence répétitive "répéter...jusqu'à..."
<pre>     Début           Traitement 1           Traitement 2           Fin   </pre>	<pre>     Condition     /   \     Vrai  Faux                 Traitement 1  Traitement 2           +-----+   </pre>	<pre>     Condition     /   \     Vrai  Faux                 Traitement           +-----+   </pre>	<pre>     Traitement           Condition     /   \     Vrai  Faux                 +-----+   </pre>
Début <ul style="list-style-type: none"> <li>• "Traitement 1"</li> <li>• "Traitement 2"</li> </ul> Fin	Si "condition" <ul style="list-style-type: none"> <li>• alors "Traitement 1"</li> <li>• sinon "Traitement 2"</li> </ul> Fin si	Tant que "condition" <ul style="list-style-type: none"> <li>• faire "traitement"</li> </ul> Fin tant que	Répéter "traitement"         jusqu'à "condition"

## FAIRE L'ORGANIGRAMME DU PROGRAMME bouton.c

Nous avons déjà rencontré la problématique du debuggage dans les systèmes embarqués, la meilleure approche est de partitionner au maximum les problèmes: simuler pour isoler les problèmes logiciels puis tester sur la cible pour traiter ensuite les problèmes matériels. Cette même approche doit nous inspirer lors du développement logiciel. Il faut séparer la phase de réflexion, où on va chercher une solution au problème (phase de conception), de la phase codage où on aborde les problèmes liés au langage de programmation choisi. Donc toujours commencer par décrire la solution sur papier via un organigramme. Commencer à coder sans savoir où on va est **catastrophique** et même si le programme finit par marcher on aboutit le plus souvent à un mouton à 5 pattes, non maintenable et instable (dangereux dans une entreprise).

La description du programme par un organigramme permet de :

- gagner en efficacité lors de la phase de codage,
- d'optimiser la structure du programme,
- de clarifier le fonctionnement du programme,
- le rendre compréhensible à une personne extérieure.

notes



- Rôle du linker ?
- Pourquoi travaille t-on uniquement en simulation dans ce cours ?
- Rôle du build ?
- Qu'appelle t-on pas à pas ?
- Intérêt des points d'arrêt?
- Citer 3 outil permettant de debugger, programmer une cible ?
- Intérêt des organigrammes ?
- La simulation est-elle la panacée ?  ?
- Différence entre mode debug et release ?  ?
- Qu'appelle t-on ICSP ?

notes





P 1

## Introduction

sommaire - organisation - contexte

P 10

## CPU

Architecture – mapping – variables

P 21

## Outil MPLAB

Description – 1<sup>er</sup> programme - debug

P 41

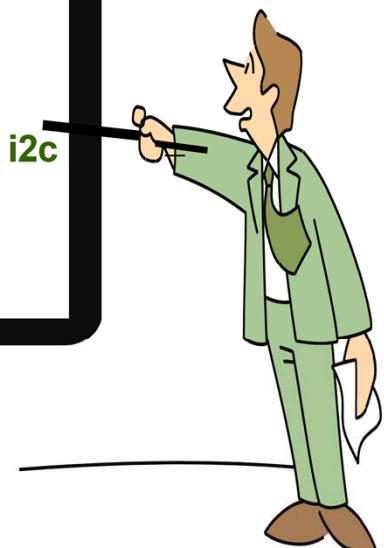
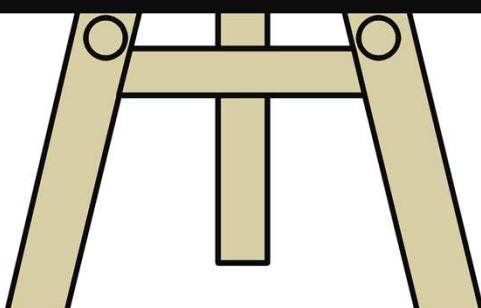
## Pérophériques

horloge - ports - interruptions - timers - can - uart - i2c

P 78

## Robot

Contrat 0 ...

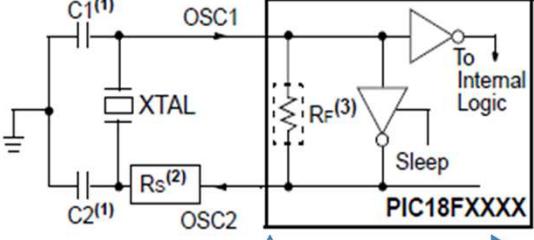


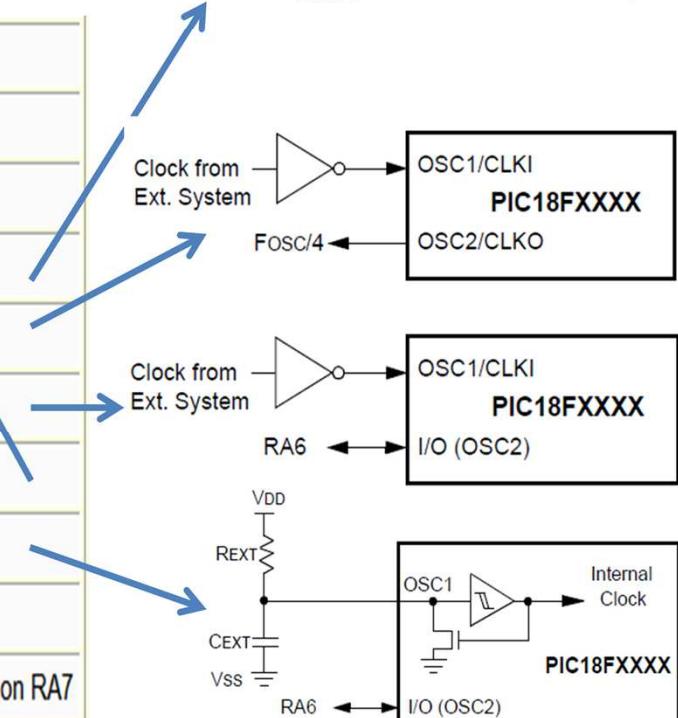
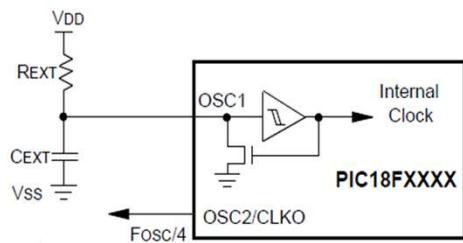
notes



## Projet Pilotage Robot

## PERIPHERIQUES: horloge - ports - interruptions - timer - can - uart - i2c

	
OSC = LP	LP oscillator
OSC = XT	XT oscillator
OSC = HS	HS oscillator
OSC = RC	External RC oscillator, CLKO function on RA6
OSC = EC	EC oscillator, CLKO function on RA6
OSC = ECI06	EC oscillator, port function on RA6
OSC = HSPLL	HS oscillator, PLL enabled (Clock Frequency = 4 x FOSC1)
OSC = RCI06	External RC oscillator, port function on RA6
OSC = INTI067	Internal oscillator block, port function on RA6 and RA7
OSC = INTI07	Internal oscillator block, CLKO function on RA6, port function on RA7



Pour l'horloge principale, 10 choix sont possibles suivant le branchement retenu. Le choix le plus commun est d'utiliser un oscillateur à quartz branché en parallèle sur OSC1 et OSC2. Suivant la fréquence du quartz il faut choisir la configuration LP, XT ou HS par ordre croissant de fréquence. Un autre choix possible est d'utiliser le bloc d'horloge interne, économisant ainsi un quartz et libérant 2 pattes ( OSC1 et OSC2 ) au détriment de la précision de l'horloge.

Je vous rappelle que le choix de la configuration de l'horloge se fait via un #pragma dans votre main, pour avoir le détail des possibilités voir **Help/Topics... /Pic18 Config settings** et choisir le pic18f2520.

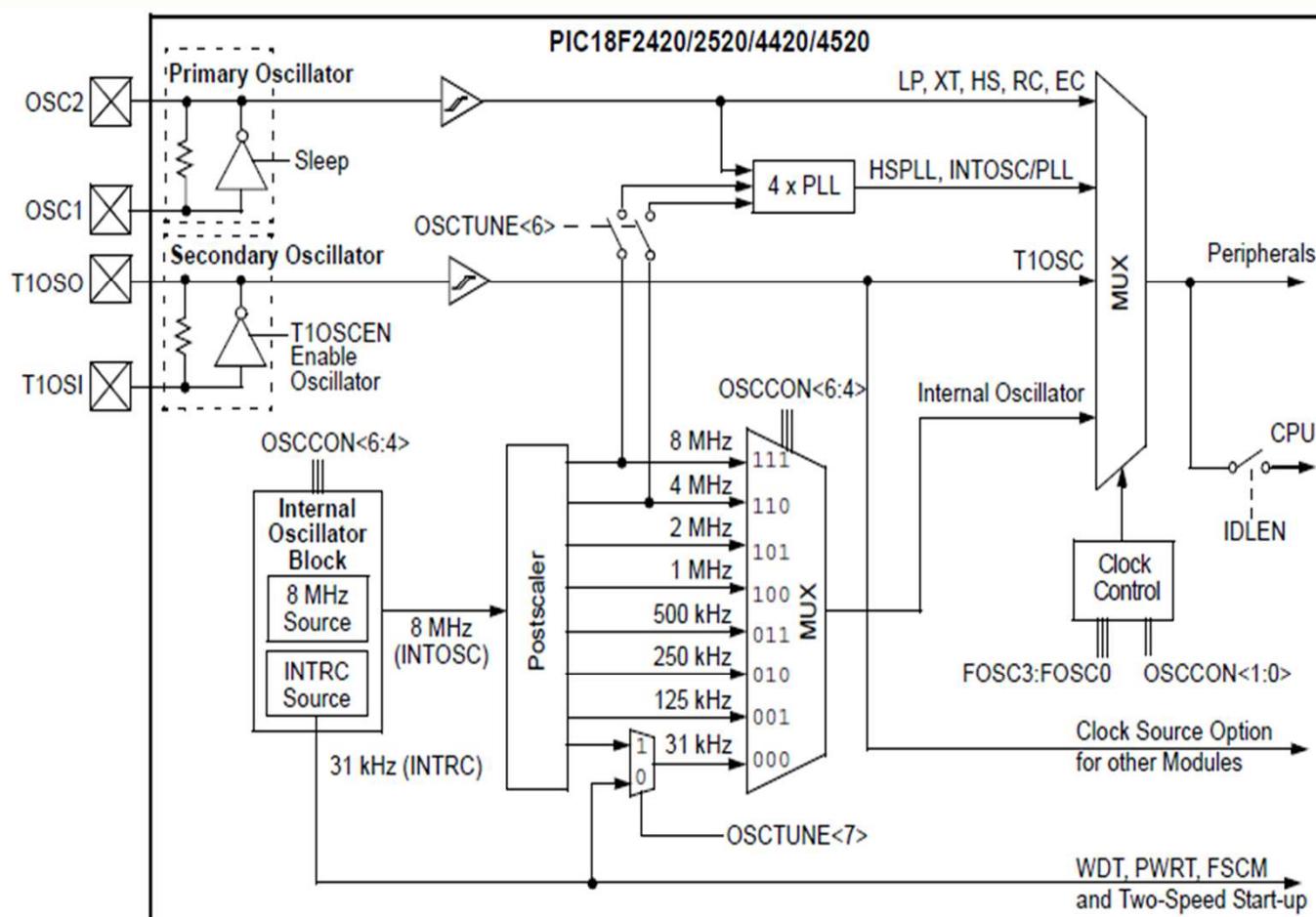
notes



## Projet Pilotage Robot

## PERIPHERIQUES: horloge – ports – interruptions – timer – can – uart – i2c

SYNOPTIQUE



Il existe 3 sources d'horloge:

Principale: en configuration standard elle peut monter jusqu'à 4 MHz, en mode High elle peut monter à 40 MHz. Le mode Low Power permet de baisser drastiquement la consommation. Pour une solution bas cout, on peut opter pour le mode RC externe.

Secondaire: nécessaire pour faire fonctionner le périphérique Real Time Clock calculant l'heure et la date courante, une horloge à 32,768 kHz est alors nécessaire pour produire la seconde ( $/ 2^{15}$ ).

Interne: L'horloge est alors réglable de 8 MHz à 31 kHz via le registre OSCON, 1 MHz par défaut.

Remarque: les entrées d'horloge occupent des ports d'entrée/ sortie, on peut libérer ces broches si elles ne sont pas utilisées.

Le choix initial de la configuration de l'horloge se fait via un #pragma dans votre main, pour avoir le détail des possibilités voir **Help/Topics... /Pic18 Config settings** et choisir le pic18f2520.

notes

\*

## \* FONCTION: Horloge

\*

Reprendre le programme bouton.c

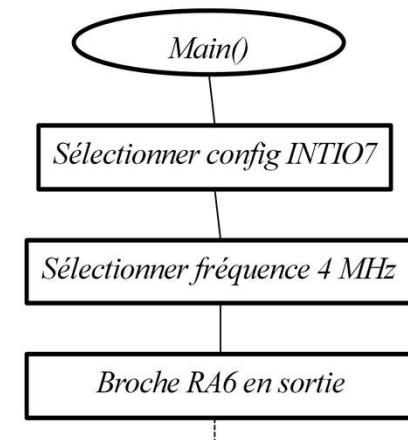
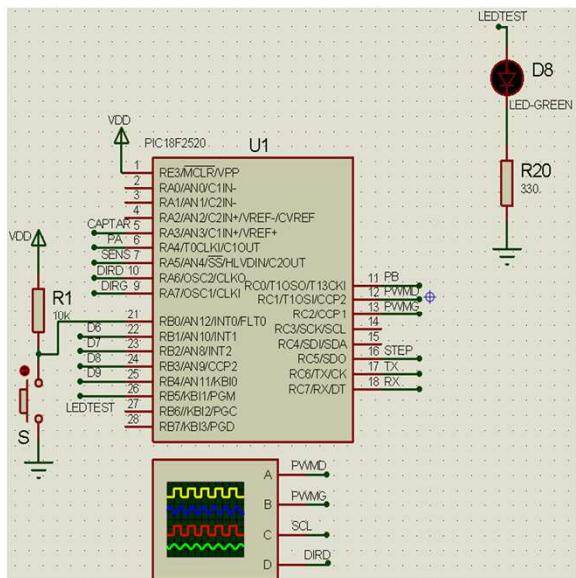
\*

A quelle fréquence fonctionne le Microcontrôleur ?

\*

Modifier-le pour obtenir une fréquence d'horloge de 4 MHz et la visualiser sur la broche CLKout ( RA6 ) ( CLKout = Fosc / 4 ).

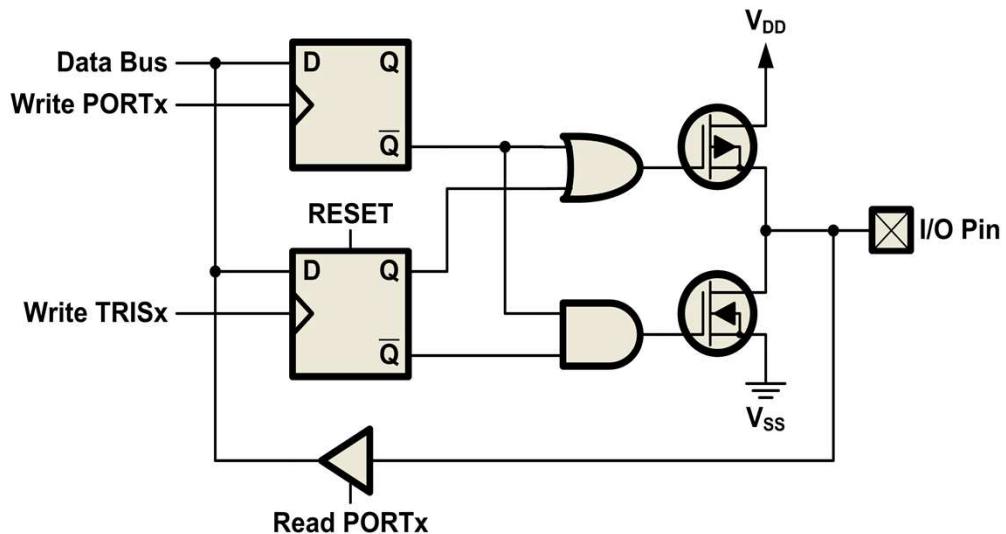
\*



notes



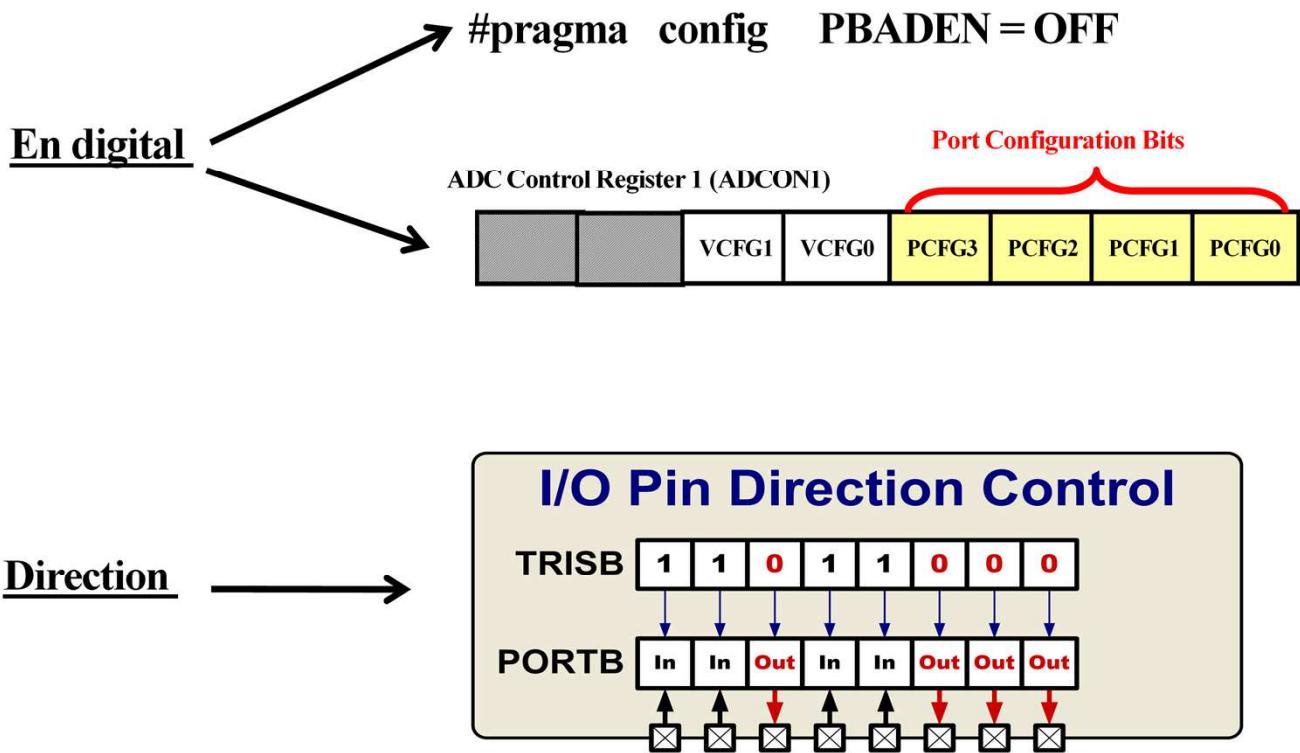
- Broches I/O bi-directionnelle multiplexée avec des périphériques
- Sortance : 25mA
- Modifiable bit à bit en 1 cycle
- Protection décharge électrostatique à diodes : 4kV
- Au reset : PIO en entrée (Hi-Z) et ANALOGIQUES



Le PIC18F2520 dispose de 3 ports: PORT A, B, C

Remarque: en entrée le port B peut se voir doter de résistances de pull up (flag RBPU = 0). Valable que sur le port B, cette résistance est automatiquement désactivée lorsque la patte est en sortie.

notes



2 configurations sont systématiquement à faire sur les ports E/S: choix digital / analogique et choix entrées/ sorties.

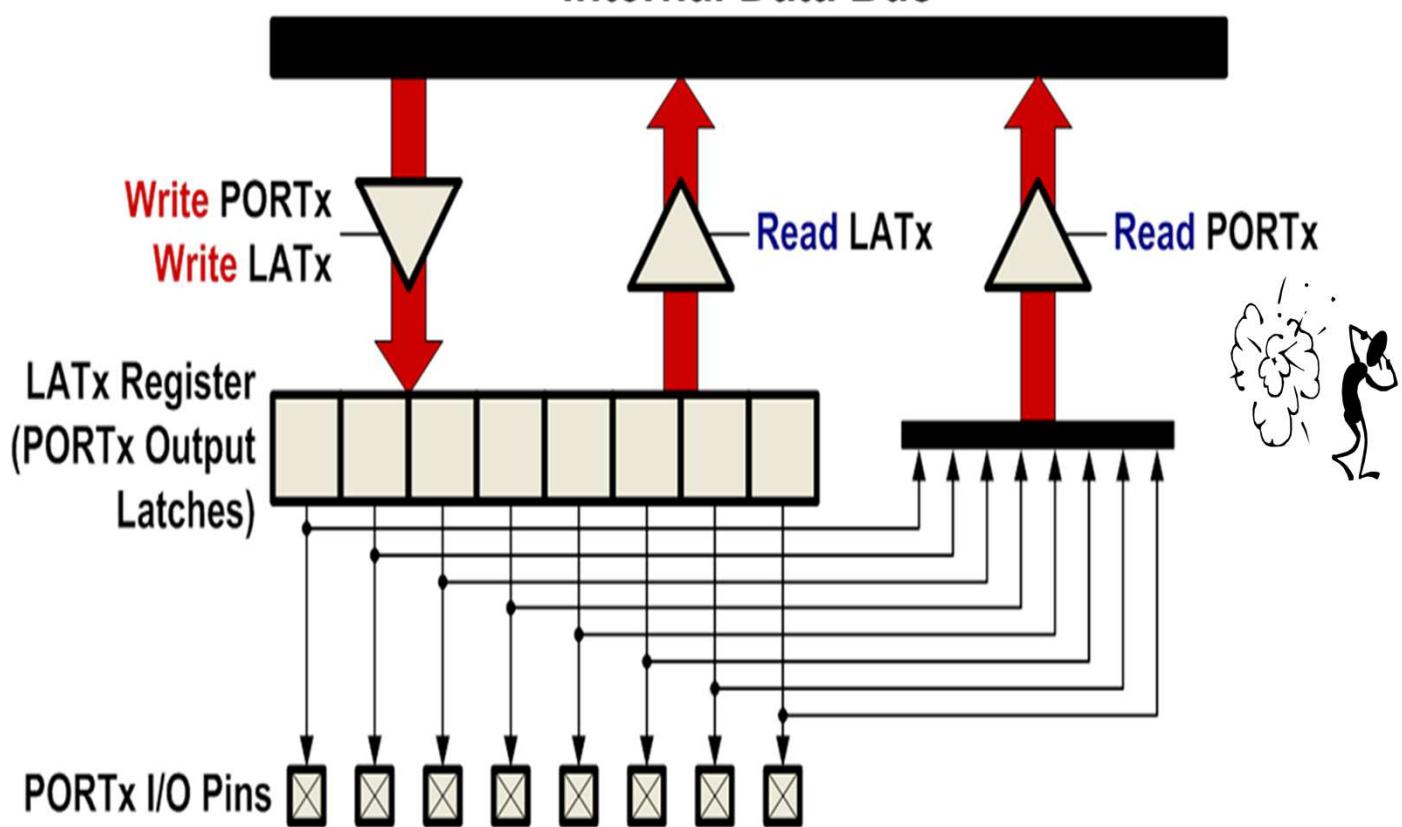
Les broches qui ont la double propriété analogique et digitale sont par défaut analogique, pour les passer en digital il faut utiliser le registre ADCON1. Les broches 0 à 4 du PORT B sont un cas particulier, car elles peuvent être configurée en digital via un #pragma en début de programme.

