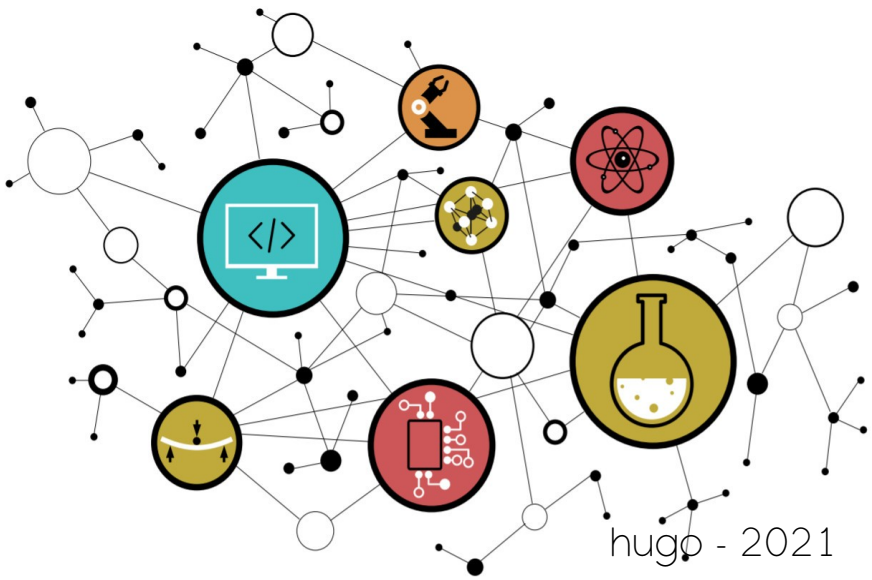
TRAVAUX PRATIQUES

MODULE DE COMPTAGE TIMER ET GESTION DES INTERRUPTIONS



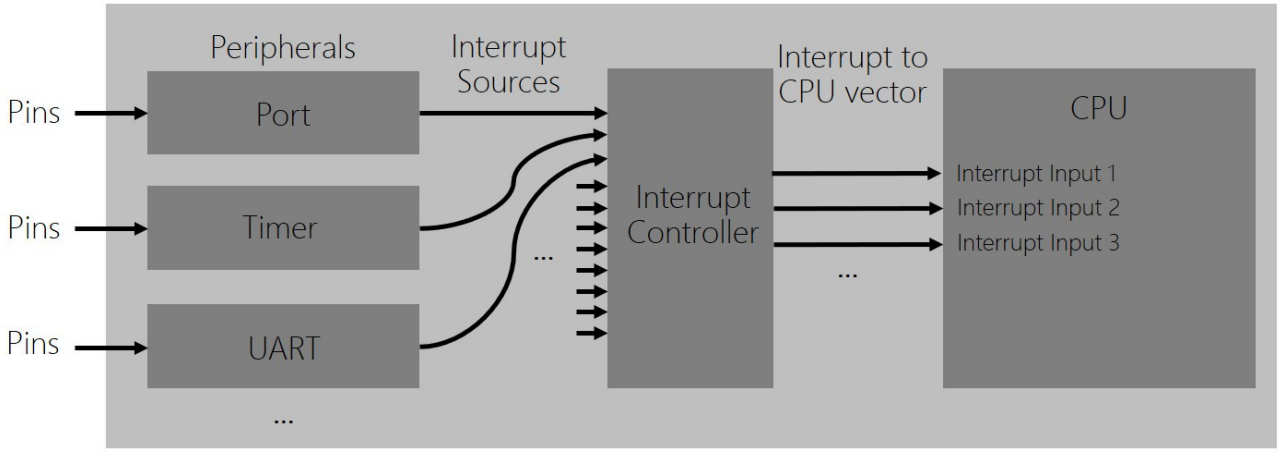
hugo - 2021

SOMMAIRE

MODULE DE COMPTAGE TIMER ET GESTION DES INTERRUPTIONS

3. MODULE DE COMPTAGE TIMER ET INTERRUPTIONS

3.1. Interruption matérielle



Le concept d'interruption matérielle est essentiel sur tout processeur conçu autour d'un CPU. Il s'agit de la capacité des fonctions périphériques à interrompre le CPU en cours de traitement pour lui affecter une nouvelle mission potentielle. Une interruption matérielle sera toujours rattachée à une fonction matérielle périphérique et correspond à l'occurrence d'un événement physique dans le système. Elle est toujours envoyée par un périphérique vers le CPU. Une fois configuré pour une mission dédiée, un périphérique est un composant indépendant et autonome dans le processeur lui-même. Chaque périphérique possède donc un rôle et une tâche spécifique dans le système (communiquer, convertir ou traiter).

