



Nous allons, dans cet exercice, nous intéresser à une problématique extrêmement importante dans le domaine de l'embarqué ... la gestion de bases de temps. Prenons l'exemple simple de l'exercice 2 dans lequel nous allons chercher à faire clignoter une LED. Pour arriver à nos fins il existe deux techniques :

software

ou

hardware

Solution n°1 (SOFT) : La première technique, la plus mauvaise, consiste à exécuter une section de code qui décrémente une valeur jusqu'à son passage à "0" avant d'exécuter la suite du programme. La temporisation est donc purement logicielle. Nous devons donc **ABSOLUMENT** connaître le temps d'exécution de chaque instruction ainsi que le nombre exact d'instructions de notre boucle.

Solution n°2 (HARD): La deuxième solution, bien meilleure, consiste à utiliser un Timer qui est un périphérique interne au MCU. Ce dernier est en fait un compteur "hardware" indépendant du CPU. Une fois configuré et démarré ce périphérique pourra, une fois sa tâche accomplie, interrompre le CPU en lui envoyant des demandes d'interruptions. Nous sommes donc gagnant sur deux aspects :

1. Nous ne monopolisons pas le CPU, et nous pouvons donc exécuter un programme en parallèle
2. Nous gagnons en précision, en temps de développement et nous pouvons dans certains cas effectuer des temporisations ou des cadencements très précis.

Nous allons donc dans un premier temps nous intéresser à l'implémentation de temporisations logicielles ...

1. Comment connaître le temps d'exécution d'une instruction ?

Si nous souhaitons connaître la durée exacte d'exécution d'une instruction, nous devons avant tout connaître la durée d'un "cycle instruction" : **Tcy. Chaque instruction possède un temps d'exécution multiple de Tcy.** Prenons l'exemple de l'instruction GOTO.

Voici ci-dessous une partie des informations fournies dans la documentation du PIC18F4550 concernant l'instruction GOTO. Nous constatons que le temps d'exécution d'un GOTO est de deux Tcy.

GOTO	Unconditional Branch
Syntax:	GOTO k
Operands:	$0 \leq k \leq 1048575$
Operation:	$k \rightarrow PC<20:1>$
Words:	2
Cycles:	2 ← 2xTcy
Example:	GOTO THERE
After Instruction PC = Address (THERE)	

figure 1 : informations sur l'instruction GOTO (cf. datasheet PIC18F4550)

Pour connaître la valeur de Tcy il faut maintenant connaître la valeur de **Tosc=1/Fosc**, sachant que **Tcy=4xTosc**. **Fosc** est la fréquence interne de travail de notre MCU. Dans notre cas, durant la totalité des TP, **Fosc=48MHz** ceci étant la vitesse maximale de travail de notre micro-contrôleur. La configuration de la vitesse de travail se fait TOUJOURS en en-tête d'un programme ... nous l'avons d'ailleurs déjà rencontré durant l'exercice 1 :

Configuration bits	
CONFIG	PLLDIV=2, CPUDIV=OSC1_PLL2, FOSC=HSPLL_HS

En fixant les champs **PLLDIV**, **CPUDIV** et **FOSC** aux valeurs données ci-dessus nous obtenons une fréquence interne de travail Fosc égale à 48MHz. Pour les curieux souhaitant connaître le sens de ces valeurs, vous pourrez en séance aller voir le document fourni à l'installation du compilateur C18 et présent dans le répertoire **C:\MCC18\doc\hlpPIC18ConfigSet.chm**. Il vous faut uniquement savoir que nous travaillons avec un oscillateur externe à 8MHz (cf. **figure 2**).