Chaque broche est configurable de façon indépendante via le registre TRISB: 1 = Input, 0 = Output

notes



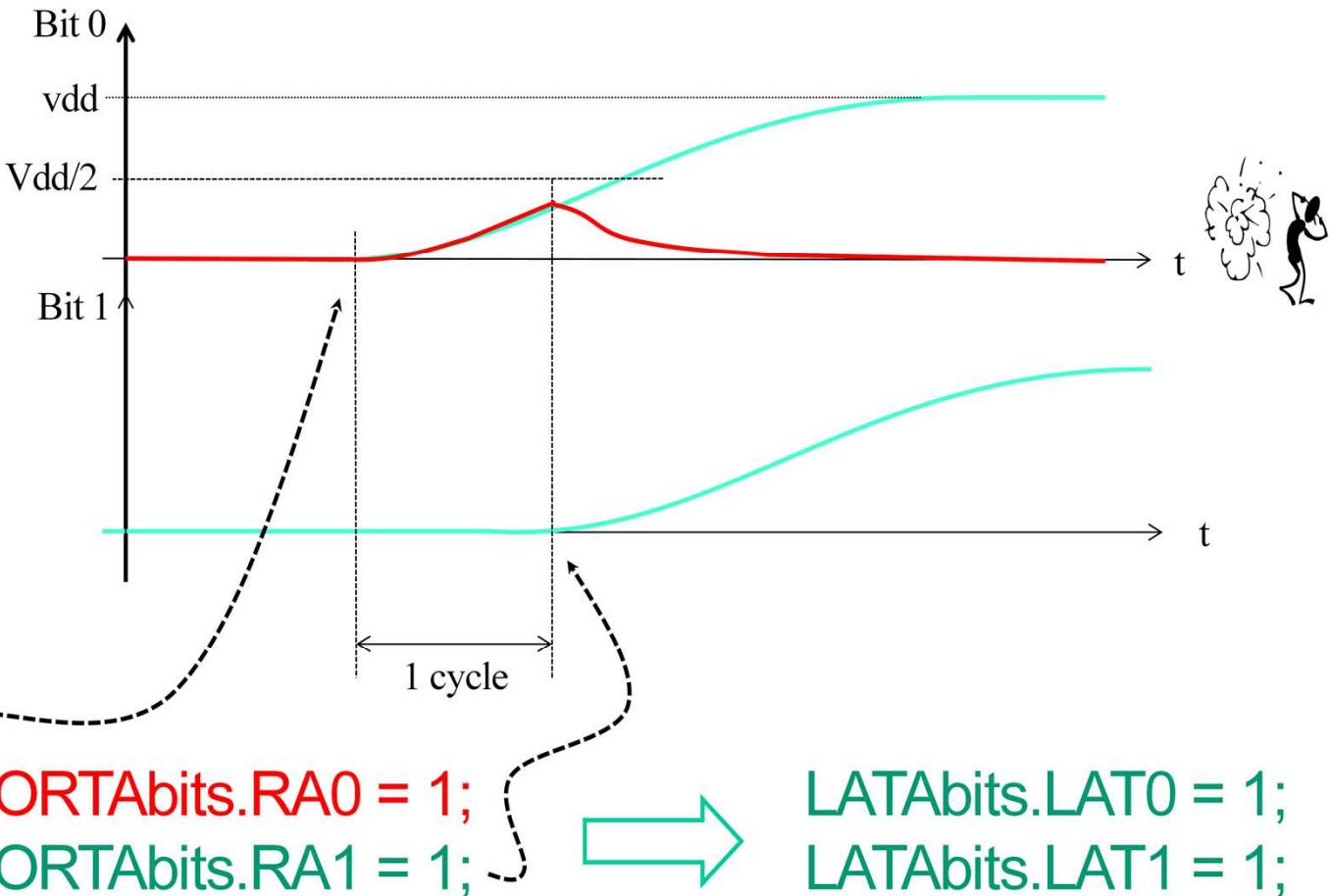
## Internal Data Bus



Le registre LATx est un registre de mémorisation des sorties (LATCH). Il est recommandé d'utiliser le registre LATx en lecture plutôt que PORTx, car les broches PORTx sont soumises aux phénomènes physiques tels que la charge capacitive par exemple et leur niveau n'est pas toujours prédictible donc source de problème logiciel.

notes





La principale raison de l'ajout des registres LAT est la protection contre les problèmes issus des instructions read-modify-write. Contrairement à ce que vous pouvez penser, les instructions sur les bits ou les rotations de bits agissent en lisant l'ensemble du port, en modifiant le bit et en écrivant sur l'ensemble du port à nouveau. En fait on ne modifie jamais directement le bit.

Exemple:

`PORTAbits.RA0 = 1;` s'exécute en fait en interne comme ceci:

`PORTAbits.RA1 = 1;`

`PORTA = PORTA|0x01;`

`PORTA = PORTA|0x02;`

Et donc en cas de forte charge capacitive ou de fréquence élevée le problème décrit dans le transparent survient. La solution consiste à utiliser les registres LATx en lieu et place des registres PORTx.

notes

\*

### \* FONCTION: Clignotement

\*

Créer un fichier clignotement.c

\*

Faire clignoter la led D8 à environ 1s ou 2s suivant le bouton poussoir S0

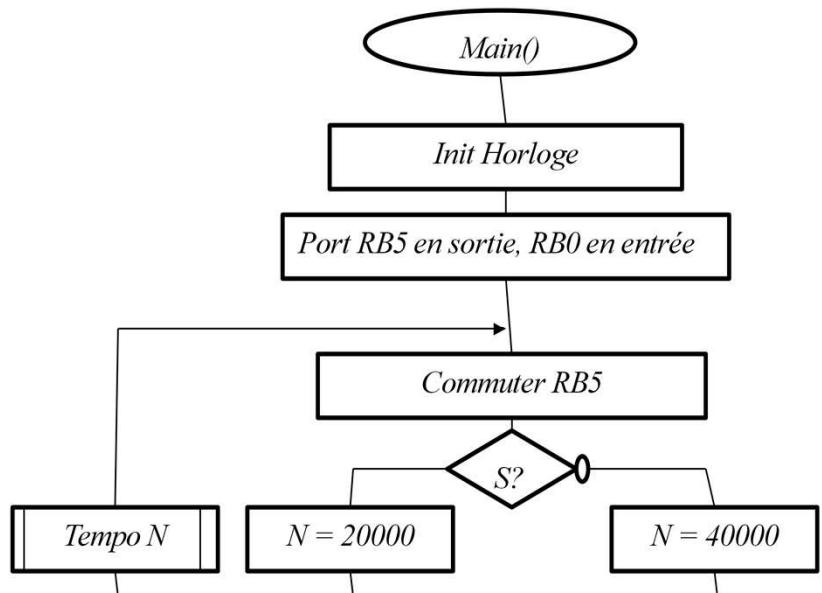
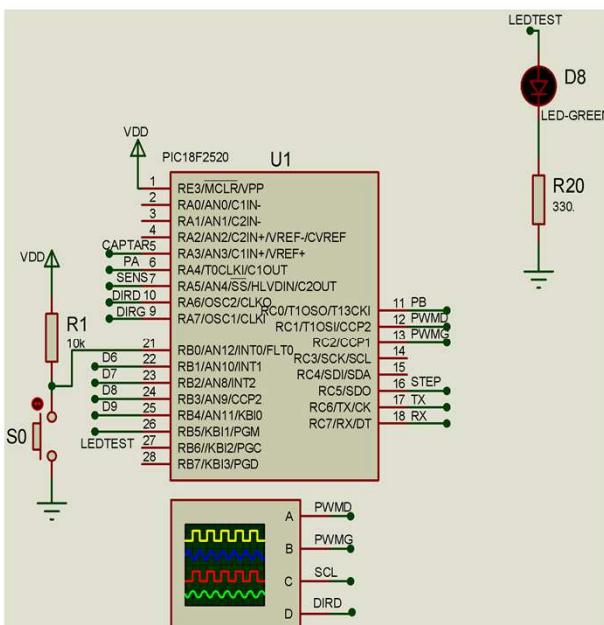
\*

Une boucle sera utilisée pour la temporisation.

\*

Visualiser LEDTEST sur l'oscilloscope

\*



notes



\*

**\* FONCTION: Clignotement**

\*

Créer un fichier clignotement2.c

\*

Faire clignoter la led D8 à environ 1s avec un rapport cyclique ¼ si S0 est appuyé et ½ sinon.

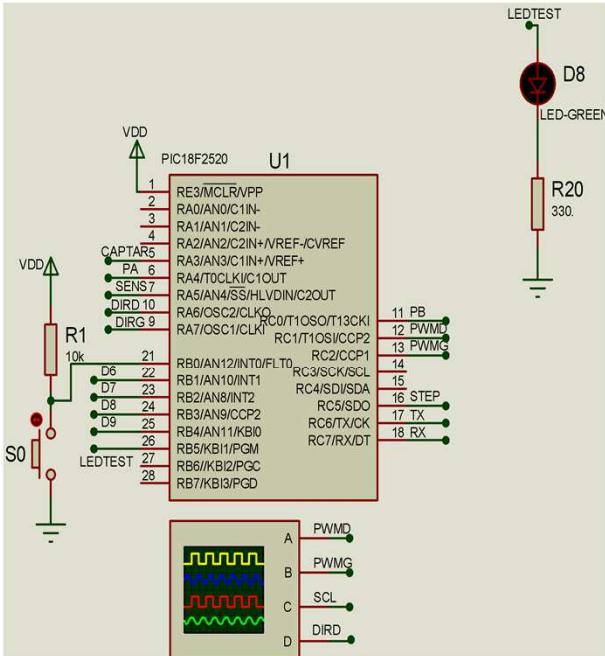
\*

Dessiner l'organigramme avant de coder

\*

Visualiser LEDTEST sur l'oscilloscope

\*



\*

**\* FONCTION: Comptage**

\*

Créer un fichier comptage.c

\*

Compter le nombre d'appui sur le bouton S0

\*

Afficher ce nombre sur le port C

\*

Dessiner l'organigramme avant de coder

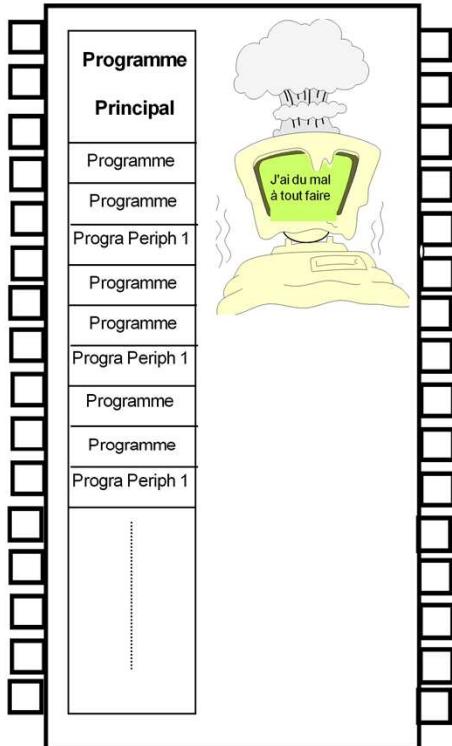
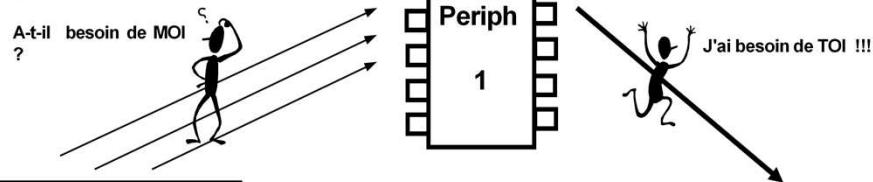
\*

\*

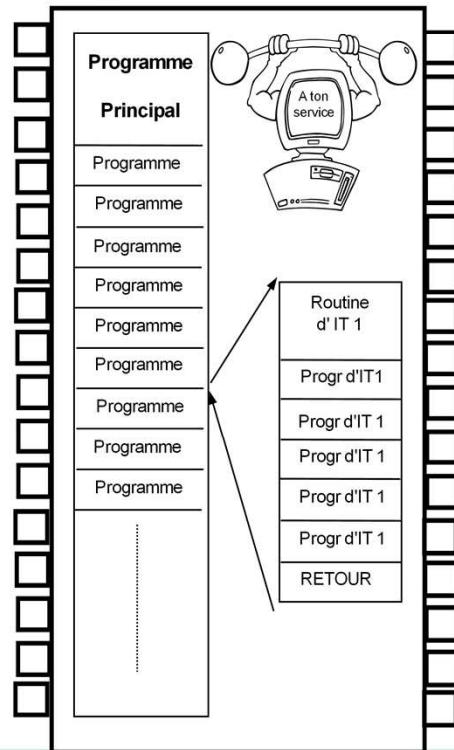
notes



## ○ Polling



## ○ **Interruption**



Pour expliquer les interruptions, prenons l'exemple d'un robot de soudage automobile avec un capteur de température déporté. Comment gérer ce capteur de température chargé de détecter les débuts d'incendie sans trop perturber le programme d'application chargé d'asservir le robot ?

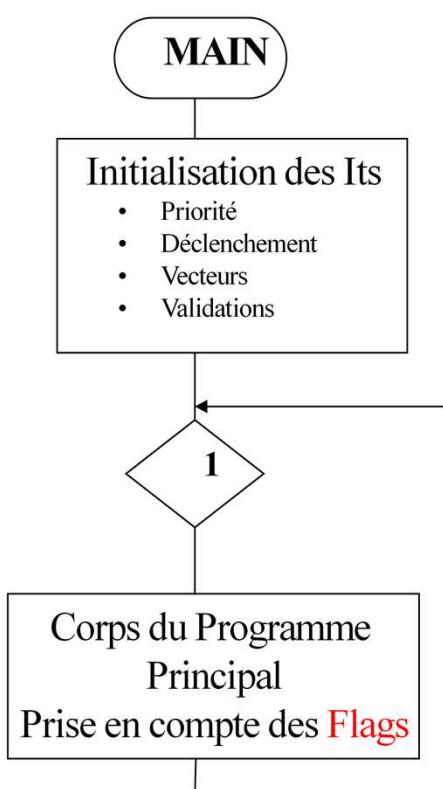
Première solution: gestion par polling ou scrutation, le programme d'application est instrumenté périodiquement avec des instructions de lecture du capteur. Inconvénient la charge CPU augmente pour rien puisque la plupart du temps il n'y a pas d'incendie (faut espérer), pas de temps réel, le temps de réaction du système dépend de la période de scrutation.

Deuxième solution: Le périphérique n'enverra un signal que si la température dépasse un seuil. Donc en fonctionnement normal le programme d'application n'est pas impacté et si un incendie est détecté un signal est reçu sur une broche particulière du microcontrôleur, appelé entrée d'interruption, qui déroutera automatiquement et instantanément le programme vers un sous-programme d'interruption de sécurité. Avantages: charge CPU non impacté, temps réel.



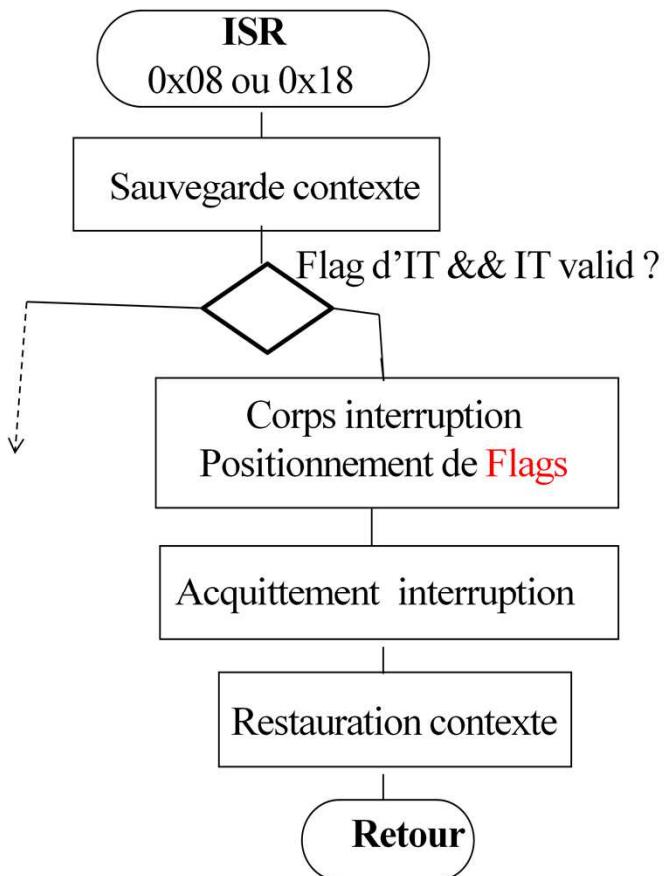


## ○ Programme principal



Flags = variables globales de type volatile

## ○ Routine d'interruption



Soyez méthodique, avant de programmer, un organigramme ou un diagramme d'activité UML est bienvenu. Toujours la même démarche isoler les problèmes: problème d'algorithme, d'architecture de logiciel et problème purement de codage. Les routines d'interruptions sont décrites à part. Il faut toujours chercher à minimiser le corps des routines d'interruption, c'est pourquoi il est conseillé de simplement positionner des flags dans la routine d'interruption que l'on testera dans la boucle principale du main. L'avantage de cette démarche est qu'elle permet de tester les différentes boucles du programme même si l'interruption ne fonctionne pas, en positionnant manuellement les flags. L'autre avantage c'est qu'en minimisant les traitements dans la routine d'interruption, on minimise également le contexte à sauvegarder donc commutation dans l'interruption plus rapide et moins de mémoire pile utilisée.

Si on utilise les interruptions, le programme principal est forcément constitué d'une boucle.

**On n'appelle jamais une fonction d'interruption dans le main.** Une interruption est en C une fonction qui ne prend aucun paramètre et ne renvoie aucun paramètre, les échanges avec le main se feront via des variables globales, attention elles devront être de type volatile. Une fonction d'interruption n'appelle pas d'autres fonctions. Bien entendu avant de pouvoir utiliser la fonction d'interruption il faut l'initialiser dans le main: Priorité, type de déclenchement, masque et enfin en dernier validation globale.

Sur les pic18 il n'existe que 2 vecteurs 0x08 et 0x18 donc toutes les interruptions renvoient vers ces l'un de ces 2 vecteurs d'où la nécessité dans la routine d'interruption de scruter les flags d'interruption pour déterminer quel événement traiter.

notes



```
void HighISR(void);
```

- Prototype

```
#pragma code HighVector=0x08
void IntHighVector(void)
{
    _asm goto HighISR _endasm
}
#pragma code
```

- Vecteur d'interruption
  - High Priority = 0x08
  - Low Priority = 0x18

```
#pragma interrupt HighISR save=var
void HighISR(void)
{
    /* VOTRE CODE ICI */
}
```

- Sauvegarde contexte
- High priority interrupt  
#pragma interrupt
- Low priority interrupt  
#pragma interruptlow

Nous sommes presque prêts à écrire notre routine d'interruption, mais nous devons d'abord dire au compilateur que notre fonction est une routine d'interruption pour qu'il puisse insérer la sauvegarde et la restauration du contexte approprié constituée des registres critiques. Il doit également mettre fin à cette fonction avec une instruction retfie au lieu d'une instruction de retour ordinaire. Nous accomplissons ceci de deux façons suivant la priorité haute ou basse de la fonction d'interruption. Pour une interruption de haute priorité, nous utilisons **#pragma interrupt functionName**, où functionName est le nom de notre routine d'interruption. Pour une interruption de faible priorité nous utilisons **#pragma interruptlow functionName**. La raison de ces directives différentes est qu'une interruption haute priorité enregistre toujours les registres critiques du contexte dans des registres miroirs de manière matérielle donc rapide, alors qu'une interruption de faible priorité n'a pas ce mécanisme matériel automatique, et donc le compilateur doit générer du code pour accomplir la même tâche.

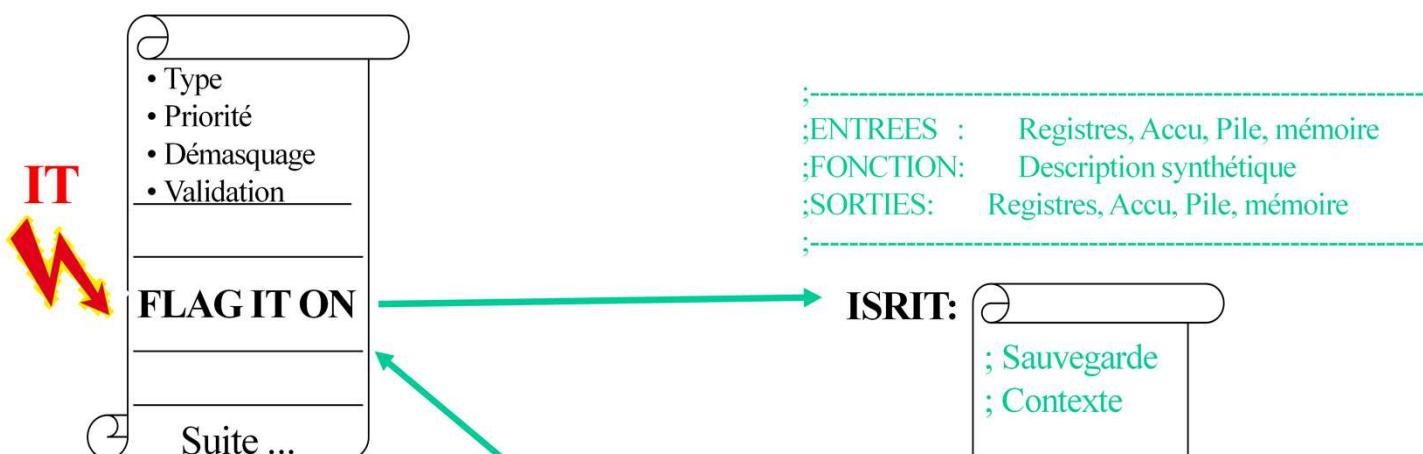
Le nom de la fonction d'interruption peut être quelconque, la seule condition c'est qu'elle ne prenne et ne renvoie aucun paramètre. Les échanges avec le main se font via des variables globales. **Attention à déclarer ces variables globales de type volatile**, en effet le compilateur est incapable de déterminer quand ces variables vont être modifiées et donc il faut désactiver ses optimisations sous peine de catastrophe.

Remarque: seul le contexte principal est sauvegardé, par le compilateur le contexte additionnel peut-être sauvegardé via la commande save (exemple la variable var).

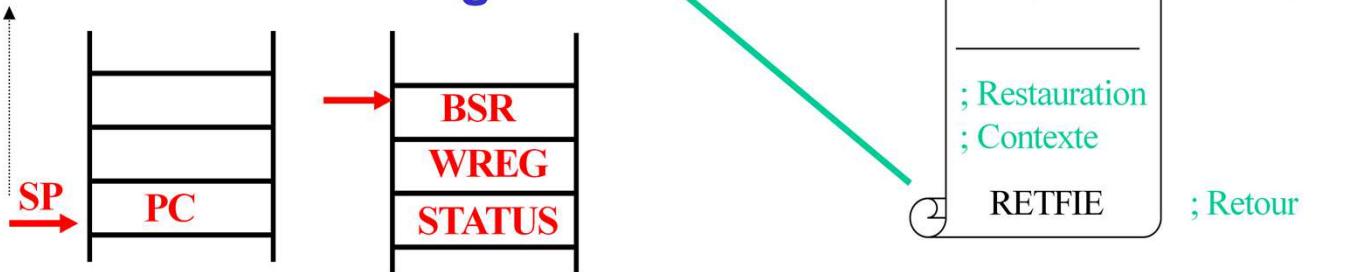
Notes



## ○ Mécanisme



## ○ Pile matérielle / logicielle



Le contexte c'est l'ensemble des registres et accumulateurs utilisés par l'ISR. C'est très important de bien sauvegarder ce contexte, car l'IT peut interrompre l'exécution du main n'importe où et donc une mauvaise sauvegarde/ restauration de contexte plante complètement le main. En assembleur c'est au programmeur de gérer la sauvegarde et la restauration du contexte en RAM, par contre en C cette sauvegarde et restauration est automatiquement faite par le compilateur en début et fin d'ISR.

