Aix*Marseille université

Licence SPI – parcours EEA

Semestre 5

Automatisme et...

Informatique Industrielle

Micro-contrôleur et langage de haut niveau

Julien Marot
Institut Fresnel

Mél. : julien.marot@fresnel.fr

Zouhair Haddi
LSIS

Mél. : zouhair.haddi@lsis.org

Les objectifs du cours...

Vous avez étudié ...

1. Algorithmie / programmation en langage *de haut niveau* (langage C ANSI)...

2. Introduction aux micro-contrôleurs (programmation *bas niveau* en assembleur)

Ce cours s'intéresse à la programmation d'applications pour microcontrôleurs avec un langage de haut niveau (ici, le langage C).

Ce cours illustre *modestement* comment sont conçues aujourd'hui les applications embarquées sur les composants programmable (« smart »-phone, GPS, etc.).

Soyez à l'heure en cours et en TP !

6 h de cours [Julien Marot]

Rappel sur la programmation en Assembleur.
Système micro-programmé et langage C.
Exercices d'application

4 h de travaux pratiques [Zouhair Haddi]

Mise en pratique des connaissances sur la carte de démonstration PICDEM2 plus

Note : Le cours, les exercices et les TP sont basés sur la programmation en C ANSI du PIC 18F4520 de Microchip.

Votre boite à outils...

Pour ce cours, vous avez besoin de maîtriser les notions suivantes...

Pour les systèmes micro-programmée :

Architecture	(Bus, gestion mémoire, ports E/S, registres)
Jeu d'instruction	(mode d'adressage, cheminement des données)
Interruption	(utilité, principe, contrôle/gestion des sources d'interruption)
Assembleur.	

Programmation de haut-niveau :

Algorithmie	(algorigramme, analyse ascendante et descendante)
Langage C ANSI	(variable, concept de variable et de fonction...).

Supposé assimilé (pas de rappel) : conversions entre binaire et hexadécimal, tables de vérité des opérateurs AND/OR/XOR.

Contrôle des connaissances

Vous êtes évalués sur la base des TP (10 points/20)

(1) Avant toute chose, vous devez rédiger les **algorigrammes requis**,

(2) les programmes écrits doivent être **commentés**,

(3) **vérification des programmes** en simulation et sur carte d'essai,

(4) chaque étudiant sera noté individuellement ;
nous évaluerons la participation de chacun au sein d'un binôme constitué.

Et d'un Examen (10 points/20) : contrôle des connaissances avec poly de cours sur les notions vues en cours (exercices inclus) et en TP.

Plan du cours

1 Programmation d'un microcontrôleur en langage C

* Un langage évolué : intérêt et limitation.

* Éléments de programmation en C / exercices

2 Retour sur quelques points clés...

* les interruptions
* les directives pragma

3 Programmation en langage C des interruptions

* Le C18 et la programmation des interruptions

* Utilisation du TIMER0 (exercice)

Programmation d'un microcontrôleur

En langage C

De plus en plus, les programmes pour micro-contrôleur sont écrits en langage C.
Ce langage permet de développer rapidement des programmes,
des bibliothèques qui pourront être ensuite utilisées sur différentes machines.

Pourquoi un langage tel que le C ?

Universel : il n'est pas dédié à une application !

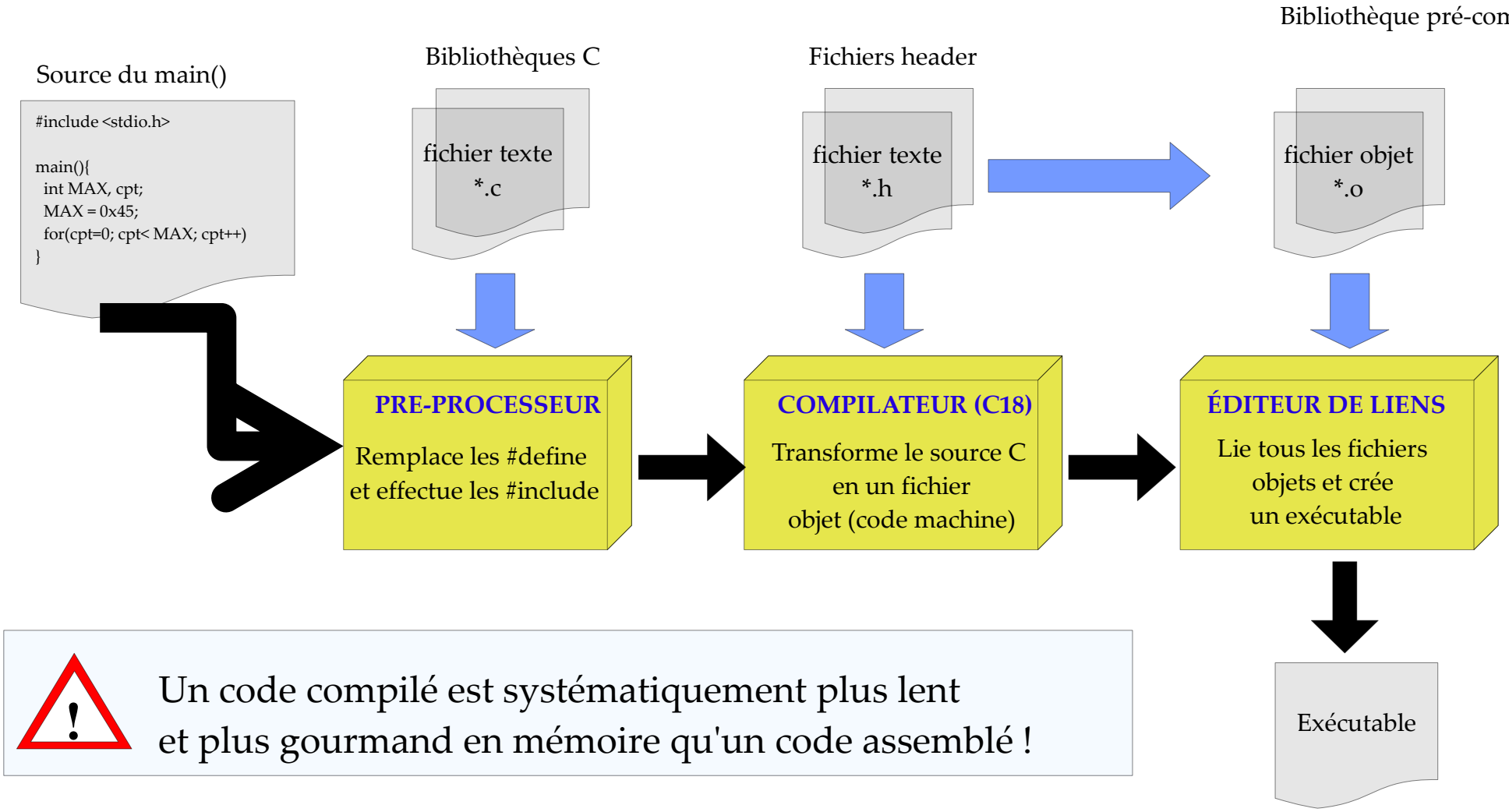
Moderne : structuré, déclaratif, récursif.
Proche de la machine : manipulations de bits, pointeurs,
possibilité d'incorporer de l'assembleur, etc.

Portable : le même code peut être utilisé sur plusieurs machines
ceci est moins vrai lorsque l'on programme pour des microcontrôleurs
Extensible : il est possible de créer des bibliothèques ou d'en incorporer.

Documentation Microchip : « *MPLAB C18 C COMPILER USER'S GUIDE (DS51288J)* »

Construction d'un exécutable (1)

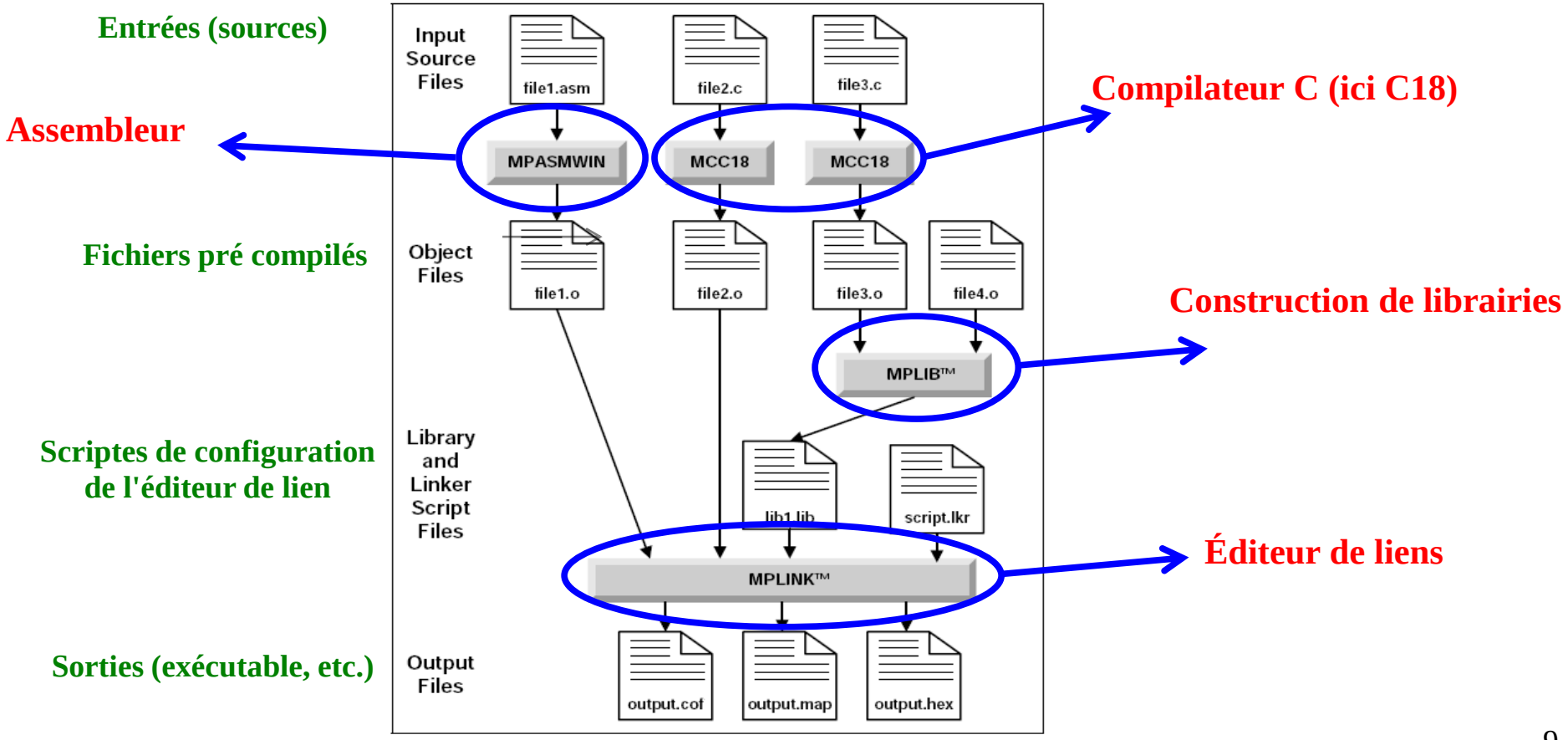
Alors que l'assembleur fait une conversion *directe* de mnémoniques en langage machine,
le compilateur C doit construire le code machine à partir d'une syntaxe de plus haut niveau.
Le recours à des *librairies pré-compilées* est permis par l'éditeur de lien qui construit un exécutable
à partir des différents *fichiers objets*.