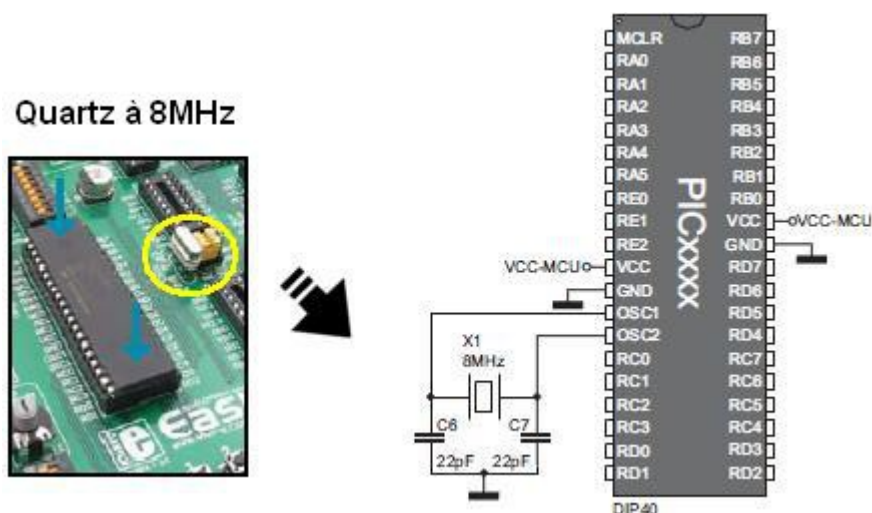


figure 2 : schéma de câblage de l'oscillateur externe utilisé (quartz à 8MHz)



Reprenons l'exemple de l'instruction *GOTO* et calculons son temps d'exécution :

$$\text{Durée GOTO} = 2 * T_{cy} = 2 * (4 * T_{osc}) = 2 * (4 * (1/48 * 10^6)) \approx 166,66 \text{ ns}$$

avec : $T_{osc} \approx 1/F_{osc} = 20,8 \text{ ns}$ $T_{cy} \approx 83,33 \text{ ns}$

Vous disposez maintenant de tous les éléments pour mener à bien la première partie de l'exercice 2. Pour cette partie, nous vous conseillons fortement d'utiliser l'instruction *decfsz*. Il vous faudra donc aller voir dans la documentation technique du PIC18F4550 son fonctionnement ainsi que le nombre de cycles correspondant à son exécution.

2. Qu'est-ce qu'un Timer ?

Un Timer est un périphérique interne du MCU. Il s'agit d'un compteur avec pré-chargement déclenché sur fronts montants d'une horloge de référence. Notre micro-contrôleur possède quatre Timers (Timer0 à Timer3). Nous allons dans le cadre de ce TP utiliser le Timer0 qui peut fonctionner selon deux modes (8bits ou 16bits). Le mode 16bits est celui utilisé en TP et est légèrement plus complexe que le mode 8bits.



Voilà **figure 3** ce que l'on peut trouver derrière le Timer0. Ce schéma fonctionnel peut se découper en deux parties. La première permettant de fixer l'horloge de référence pour le comptage. La seconde partie permet de fixer la valeur de départ pour le compteur. Au maximum, sur 16bits, le compteur pourra compter de 0x0000 à 0xFFFF. Dans la très grande majorité des cas, nous le ferons partir d'une valeur initiale chargée dans les registres *TMR0H* et *TMR0L*.