Les 3 registres principaux ( WREG, BSR, STATUS ) sont sauvegardés automatiquement par le compilateur, si d'autres contextes additionnels sont à sauvegarder, il faudra le faire explicitement par programmation.

BSR ( Bank Select Register ) : contient le numéro d'une des 16 pages de données.

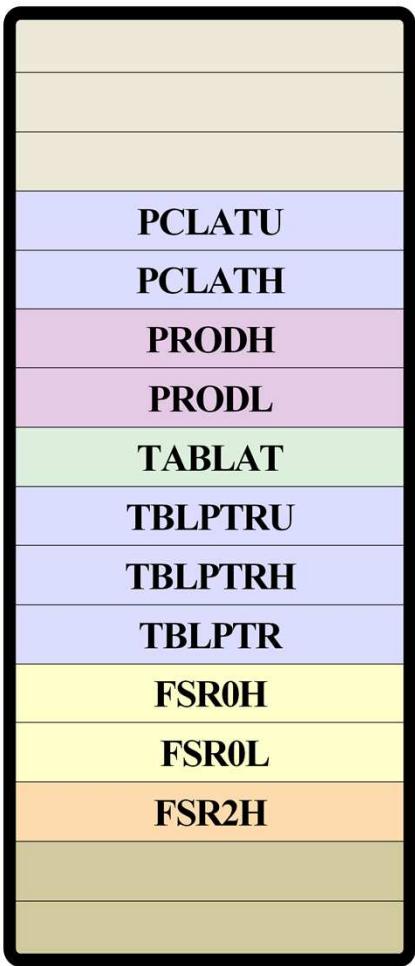
WREG ( working REGister ) : c'est l'accumulateur 8 bits

STATUS : registre contenant les flags de statut associé à l'accumulateur

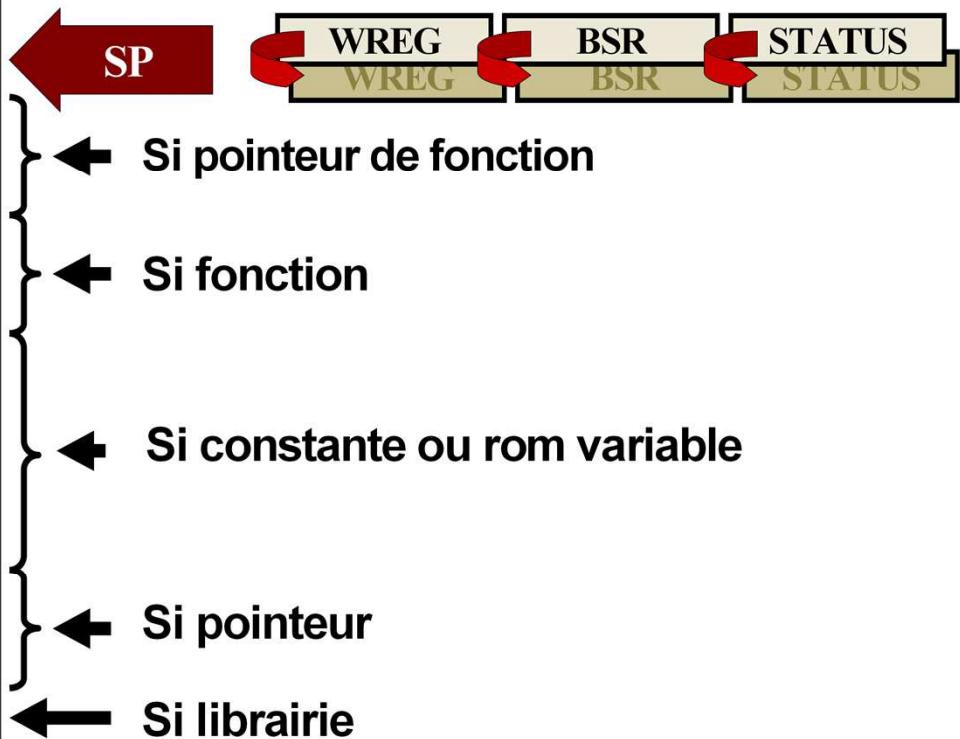
notes



Increasing Addresses ↑



## SI High Priority Interrupt



Dans les PIC18 il existe 2 piles: une pile matérielle de 31 mots qui sauvegarde uniquement l'adresse de retour ( PC ) et une pile logicielle qui sauvegardera le reste du contexte c'est-à-dire tous les registres critiques modifiés par la routine d'interruption. En fait la pile logicielle n'est pas une vraie pile, mais une zone de Ram pointée par un registre FSR. Malgré tout la sauvegarde et la restauration se font de manière automatique par le compilateur à condition d'avoir réservé suffisamment de place dans la mémoire.

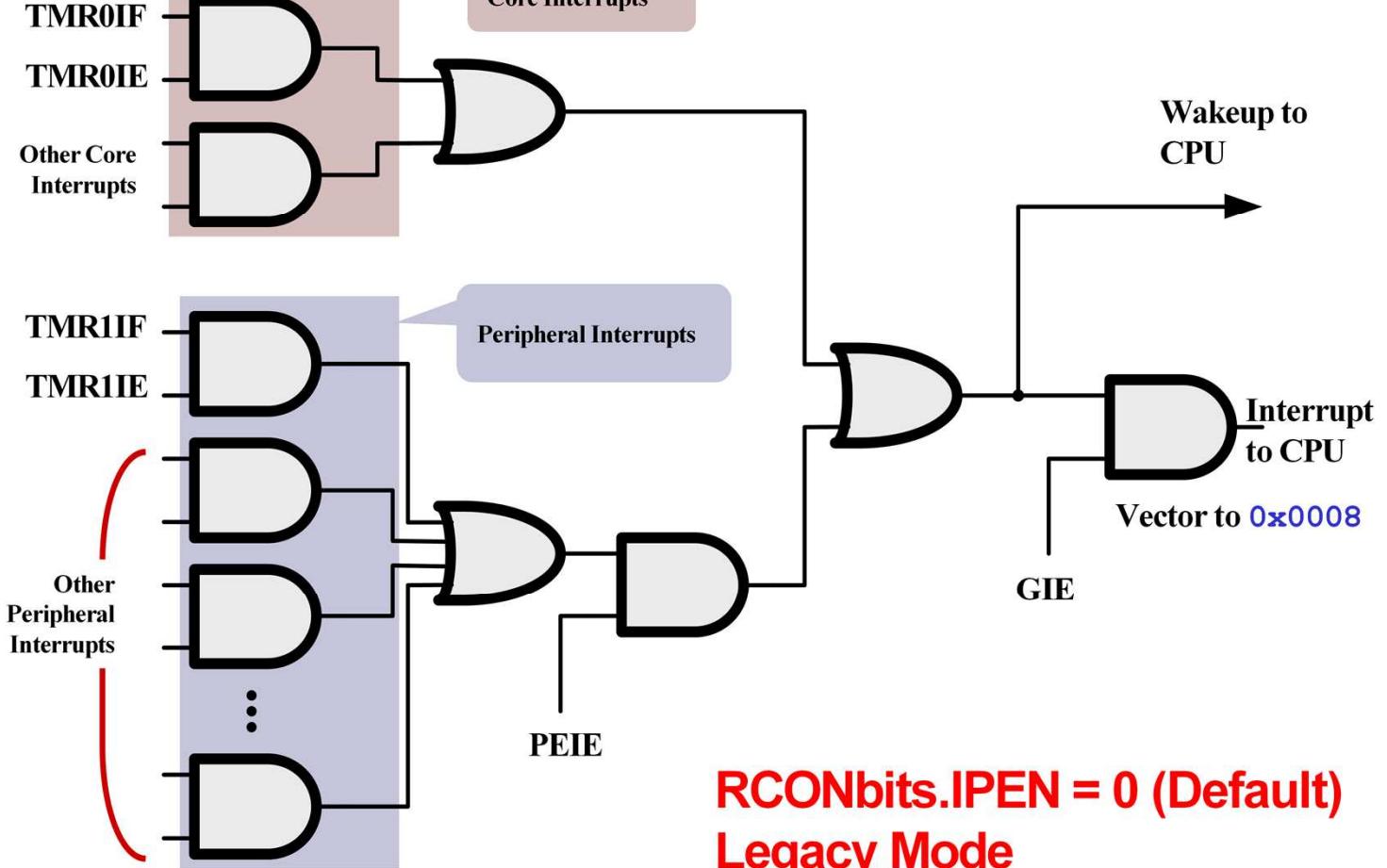
L'exemple ci-dessus montre la montée de la pile dans un cas extrême, mais pas le pire. À noter que les 3 registres critiques, qui sont toujours sauvegardés, le sont dans des registres miroirs lorsque l'interruption est prioritaire ( plus rapide ) et dans la pile lorsque l'interruption est de basse priorité.

WREG ( Working REGister ) : accumulateur 8 bits

BSR ( Bank Select register ): Designe la page de données utilisées.

STATUS: Flag de statut associé à l'accumulateur

notes

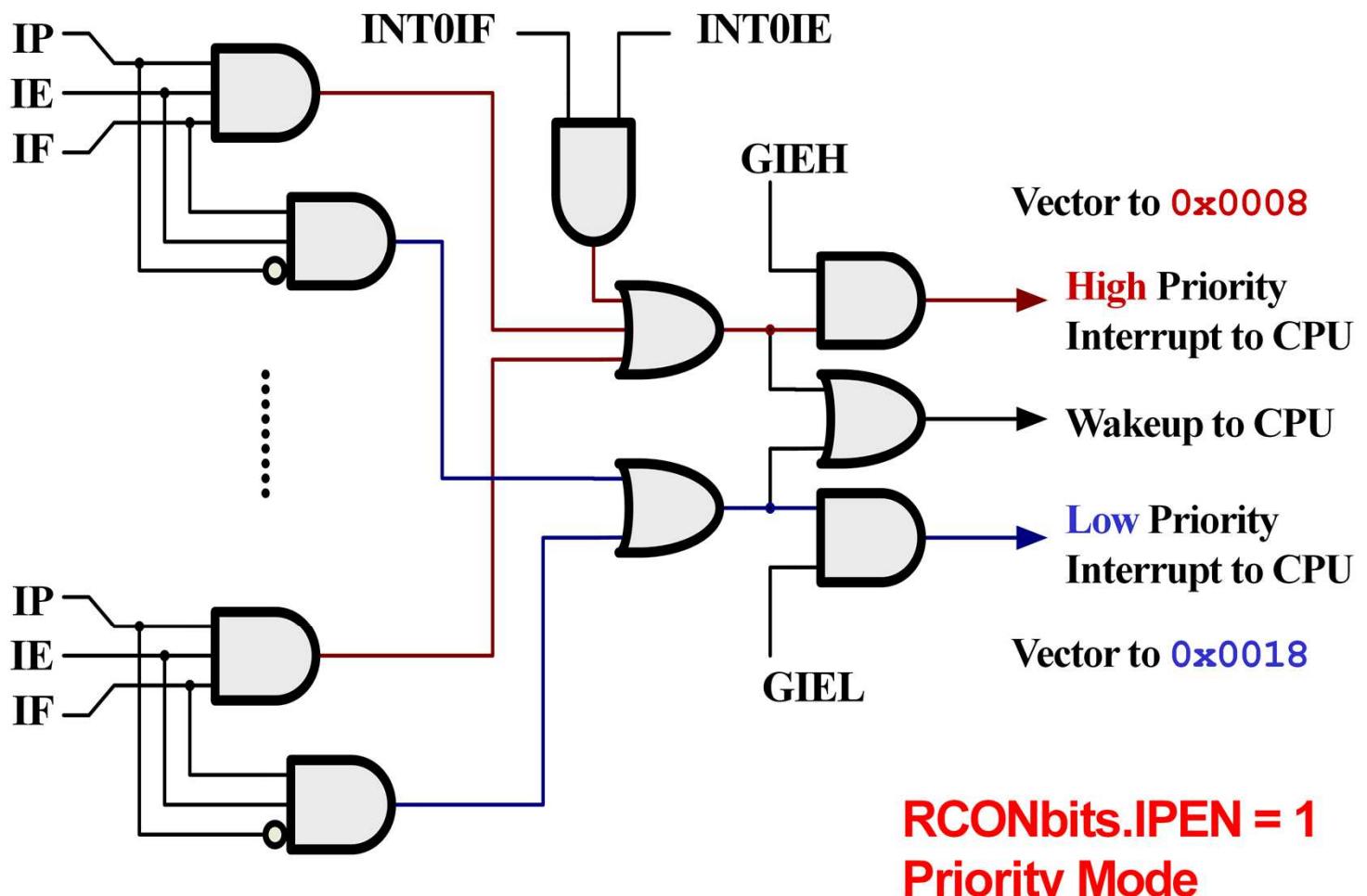


**RCONbits.IPEN = 0 (Default)  
Legacy Mode**

Il existe 2 modes pour les interruptions legacy et priority. Legacy est le mode de compatibilité avec la famille PIC16 ou un seul vecteur existe et pas de priorité. On remarque que le timer 0 est dans le cœur du PIC18 alors que le timer1 fait partie des périphériques et donc attention à valider PEIE = 1.

Dans ce mode 2 flags caractérisent les interruptions Flag d'IT et Flag de validation. On remarque le flag de validation globale GIE. Toutes les interruptions peuvent réveiller le microcontrôleur, le sortir du mode basse consommation (< 100 nA). Seul le vecteur 0x08 est valide.

notes



Dans le priority mode, 3 flags caractérisent les interruptions: Flag d'IT, Flag de validation et Flag de priorité. Interruption de haute et de basses priorités ont des vecteurs différents. On remarque que l'on peut valider les interruptions de haute priorité et de basse priorité de façon indépendante (GIEH et GIEL). L'interruption externe INT0 est par construction de type prioritaire.

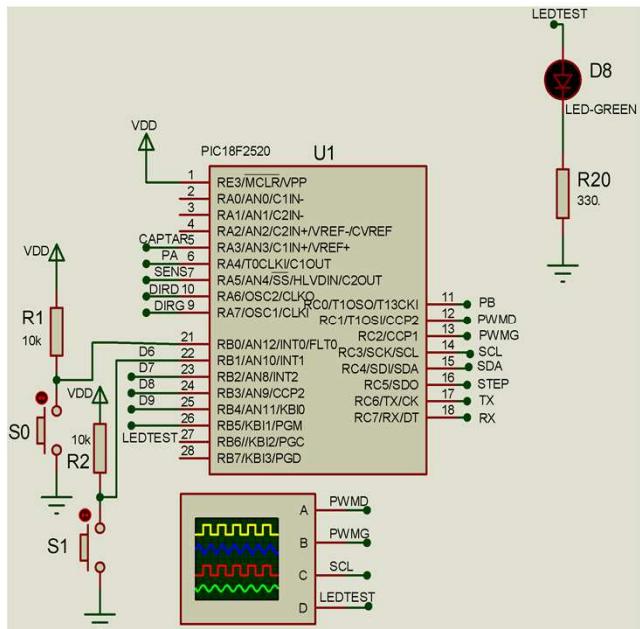
Le pic18f2520 possède 3 entrées d'interruptions externes INT0, INT1, INT2, le type de déclenchement de ces interruptions, front montant ou front descendant, sont positionnés par les flags INTEDGx.

notes

\*

## \* FONCTION1: Interruption

- \* Ajouter interruption.c dans MPLAB
- \* Analyser et commenter le programme
- \* Dessiner l'organigramme du programme
- \* Tester le avec le simulateur proteus
- \*



notes





\*

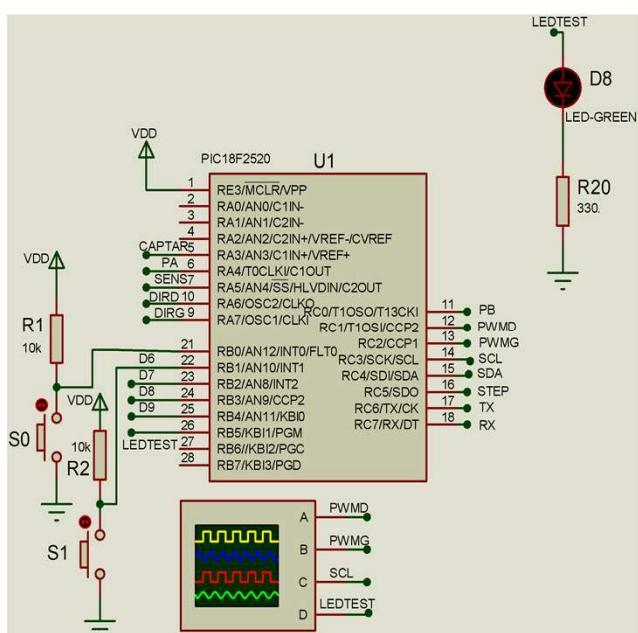
## \* FONCTION2: Interruption

Ajouter une interruption INT1 sur front montant faisant clignoter D8 à 1s, INT0 doit continuer à fonctionner. Utiliser le mode legacy

Dessiner l'organigramme du programme avant de programmer

Tester le avec le simulateur proteus

\*



\*

## \* FONCTION3: Utiliser le mode priority

INT1: priorité 1

INT0: priorité 0

\*Compter le nombre d'appui sur INT0 ,

\* l'afficher sur le port C

\*

notes



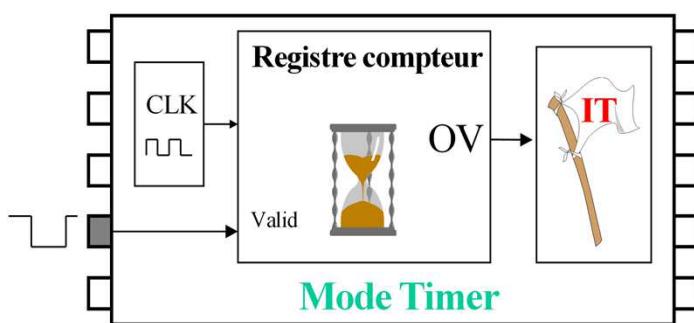
- Comptage d 'événements ( mode compteur )
- Base de temps ( mode timer )
  - Temporisation
  - Séquenceur
- Génération de signaux ( mode comparaison )
  - Périodiques
  - Modulé en rapport cyclique ( signal PWM )
  - Modulé en fréquence ( signal FM )
  - Impulsions calibrée
- Mesure signaux ( mode capture )
  - Mesure période, fréquence
  - Mesure intervalles de temps, retards

Les timers sont certainement les périphériques les plus utilisés dans les microcontrôleurs, ils introduisent la notion de temps. 4 modes de fonctionnement sont possibles: mode compteur le plus simple, compte le nombre d'impulsions sur une patte externe, mode timer piloté par un signal d'horloge, fourni, une information temporelle précise, à la période d'horloge prête. En mode comparaison la génération de signaux PWM ou FM est possible et en mode capture la mesure de périodes ou d'intervalles de temps est possible.

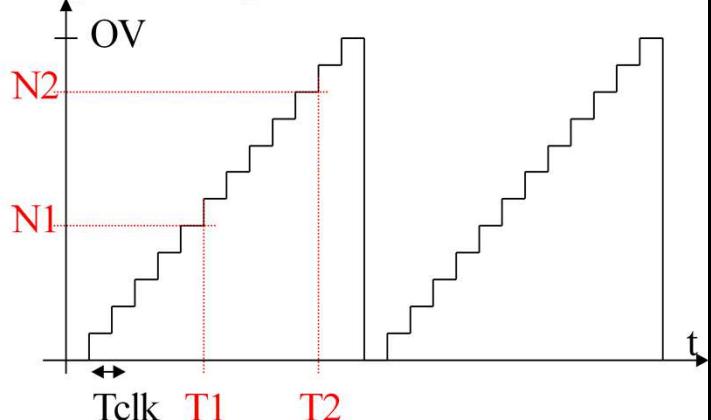


notes

## ○ Principe



Registre Compteur



## ○ Mesure temporelle

$$T_2 - T_1 = (N_2 - N_1) \times T_{clk}$$

$$(T_2 - T_1)_{\max} = (\text{Capacité de comptage max}) \times T_{clk}$$

Sinon tenir compte des OVERFLOWS

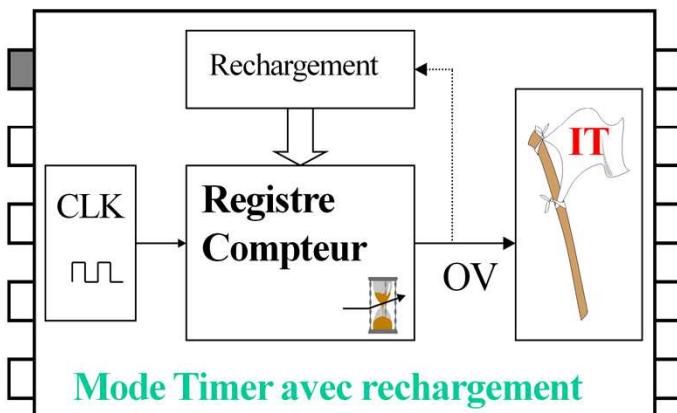
Le principe du mode timer est illustré ici, le compteur est piloté par une horloge interne ou externe, son contenu varie donc linéairement avec des overflow périodiques. Il est possible sur les overflow de déclencher une interruption réalisant ainsi un séquenceur périodique. En lisant à la volée le compteur, on peut mesurer l'intervalle de temps écoulé à la période d'horloge prête. Si l'intervalle de temps dépasse la capacité de comptage maximale du compteur, il faudra prendre également en compte le nombre d'overflow dans l'intervalle.

notes

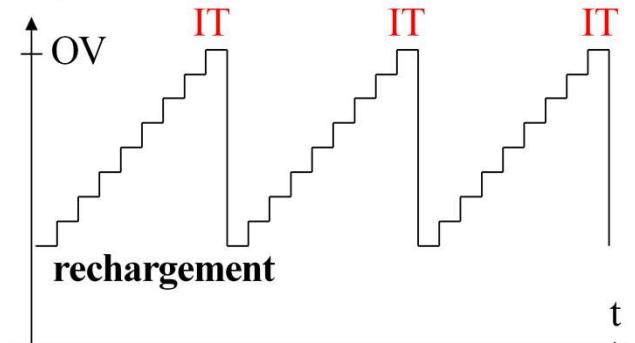




## ○ Avec rechargement



Registre  
Compteur



## ○ Base de temps

$$\text{Période} = (\text{OV} - \text{rechargement}) \cdot \text{Tclk}$$

L'inconvénient du fonctionnement précédent c'est que l'on ne peut pas régler la période des interruptions, des overflow. Pour cela un fonctionnement timer avec recharge est nécessaire. La valeur de recharge à l'overflow fixe la période des interruptions, le timer peut alors servir de base de temps.

Remarque: selon le périphérique le recharge peut-être matériel (automatique) ou logiciel. Il faut dans ce dernier cas utiliser l'interruption d'overflow pour initialiser le timer à la valeur de recharge.