Un code compilé est systématiquement plus lent
et plus gourmand en mémoire qu'un code assemblé !

Construction d'un exécutable (2)

Assembleur et compilateur peuvent néanmoins être utilisés pour construire un seul exécutable, *cf.* ci-dessous.
Là encore, l'éditeur de lien s'occupe de construire le code machine exécutable à partir des différents fichiers objets.



Vous avez accès à la majorité des fonctionnalités du C de la norme ANSI C.
Vous pouvez donc écrire quelque chose comme

```
vitesse = 0x27
```

pour faire *l'affectation d'une variable* codée sur 8 bits, ou encore

```
if(vitesse == limite)
```

pour effectuer un *test d'égalité* entre deux variables *entières*
codées sur un nombre de bits appropriés.

Plus précisément, vous avez accès (*cf.* cours de C++)

aux déclaration de *variables, constantes, tableaux, pointeurs, structures, etc.*

aux opérateurs *arithmétiques* et *logiques*,

aux opérateurs d'*affectation, incrémentation* et *décrémentation*,

aux opérateurs de contrôle de flux (*test* et *boucles*),

à la *conversion de type*, à l'*intégration de routines en assembleur*.

Par contre, les instructions gérants les *entrées* et *sorties*
(écran, clavier, disque, *etc.*) n'ont pas toujours de sens dans ce type de contexte...

```
//================================
// Filename: PremierProgramme.C
//================================
// Author:   marc ALLAIN
// Company:  Universite Paul Cezanne
// Revision: 1.00
// Date:   2006/07
//================================

#include <p18f4520.h>
#define duree 10000
#pragma config WDT = OFF

// Déclaration des variables globales
char c;
float pht;

// Prototypes des fonctions
void tempo(unsigned int count);

// Programme Principal
void main()
{
          PORTB = 0x00;
          TRISB = 0x00;
          while(1){
            PORTB++;
            tempo(duree);
          }
}

tempo(unsigned int count){
          while((count--)>0){
// Boucle qui tourne jusqu'à
// avoir count- égal à 1
          }
}
```

Fichier des en-têtes de fonction pré-compilées spécifiques au microcontrôleur considéré. Contient notamment les équivalences nom & adresses.

Déclaration d'un alias au préprocesseur (i.e., duree sera systématiquement remplacé par 10000).

Directive pour adresser des emplacement spécifiques de la mémoire programme ou données (cf. plus loin).

Déclaration de variables globales

Déclaration d'une fonction

Affectation de registres du microcontrôleur ; PORTB et TRISB sont déclarés dans p18f4520.h

Déclaration de types de données

Le tableau ci-dessous présente les types de variables supportés par le compilateur C18 ainsi que leur format de codage.

TABLE 2-1: INTEGER DATA TYPE SIZES AND LIMITS

Type	Size	Minimum	Maximum
`char`(1,2)	8 bits	-128	127
`signed char`	8 bits	-128	127
`unsigned char`	8 bits	0	255
`int`	16 bits	-32,768	32,767
`unsigned int`	16 bits	0	65,535
`short`	16 bits	-32,768	32,767
`unsigned short`	16 bits	0	65,535
`short long`	24 bits	-8,388,608	8,388,607
`unsigned short long`	24 bits	0	16,777,215
`long`	32 bits	-2,147,483,648	2,147,483,647
`unsigned long`	32 bits	0	4,294,967,295

Note 1: A plain *`char`* is signed by default.
2: A plain *`char`* may be unsigned by default via the `-k` command-line option.

Le format est celui du *bus de donnée* pour les types char et unsigned char.

L'utilisation de variables « plus grandes » est néanmoins permise.
Par exemple, une déclaration de la forme

```
#pragma idata test=0x0200
long l=0xAABBCCDD;
```

conduit au stockage mémoire suivant

Address	*0x0200*	*0x0201*	*0x0202*	*0x0203*
Content	0xDD	0xCC	0xBB	0xAA

Opérateurs *d'accès à la mémoire, etc.*

Opérateur	Traduction	Exemple	Résultat
`&`	Adresse de	`&x`	l'adresse mémoire de x
`*`	Indirection	`*p`	l'objet (ou la fonction) pointée par p
`[ ]`	Elément de tableau	`t[i]`	L'équivalent de `*(x+i)`, l'élément d'indice i dans le tableau t
`.`	Membre d'une structure ou d'une union	`s.x`	le membre x dans la structure ou l'union s
`->`	Membre d'une structure ou d'une union	`p->x`	le membre x dans la structure ou l'union pointée par p

Opérateur	Traduction	Exemple	Résultat
`()`	Appel de fonction	`f(x,y)`	Exécute la fonction f avec les arguments x et y
`(type)`	cast	`(long)x`	la valeur de x avec le type spécifié
`sizeof`	taille en bits	`sizeof(x)`	nombre de bits occupé par x
`? :`	Evaluation conditionnelle	`x?:y:z`	si x est différent de 0, alors y sinon z
`,`	séquencement	`x,y`	Evalue x puis y

Structures & champs de bits

Les structures sont des types composites dans lesquels des variables de types distincts peuvent cohabiter...

```
struct prof {
        char nom[30];
        char prenom[30];
        char labo[30];
        int tel;
        int HETD;
};
```

```
int main(){
        struct prof DEP_SDM[10];

        DEP_SDM[0].tel = 2878;
        strcpy(DEP_SDM[0].nom,"marot");
        strcpy(DEP_SDM[0].prenom,"julien");
        ...
}
```

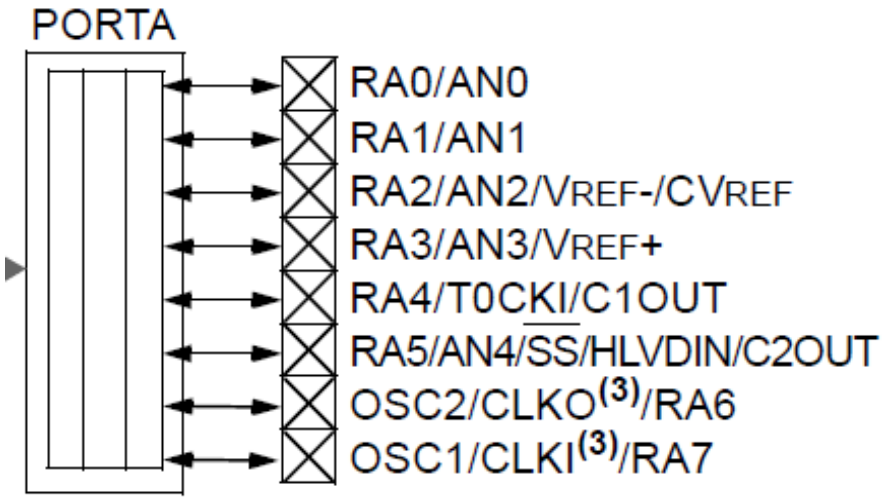
Les structures « champs de bits » permettent d'accéder explicitement à des sous-ensembles d'une variable : le premier champ correspond au bit 0 et le nom de l'élément est suivi par le nombre de bits associé.

```
struct {
   unsigned RB0:1;
   unsigned RB1:1;
   unsigned RB2:1;
   unsigned RB3:1;
 }PORTBbits;
```

Exemple : on accède à la broche RB0 par une instruction PORTBbits.RB0

Structures:

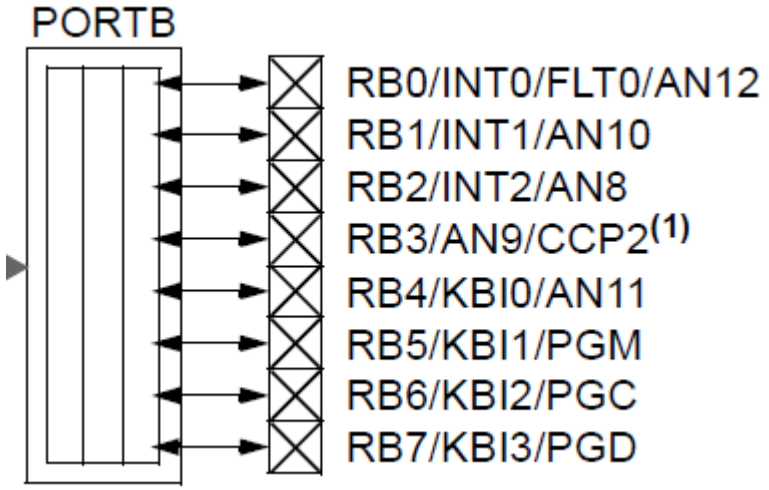
intérêt pour l'accès aux ports



1 PORT:

Une structure à 8 champs
Car il y a 8 broches.