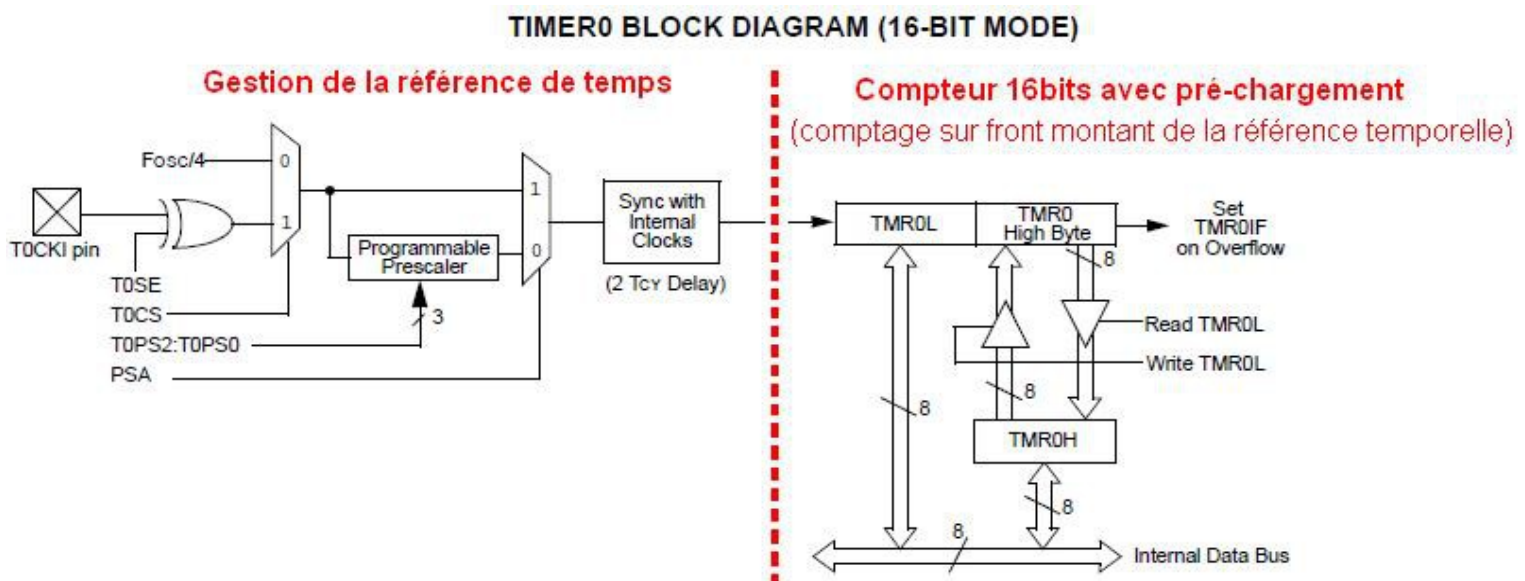


figure 3 : schéma fonctionnel du Timer0 en mode 16bits

Quelque soit le périphérique, avant de l'utiliser il faut le configurer. Une fois configuré le périphérique accomplira la tâche pour laquelle il a été configuré de façon entièrement autonome. Pour configurer un périphérique et donc notre Timer0, il faut écrire dans un ou plusieurs registres de contrôle. Voici ci-dessous le descriptif de l'un des principaux registre de contrôle du Timer0, le registre **T0CON** (Timer0 CONTROL).

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

figure 4 : registre de contrôle T0CON associé au Timer0



*Par exemple, si nous souhaitons travailler avec l'horloge interne déterminée à partir des "bits de configuration" (cf. 1. Comment connaître le temps d'exécution d'une instruction), il nous suffit de mettre le bit **TOCS** du registre **T0CON** à '0'. Nous pouvons d'ailleurs constater sur la **figure 3** que le bit **TOCS** permet de commuter l'horloge source ($F_{osc}/4$ ou externe sur la broche **T0CKI**). En langage "C", nous écririons cette configuration comme suit :*

T0CONbits.TOCS=0;

Vous trouverez **figure 5** le détail de tous les registres associés au Timer0 dans le cadre d'une utilisation avec Fosc/4 comme horloge de référence.

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Reset Values on page
TMR0L	Timer0 Module Low Byte Register								52
TMR0H	Timer0 Module High Byte Register								52
INTCON	GIE/GIEH	PEIE/GIEL	TMR0IE			TMR0IF			51
T0CON	TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0	52

figure 5 : détail des registres associés au Timer0

Parmi les registres présentés ci-dessus, le registre T0CON a été rencontré précédemment et les registres TMR0L et TMR0H ont été rapidement évoqués. Prenons un exemple concret d'utilisation de ces registres, celui de l'exercice 4 dans lequel nous chercherons à réaliser un DIGICODE. Notre application exige d'exécuter à intervalle de temps régulier une fonction (toutes les ~5ms). Intéressons nous à la configuration du Timer0, **mais attention, cette configuration n'est pas unique, ce n'est qu'une proposition !**

Langage "C"
Configuration du Timer0 – génération d'une base de temps de ~5ms
<pre> /** Timer0=OFF, mode 16bits, synchro. sur Fosc/4, prescaler=ON, prescale value 1:256 */ T0CON = 0x07; /** Pré-chargement de la valeur initiale pour le comptage dans TMR0H puis TMR0L */ TMR0H = 0xFF; TMR0L = 0x15; /** Démarrage du Timer0 */ T0CONbits.TMR0ON = 1; </pre>

Dans l'exemple donné ci-dessus, comment avons nous trouvé la valeur à charger dans les registres TMR0H et TMR0L ? Voici le détail du calcul :

$$5\text{ms} = \text{Timer0 clock source} \times \text{prescale value} \times \text{count value} \sim 1/(\text{Fosc}/4) \times 256 \times 234$$

Notre compteur devra donc compter **234** coûts d'horloge afin d'obtenir approximativement 5ms. Sachant qu'il compte de la valeur initiale jusqu'à 0xFFFF=65535, la valeur initiale doit donc être égale à **65535-234=65301=0xFF15=TMR0H+TMR0L**.