Le PIC18f2520 possède 4 timers 8 ou 16 bits avec les modes de fonctionnement suivant:

Timers 0: Compteur/Timer 8/16 bits

Timer 1 et 3: Possède en plus un mode capture et comparaison

Tilmer 2: optimisé pour la génération PWM.

notes



# Timer Comparison

Projet Pilotage Robot

	TIMER 0	TIMERS 1 & 3	TIMERS 2
<b>SIZE OF REGISTER</b>	8-bits or 16-bits	16-bits	8-bits
<b>CLOCK SOURCE (Internal)</b>	Fosc/4	Fosc/4	Fosc/4
<b>CLOCK SOURCE (External )</b>	T0CKI pin	T13CKI pin or Timer 1 oscillator (T1OSC)	None
<b>CLOCK SCALING AVAILABLE (Resolution)</b>	Prescaler 8-bits (1:2→1:256)	Prescaler 2-bits (1:1, 1:2, 1:4, 1:8)	Prescaler (1:1,1:4,1:16) Postscaler (1:1→1:16)
<b>INTERRUPT EVENT</b>	On overflow FFh→00h	On overflow FFFFh→0000h	TMR REG matches PR2
<b>CAN WAKE PIC FROM SLEEP?</b>	NO	YES	NO

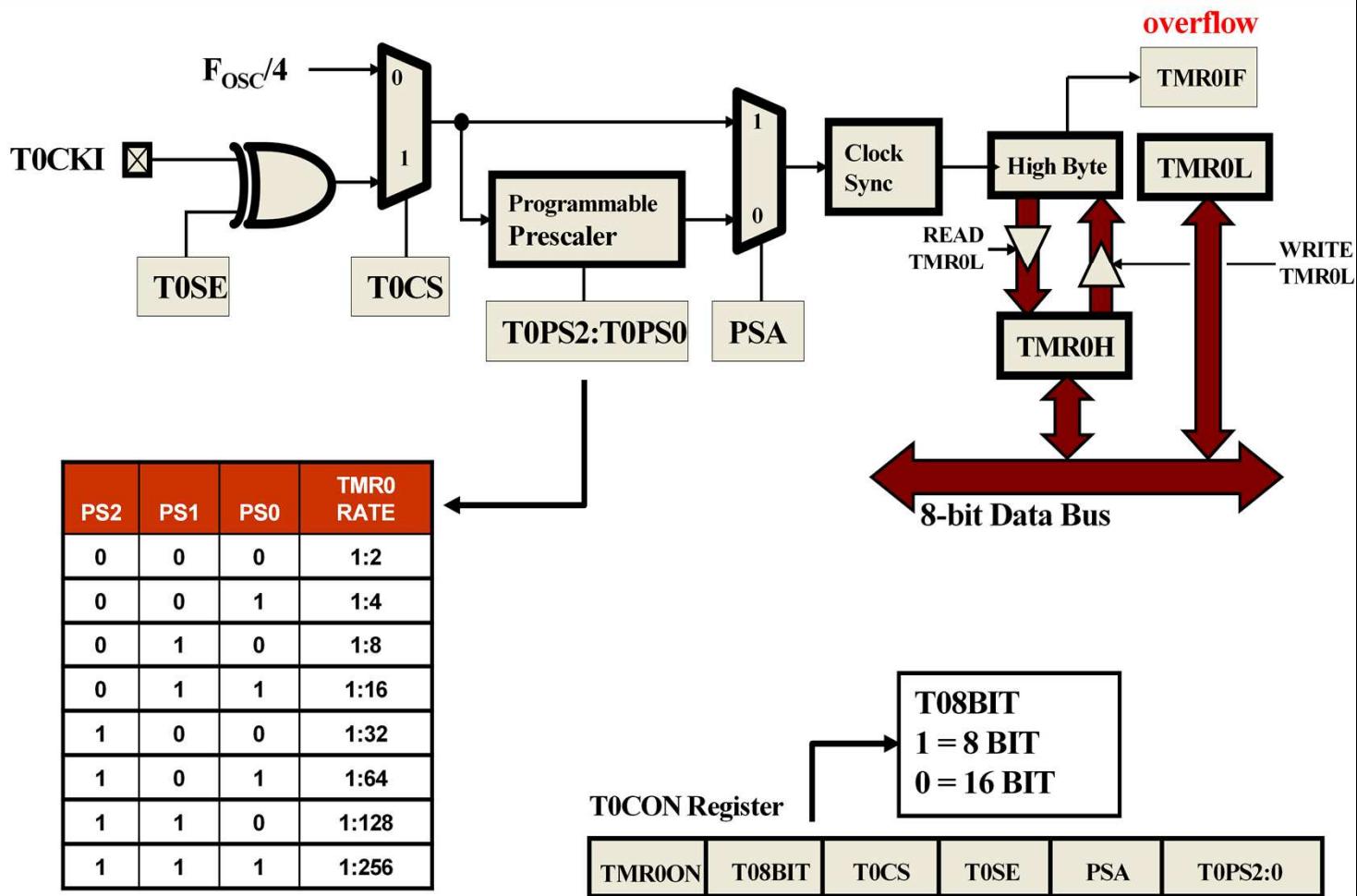
notes





## Projet Pilotage Robot

PERIPHERIQUES: horloge - ports - interruptions - timer - can - uart - i2c



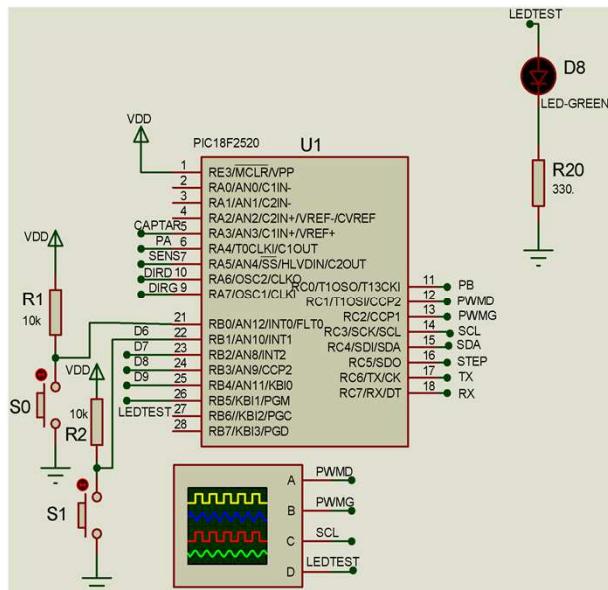
Le timer 0 peut fonctionner en 8 bits (overflow 0xFF) ou 16 bits (overflow 0xFFFF) suivant le flag T08BIT. Le périphérique peut fonctionner en timer via l'horloge interne Fosc/4 ou en compteur via la broche RA4 (T0CKI). Le flag T0SE permet alors de sélectionner une incrémentation sur front montant ou descendant. Un prescaler permet de régler l'horloge en appliquant un rapport de division allant de 1 à 256. Attention pour faire fonctionner le timer 0 il faut valider le périphérique via le flag TMR0ON = 1.

notes





- \* \* FONCTION1: Timer 0
- \* Créer un fichier timer0.c dans MPLAB
- \* Réaliser un programme faisant clignoter D8 à 2 s précise
- \* L'horloge reste positionnée à 4 Mhz
- \* Dessiner un organigramme avant de programmer
- \* Mesurer la période avec l'oscilloscope.





\*

## \* FONCTION2: Timer 0

\*

\*

\*

\*

\*

\*

\*

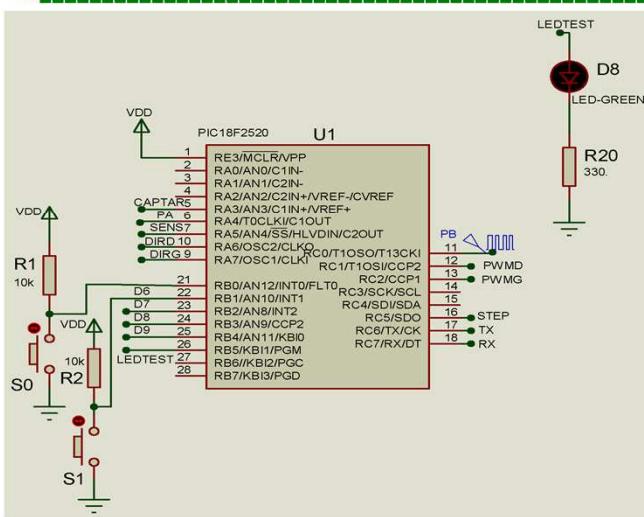
\*

\*

**Sur le robot le déplacement est mesuré en comptant le nombre d'impulsions sur les entrées PA et PB. Ces impulsions sont issues des roues codeuses positionnées sur les moteurs des chenilles.**

**Compter le nombre d'impulsions sur PB. Les impulsions seront simulées via la sonde Isis dpattern.**

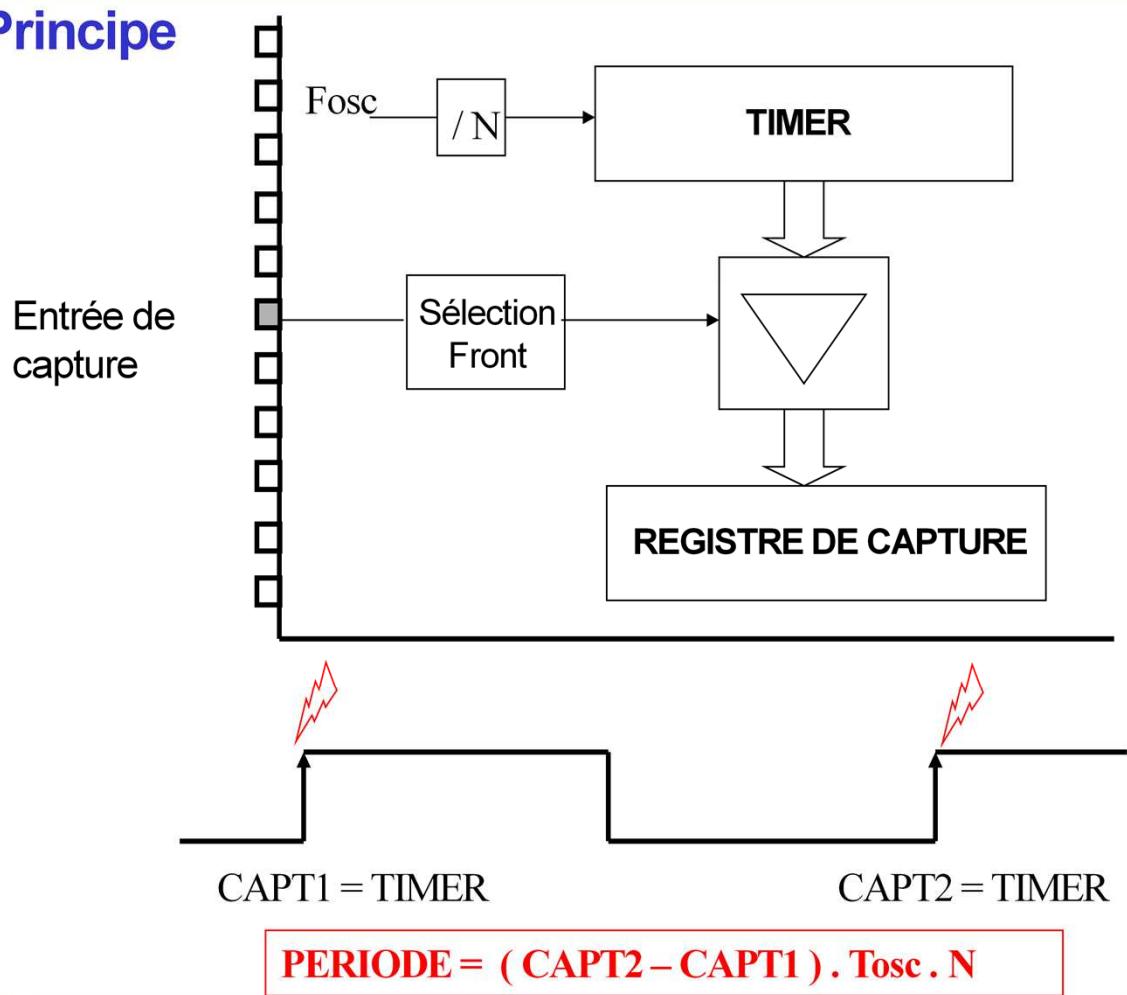
**L'appui sur S0 entraîne le clignotement ( 2 s ) de la led D8 un nombre de fois identique. S0 sera géré par interruption.**



notes



## ○ Principe



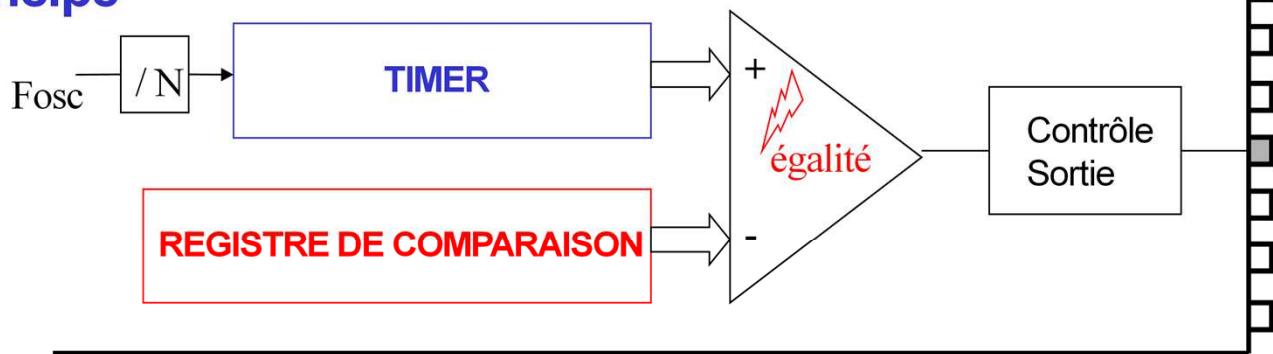
Le mode capture se caractérise par un registre de capture chargé de stocker le contenu du timer à un instant donné. Cette méthode est beaucoup plus précise que de lire à la volée le timer par logiciel, même en utilisant les interruptions. Les temps de latence faussent le résultat.

notes

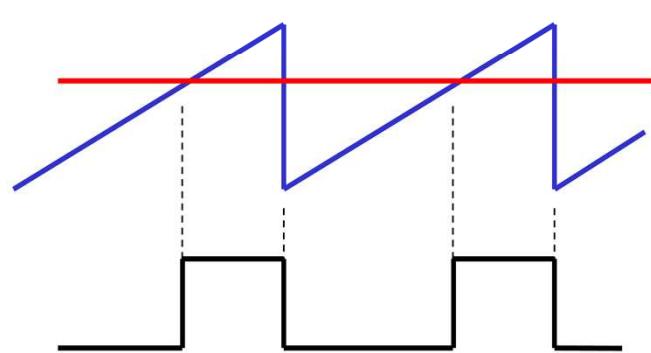




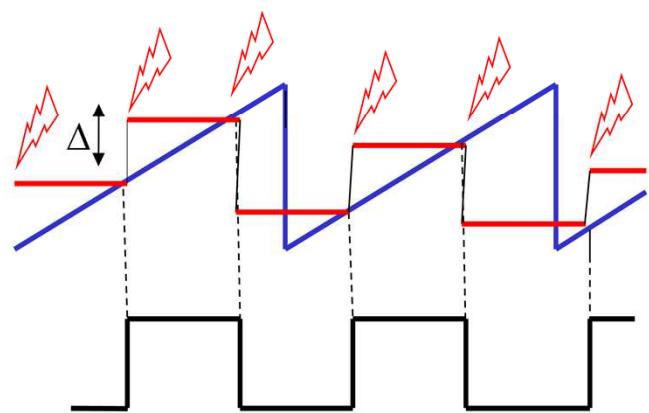
## ○ Principe



PWM



FM



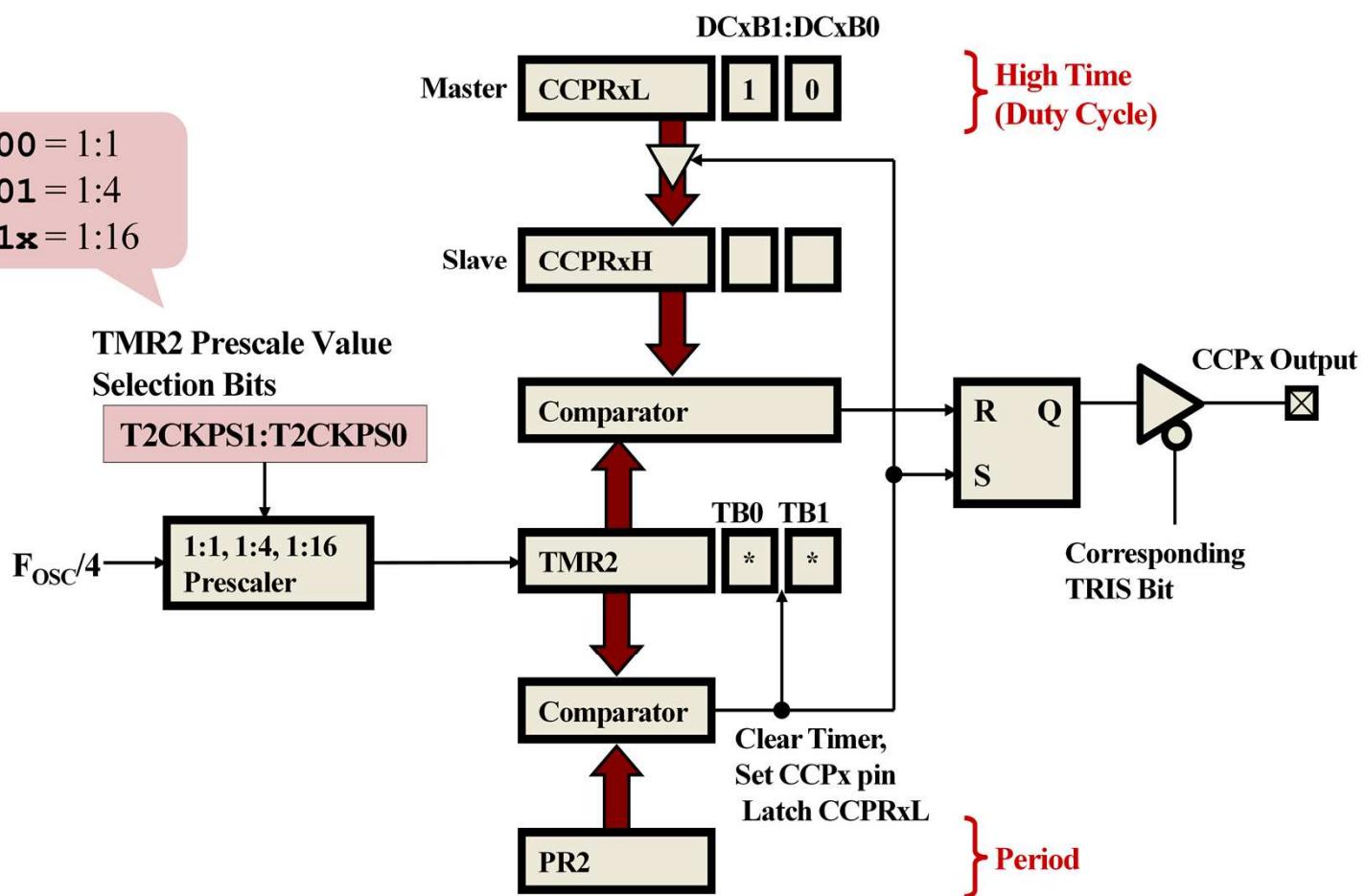
Le mode comparaison est basé sur un comparateur entre le timer et un seuil programmé dans un registre. Le résultat de cette comparaison est un créneau périodique dont le rapport cyclique est fixé par le seuil de comparaison (signal PWM). La fréquence du PWM peut être ajustée si on utilise un timer avec recharge. Pour générer un signal FM, on peut utiliser l'interruption « égalité » entre le timer et le registre de comparaison pour venir à chaque fois modifier le seuil, le delta fixe la fréquence ( $F = 1/2 \cdot \Delta \cdot T_{osc} \cdot N$ ).

notes





**00 = 1:1**  
**01 = 1:4**  
**1x = 1:16**



Seul le timer 2 permet la génération d'un signal PWM sur 2 sorties possibles CCP1 et CCP2, respectivement RC2 et RC1 par défaut. Remarque la sortie CCP2 peut être redirigée en RB3 via la configuration #pragma config CCP2MUX = OFF.

PR2 permet de fixer la période du signal PWM avec une résolution de 8 bits.

CCPR1L permet de fixer le rapport cyclique avec une résolution augmentée de 10 bits sur CCP1.

CCPR2L permet de fixer le rapport cyclique avec une résolution augmentée de 10 bits sur CCP2.

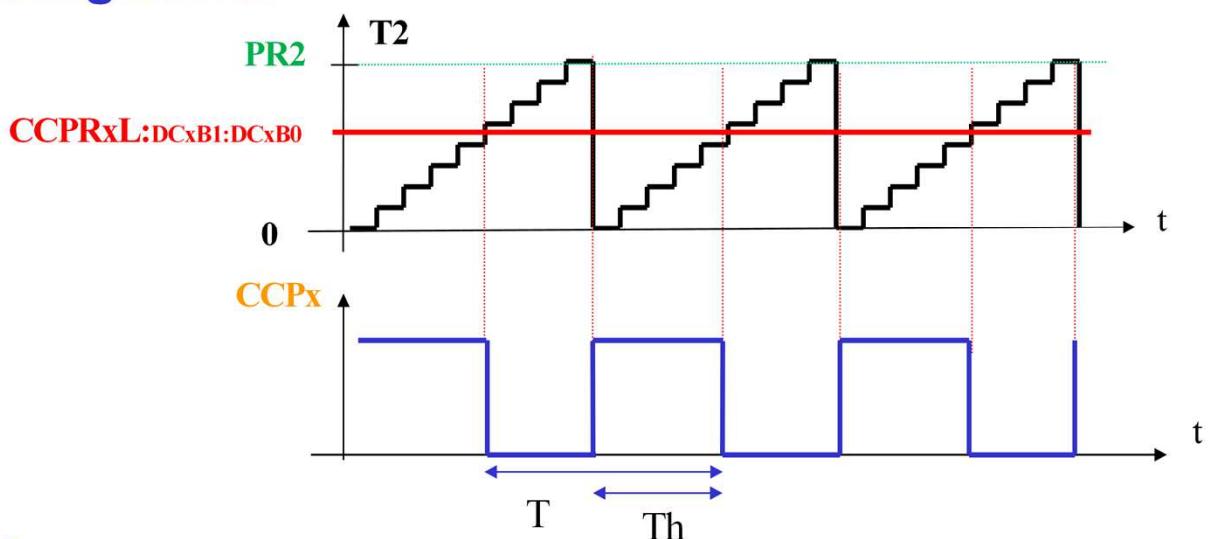
Le périphérique peut donc générer 2 PWM simultanément de même fréquence (fixée par PR2), mais de rapport cyclique programmable indépendamment via CCPR1L ou CCPR2L.

Les bits TB0 et TB1 ne sont pas accessibles, ils ont été ajoutés pour former un compteur 10 bits et ainsi augmenter la résolution du PWM à 10 bits

notes



## ○ Chronogramme



## ○ Timings

$$PR2 = \frac{f_{OSC}}{4 \cdot f_{PWM} \cdot \text{Prescaler}} - 1$$

$$CCPRxL:DCxB1:DCxB0 = \frac{\%RC_{PWM} \cdot f_{OSC}}{100 \cdot \text{Prescaler} \cdot f_{PWM}}$$

Remarque: CCPRxL est mémorisé dans CCPRxH à chaque overflow du timer, ce qui permet de synchroniser la mise à jour du seuil de comparaison avec le timer et donc d'éviter les glitches sur les signaux PWM lors des modifications de seuil à la volée.

Critère de choix du prescaler: PR2 le plus grand possible inférieur à 255

notes





\*

## \* FONCTION: PWM

\*

Créer un fichier **timer2.c** dans MPLAB

\*

Dans le projet robot PWMD et PWMG permettent de commander respectivement les chenilles droite et gauche du robot.