1 champ – 1 broche



Unions

Dans une union, les champs partagent la même adresse.
Cette construction permet d'utiliser des dénominations différentes pour adresser les même bits.

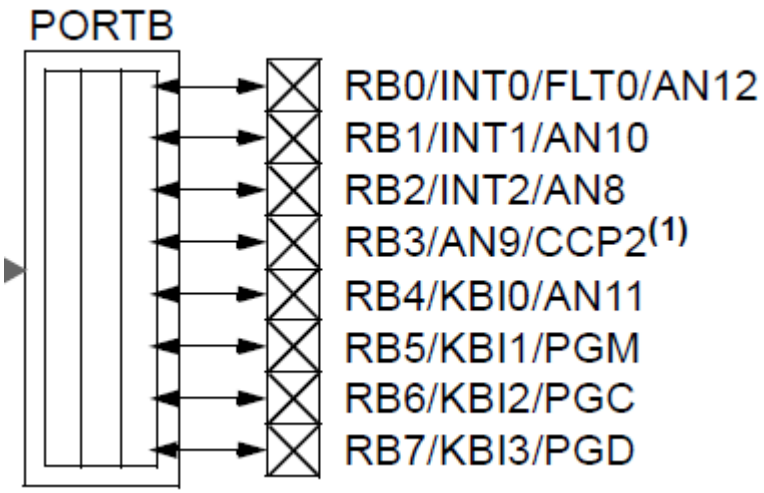
```
extern volatile near unsigned char       PORTB;
extern volatile near union {
  struct {
    unsigned RB0:1;
    unsigned RB1:1;

...
  };
  struct {
    unsigned INT0:1;
    unsigned INT1:1;

...
  };
  struct {
    unsigned AN12:1;
    unsigned AN10:1;

...
  };
} PORTBbits;
```

Exemple : PORTBbits.RB0 et PORTBbits.INT0 partagent la même adresse.

Intérêt pour la paramétrage des PORTS

Extrait du fichier header p18f4520.h

```
extern volatile near unsigned char       PORTB;
extern volatile near union {
  struct {
    unsigned RB0:1;
    unsigned RB1:1;
    unsigned RB2:1;
    unsigned RB3:1;
    unsigned RB4:1;
    unsigned RB5:1;
    unsigned RB6:1;
    unsigned RB7:1;
  };
  struct {
    unsigned INT0:1;
    unsigned INT1:1;
    unsigned INT2:1;
    unsigned CCP2:1;
    unsigned KBI0:1;
    unsigned KBI1:1;
    unsigned KBI2:1;
    unsigned KBI3:1;
  };
  struct {
    unsigned AN12:1;
    unsigned AN10:1;
    unsigned AN8:1;
    unsigned AN9:1;
    unsigned AN11:1;
    unsigned PGM:1;
    unsigned PGC:1;
    unsigned PGD:1;
```



1 broche:

plusieurs appellations

plusieurs dénominations
pour la même adresse
dans l'espace mémoire

La conversion de type (cast)

Une conversion de type intervient lorsqu'un opérateur doit agir sur des opérandes de types différents. En général, les opérandes sont alors converties selon la règle suivante :

> ***L'opérande la plus petite est convertie dans le type de l'opérande la plus grande.***
>
> char < int < long < float < double

Il est aussi possible de forcer une conversion de type :

```
float ecart, distance      = 11;
int tronque, nbp           = 5;

ecart                      = distance / (float) (nbp – 1);
tronque                    = (int) ecart;
```

ecart=2.75 → ecart

ecart → tronque

ecart=2

Les opérateurs du langage C

Opérateurs *arithmétiques* et *relationnels* :

Opérateur	Traduction	Exemple	Résultat
+	Addition	`x + y`	l'addition de x et y
-	Soustraction	`x - y`	la soustraction de x et y
*	Produit	`x * y`	la multiplication de x et y
/	Division	`x / y`	le quotient de x et y
%	Reste	`x % y`	Reste de la division euclidienne de x par y
+(unaire)	Signe positif	`+x`	la valeur de x
-(unaire)	Signe négatif	`-x`	la négation arithmétique de x
++(unaire)	Incrément	`x++` ou `++x`	x est incrementé ($x = x + 1$). L'opérateur préfixe `++x` (resp. suffixe `x++`) incrémente x avant (resp. après) de l'évaluer
--(unaire)	Decrément	`x--` ou `--x`	x est décrementé ($x = x - 1$). L'opérateur préfixe `--x` (resp. suffixe `x--`) décrémente x avant (resp. après) de l'évaluer

Opérateur	Traduction	Exemple	Résultat
<	inférieur	`x < y`	1 si x est inférieur à y
<=	inférieur ou égal	`x <= y`	1 si x est inférieur ou égal à y
>	supérieur	`x > y`	1 si x est supérieur à y
>=	supérieur ou égal	`x >= y`	1 si x est supérieur ou égal à y
==	égalité	`x == y`	1 si x est égal à y
!=	non inégalité	`x != y`	1 si x est différent de y

Opérateurs *logiques* et *de manipulation de bits* :

Opérateur	Traduction	Exemple	Résultat (pour chaque position de bit)
&	ET bit à bit	`x & y`	1 si les bits de x et y valent 1
|	OU bit à bit	`x | y`	1 si le bit de x et/ou de y vaut 1
^	XOR bit à bit	`x ^ y`	1 si le bit de x ou de y vaut 1
~	NON bit à bit	`~x`	1 si le bit de x est 0
<<	décalage à gauche	`x << y`	décale chaque bit de x de y positions vers la gauche
>>	sécalage à droite	`x >> y`	décale chaque bit de x de y positions vers la droite

Opérateur	Traduction	Exemple	Résultat
&&	ET logique	`x && y`	1 si x et y sont différents de 0
||	OU logique	`x || y`	1 si x et/ou y sont différents de 0
!	NON logique	`!x`	1 si x est égal à 0. Dans tous les autres cas, 0 est renvoyé.

Variables globales, locales...

La programmation en langage C se base entièrement sur des fonctions dont la principale est main().
La déclaration de variable peut être faite à l'*intérieur* ou à l'*extérieur* d'une fonction avec des effets distincts (notion de *classes des stockage*).

A l'extérieur d'une fonction, une déclaration est **globale** :
la variable est visible pour toutes les fonctions et l'adresse allouée est fixe.

ex.
```
#include <pic18f4520.h>
int MAX;
main(){...}
```

A l'intérieur d'une fonction, la variable n'est pas visible de l'extérieur et les mots clés suivant détermine la manière dont la variable est stockée :

`static` : *variable stockée en RAM à une adresse fixe.*

ex.
```
void function()
{
        static int MAX;...
}
```

De l'assembleur dans du C ?

C18 permet l'utilisation d'instructions en assembleur presque comme si vous utilisiez MPASM.

Ces instructions doivent être dans un « bloc » délimité par _asm et _endasm.

```
_asm                                          /* User assembly code */
        MOVLW 10                        // Move decimal 10 to count
        MOVWF count, 0
start:                                        /* Loop until count is 0 */
        DECFSZ count, 1, 0
        GOTO done
        BRA start
done:
_endasm
```

Etiquette= label

Cet assembleur diffère néanmoins de MPASM sur les points suivants :

pas les directives, commentaires rédigés dans la syntaxe du C,

pas de valeur par défaut, *i.e.*, les arguments d'une instruction doivent être spécifiés,
la convention de notation en hexadecimal est 0xNN,
les étiquettes doivent comporter un « : »,
pas d'adressage indexé.

Quelques astuces/pièges en langage C

Utiliser le passage des arguments par adresse

La fonction ne travaille pas sur une copie de la valeur de l'argument effectif mais directement sur celui-ci (gain de place en mémoire).

Les fonctions récursives

Elles sont à éviter pour la programmation des systèmes embarqués à cause d'un débordement de pile matériel.
En effet, on rappelle que sur un micro-contrôleur,
le nombre d'appels de fonctions imbriqués est limité (31 pour PIC).

Plan du cours

1 Programmation d'un microcontrôleur en langage C

* Un langage évolué : intérêt et limitation.