*Il faut cependant être prudent sur un point. Le registre TMR0H doit absolument être chargé avant TMR0L. Si vous regardez bien la **figure 3**, le registre réellement incrémenté est le registre 16 bits TMR0 dont les 8 bits LSB sont ceux de TMR0L et les 8 bits MSB sont ceux de TMR0H après chargement. Pour charger la valeur de TMR0H dans TMR0, il suffit d'écrire dans le registre TMR0L (cf. **figure 3**). Il nous faut donc initialiser le registre TMR0H avant TMR0L !*

3. Qu'est-ce qu'une interruption (IT)?

Nous avons jusqu'à présent réussi à configurer le Timer0. Rappelons que le Timer0 est un périphérique interne de notre micro-contrôleur et qu'il est totalement indépendant du CPU. Une fois configuré et démarré, il effectue la tâche pour laquelle il a été configuré, à savoir compter, jusqu'à ce qu'elle soit finie.

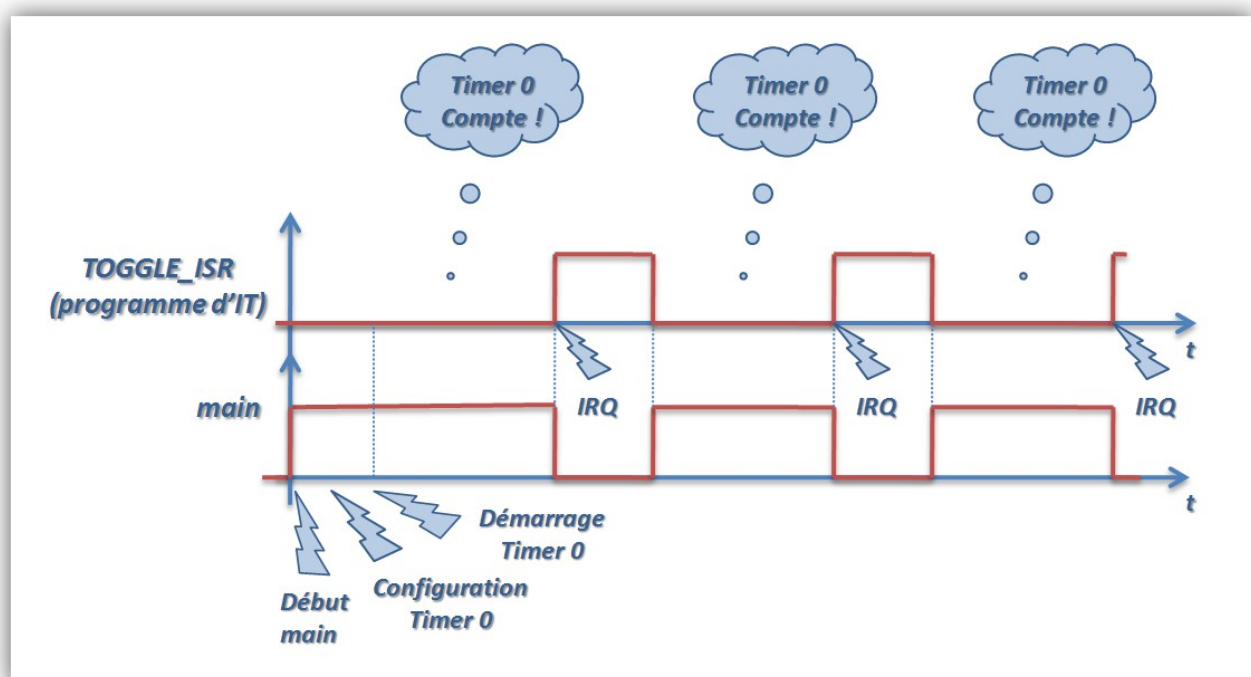


*Nous allons maintenant découvrir la technique utilisée par **n'importe quel** périphérique pour communiquer avec le CPU ... les **INTERRUPTIONS** !*

Un périphérique est un composant "hardware" capable de travailler de façon autonome. Il est important de comprendre qu'à aucun moment un périphérique n'est capable d'exécuter une instruction, c'est le rôle du CPU. Si un périphérique veut communiquer avec le CPU et donc interrompre le programme en cours d'exécution, il lui envoie une **demande d'interruption** ou IRQ. Une demande d'interruption est un **signal physique** passant de l'état inactif à l'état actif et allant du périphérique vers le CPU. Si le CPU est sensible aux demandes d'interruptions d'un périphérique, à chaque demande d'IT il devra exécuter le programme d'interruption correspondant. Nous devons donc répondre aux questions suivantes :

- ➔ Comment rendre le CPU sensible aux demandes d'IT d'un périphérique . Nous parlerons de **démasquage des interruptions** !
- ➔ Comment exécuter le bon programme d'IT. Nous parlerons de configuration du **vecteur d'interruption** !