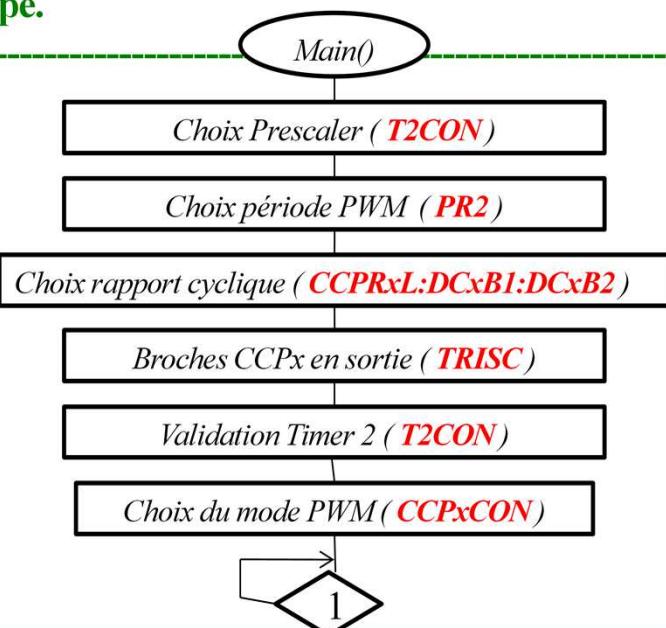
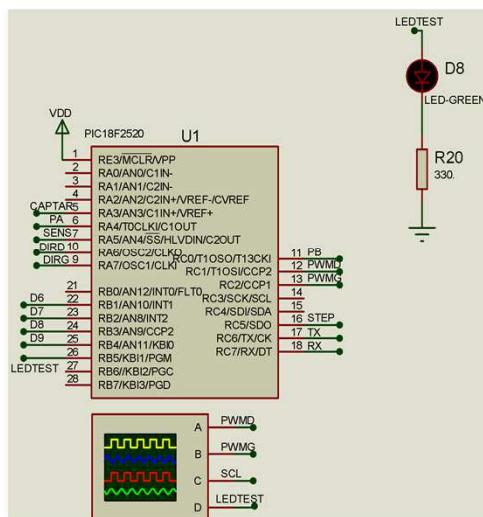
\*

Réaliser un programme commandant le moteur gauche à une vitesse correspondant à un rapport cyclique de 20 % (  $F_{PWM} = 2 \text{ KHz}$  ).

\*

Vérifier PWMG à l'oscilloscope.

\*



Choisir le prescaler le plus faible possible pour obtenir la plus grande précision possible sur le rapport cyclique.

Remarque: vous pouvez également utiliser l'analyseur logique de Mplab pour visualiser le PWM: **Windows / Simulateur/ Analyzer**.

notes



\*

### \* FONCTION: PWM amélioration

\*

La vitesse doit augmenter de 10 % à chaque appui sur le bouton S0.

\*

La vitesse doit diminuer de 10 % à chaque appui sur le bouton S1

\*

Utiliser les interruptions INT0 et INT1

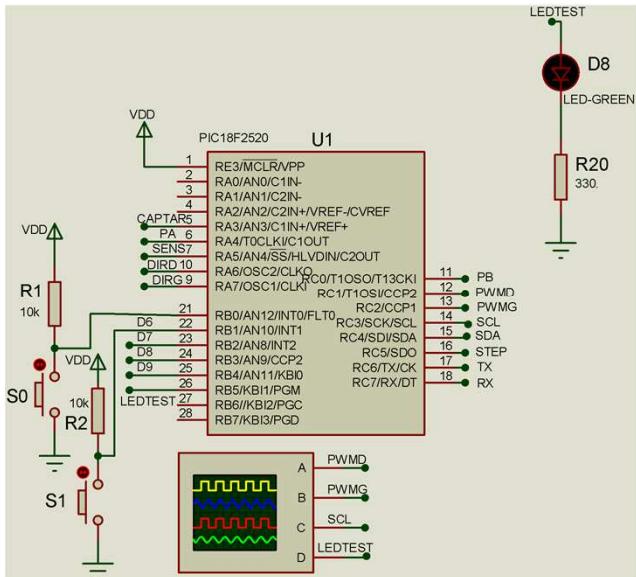
\*

Dessiner un organigramme avant de programmer

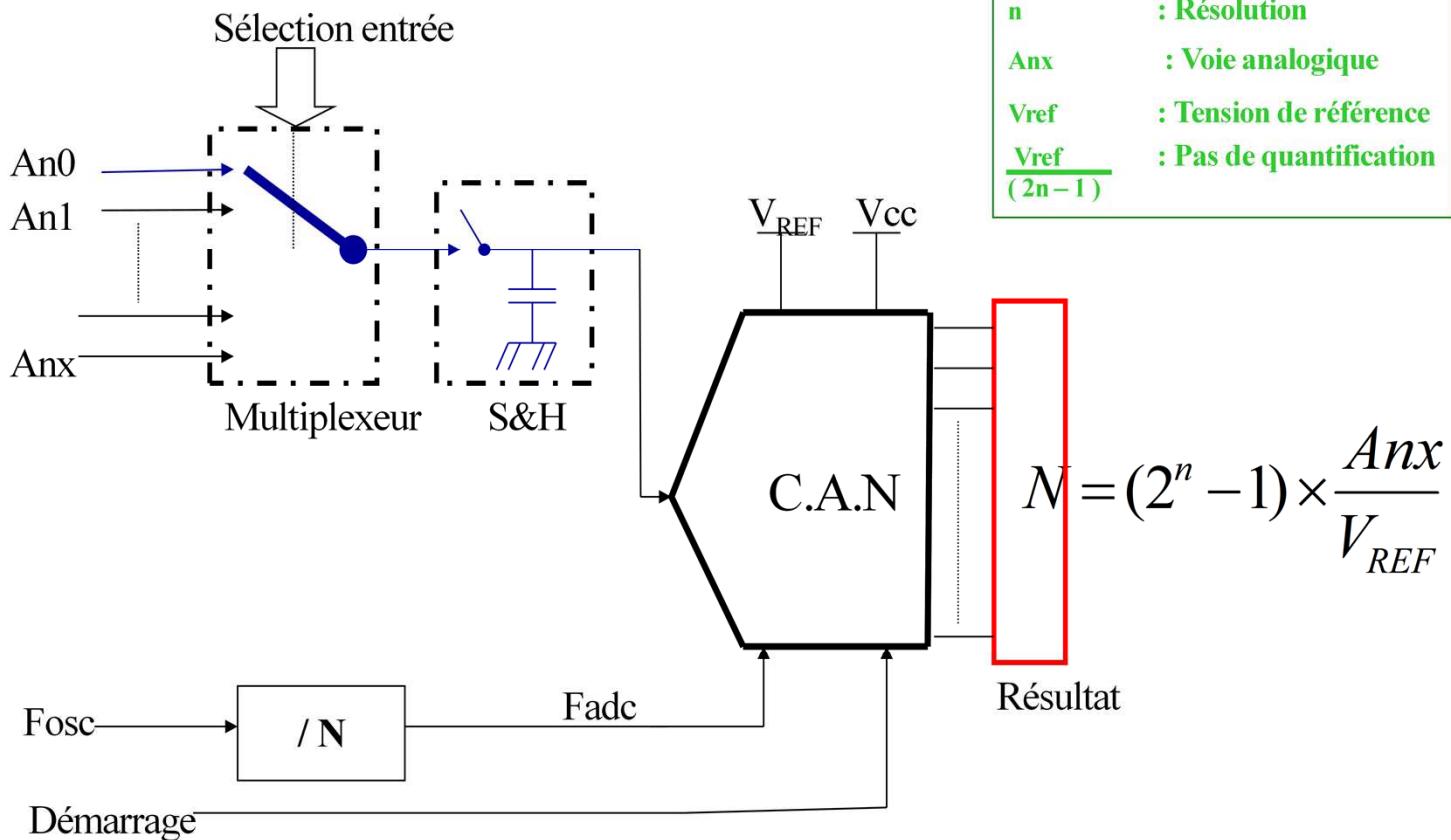
\*

Vérifier les PWM à l'oscilloscope.

\*

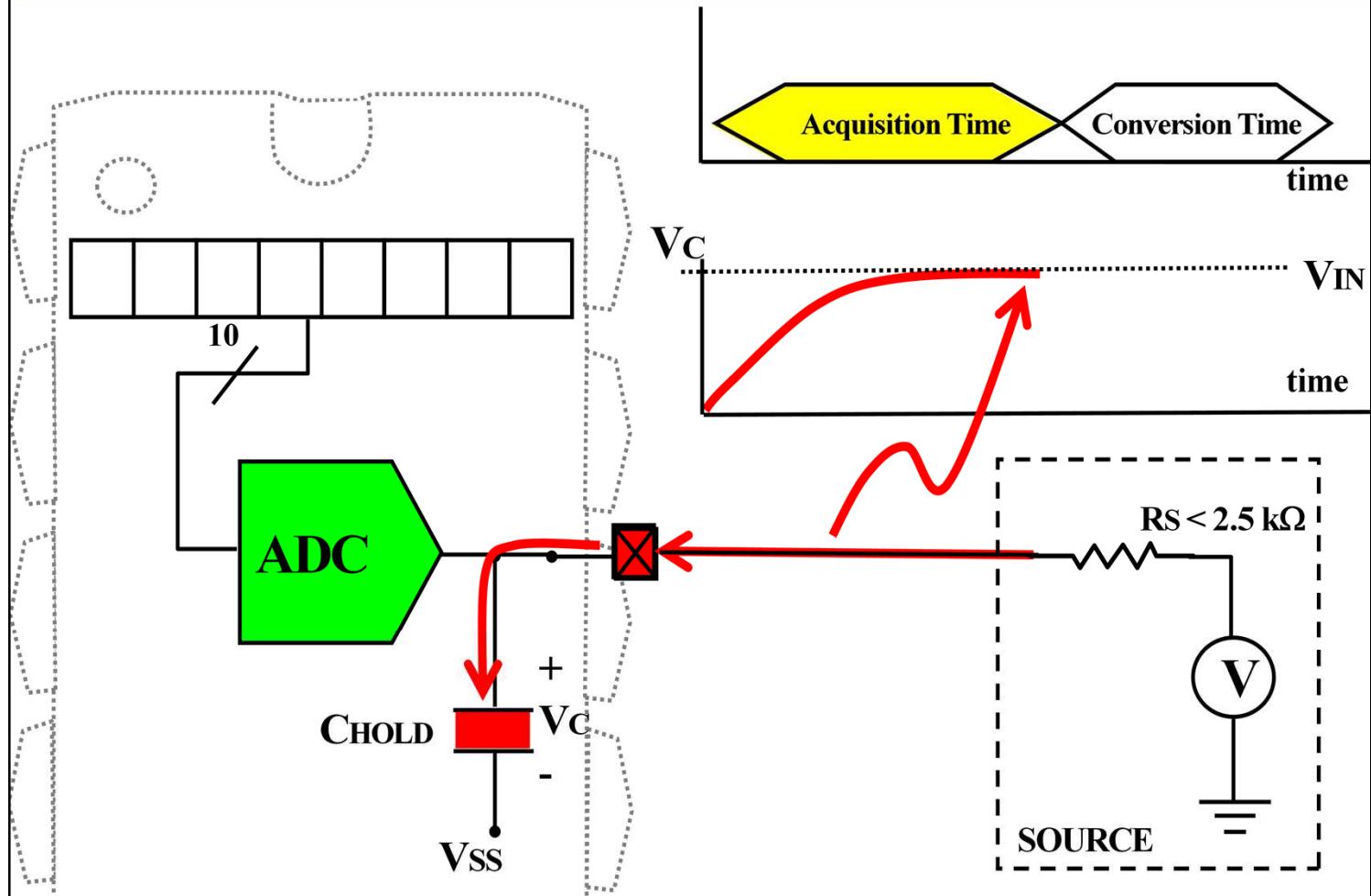


notes



Un seul adc peut être associé à plusieurs entrées analogiques via un multiplexeur. Un circuit échantillonneur bloqueur d'entrée permet de prélever et de maintenir constante la tension analogique pendant tout le temps de conversion. Le résultat est fourni dans un registre dont la résolution dépasse rarement 12 bits. Au-delà de 12 bits sur une carte standard les bits de poids faibles ont peu de signification sont noyés dans le bruit. La tension de référence des convertisseurs est généralement séparé de la tension d'alimentation ce qui permet de fixer sa valeur et de prendre des précautions plus draconniennes concernant sa stabilité, notamment en température, et sa précision, la précision du résultat numérique en dépendant directement comme le montre la formule de conversion. Un adc possède une fréquence maximale de travail qu'il faut veiller à ne pas dépasser, choix du rapport de division N. Il faut toujours démarrer la conversion soit de façon logicielle via un flag soit matériel via une broche d'entrée. Certains modes de fonctionnement comme le mode continu ne nécessitent qu'un seul démarrage alors que le mode single (ou monocoup) nécessite un démarrage pour chaque conversion.

notes

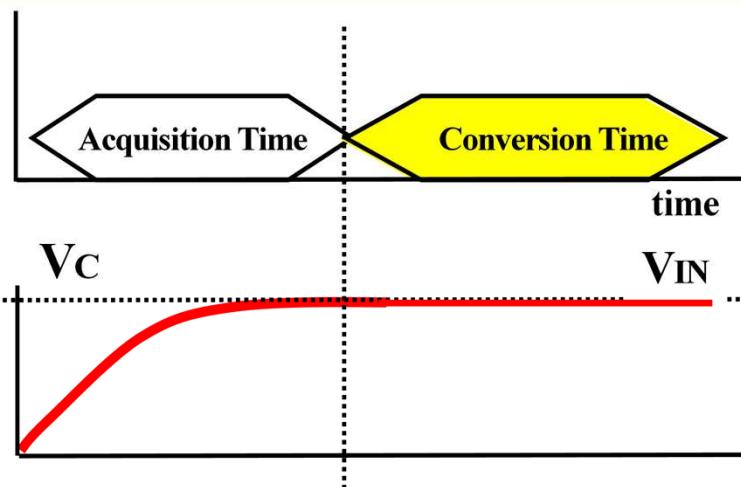
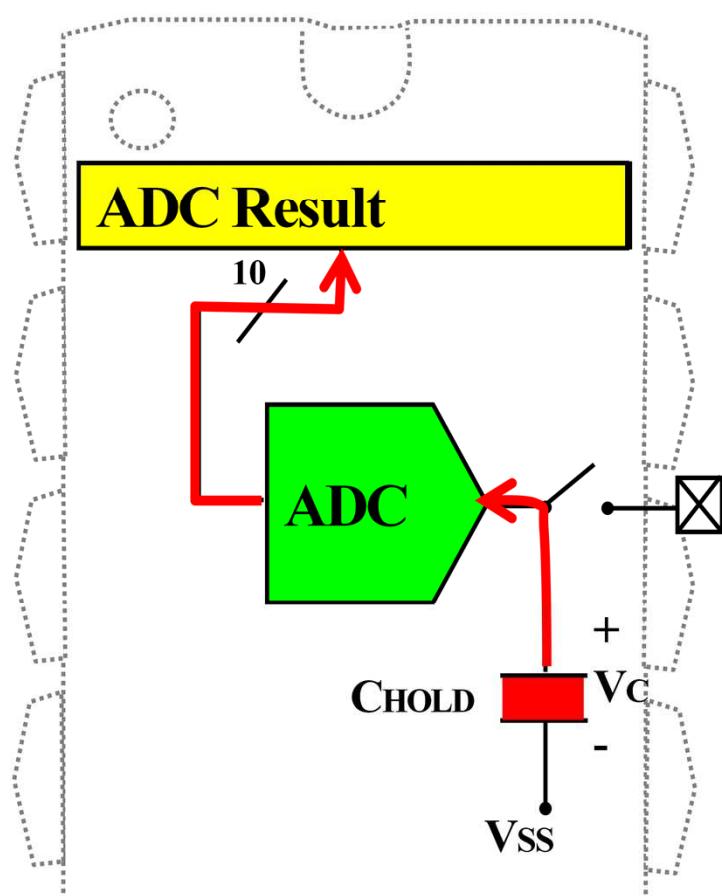


Lorsqu'on lance une conversion, la problématique est de savoir quand peut-on venir lire le résultat, il faut avoir une idée du temps total de conversion. Celui-ci se décompose en 2 parties: temps d'acquisition + temps de conversion.

Le temps d'acquisition correspond au temps de maintien de l'échantillonneur permettant la charge complète du condensateur de maintien. Ce temps d'acquisition dépend bien entendu de l'impédance de sortie de la source. Beaucoup d'imprécision dans les conversions analogiques numériques provient du fait que la conversion est lancée alors que la tension analogique ne s'est pas stabilisée. Attention à bien estimer ce temps d'acquisition qui varie en fonction de la source analogique. Sur certains microcontrôleurs ce temps d'acquisition est programmable pour s'adapter à différentes sources analogiques. C'est le cas sur le PIC18f2520.

Ce temps  $T_{ACQ}$  est à choisir en fonction de la charge, si on se place dans le pire des cas ( $R_s = 2.5 \text{ k}\Omega$ ) on obtient  $T_{ACQ\ min} = 2.4 \mu\text{s}$  (voir data sheet). Il est programmable via les flags ACQT2:0 (ADCON2).

notes



ADC ConversionClock cycles (TAD)

Remarque: le temps de conversion prend toujours le même nombre de cycles, cela ne dépend pas de la valeur à convertir (conversion par approximations successives).

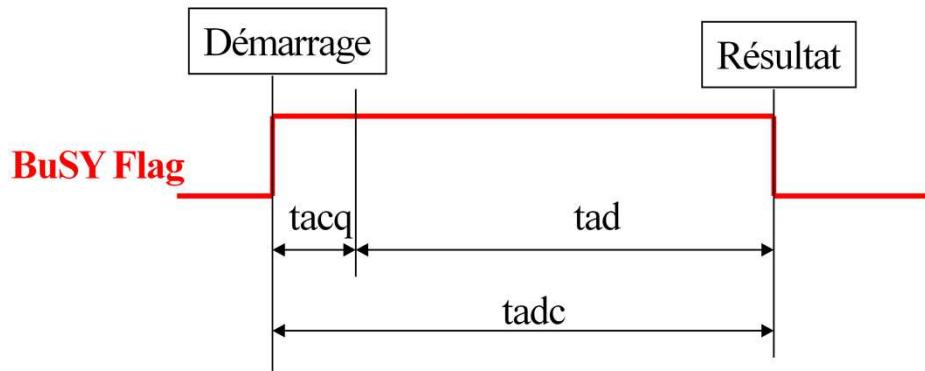
Temps de conversion =  $11 T_{AD}$  pour une résolution de 10 bits. Ce temps doit être choisi le plus faible possible tout en respectant le minimum imposé par le matériel. La data-sheet nous indique un minimum de  $0.7 \mu s$ . Il est programmable via les flags ADCS2:0 (ADCON2).

notes





- Tadc



- Problème : **Quand venir lire le résultat ?**

- 4 Méthodes

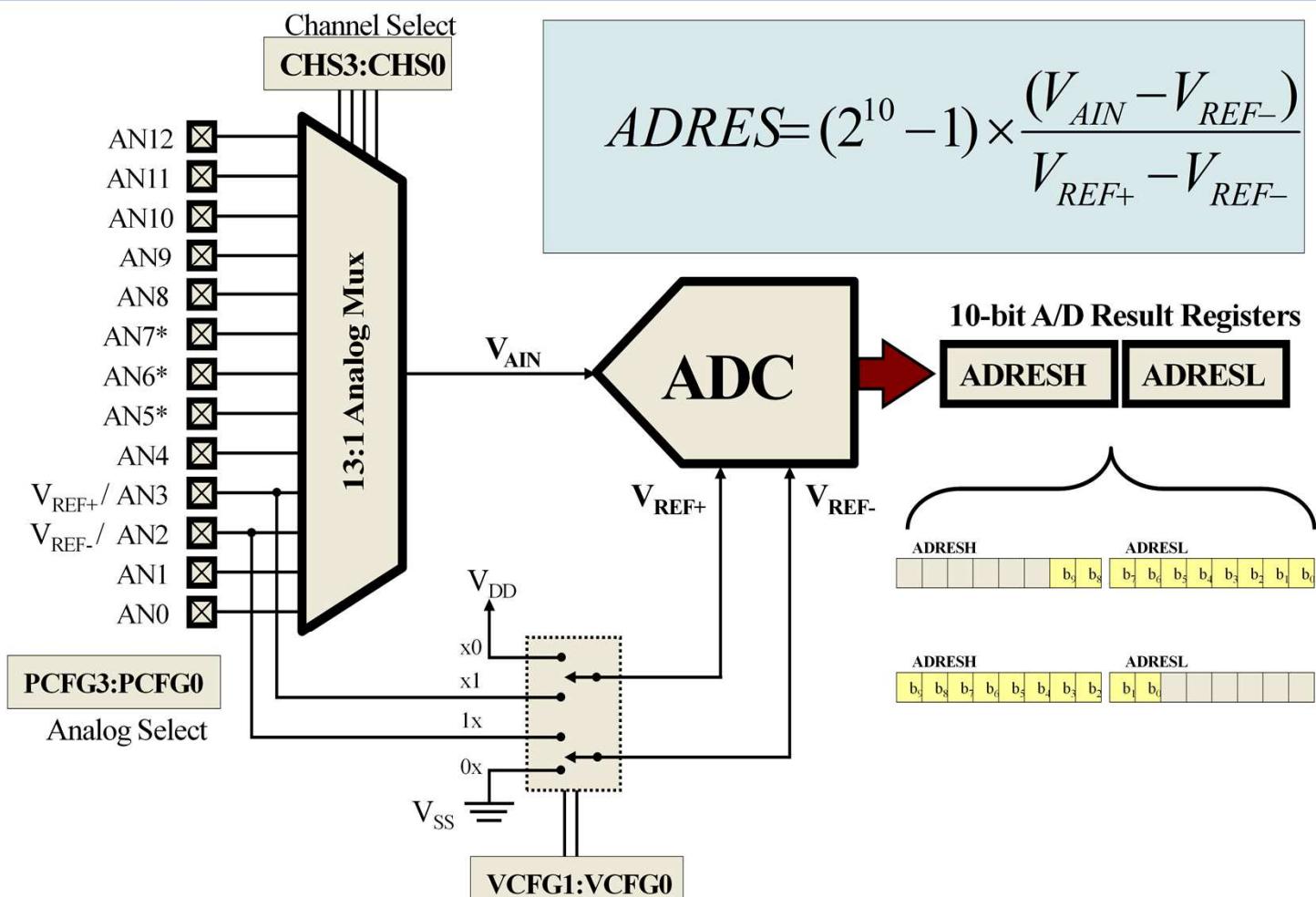
- Boucle d 'attente
- Par Scrutation du Flag BSY
- Par interruption
- Mode continu

Typiquement il existe 4 méthodes pour gérer l'adc:

- 1) Boucle d'attente dimensionnée pour correspondre au temps de conversion totale.
- 2) Scrutation du flag BSY pour savoir quand on peut venir lire le résultat. Attention ne fonctionne qu'en mode single, en mode continu le flag BSY est toujours à 1.
- 3) Validation de l'interruption interne fin de conversion de l'adc, lecture du résultat dans la routine d'interruption (le plus efficace).
- 4) En mode continu, après un démarrage, le résultat peut-être lu à la volée dans le registre résultat, au pire il sera vieux d'un temps de conversion. Inconvénient le périphérique fonctionne en permanence d'où une consommation plus importante.

notes





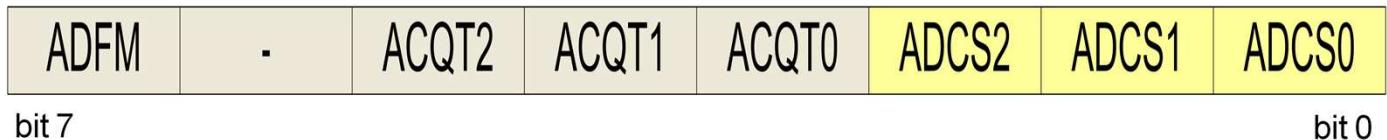
Le PIC18f2520 dispose de 10 voies analogiques réparties entre le port A et B. Dans un premier temps il faut configurer la voie en entrée analogique via les flags PCFG3:0 du registre ADCON1 puis positionner le multiplexeur d'entrée sur la voie en question via les flags CHS3:0 du registre ADCON0. Enfin ne pas oublier de fixer la tension de référence. Remarque cette tension de référence peut-être quelconque en redirigeant Vref+ et Vref- sur AN3 et AN2 via les flags VCFG1:0. Le résultat est sur 10 bits accessibles via 2 registres ADRESH:ADRESL, il peut être justifié à droite ou à gauche en fonction du flag ADFM (ADCON2).