* Éléments de programmation en C / exercices

2 Retour sur quelques points clés...

* les interruptions
* les directives pragma

3 Programmation en langage C des interruptions

* Le C18 et la programmation des interruptions

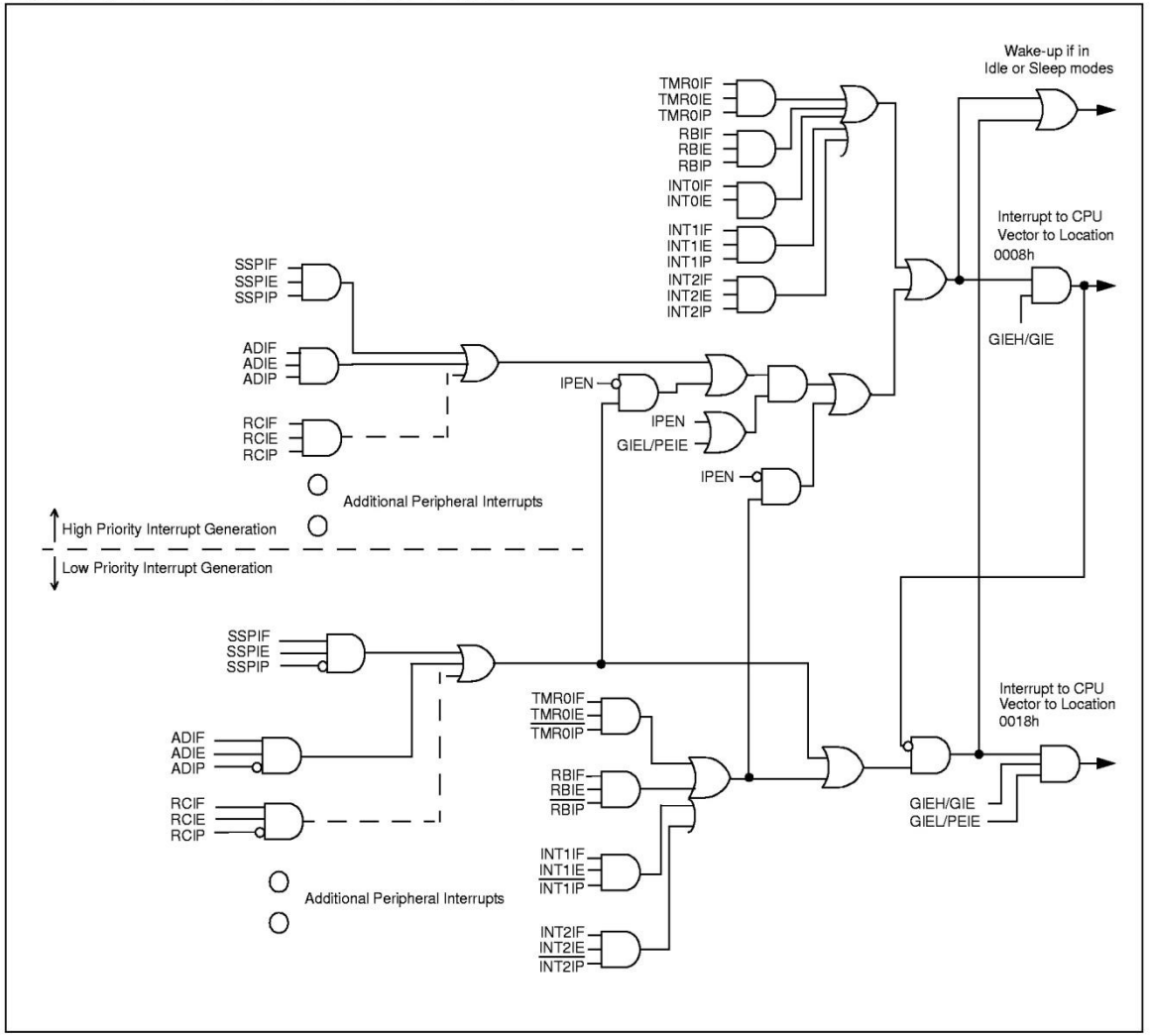
* Utilisation du TIMER0 (exercice)

Les interruptions (ou traitement d'exception)

Les interruptions matérielles sont utilisées lorsqu'il est nécessaire de pouvoir réagir en temps réel à un événement asynchrone...»

(Source : *Wikipédia*)

FIGURE 9-1: PIC18 INTERRUPT LOGIC



Une interruption peut avoir différentes sources : périphérique d'entrée/sortie, *timer*, *watchdog*, ...

Le contrôle des interruptions

1. Interruptions de priorité hautes et **basses...**

* un **vecteur d'interruption** est associé à chaque niveau de priorité pour permettre une gestion différente selon la priorité.

* Si une interruption de haute priorité se produit pendant une interruption de priorité basse, l'interruption de haute priorité « prend la main ».

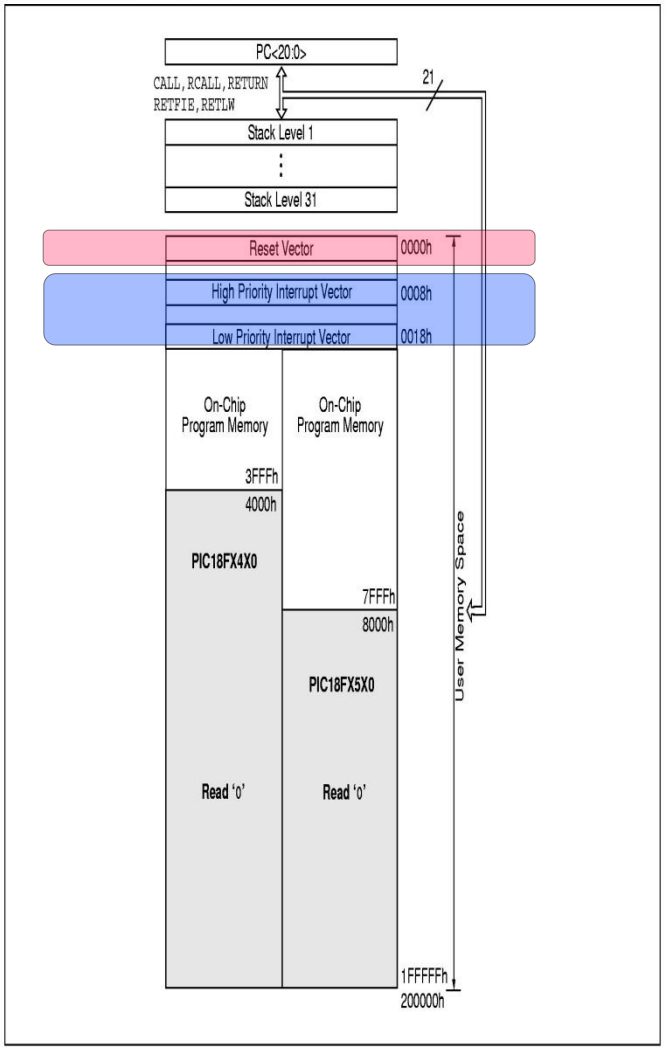
2. Les interruptions sont contrôlées par 3 bits...

* **bit de flag** : drapeau qui identifie l'origine de l'interruption,

* **bit d'activation** : permet à une source donnée de générer des interruptions...

* **bit de priorité** : permet de sélectionner la priorité d'une source d'interruption...

FIGURE 5-1: PROGRAM MEMORY MAP AND STACK FOR PIC18F2420/2520/4420/4520 DEVICES



Déroulement d'une interruption

(1). **Réception de l'interruption :** le micro-contrôleur reçoit une interruption.

(2). **Sauvegarde des données :** le micro-contrôleur sauve une partie variable (en fonction du type d'interruption) de son état interne dans la pile, notamment l'adresse dans la mémoire programme où le micro-contrôleur s'est arrêté.

(3). **Lecture de l'adresse du vecteur d'interruption et chargement dans le PC.**

(4). **Exécution de la routine d'interruption,**

Attention !! L'utilisateur doit penser à effectuer une **sauvegarde de données** du programme principal pour ne pas les effacer pendant la routine d'interruption et également à **supprimer le flag d'interruption** qui a déclenché l'interruption.

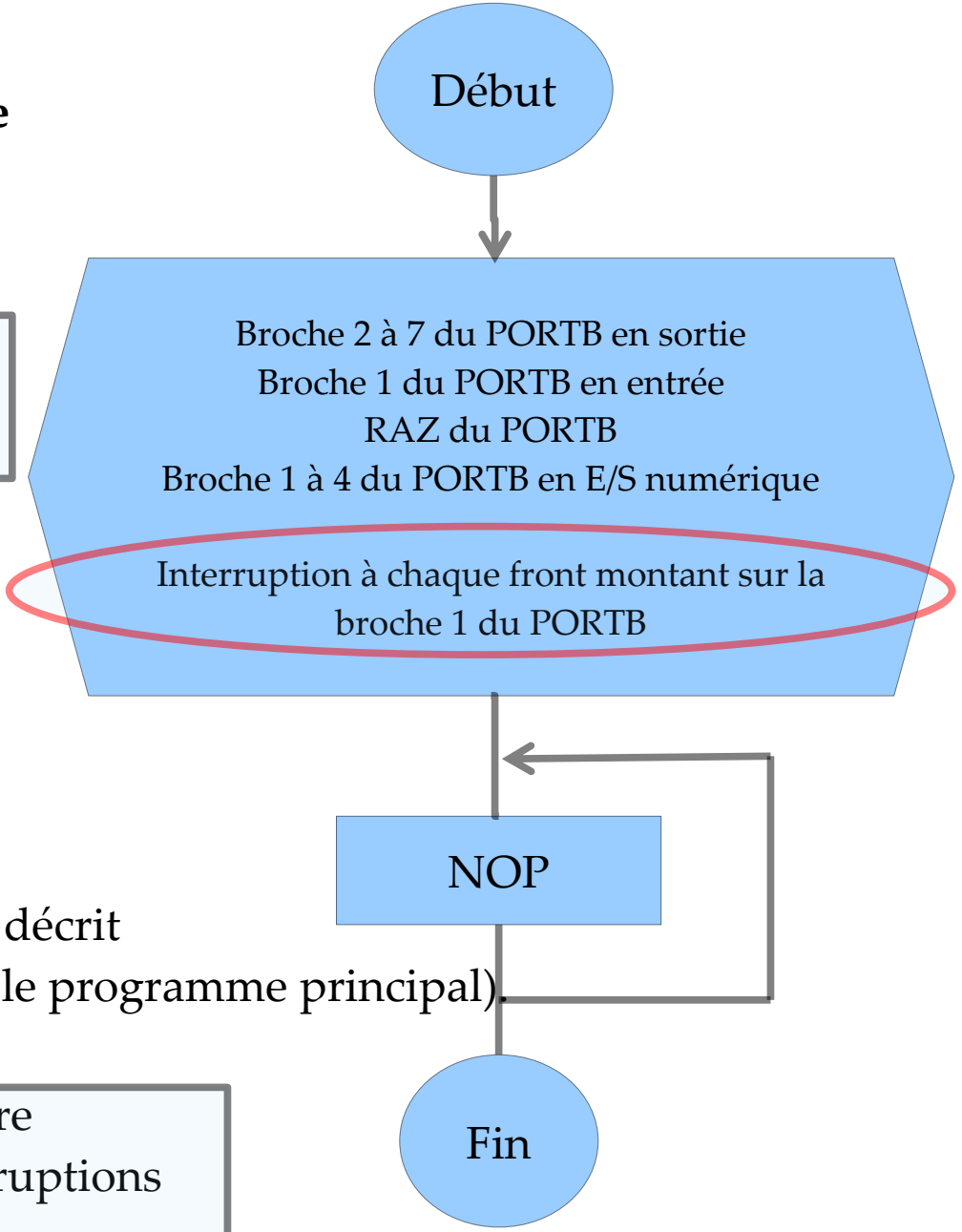
(5). **Rétablissement des données :** le micro-contrôleur rétablie les données stockées dans la pile.