Illustrons le concept des interruptions dans le cadre d'une utilisation du Timer 0 :



4. Qu'est ce qu'un vecteur d'interruption ?

Un vecteur d'interruption est une **"petite" zone mémoire**. Lorsqu'une demande d'IT arrive au CPU, celui-ci **termine l'instruction en cours d'exécution puis exécute le code se trouvant dans le vecteur d'interruption** correspondant. Un vecteur d'IT contient notamment une instruction de saut vers le programme d'IT (ISR). La zone mémoire contenant les différents vecteurs d'IT est souvent nommée table des vecteurs. Sur notre MCU la gestion des interruptions est relativement simple car il n'existe que trois vecteurs d'IT.

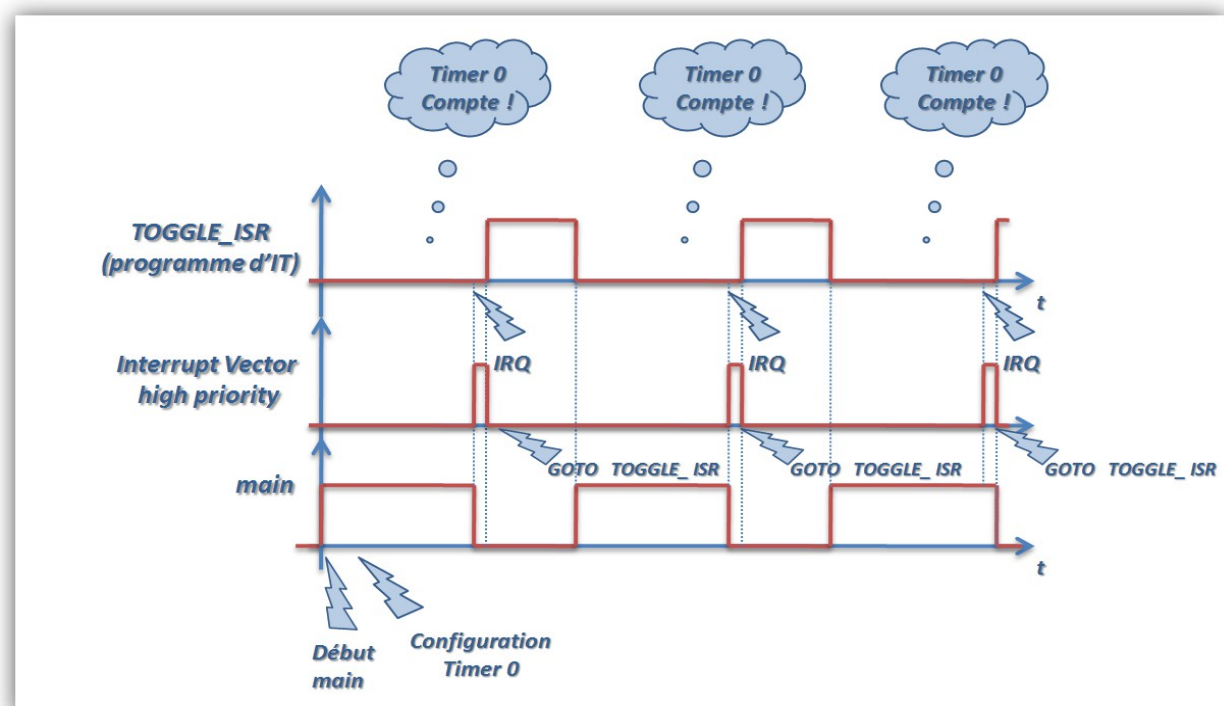


figure 6 : mapping de la mémoire programme



De façon générale, quelque soit le MCU, DSP ou MPU, les vecteurs d'IT se trouvent aux adresses les plus basses en mémoire. Il est cependant possible de les tradater dans le cadre de l'utilisation d'un bootloader.

Reprenons l'exemple du Timer 0. Avant d'exécuter le code du programme d'interruption (TOGGLE_ISR), nous exécutons celui du vecteur d'interruption. Celui choisi en TP est le vecteur de priorité haute :



Nous pouvons d'ailleurs maintenant mieux comprendre le code donné ci-dessous et présent dans l'exercice 1. Ces trois lignes de code permettent de signaler qu'à l'adresse 0x0000 de la mémoire programme (adresse du vecteur d'IT lié au Reset) correspondra l'étiquette "RESET_V:" (que nous aurions pu nommer TOTO_V:) et que nous y trouverons l'instruction "goto MAIN".