\*AN5-AN7 non disponible sur la version 28 broches.

notes



## ADCON2 Register



bit 2-0 **ADCS2:ADCS0:** A/D Conversion Clock Selection bits

111 = F<sub>RC</sub> (Clock derived from A/D RC oscillator)

$$110 = F_{osc}/64$$

$$101 = F_{\text{osc}}/16$$

$$100 = F_{osc}/4$$

011 = F<sub>RC</sub> (Clock derived from A/D RC oscillator)

$$010 = F_{osc}/32$$

$$001 = F_{osc}/8$$

$$000 = F_{osc}/2$$

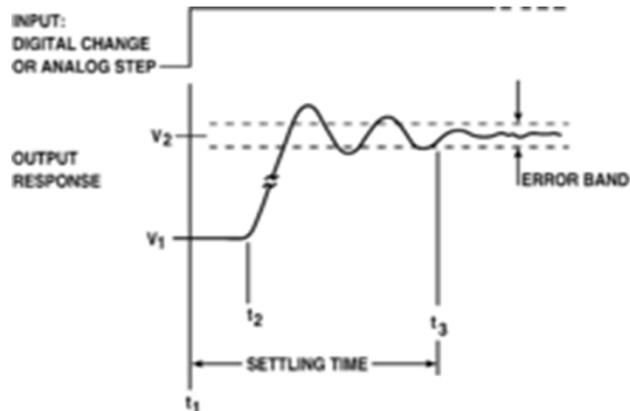
**Minimum  $T_{AD}$  = 0.7  $\mu$ s**

Il faut choisir  $T_{AD}$  le plus faible possible tout en étant supérieur au minimum autorisé ( 0,7 µs, voir data sheet)





$$T_{ACQ} = T_{AMP} + T_C + T_{COFF}$$



For R<sub>s</sub> = 2.5k Ω  
Temp = 85° C

$$T_{ACQ} = 2 + 1 + 1.2 = 4.2 \mu s$$

T<sub>AMP</sub> = Amplifier settling time = Tamp internal + Tamp external. Pour le PIC 18F2520 Tamp internal = 2 μs ( voir data sheet ). Si vous utilisez des composants pour pré-conditionner le signal il faudra ajouter le Tamp external.

T<sub>C</sub> : Charging time. Dans le pire des cas avec une impédance de source maximale de 2,5 K, on obtient: T<sub>C</sub> = 1.05 μs ( voir data sheet ).

T<sub>COFF</sub> : température coefficient = ( temp – 25°C ) . ( 0.02μS/°C ). Dans le pire des cas à 85 °C, on obtient T<sub>COFF</sub> = 1.2 μS

notes



## ADCON2 Register

ADFM	-	ACQT2	ACQT1	ACQT0	ADCS2	ADCS1	ADCS0
bit 7				bit 0			

bit 5-3 ACQT2:ACQT0: A/D Acquisition Time Selection bits

111 = 20 T<sub>AD</sub>

110 = 16 T<sub>AD</sub>

101 = 12 T<sub>AD</sub>

100 = 8 T<sub>AD</sub>

011 = 6 T<sub>AD</sub>

010 = 4 T<sub>AD</sub>

001 = 2 T<sub>AD</sub>

000 = 0 T<sub>AD</sub>

Vous avez choisi T<sub>AD</sub>, vous avez calculé T<sub>ACQ</sub>, vous en déduisez ACQT2:ACQT0.

notes





## Projet Pilotage Robot

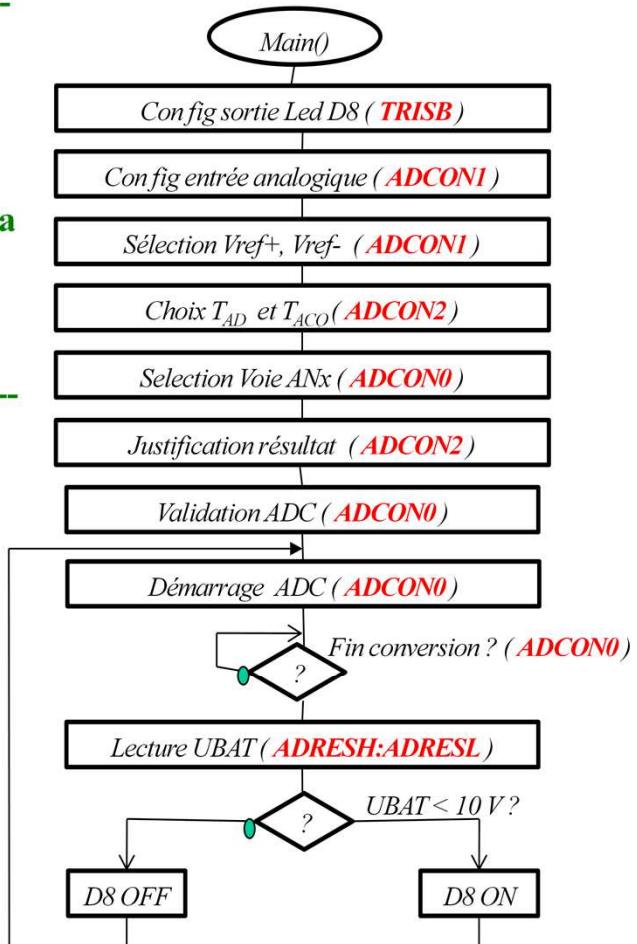
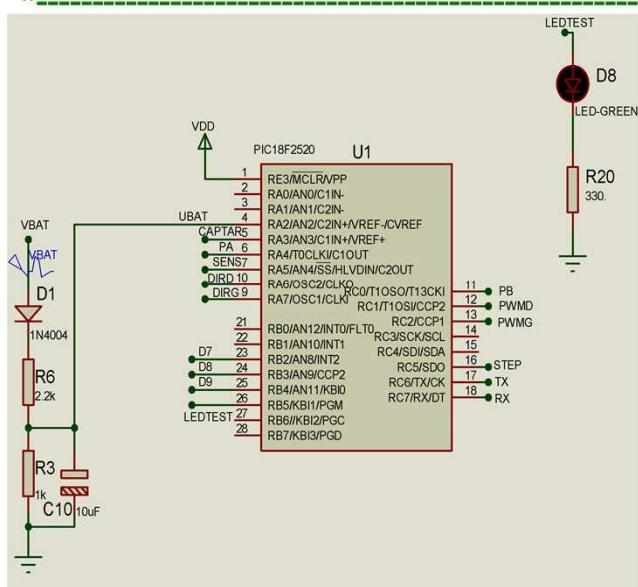
PERIPHERIQUES: horloge - ports - interruptions - timer - **Can** - uart - i2c

TP

\*

### \* FONCTION: CAN

- \* Créer un fichier adc.c dans MPLAB
- \* Dans le projet robot UBAT permet de surveiller la tension de la batterie de 12 V.
- \* Réaliser un programme allumant la led D8 lorsque la tension de batterie est inférieure à 10 V.
- \* Tester le programme en utilisant les générateurs de tension du simulateur ISIS
- \*



notes

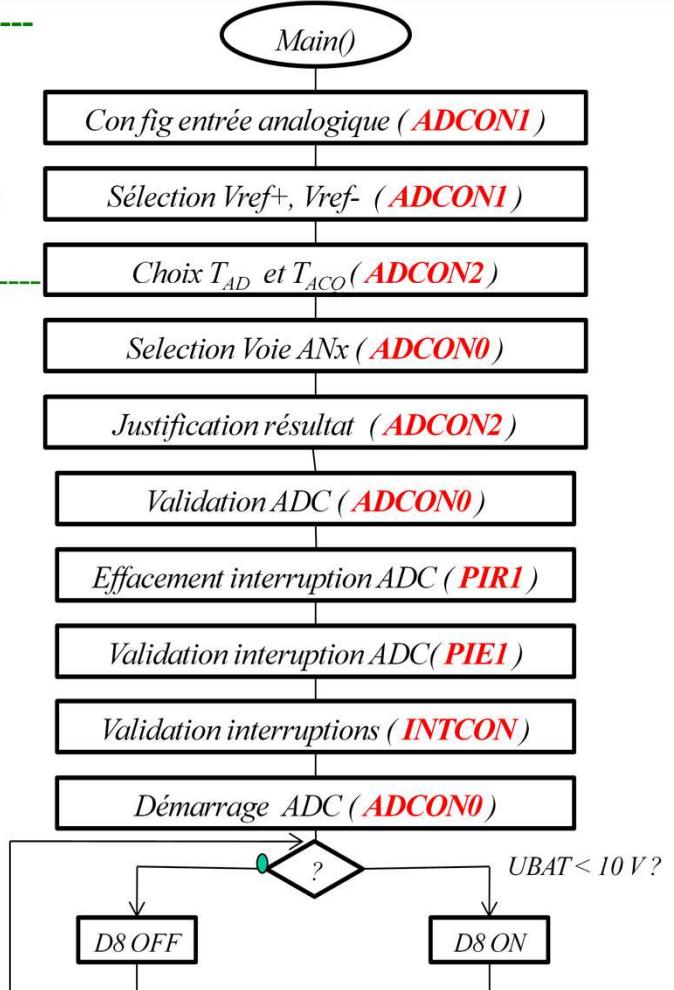
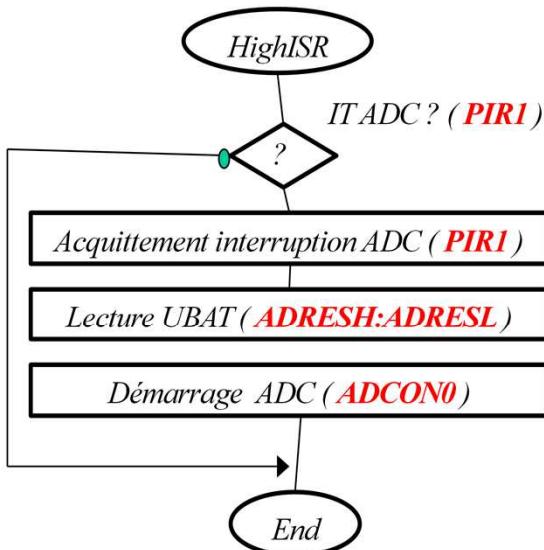


\*

## \* FONCTION: CAN

- \* Améliorer le programme précédent en utilisant les interruptions pour gérer l'ADC
- \* Tester le programme en utilisant les générateurs de tension du simulateur ISIS

\*



notes



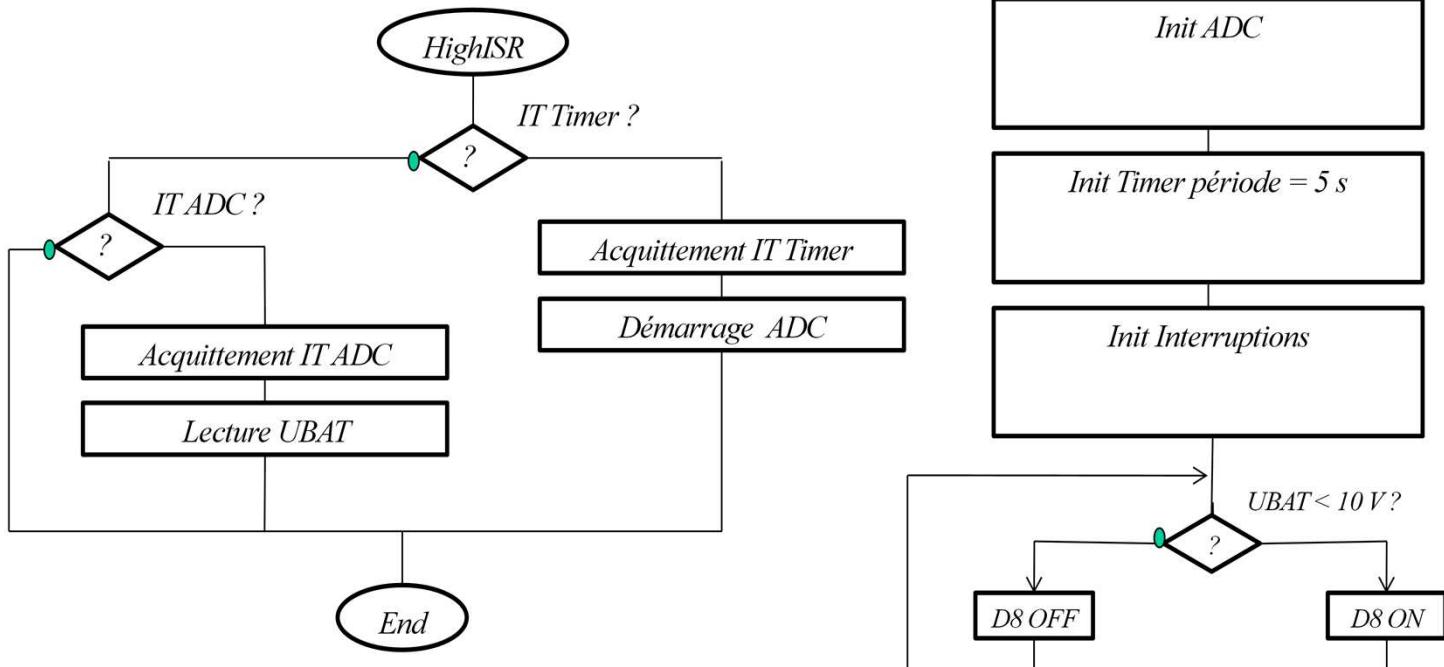


\*

**\* FONCTION: CAN**

- \* Améliorer le programme précédent pour surveiller la tension de batterie périodiquement, toutes les 5 s
- \* Tester le programme en utilisant les générateurs de tension du simulateur ISIS

\*



Cette solution à l'avantage de diminuer la consommation du périphérique ADC puisqu'il ne fonctionne plus en continu. A privilégier donc surtout dans le cadre de votre robot alimenté par une batterie.

notes

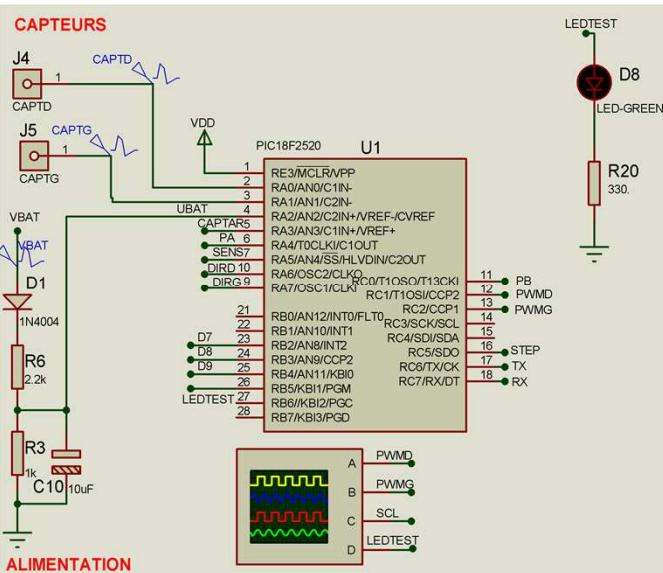




\*

### \* FONCTION: CAN

- \* Modifier le fichier pour implémenter la fonctionnalité suivante:
- \* Les signaux issus des capteurs ( CAPTD et CAPTG ) doit être transformé en PWM ( PWMD et PWMG ).
- \* Tester le résultat avec le simulateur ISIS, utilisation des générateurs de signaux et de l'oscilloscope.
- \*

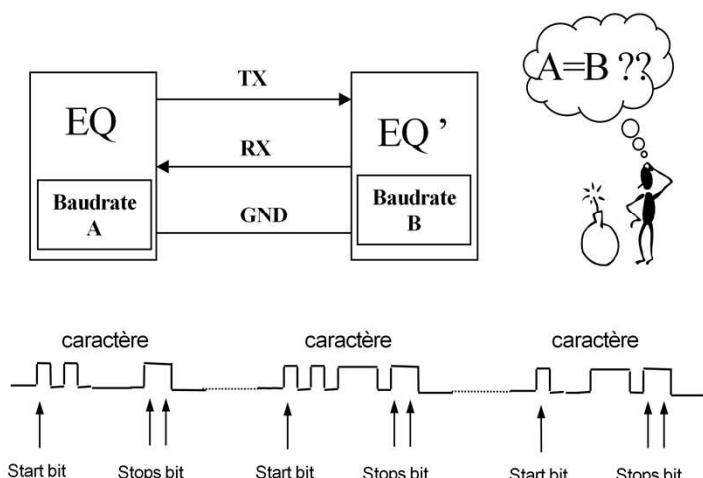


notes

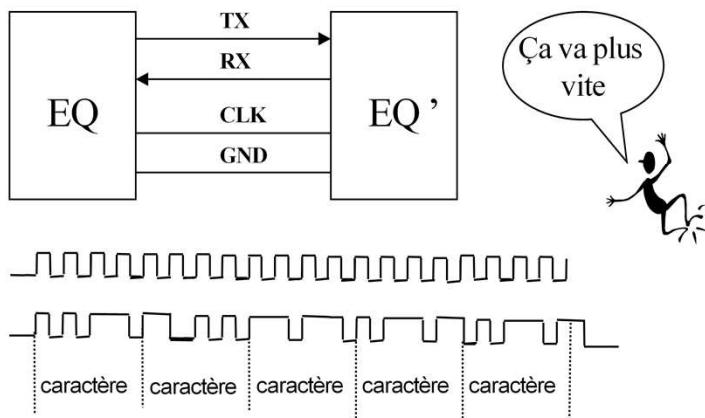




## ○ Asynchrone



## ○ Synchrone



## ○ Applications

### Equipements

- PC
- Imprimante
- Instrumentation
- ...

### Mais moins loin !

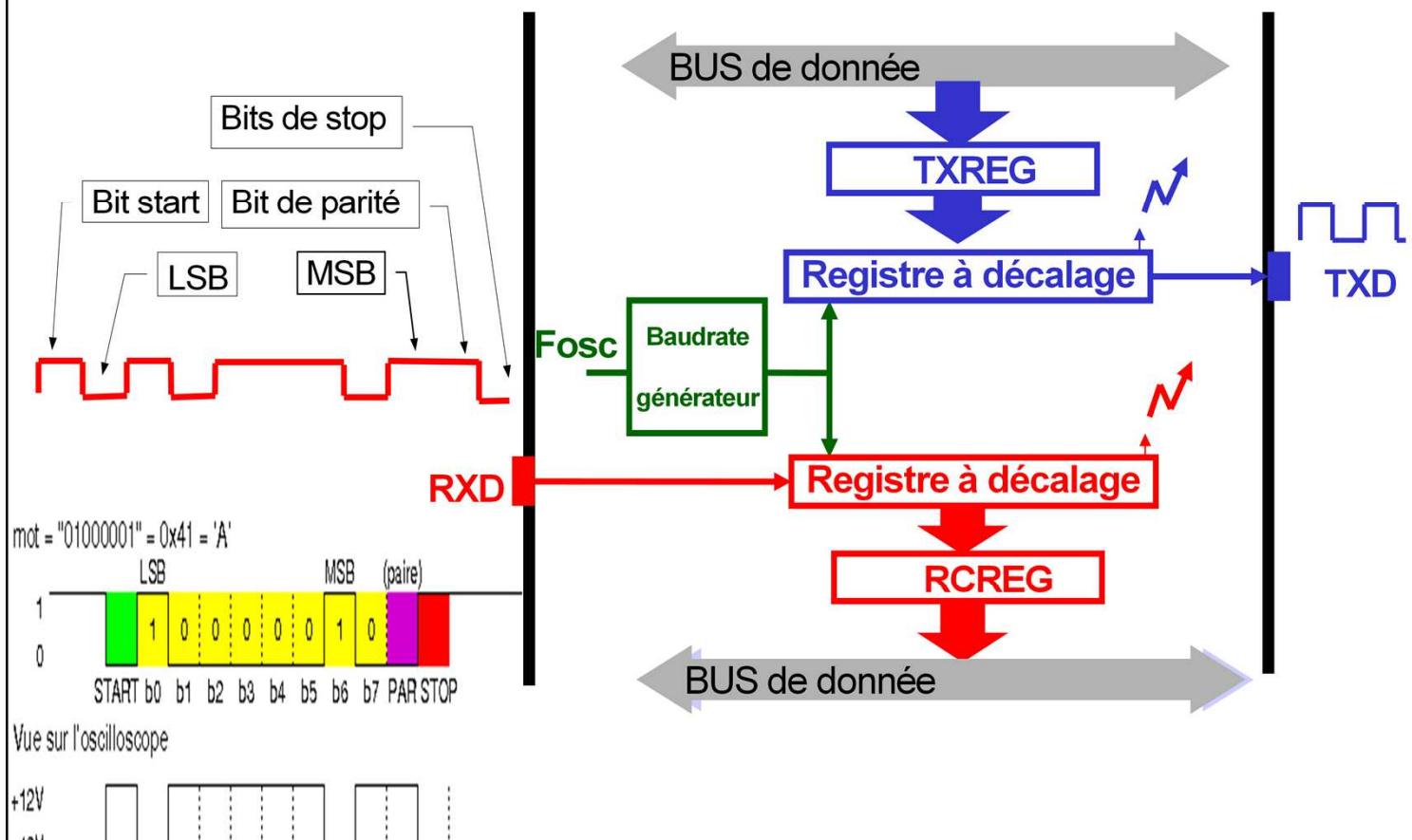
### Périphériques

- EEPROM
- Multi µC
- Convertisseur A. N.
- ...

Le périphérique gérant la liaison série est appelé UART (Universal Asynchronous Receiver Transceiver) ou USART (Universal Synchronous Asynchronous Receiver Transceiver). En mode asynchrone l'horloge n'est pas transmise entre les 2 équipements, ils seront synchronisés par leur horloge locale. Il faudra bien entendu régler les 2 équipements à la même vitesse de transmission (baudrate), même horloge. Les horloges étant sujet à des dérives il faut sans cesse les resynchroniser sur le signal lui-même via des bits de start ou de stop. En mode synchrone, l'horloge étant transmise entre les équipements, ces signaux de protocole ne sont plus nécessaires, d'où un plus grand débit utile, mais généralement sur de courtes distances. Les horloges ne peuvent pas être propagées sur de longue distance, elles sortent rarement des cartes.