(6). **Le micro-contrôleur reprend son fonctionnement normal...**

Un exemple...

Un programme avec interruption se décompose en deux parties distinctes...

La première spécifie le fonctionnement « hors interruption » de l'algorithme.



Voici un exemple (très) simple...

On notera que...

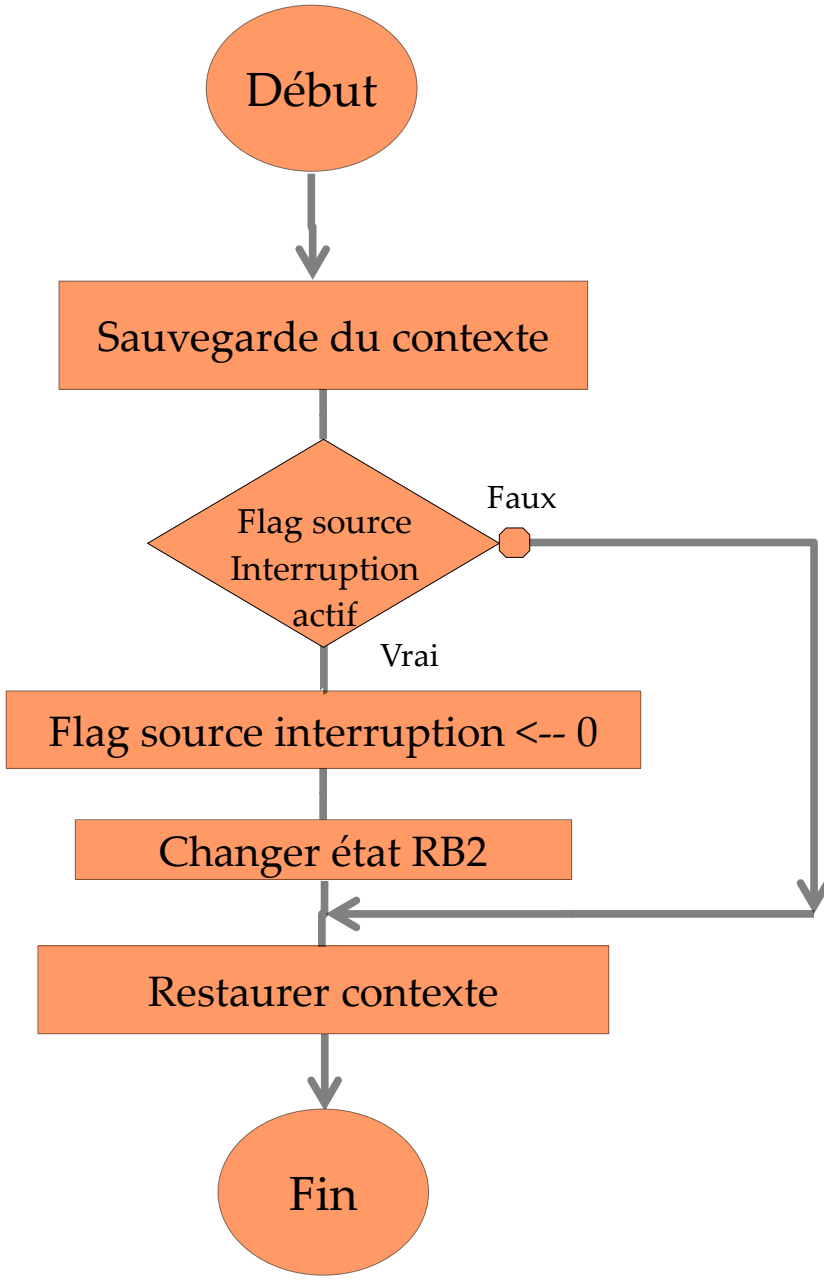
1. ce mode de fonctionnement est généralement décrit par plusieurs algorigrammes (un par fonction + le programme principal).

2. l'initialisation (programme principal) configure le microcontrôleur de manière à ce que les interruptions puissent se produire.

Un exemple...

Un programme avec interruption se décompose en deux parties distinctes...

La seconde spécifie le fonctionnement d'exception de l'algorithme, c.à.d. Ce qui doit être fait si l'interruption a lieu.



Voici un exemple (très) simple...
On notera que...

1. le programme principal ne faisant rien (!),
la sauvegarde du contexte est ici « virtuelle ».

2. on teste systématiquement l'origine de l'interruption.
Si plusieurs sources sont possibles,
plusieurs tests doivent être menés...

Le début d'un programme en assembleur, avec interruption,
reste très proche de celui d'une version sans interruption.
On peut tout de même remarquer des directives de **réservation d'emplacements mémoire**
en prévision de la sauvegarde du contexte lors de l'interruption.

```
; Filename : premier_programme_interruption.asm
; Change l'état de la broche 2 du PORTB à chaque front
; montant sur la broche 1 du PORTB (gestion par interruption
; Author:       Eric Magraner
; Company:      Université Paul Cézanne
; Revision:     1.00
; Date:                 2006/07
```

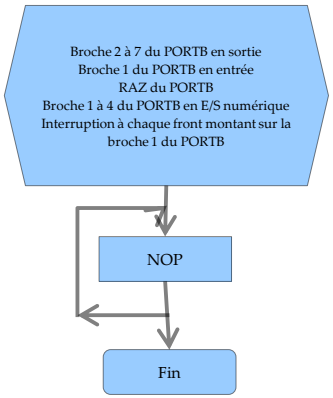
Début

```
list p=18f4520
; Définition du micro-contrôleur utilisé
```

```
#include <p18f4510.inc>
; Définitions des emplacements mémoires des registres
; et configurations matérielles par défaut

#include <MA_CONFIG.inc>
; Modification des configurations matérielles par défaut
W_TEMP                  RES     1; Réservation d'un octet en mémoire
STATUS_TEMP             RES     1; Réservation d'un octet en mémoire
BSR_TEMP                RES     1; Réservation d'un octet en mémoire
```

Du code du programme principal, on distingue les étapes classiques d'initialisation du vecteur RESET et du PORT B.
On note aussi les parties propre aux interruptions : **initialisation du vecteur et du registre d'INTERRUPTION**.

```
        org     h'0000'                 ; Init. du vecteur RESET
                goto    init

                org             h'0008' ; Init. du vecteur INTERRUPTION
                goto    routine_interruption

init            clrf    PORTB
                movlw   b'00000001'
                movwf   TRISB           ; Config. de la dir. du PORTB
                clrf    LATB
                movlw   0Fh
                movwf   ADCON1          ; Broche 1à4 du PORTB en E/S num.
                movlw   b'10010000'     ; 0x90 -> w
                movwf   INTCON          ; w -> INTCON
                                        (Init. du registre d'interrup.)

boucle          nop
                goto    boucle
                END
```



Le registre d'interruption INTCON permet, *d'une part* d'activer les interruptions (bit 7), et *d'autre part* d'activer le mode interruption externes INT0 (bit 4).
Dans ce cas, l'interruption sera détectée sur la broche 0 du port B (*cf. datasheet*).

REGISTER 9-1: INTCON REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF
bit 7							bit 0

bit 7 GIE/GIEH: Global Interrupt Enable bit
When IPEN = 0:
1 = Enables all unmasked interrupts
0 = Disables all interrupts
When IPEN = 1:
1 = Enables all high priority interrupts
0 = Disables all interrupts

bit 6 PEIE/GIEL: Peripheral Interrupt Enable bit
When IPEN = 0:
1 = Enables all unmasked peripheral interrupts
0 = Disables all peripheral interrupts
When IPEN = 1:
1 = Enables all low priority peripheral interrupts
0 = Disables all low priority peripheral interrupts

bit 5 TMR0IE: TMR0 Overflow Interrupt Enable bit
1 = Enables the TMR0 overflow interrupt
0 = Disables the TMR0 overflow interrupt

bit 4 INT0IE: INT0 External Interrupt Enable bit
1 = Enables the INT0 external interrupt
0 = Disables the INT0 external interrupt

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

bit 2 TMR0IF: TMR0 Overflow Interrupt Flag bit
1 = TMR0 register has overflowed (must be cleared in software)
0 = TMR0 register did not overflow

bit 1 INT0IF: INT0 External Interrupt Flag bit
1 = The INT0 external interrupt occurred (must be cleared in software)
0 = The INT0 external interrupt did not occur

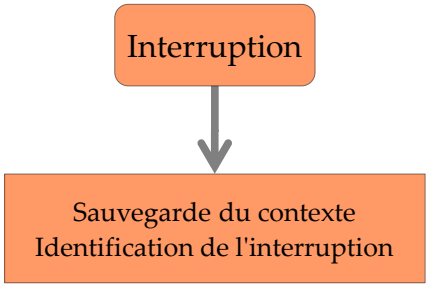
bit 0 RBIF: RB Port Change Interrupt Flag bit
1 = At least one of the RB7:RB4 pins changed state (must be cleared in software)
0 = None of the RB7:RB4 pins have changed state

Note: A mismatch condition will continue to set this bit. Reading PORTB will end the mismatch condition and allow the bit to be cleared.