Assembleur	
Mise en place du vecteur d'interruption	
;*** Mise en place du vecteur d'IT lié au Reset ***	
org	0x0000
RESET_V:	
goto	MAIN



En cas de Reset, le CPU finira donc d'exécuter l'instruction en cours puis exécutera l'instruction 'goto main' se trouvant à l'adresse 0x0000. Notre programme sera donc lancé à nouveau !

5. Comment démasquer les IT sur notre MCU ?

Le démasquage des interruptions consiste à rendre notre CPU sensible aux demandes d'IT d'un périphérique. Par défaut, toutes les demandes d'IT son masquées, sauf celle du Reset qui n'est pas masquable. Notre MCU possède un grand nombre de périphérique (Timers, UART, ADC ...) et selon l'application tous ne sont pas forcément utilisés. Dans notre cas par exemple nous n'allons chercher qu'à démasquer les demandes d'IT arrivant du Timer0.

Avant de manipuler les interruptions de notre MCU, il vous faut savoir qu'elles peuvent être gérées selon deux modes de fonctionnement (**Legacy mode** ou **priority mode**). Le mode **legacy** est un mode limité qui n'existe que pour rendre compatible aux PIC18Fxxx les codes développés sur PIC16Cxxx. **Dans notre cas la question ne se pose pas et nous devons utiliser le priority mode** et ainsi utiliser nos deux vecteurs d'IT (haute priorité et basse priorité). Afin de se placer dans le priority mode, il suffit de mettre le bit **IPEN** du registre **RCON** à "1".

RCON: RESET CONTROL REGISTER

R/W-0	R/W-1 ⁽¹⁾	U-0	R/W-1	R-1	R-1	R/W-0 ⁽²⁾	R/W-0
IPEN	SBOREN	—	RI	TO	PD	POR	BOR
bit 7							bit 0

bit 7 **IPEN:** Interrupt Priority Enable bit

1 = Enable priority levels on interrupts

0 = Disable priority levels on interrupts (PIC16CXXX Compatibility mode)

Il existe un grand nombre de registres pour le démasquage des interruptions ... tout simplement car il existe un grand nombre de périphériques dans notre MCU. Voici ci-dessous probablement le registre le plus important correspondant notamment au démasquage des IT pour le Timer0, le registre **INTCON** (**INT**errupt **CON**trol).

INTCON: INTERRUPT CONTROL 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 high priority 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



Prenons l'exemple de notre application et du démasquage des interruptions concernant le Timer0. Les bits nous intéressant sont GIEH, GIEL, TMR0IE et TMR0IF. GIE correspond au legacy mode et GIEH au priority mode ! ... idem pour PEIE et GIEL.

Voici ci-dessous un schéma détaillant les mécanismes de démasquage des interruptions concernant le Timer0. Sur ce schéma, il existe un dernier bit dont nous n'avons pas encore parlé, le bit **TMR0IP** du registre **INTCON2**. Ce bit permet de sélectionner le vecteur d'interruption utilisé par le Timer0. Si TMR0IP est à "1", on utilisera le vecteur de haute priorité, sinon ce sera celui de priorité basse.