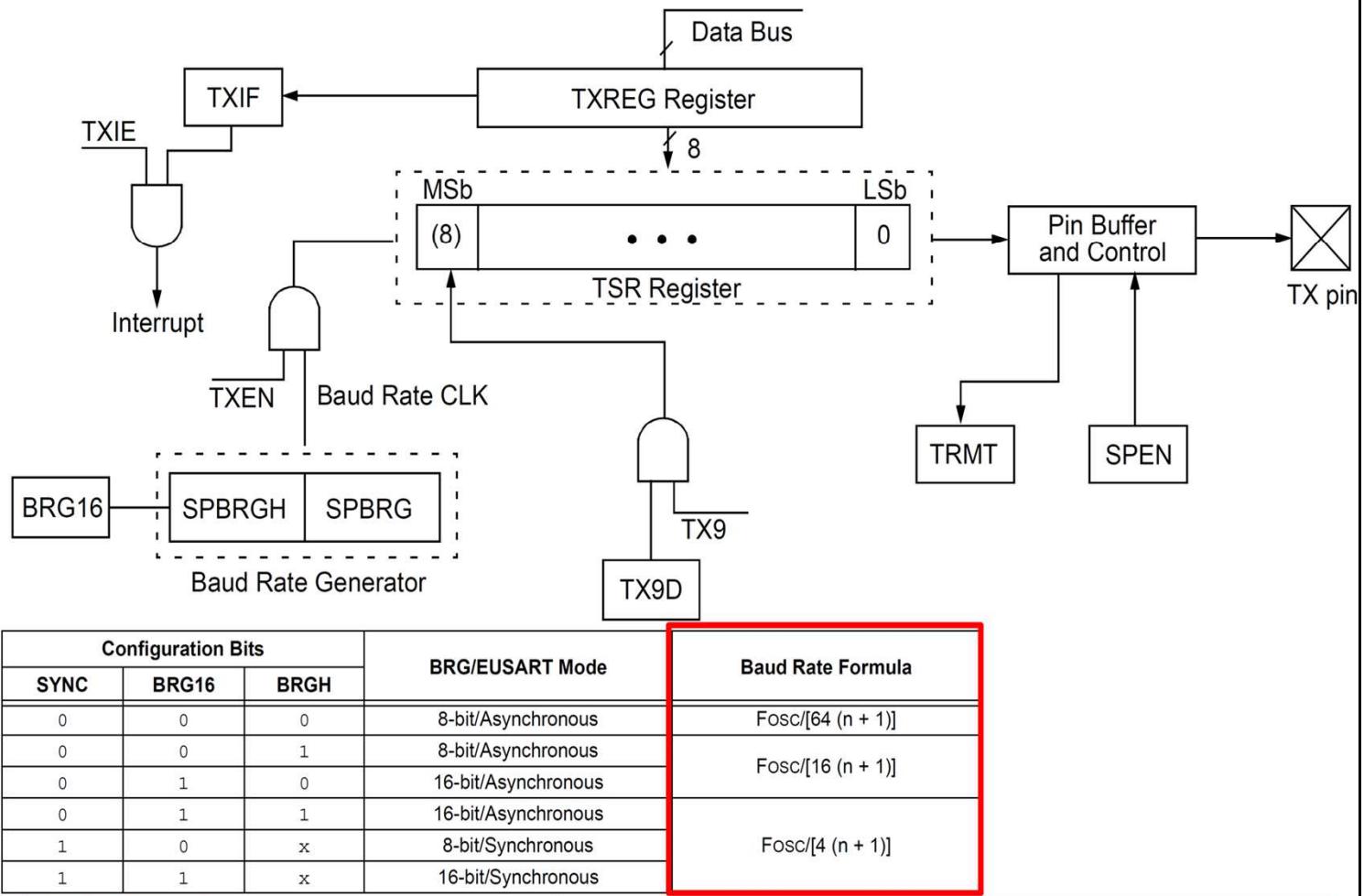
La liaison série RS232 est la plus utilisée, il s'agit de la liaison série asynchrone la plus simple, 3 fils suffisent. Elle permet des débits allant jusqu'à 115200 bauds sur 15 m. À noter qu'un circuit d'interface est nécessaire entre les sorties 5 v du microcontrôleur et les lignes RX, TX ou la norme RS232 impose des niveaux de tensions de +/- 10 V.

notes



Le protocole de la liaison série impose un bit de start, 7, 8 ou 9 bits de données, un bit optionnel de parité et un ou deux bit de stop. Le bit de parité permet une détection d'erreur simple. Un mode de parité est choisi entre les 2 équipements pair ou impair, l'UART utilise alors le bit de parité pour que le message transmis observe la parité choisie. À la réception l'UART teste cette parité pour détecter des erreurs de transmission éventuelles. Le bit de start sert à synchroniser l'horloge locale et les bits de stop laisse le temps à l'UART de déserialiser la donnée en cours avant de recevoir la prochaine. Pour lire la donnée déserialisée, il suffit de lire le registre RCREG. Une interruption de réception est possible pour avertir qu'un message est arrivé ou on peut scruter le flag RI. Dès qu'une écriture est effectuée dans TXREG, une transmission démarre avec une sérialisation de la donnée. La fin de transmission est signalée par un flag TI ou une interruption.

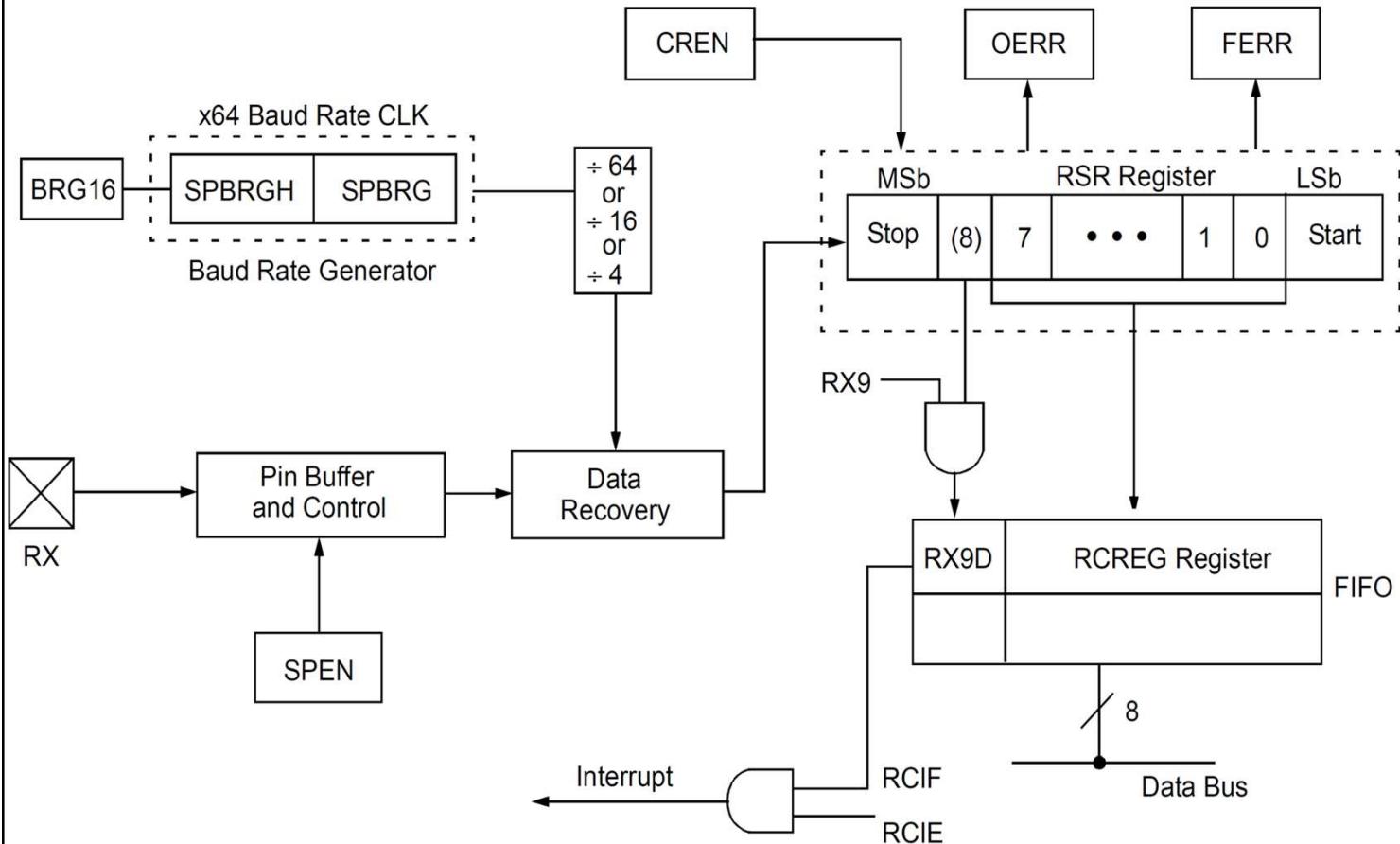
notes



Remarque: l'UART du PIC18 n'a pas de bit de parité, le 9ème bit peut être utilisé comme tel.

TXREG	: Registre de transmission
TSR	: Transmit Shift register, registre à décalage pour la transmission
TXIF	: Flag d'interruption, indique que TXREG est vide
TXIE	: autorise l'interruption
TMRT	: Indique si TSR est vide
SPEN	: Validation / inhibition patte TX pour liaison série
TX9D	: 9 éme bit si utilisé
TX9	: valide 9 éme bit
BRG16	: générateur baudrate sur 8 ou 16 bits
SPBRG, n	: valeur définissant la vitesse de transmission (baudrate) sur 8 ou 16 bits suivant BRG16
TXEN	: Validation / inhibition transmission
SYNC	: Sélectionne le mode synchrone pu asynchrone de l'uart
BRGH	: Sélectionne High ou Low speed baudrate

notes



BRG16	: générateur baudrate sur 8 ou 16 bits
SPBRG	: valeur définissant la vitesse de transmission (baudrate) sur 8 ou 16 bits suivant BRG16
SPEN	: Validation / inhibition patte RX pour liaison série
CREN	: Validation / inhibition réception
OERR, FERR	: Overrun error, Framing error
RX9	: Valide le 9 éme bit
RCIF	: Flag interruption si réception d'une donnée
RCIE	: Validation interruption réception donnée

Remarque: Le bloc réception possède une FIFO de 2 emplacements pour accélérer les réceptions.

notes



\*

## \* FONCTION: UART

\*

Créer un fichier **uart.c** dans MPLAB

\*

Afficher via la liaison série ( 9600 bauds ) le message suivant

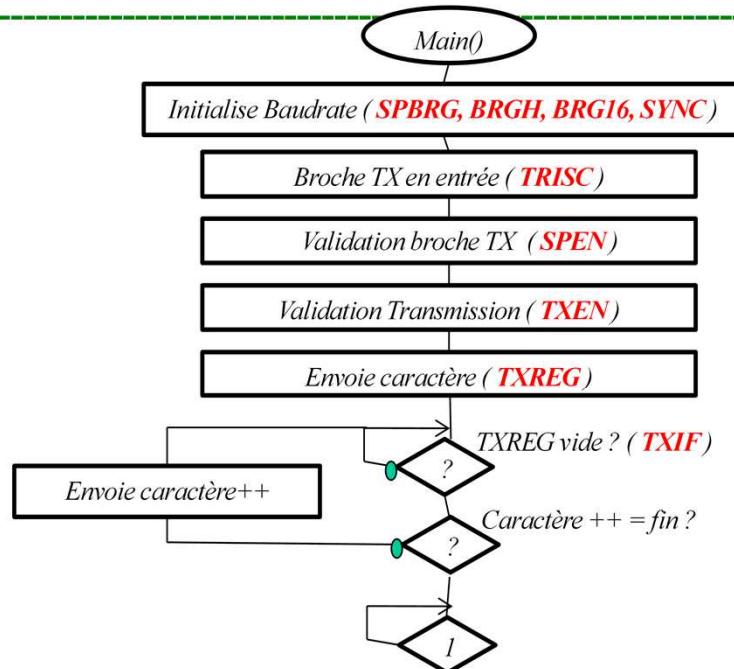
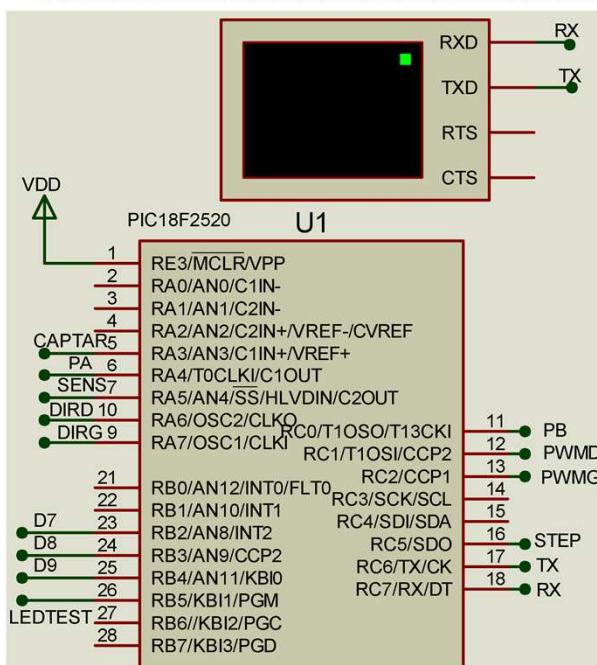
\*

« Entrer vitesse: »

\*

Tester le programme avec ISIS et son terminal série

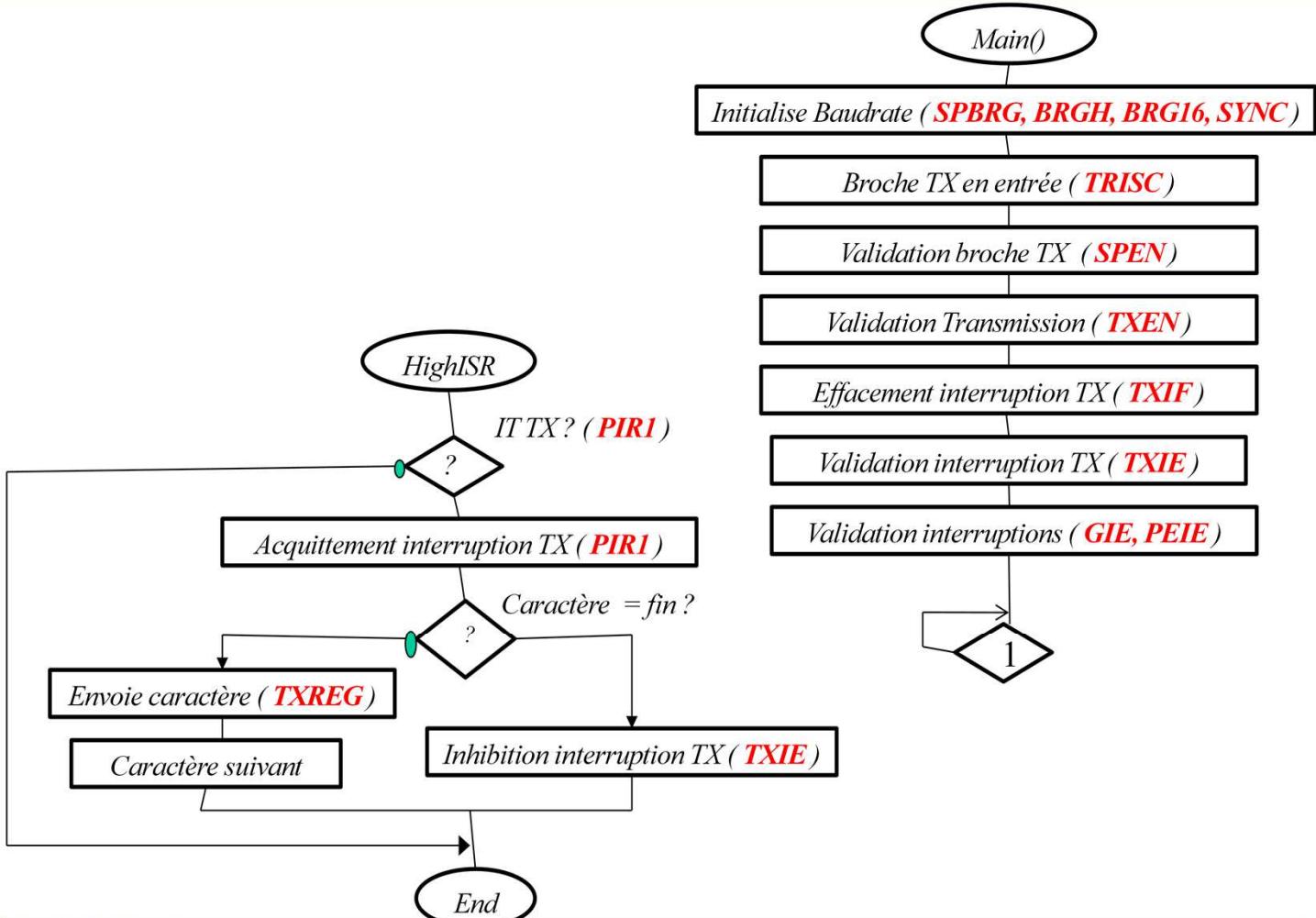
\*



Remarque 1: Les broches TX et RX doivent être configurées en entrée pour fonctionner avec l'USART. Le fait de positionner le flag PSEN, automatiquement, basculera TX en sortie.

Remarque : la fonction printf de la bibliothèque stdio.h a pour sortie par défaut l'USART. Donc un printf affichera sur le terminal série.

notes



notes





\*

### \* FONCTION: UART

\*

Améliorer le programme précédent pour faire l'acquisition d'une vitesse via le terminal série suite à l'invite

\*

Transférer cette vitesse sur PWMD et PWMG

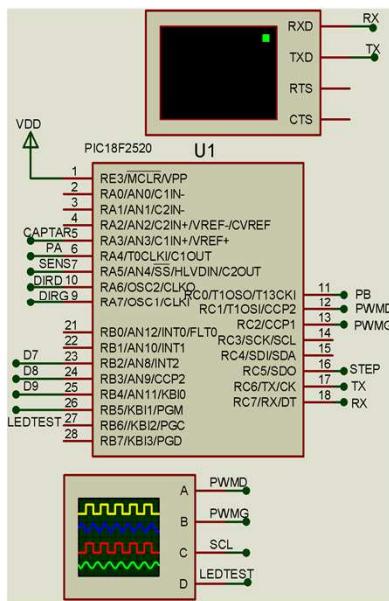
\*

Faire un organigramme avant de coder

\*

Tester le programme avec ISIS , son terminal série et son oscilloscope

\*

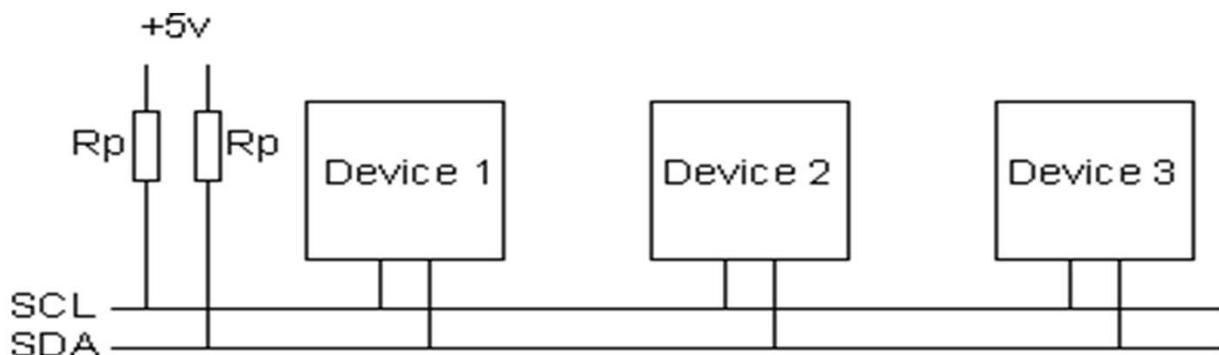


notes





- Bus série synchrone 8 bits, bidirectionnel
- 3 fils: Signal DAta, Signal Clock, Masse
- Communication inter-circuit
- 100 kb/s ( standard ), 400 kb/s ( fast ), 3,2 Mb/s ( high-speed )
- Multi-points
- Multi-maitres
- Sorties collecteurs ouverts



Le bus I<sup>2</sup>C (Inter Integrated Circuit) fait partie des bus série : 3 fils pour faire tout passer. Il a été développé au début des années 1980, par [Philips](#) pour minimiser les liaisons entre les circuits intégrés numériques de ses produits (Téléviseurs, éléments Hifi, magnétoscopes ...).

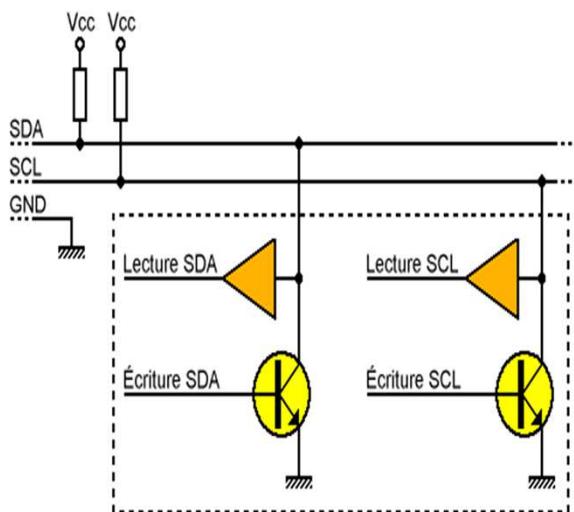
Pour se connecter à un bus I<sup>2</sup>C il faut une masse, et deux fils de communication. Le premier fil, SDA (Signal DAta), est utilisé pour transmettre les données. L'autre fil, SCL (Signal CLock) est utilisé pour transmettre un signal d'horloge synchrone (signal qui indique le rythme d'évolution de la ligne SDA)

notes

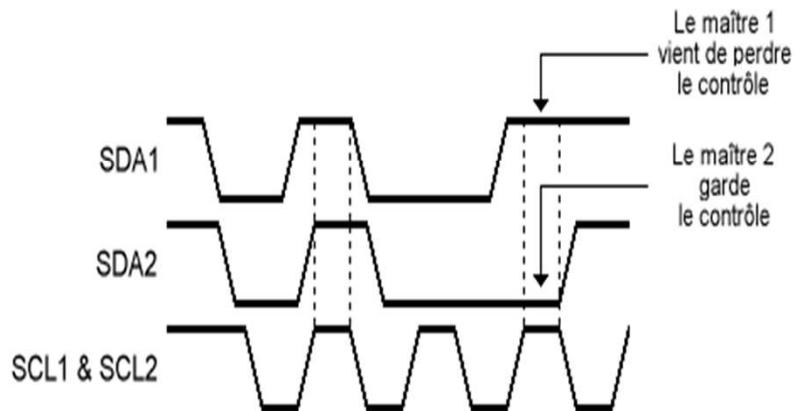
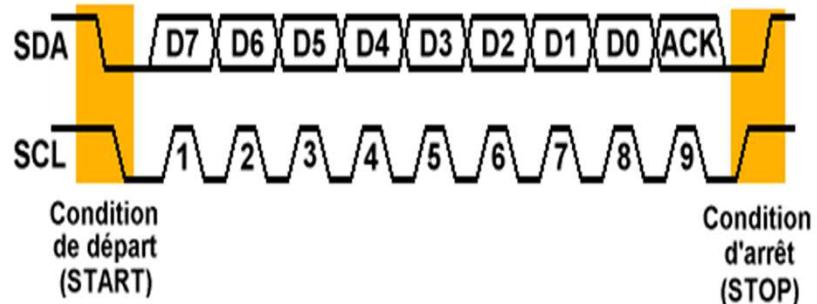




## ○ Niveau électrique



## ○ Niveau données



Il reste maintenant un problème crucial. Comment permettre à plusieurs circuits logiques de connecter leurs sorties ensemble (multipoints), sachant que certains circuits voudront imposer un niveau haut tandis que d'autres voudront imposer un niveau bas ? La réponse est connue depuis longtemps. Il faut utiliser des sorties à collecteur ouvert (ou à drain ouvert pour des circuits CMOS). Le niveau résultant sur la ligne est alors une fonction « ET » de toutes les sorties connectées. Attention ne pas oublier **les résistances de pull up** sur les lignes SDA et SCL.

Au repos, tous les circuits connectés doivent imposer un niveau haut sur leurs sorties respectives. Avant de tenter de prendre le contrôle du bus, un circuit doit vérifier que les lignes SDA et SCL sont au repos, c'est-à-dire à l'état haut. Si c'est le cas, le circuit indique qu'il prend le contrôle du bus en mettant la ligne SDA à 0. À partir de ce moment-là, les autres circuits savent que le bus est occupé et ils ne devraient pas tenter d'en prendre contrôle. Le circuit qui vient de prendre le contrôle du bus en devient le maître (en anglais « master »). C'est lui qui génère le signal d'horloge, quel que soit le sens du transfert.

Il existe la possibilité que deux maîtres prennent le contrôle du bus en même temps. Si cela ne pose pas de problème sur le plan électrique grâce à l'utilisation de collecteurs ouverts, il faut pouvoir détecter cet état de fait pour éviter la corruption des données transmises. Comment cela se passe-t-il ?