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

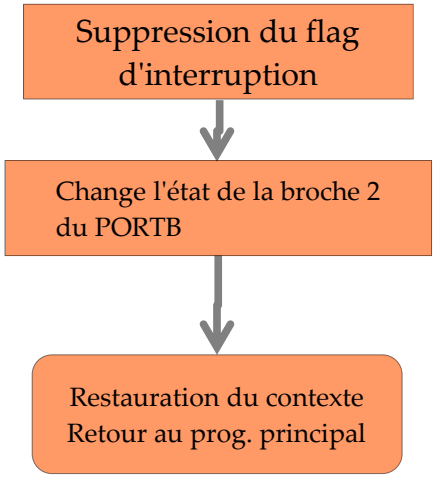
Le déclenchement d'une interruption conduit le microcontrôleur
à sauver l'adresse de l'instruction courante dans la pile, puis à charger le vecteur d'interruption dans le PC.

Dès lors, il est **systématiquement** nécessaire de
(1) **sauvegarder le contexte** et (2) **identifier l'origine de l'interruption**.




```
routine_interruption

        ; Sauvegarde du contexte
        movwf   W_TEMP                  ; Sauvegarde de W
        movff   STATUS, STATUS_TEMP     ; Sauvegarde de STATUS
        movff   BSR, BSR_TEMP           ; Sauvegarde de BSR

        ; identification de l'origine de l'interruption
        btfsc   INTCON,1
        goto    interruption_INT0
        bra             restauration_contexte
```

Il faut ensuite **systématiquement**
(3) mettre à **zéro le bit d'interruption** puis,
(4) **exécuter la fonction** pour laquelle l'interruption a été prévue, et enfin
(5) faire la **restauration du contexte**
(6) et retourner au programme principal.




```
interruption_INT0

        bcf     INTCON,1; Suppression du flag d'interruption

        movlw   0x02            ; 0x02 -> w
        xorwf   PORTB           ; w xor PORTB -> PORTB
        goto    restauration_contexte

        ;       Restauration du contexte
restauration_contexte
        movff   BSR_TEMP, BSR   ; Restauration de BSR
        movff   W_TEMP, W       ; Restauration de W
        movff   STATUS_TEMP, STATUS ; Restauration de STATUS

        retfie
```

Les directives « pragma » du C18

La norme ANSI C ne permet pas de contrôler les adresses mémoires où sont placées des variables ou des instructions.
Pour pallier ce problème, C18 introduit les directives #pragma qui permettent d'accéder spécifiquement à des adresses de la mémoire donnée et programme ainsi que de configurer le microcontrôleur.

Ainsi la directive

```
#pragma config WDT = OFF
```

permet de désactiver la fonctionnalité « watch-dog » du microcontrôleur ;
la liste des fonctionnalités et de leur valeurs est disponible dans la documentation DS51537.

Cette directive permet de placer un morceau de code
à une adresse de la mémoire programme que l'on choisit.

```
#pragma code    mon_prog = 0x@                    // après : placement imposé dans la mémoire
void mon_prog(void){
  instructions;
}
#pragma code                                      // après : placement libre dans la mémoire
```


permet de placer `instructions;` à l'adresse `0x@` de la **mémoire programme**.

Plan du cours

1 Programmation d'un microcontrôleur en langage C

* Un langage évolué : intérêt et limitation.

* Éléments de programmation en C / exercices

2 Retour sur quelques points clés...

* les interruptions
* les directives pragma

3 Programmation en langage C des interruptions

* Le C18 et la programmation des interruptions

* Utilisation du TIMER0 (exercice)

Gérer les interruptions avec C18

Les directives pragma permettent en particulier de configurer les d'interruptions :
une première directive permet d'*initialiser le vecteur d'interruption* (haute ou basse)
une seconde directive identifie l'adresse mémoire de cette fonction :

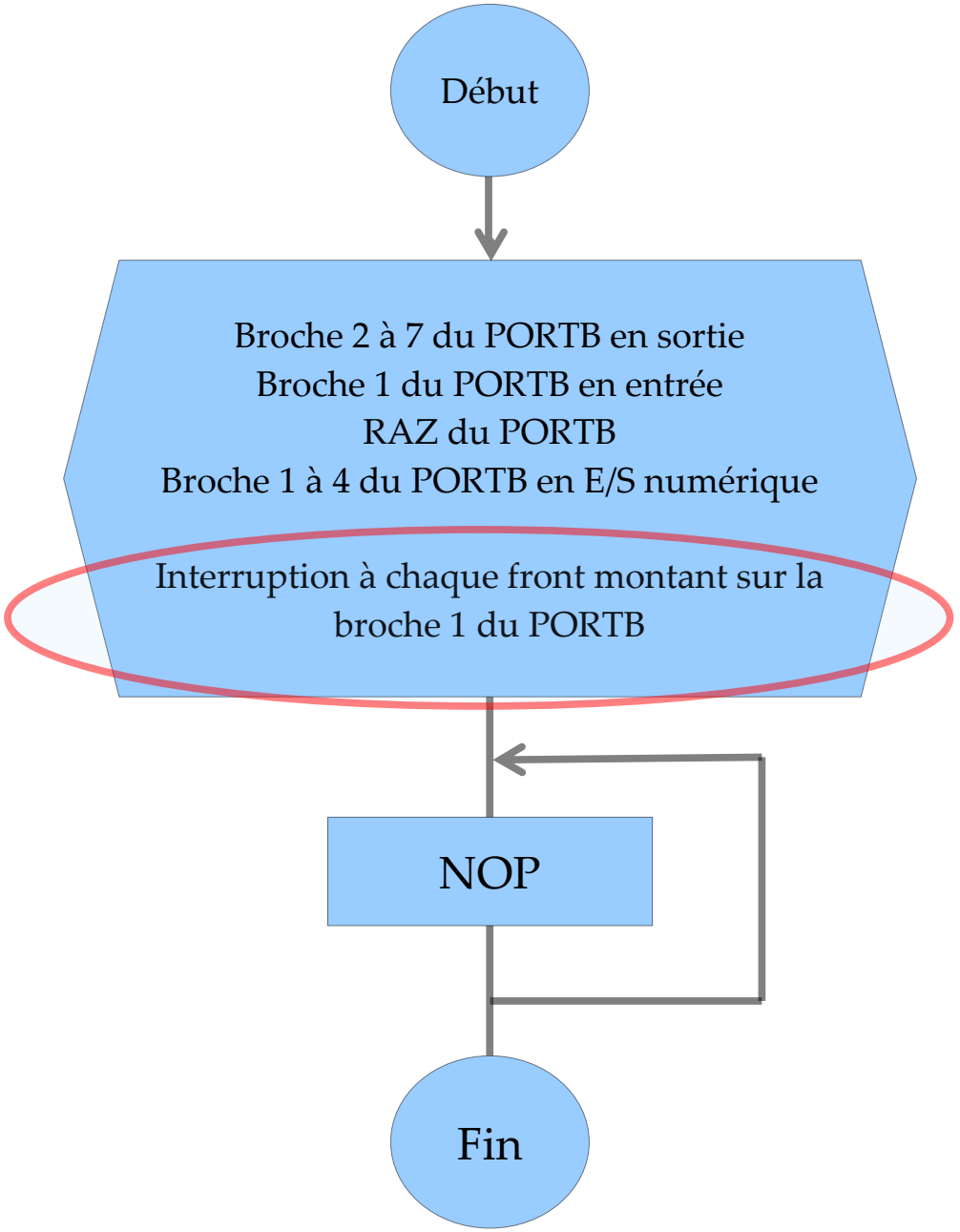
```
#pragma code  vecteur_low = 0x18   ← Case mémoire (18)h

void vecteur_low(){
    _asm goto routine_int_low _endasm
};

#pragma code


#pragma interruptlow routine_int_low
void routine_int_low(){
...};
```