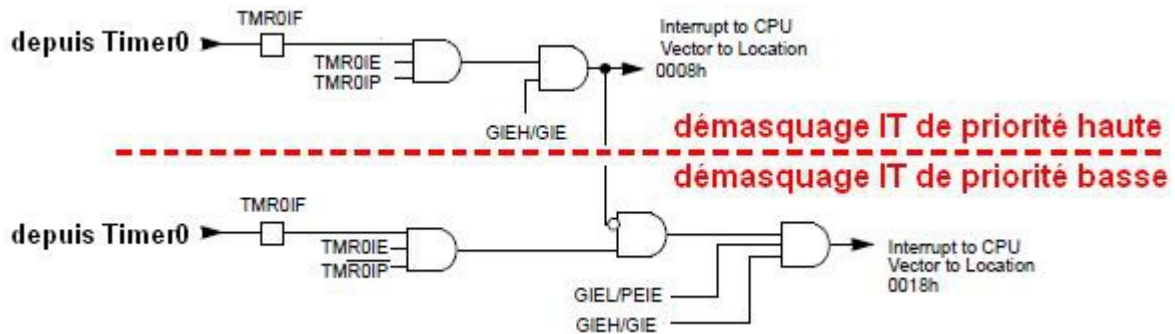


figure 7 : mécanismes de démasquage des interruptions pour le Timer0

Par exemple, imaginons que le Timer0 vienne de finir de compter jusqu'à 0xFFFF. Il envoie alors une demande d'IT au CPU. Cette demande sera "latché" (sauvé) dans le bit d'état **TMR0IF** du registre **INTCON**. Si on souhaite exécuter le code contenu dans le vecteur d'interruption de priorité haute (par exemple : **goto Timer0_ISR**), il nous faut avant cela avoir positionné les bits **TMR0IE**, **TMR0IP** et **GIEH** à "1" et alors le CPU sera sensible aux demandes d'IT du Timer0. Une fois le démasquage effectué, le CPU pourra aller exécuter le code contenu dans le vecteur d'IT de haute priorité puis le programme d'interruption correspondant à l'étiquette/adresse "Timer0_ISR". Une fois dans le programme d'IT, **il ne faudra pas oublier de remettre le flag TMR0IF à "0" !**



Vous disposez maintenant de tous les éléments pour mener à bien la dernière partie de l'exercice 2 ... à vous de jouer maintenant !



Bien, vous êtes maintenant prêts à attaquer le TP ! ... vous pouvez d'ailleurs dès à présent commencer à compléter les fichiers sources présents juste après cette partie.

Cependant, dans un soucis pédagogique, il vous est demandé un petit travail préparatoire. Il sera vérifié en début de séance et vous permettra d'anticiper une perte de temps non négligeable dans l'avancement des TP (nb ★ = difficulté).

1. (★) Pour notre MCU, donnez le temps d'exécution en "ns" de cette section de code ? Vous trouverez des détails sur ces instructions en ANNEXES.

```
mullw    2
movf     PRODL,W,A
bra      MULT
```

2. (★★) Proposez en assembleur un exemple d'implémentation de boucle durant approximativement 1s.
3. (★★) Quel est le plus petit pré-diviseur (prescale) utilisable pour configurer le timer0 afin d'obtenir une temporisation de 1s ?
4. (★★) Proposez en assembleur le code d'une fonction durant approximativement 1mn. La fonction se nommera **wait1mn()**.
5. (★★) Qu'est-ce qu'une interruption ?
6. (★★) Proposez en assembleur une configuration pour les registres T0CON, TMR0H et TMR0L du Timer0 sachant que nous souhaitons exécuter notre programme d'interruption toutes les ~0,5s
7. (★★) Dans le cas donné ci-dessus, proposez en assembleur une configuration des registres de contrôle pour la gestion des interruptions (INTCON, RCON et INTCON2). Nous souhaitons utiliser le vecteur d'interruption de priorité basse en "priority mode".
8. (★★) Peut-on générer une base de temps d'environ 10s avec le Timer0 ? Si oui, comment feriez-vous ?



Listing du programme *ex21.asm*

```

. *****
;
; @file : ex21.asm
; @brief :
; @author :
; last modification :
. *****
;
; *** Fichiers d'en-tête ***
#include <p18f4550.inc>

; *** Configuration bits - Fosc=48MHz  Tosc=1/Fosc=20,8ns  Tcy=4*Tosc=83,33ns ***
CONFIG WDT=OFF, LVP=OFF, PLLDIV=2, CPUDIV=OSC1_PLL2, FOSC=HSPLL_HS, ...

; *** définition des MACROS ***
TEMPO      EQU      ; *** à compléter ! ***

; *** Section de données - initialized data in ACCESS mode ***
idata_acs
CounterL    db      0      ; On force le compilateur à mettre nos variables en ACCESS RAM
CounterH    db      0      ; Allocation d'un octet en mémoire RAM et initialisation à "0"
CounterUH   db      0

; *** Mise en place du vecteur d'IT lié au Reset ***
org 0x0000      ; Adresse du vecteur d'IT
RESET_V:
goto MAIN      ; Vecteur d'interruption lié au Reset

; *** Section de code ***
code
*****
; *** FONCTION : Temporisation logicielle ***
*****
DELAY:
; *** Initialisation de la partie haute (CounterUH) de la valeur à décrémenter ***

; *** à compléter ! ***

DELAYLOOP:
; *** Décrémentation de la valeur à décompter ***

; *** à compléter ! ***
return

. *****
; *** Programme principal ***
. *****
MAIN:
; *** Initialisation de la broche n°0 du port B ***

; *** à compléter ! ***

; *** Faire toujours : Allumer LED >> appel fonction DELAY >> éteindre LED >> appel fonction DELAY ...

; *** à compléter ! ***
end