Si tous les maîtres qui ont pris le contrôle du bus placent le même état sur la ligne SDA, le conflit n'étant pas visible, la transmission se poursuit normalement, comme si chacun était seul. En revanche, dès qu'un maître place un niveau différent des autres sur la ligne SDA, il y aura forcément un état bas sur la ligne SDA, tandis qu'un ou plusieurs maîtres souhaitaient imposer un niveau haut. Tous les maîtres qui ont demandé un niveau haut, tandis que la ligne SDA reste à 0, vont perdre immédiatement le contrôle du bus.

notes

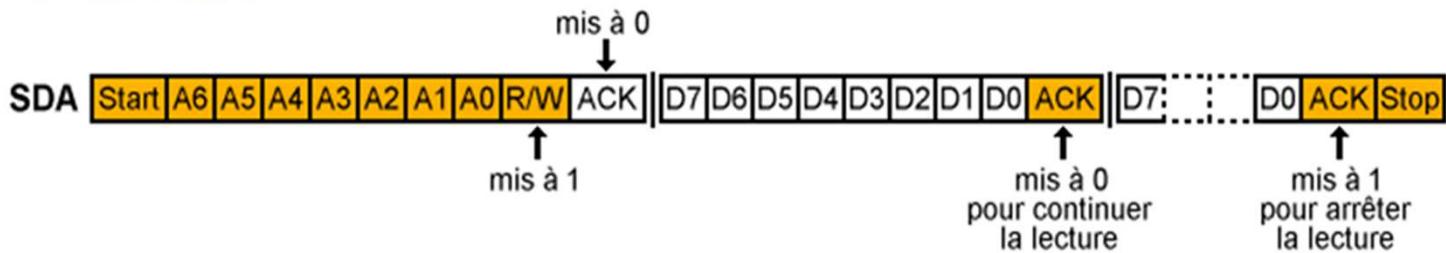


- état imposé par le maître
- état imposé par l'esclave

## ○ Ecriture



## ○ Lecture

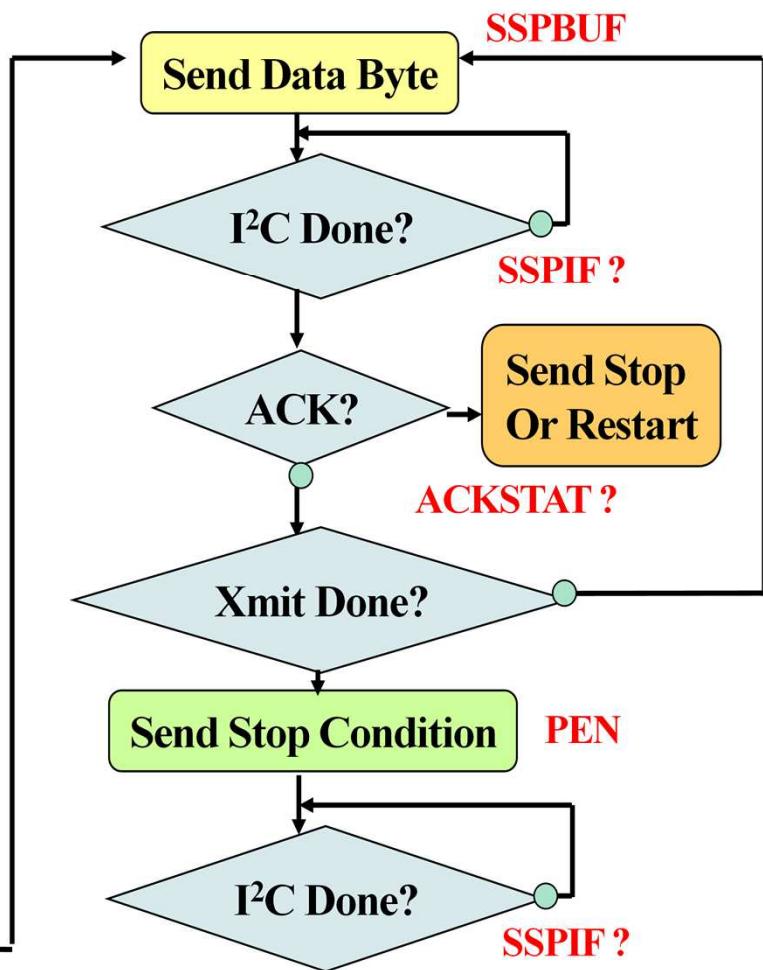
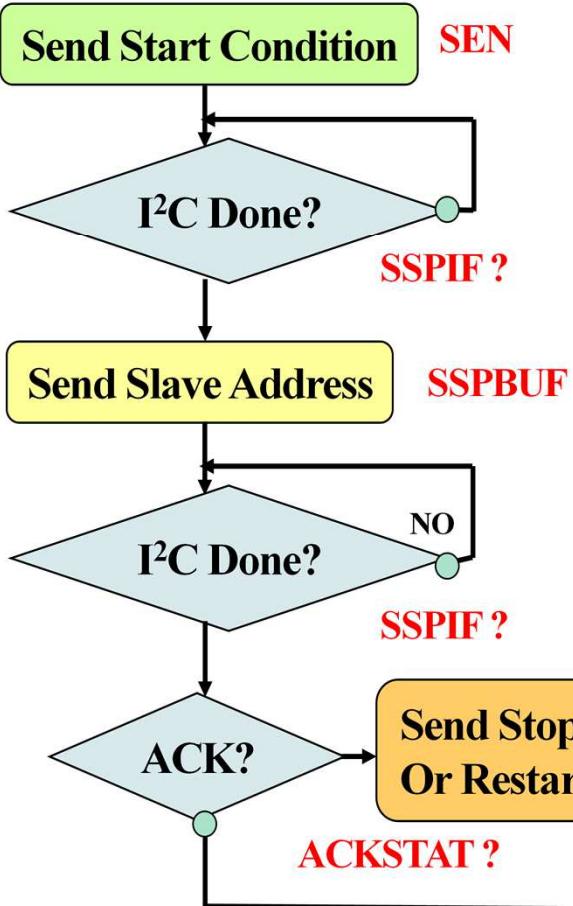


Le nombre de composants qu'il est possible de connecter sur un bus I<sup>2</sup>C étant largement supérieur à deux, le maître doit pouvoir choisir quel esclave est censé recevoir les données. Dans ce but, le premier octet que transmet le maître n'est pas une donnée, mais une adresse. Le format de l'octet d'adresse est un peu particulier puisque le bit D0 est réservé pour indiquer si le maître demande une lecture à l'esclave ou bien au contraire si le maître impose une écriture à l'esclave.

Une fois les 8 bits transmis, le circuit esclave doit imposer un bit d'acquittement ACK sur la ligne SDA. Pour cela, le maître place sa propre sortie au niveau haut, tandis que l'esclave place sa sortie au niveau bas. Puisque les sorties sont à collecteur ouvert, la ligne SDA restera au niveau bas à cause de l'esclave. Le maître relit ensuite la ligne SDA, si la valeur lue pour le bit ACK est 0, c'est que l'esclave s'est bien acquitté de l'octet reçu, sinon c'est qu'il y a une erreur et le maître doit générer la condition arrêt.

notes



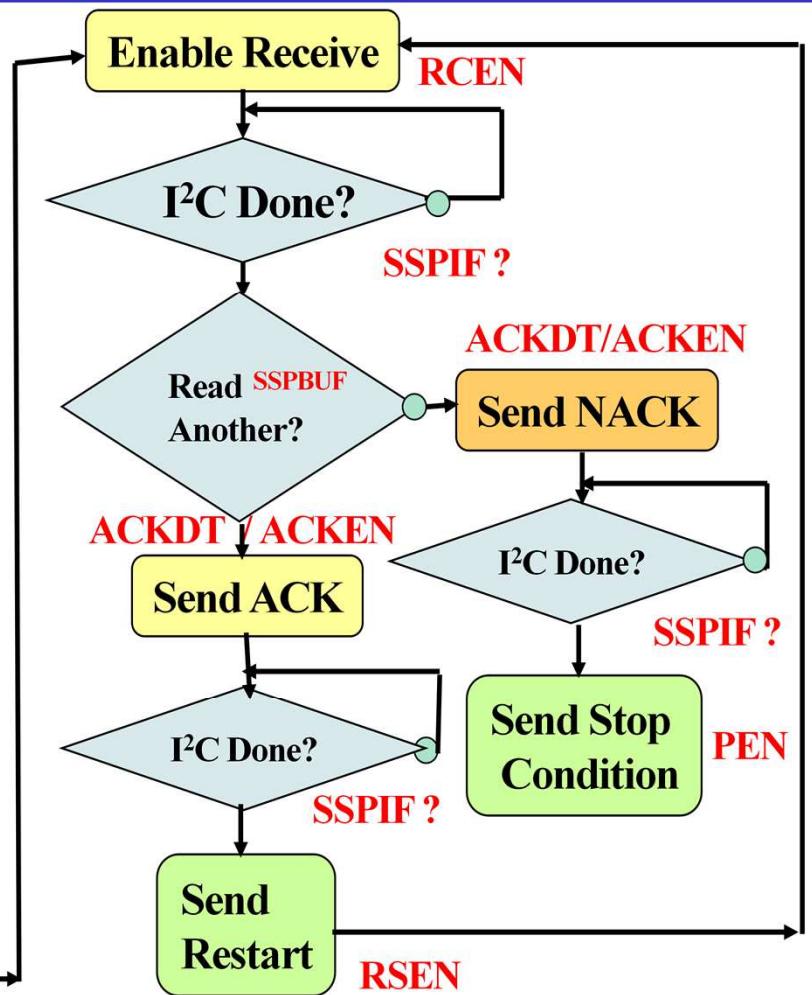
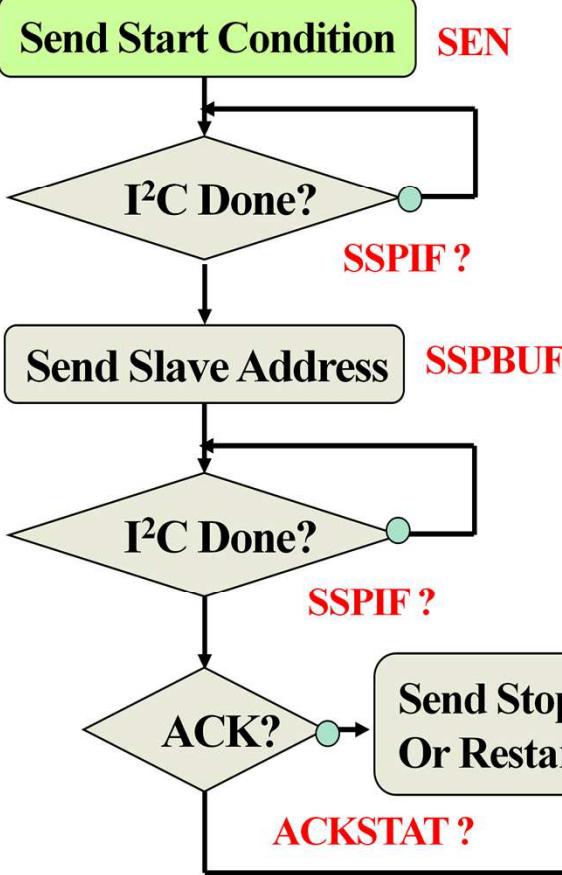


Il existe plusieurs façons de tester I<sup>2</sup>C Done. Soit on teste un flag particulier, soit on teste toujours le même flag SSPIF qui passe à 1 lorsque l'action en cours sur l'I<sup>2</sup>C se termine. Ne pas oublier de le remettre à zéro.

Attention: ACK est actif à l'état bas, ACK = 0: acquittement

notes





notes



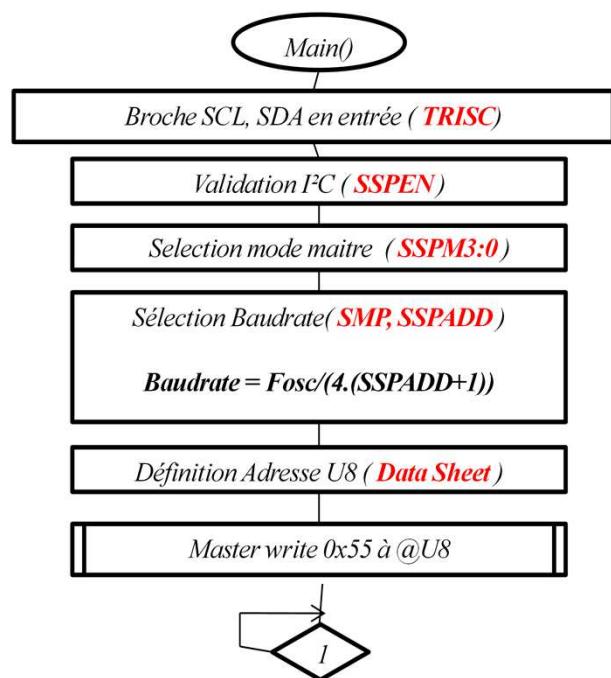
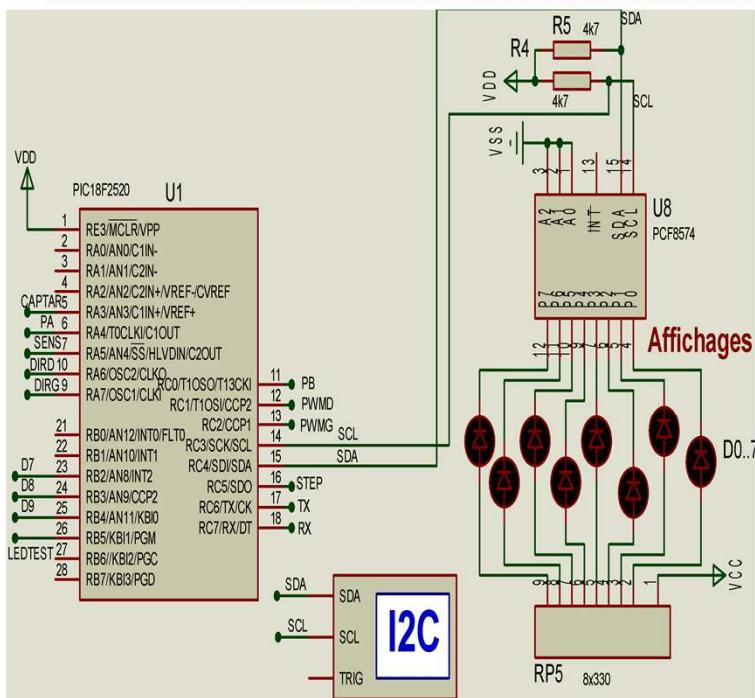


\* **\* FONCTION: I<sup>2</sup>C**

\* **Créer un fichier i2c.c dans MPLAB**

\* **Afficher 0x55 sur les leds D0..7 via la liaison I<sup>2</sup>C à 100 kb/s**

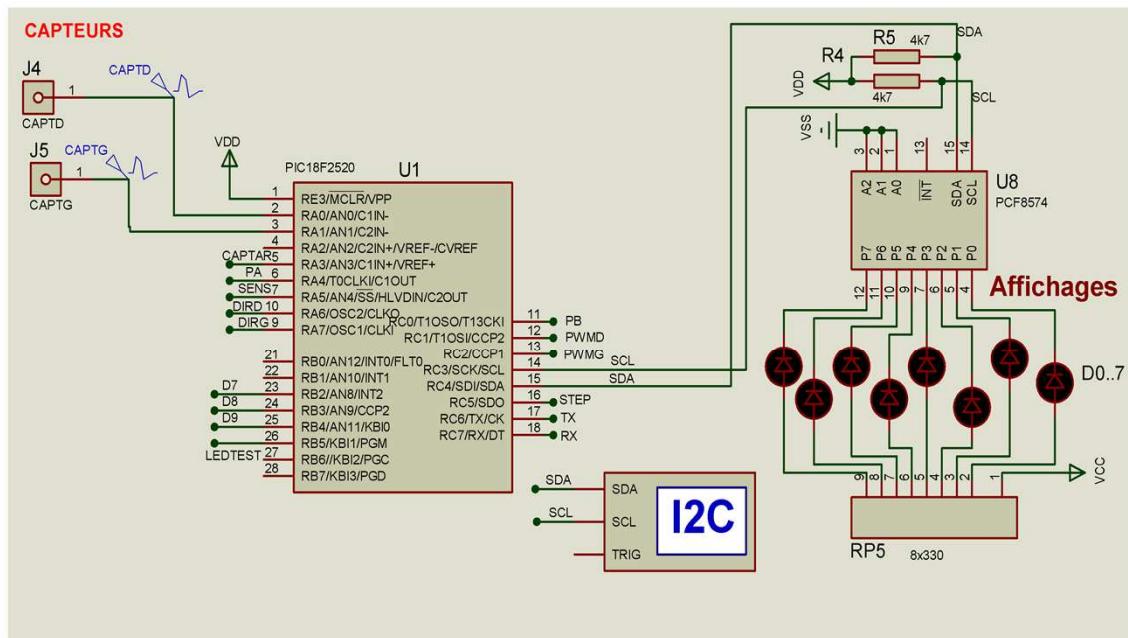
\* **Tester le programme avec ISIS**



notes



- \* **FONCTION:** le sonar est simulé de façon simplifiée par l'instrument I<sup>2</sup>C de proteus
- \* L'adresse du sonar est 0xE0. L'envoie du code 0x51 lance une mesure.
- \* Le résultat de la mesure est accessible 70 ms plus tard
- \* Ecrire le programme de lecture de la distance sonar.
- \* Faire un organigramme et tester le programme avec ISIS
- \*



Remarque: le sonar ( simulé par l'instrument I<sup>2</sup>C ) renvoie 0x00 la première fois ensuite il envoie différente valeur à chaque fois que vous relancer votre logiciel.

notes



\*

**\* AMELIORATION:**

- \* En pratique le sonar a un fonctionnement plus compliqué. Il possède 3 registres principaux: le registre de commande à l'adresse 0, le registre résultat sur 16 bits, MSB à l'adresse 2 et LSB à l'adresse 3.

\*

- \* La séquence pour faire une mesure en cm est donc la suivante:

\*

- S 0xE0 0x00 0x51 P

- Delay(70 ms )

- S 0xE0 0x02 Sr 0xE1 lecture(MSB) N P

- S 0xE0 0x03 Sr 0xE1 lecture(LSB) N P

\*

- \* Ecrire le driver, faire un organigramme au préalable.

\*

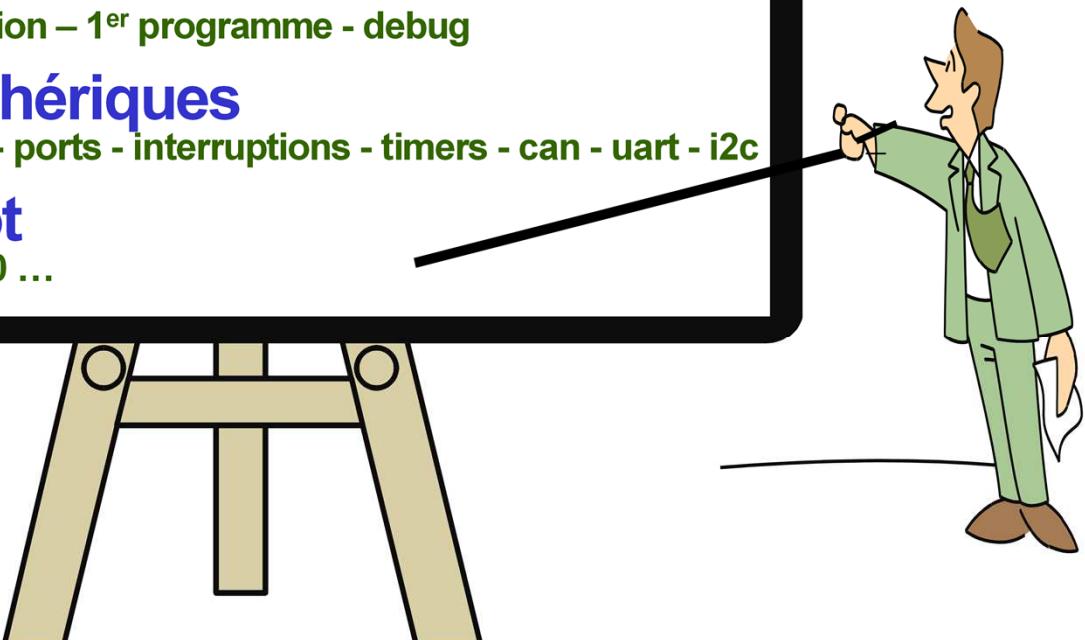
\*

notes





- P 2 **Introduction**  
sommaire - organisation - contexte
- P 10 **CPU**  
Architecture – mapping – variables
- P 21 **Outil MPLAB**  
Description – 1<sup>er</sup> programme - debug
- P 41 **Périphériques**  
horloge - ports - interruptions - timers - can - uart - i2c
- P 99 **Robot**  
Contrat 0 ...



Ce cours s'intitule Projet Pilotage Robot, même si son contenu peut vous paraître éloigné de la réalisation pratique d'un robot, il vous donne les bases informatiques nécessaires pour concevoir votre robot. Dans la mesure du possible les TP auront été pensés « robot » surtout les TP avancés, les premiers TP étant plus pédagogiques. En clair si vous faites tous les TP avancés vous aurez le logiciel correspondant au contrat 0 du projet robot.





## Projet Pilotage Robot

## ROBOT

\*  
\*  
\*  
\*  
\*  
\*  
\*

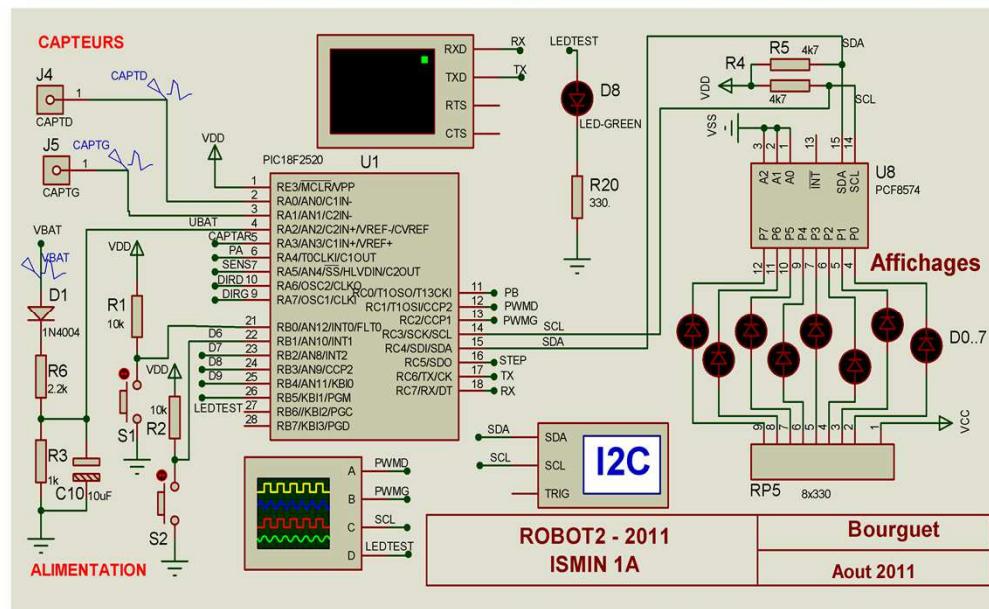
## Créer le fichier Robot.c dans MPLAB

Afficher les capteurs CAPTG et CAPTD sur D0..7, quartets poids fort

Envoyer CAPTG, CAPTD sur PWMD et PWMG

Si UBAT &lt; 10 V allumer D8

Tester le programme avec ISIS, utiliser les générateurs de signaux et l'oscilloscope



Faire un  
algorigramme  
avant de coder

notes





***FINI LA  
THEORIE EN  
AVANT  
MARCHE  
LES ROBOTS***

...