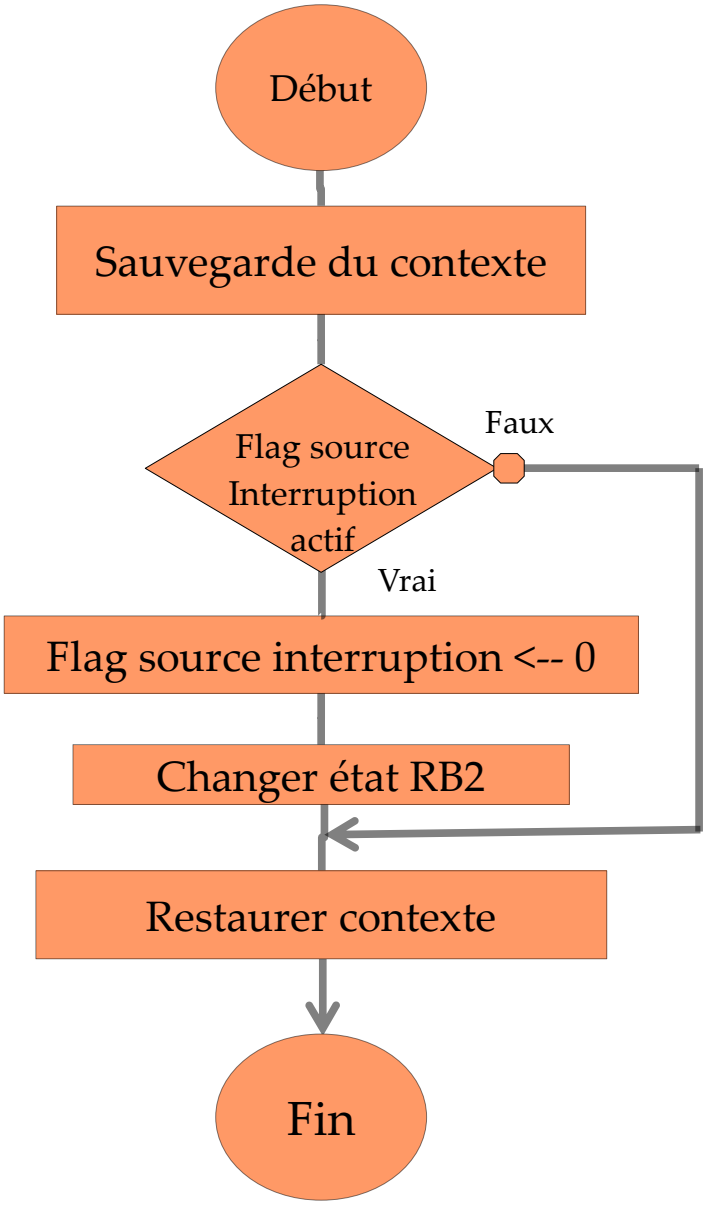
Quelques remarques :
(**1**) une fonction d'interruption ne possède ni entrée ni sortie ;
(**2**) une sauvegarde minimale du contexte est assurée automatiquement
et peut être complétée en modifiant la seconde directive comme suit

```
#pragma interruptlow routine_int_low save = ma_variable
```

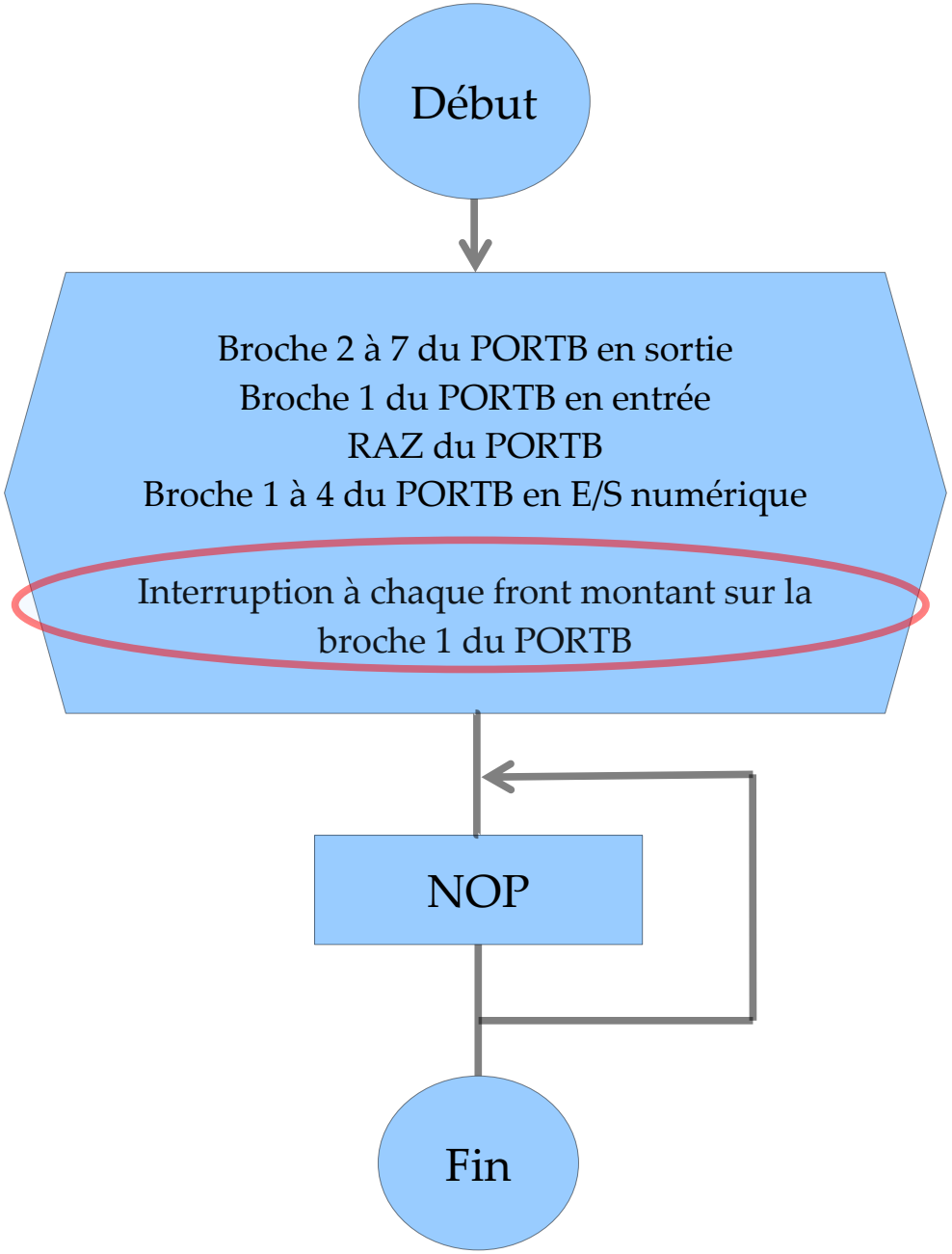

où ma_variable est une variable **globale** à sauver.

Un exemple simple (le même)



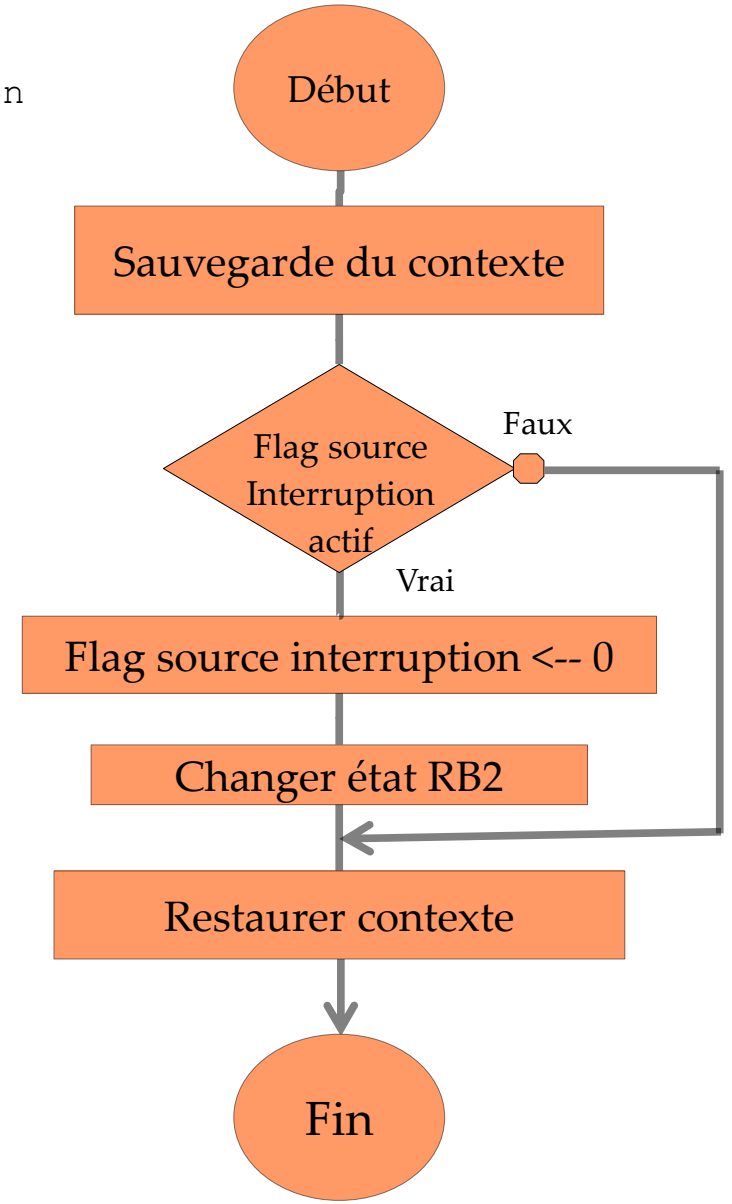


```
//=======================================
// Filename: Interruption_C.C
//=======================================
// Authors:  Marc Allain, Julien Marot
//=======================================

#include <p18f4520.h>
#pragma config WDT = OFF

// Prototypes des fonctions
void Vecteur_interruption(void);
void Routine_gestion_interruption(void);

// Programme Principal
void main()
{
        PORTB = 0;
        TRISB = 0x01;
        LATB = 0;
        ADCON1 = 0x0F;
        INTCON = 0x90;
        while(1){
        }
}
```



```
// Indique l'adresse mémoire ou débute la fonction
// décrite juste après la directive #pragma code
#pragma code Vecteur_interruption = 0x08
void Vecteur_interruption (void)
{
  _asm
    goto Routine_gestion_interruption
  _endasm
}
#pragma code
// Retour au placement libre du programme
// dans la mémoire
```

```
// Routine de gestion des interruptions
#pragma interrupt Routine_gestion_interruption
void Routine_gestion_interruption()
{
        // Vérification que l'interruption est
        // bien déclenchée par RB0
        if (INTCONbits.INT0IF)
    {
        // Suppression du flag d'interruption
        INTCONbits.INT0IF = 0;
        // Change l'état de RB1
        PORTB ^= 0x02;
    }
}
```



Chronographe basé PIC 18F4520 : le TIMER

Les timers sont des registres incrémentés à chaque réalisation d'un événement, la valeur de ces registres pouvant être pré-positionnée à une valeur initiale.