```




Listing du programme ex22.asm

```
. *****
;
; @file : ex22.asm
; @brief :
; @author :
; last modification :
; *****
;
; *** Fichiers d'en-tête ***
; #include <p18f4550.inc>
;
; *** Configuration bits - Fosc=48MHz Tosc=1/Fosc=20,8ns Tcy=4*Tosc=83,33ns ***
; CONFIG WDT=OFF, LVP=OFF, PLLDIV=2, CPUDIV=OSC1_PLL2, FOSC=HSPLL_HS, ...
;
; *** définition des MACROS ***
;
; COUNTH EQU ;*** à compléter ! ***
; COUNTL EQU ;*** à compléter ! ***
;
; *** Mise en place du vecteur d'IT lié au Reset ***
;
; org 0x0000 ; Adresse du vecteur d'IT
RESET_V:
; goto MAIN ; Vecteur d'interruption lié au Reset
;
; *** Mise en place du vecteur d'IT de priorité haute ***
;
; org 0x0008
HPRIO_V:
; goto TOGGLE_ISR ; Vecteur d'interruption de priorité haute
;
;
; *** section programme ***
;
; code
; *****
;
; *** ISR : routine d'interruption TOGGLE_ISR
; *** @brief : inverse le niveau logique sur la broche n°0 du portB
; *****
TOGGLE_ISR:
; *** Inversion du niveau logique sur la broche n°0 du port B ***
;
; *** à compléter ! ***
;
; *** Ré-initialiser le registre TMR0L pour le pré-chargement ***
;
; *** à compléter ! ***
;
; *** Mise à "0" du flag TMR0IF ***
;
; *** à compléter ! ***
;
; retfie
```

```
.*****  
;  
.*** Programme principal ***  
;  
.*****  
;
```



MAIN:

```
.*** Initialisation de la broche n°0 du port B ***  
;  
    .*** à compléter ! ***  
  
.*** Démasquage de l'interruption liée au Timer0 ***  
;  
.*** avec: priority mode, high priority vector et masquage global des IT  
  
    .*** à compléter ! ***  
  
.*** Initialisation puis démarrage du Timer0 ***  
;  
    .*** à compléter ! ***  
  
.*** démasquage global des IT ***  
  
    .*** à compléter ! ***  
  
goto $          ; branch forever  
  
end
```

Notez vos remarques :



1. Travail pratique (ex2 : partie 1) : temporisation logicielle

Dans cette première partie de l'exercice 2, votre cahier des charges est relativement simple. Vous devez configurer la broches n°0 du port B en sortie puis faire clignoter la LED connectée à celle-ci. Votre programme devra toujours faire ce qui suit : Allumer le LED connectée à RB0 puis appeler la fonction DELAY dont le temps d'exécution sera proche de 1s puis revenir au programme principal et éteindre la LED puis appeler la fonction DELAY puis ... etc

- ➔  Lire et compléter le listing du programme **ex21.asm**.
- ➔  Créer un projet **ex2** dans votre répertoire de travail (cf. **ANNEXE 1**)
- ➔ Configurer le port B
- ➔ Faire tout le temps ce qui suit ...
 - ➔ Allumer la LED connectée à RB0
 - ➔ Appeler la fonction DELAY qui dure approximativement 1s
 - ➔ Éteindre la LED connectée à RB0
 - ➔ Appeler la fonction DELAY

2. Travail pratique (ex2: partie 2) : gestion du Timer0 et de l'interruption associée

Dans cette seconde partie, nous vous demandons d'effectuer exactement le même traitement que précédemment, faire clignoter la LED avec une durée d'allumage proche de la seconde, mais en utilisant l'interruption rattachée au Timer0.

- ➔  Lire et compléter le listing du programme **ex22.asm**.
- ➔  Vous n'avez pas à créer un nouveau projet, vous devez travailler à partir du projet **ex2**. Retirez l'ancien fichier source du projet (**Remove File From Project - ex21.asm**) puis ajouter le nouveau (**Add Files to project – ex22.asm**). Vous trouverez ces commandes sous l'onglet "Project".
- ➔ Configurer le port B
- ➔ Configurer le Timer0 (cf. ANNEXE)
- ➔ Démasquer l'interruption liée au Timer0 (priority mode et vecteur d'IT de priorité haute)
- ➔ Écrire le programme d'interruption, sans oublier de **recharger le registre TMR0L** !



A vous de jouer maintenant !

Notez vos remarques :