Les événements qui *commandent* l'incrémentation sont

un cycle d'horloge, c'est la fonction « *timer* » ;

un front montant sur une broche en entrée, c'est la fonction « *counter* ».

Il en découle que le module *timer* peut remplir les fonctions suivantes,

Utilisation « timer » : permet de fournir une *référence temporelle* à partir de l'horloge du micro-contrôleur, notamment dans le cadre d'applications temps réel.

Utilisation « counter » : sert à compter un *nombre d'événements asynchrones* sur une broche d'entrée du micro-contrôleur.

Cahier des charges :

On cherche à utiliser le « module timer » du microcontrôleur pour faire clignoter une LED connectée sur le port RB1. La période est fixée à une fréquence de 1 Hz.

Une méthode générale pour utiliser les modules du microcontrôleur...

(1) Lire dans la documentation (*data-sheet*) la section traitant du module.

(2) Déduisez-en les registres à configurer lors de la phase d'initialisation.

(3) Construisez l'algorigramme préalable à l'écriture du programme.

(4) Écrivez le programme en C18, testez-le et débuggez-le...

Extrait du data-sheet du PIC18F4520, p. 123-125

11.0 TIMER0 MODULE

The Timer0 module incorporates the following features:

- Software selectable operation as a timer or counter in both 8-bit or 16-bit modes
- Readable and writable registers
- Dedicated 8-bit, software programmable prescaler
- Selectable clock source (internal or external)
- Edge select for external clock
- Interrupt-on-overflow

The T0CON register (Register 11-1) controls all aspects of the module's operation, including the prescale selection. It is both readable and writable.

A simplified block diagram of the Timer0 module in 8-bit mode is shown in Figure 11-1. Figure 11-2 shows a simplified block diagram of the Timer0 module in 16-bit mode.

REGISTER 11-1: T0CON: TIMER0 CONTROL REGISTER

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0
bit 7							bit 0

bit 7 **TMR0ON:** Timer0 On/Off Control bit
1 = Enables Timer0
0 = Stops Timer0

bit 6 **T08BIT**: Timer0 8-bit/16-bit Control bit
1 = Timer0 is configured as an 8-bit timer/counter
0 = Timer0 is configured as a 16-bit timer/counter

bit 5 **T0CS**: Timer0 Clock Source Select bit
1 = Transition on T0CKI pin
0 = Internal instruction cycle clock (CLKO)

bit 4 **T0SE**: Timer0 Source Edge Select bit
1 = Increment on high-to-low transition on T0CKI pin
0 = Increment on low-to-high transition on T0CKI pin

bit 3 **PSA**: Timer0 Prescaler Assignment bit
1 = TImer0 prescaler is NOT assigned. Timer0 clock input bypasses prescaler.
0 = Timer0 prescaler is assigned. Timer0 clock input comes from prescaler output.

bit 2-0 **T0PS2:T0PS0**: Timer0 Prescaler Select bits
111 = 1:256 prescale value
110 = 1:128 prescale value
101 = 1:64 prescale value
100 = 1:32 prescale value
011 = 1:16 prescale value
010 = 1:8 prescale value
001 = 1:4 prescale value
000 = 1:2 prescale value

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

11.1 Timer0 Operation

Timer0 can operate as either a timer or a counter; the mode is selected with the T0CS bit (T0CON<5>). In Timer mode (T0CS = 0), the module increments on every clock by default unless a different prescaler value is selected (see **Section 11.3 "Prescaler"**). If the TMR0 register is written to, the increment is inhibited for the following two instruction cycles. The user can work around this by writing an adjusted value to the TMR0 register.

The Counter mode is selected by setting the T0CS bit (= 1). In this mode, Timer0 increments either on every rising or falling edge of pin RA4/T0CKI. The incrementing edge is determined by the Timer0 Source Edge Select bit, T0SE (T0CON<4>); clearing this bit selects the rising edge. Restrictions on the external clock input are discussed below.

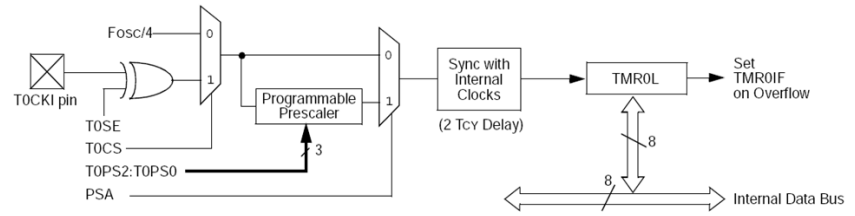
An external clock source can be used to drive Timer0; however, it must meet certain requirements to ensure that the external clock can be synchronized with the internal phase clock (TOSC). There is a delay between synchronization and the onset of incrementing the timer/counter.

11.2 Timer0 Reads and Writes in 16-Bit Mode

TMR0H is not the actual high byte of Timer0 in 16-bit mode; it is actually a buffered version of the real high byte of Timer0 which is not directly readable nor writable (refer to Figure 11-2). TMR0H is updated with the contents of the high byte of Timer0 during a read of TMR0L. This provides the ability to read all 16 bits of Timer0 without having to verify that the read of the high and low byte were valid, due to a rollover between successive reads of the high and low byte.

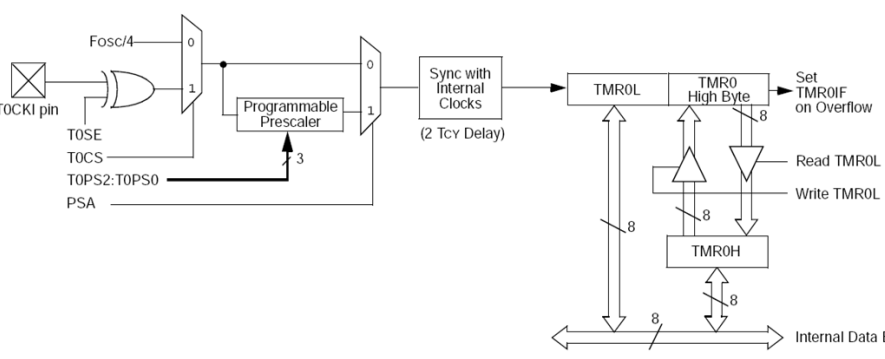
Similarly, a write to the high byte of Timer0 must also take place through the TMR0H Buffer register. The high byte is updated with the contents of TMR0H when a write occurs to TMR0L. This allows all 16 bits of Timer0 to be updated at once.

FIGURE 11-1: TIMER0 BLOCK DIAGRAM (8-BIT MODE)



Note: Upon Reset, Timer0 is enabled in 8-bit mode with clock input from T0CKI max. prescale.

FIGURE 11-2: TIMER0 BLOCK DIAGRAM (16-BIT MODE)



Note: Upon Reset, Timer0 is enabled in 8-bit mode with clock input from T0CKI max. prescale.

Extrait du data-sheet du PIC18F4520, p. 123-125

11.3 Prescaler

An 8-bit counter is available as a prescaler for the Timer0 module. The prescaler is not directly readable or writable; its value is set by the PSA and T0PS2:T0PS0 bits (T0CON<3:0>) which determine the prescaler assignment and prescale ratio.

Clearing the PSA bit assigns the prescaler to the Timer0 module. When it is assigned, prescale values from 1:2 through 1:256 in power-of-2 increments are selectable.

When assigned to the Timer0 module, all instructions writing to the TMR0 register (e.g., `CLRF TMR0`, `MOVWF TMR0`, `BSF TMR0`, etc.) clear the prescaler count.

Note:	Writing to TMR0 when the prescaler is assigned to Timer0 will clear the prescaler count but will not change the prescaler assignment.

11.3.1 SWITCHING PRESCALER ASSIGNMENT

The prescaler assignment is fully under software control and can be changed "on-the-fly" during program execution.

11.4 Timer0 Interrupt

The TMR0 interrupt is generated when the TMR0 register overflows from FFh to 00h in 8-bit mode, or from FFFFh to 0000h in 16-bit mode. This overflow sets the TMR0IF flag bit. The interrupt can be masked by clearing the TMR0IE bit (INTCON<5>). Before re-enabling the interrupt, the TMR0IF bit must be cleared in software by the Interrupt Service Routine.

Since Timer0 is shut down in Sleep mode, the TMR0 interrupt cannot awaken the processor from Sleep.

TABLE 11-1: REGISTERS ASSOCIATED WITH TIMER0

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Reset Values on page
TMR0L	Timer0 Register, Low Byte								50
TMR0H	Timer0 Register, High Byte								50
INTCON	GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF	49
T0CON	TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0	50
TRISA	RA7(1)	RA6(1)	RA5	RA4	RA3	RA2	RA1	RA0	52

Legend: Shaded cells are not used by Timer0.

Note 1: PORTA<7:6> and their direction bits are individually configured as port pins based on various primary oscillator modes. When disabled, these bits read as '0'.

Questions :

(1) Expliquez comment fonctionne le module TIMER0 ?
Comment peut-on l'utiliser pour faire basculer la sortie RB1 toute les 0.5 seconde ?

(2) Donnez les valeurs d'initialisation des différentes registres
pour rentrer dans le cahier des charges.

(3) Construisez l'algorigramme préalable à l'écriture du microcode.

(4) Finalement, écrivez le programme en assembleur.

(5) Évaluez l'erreur sur la période associée au temps d'exécution
du code et modifiez les registres en conséquence.