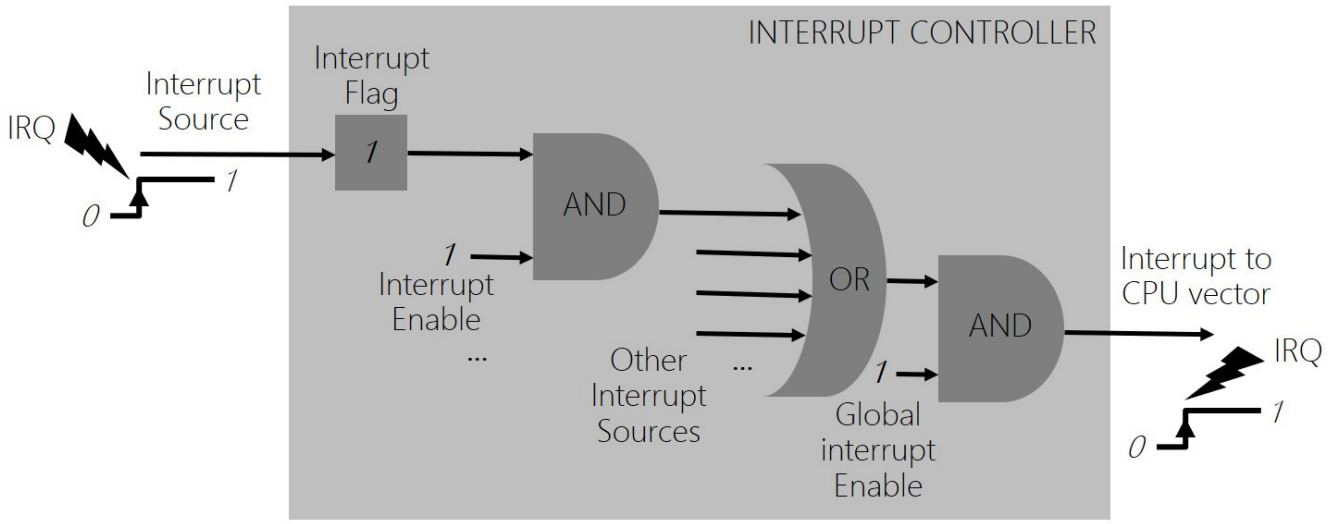
Un module périphérique Timer est par exemple conçu autour d'un compteur ou décompteur numérique. Sa mission est de compter à la place du CPU. Les modules UART, SPI, I2C ou USB sont par exemple des périphériques de communication permettant d'échanger de l'information avec l'extérieur du processeur. Communication de machine à machine. Lorsqu'un événement physique relatif à la tâche d'un périphérique se produit (fin de comptage pour un Timer, réception ou fin d'émission d'une donnée pour un UART, etc), celui-ci va tenter d'interrompre le CPU pour le prévenir de cet événement. Une interruption est alors envoyée par le périphérique vers le CPU. Une interruption est un simple signal physique électrique booléen tout ou rien. A la réception de l'interruption, le CPU sera soit en cours d'exécution d'une instruction (*fetch/decode/execute/writeback*), soit en veille (*inactif*). Pour que le CPU soit sensible et donc voit l'interruption, le développeur aura à configurer le contrôleur d'interruption du processeur. Nous allons découvrir tout ceci dans les pages qui suivent.

Une interruption est donc un signal physique partant d'un périphérique et arrivant (sous conditions) jusqu'au CPU. Les entrées d'interruption d'un CPU sont le plus souvent classées par niveau de priorité. Il est à noter qu'une ''Belle'' application, ne travaillera et ne réalisera des actions qu'au moment opportun, lorsqu'un traitement est strictement nécessaire. Le reste du temps, le système de supervision devra rester au repos (veille). La mise en veille CPU correspond à sa capacité à ne pas exécuter d'instruction (CPU inactif). Alors, seuls les périphériques restent à l'écoute des événements extérieurs (voire intérieurs) au système. Ils servent donc d'interfaces entre le système embarqué et son environnement physique extérieur.

3.2. Source et requête d'interruption IRQ

Dans la majorité des cas, il existe au moins autant de sources d'interruption que de périphériques dans le processeur. Une source d'interruption est un signal physique unidirectionnel (conducteur sur le die) allant d'un périphérique au contrôleur d'interruption. Une interruption (par abus de langage) ou requête d'interruption ou IRQ (Interrupt Request) correspond au passage d'un état logique inactif à actif sur une source d'interruption. Un périphérique envoie alors une requête au CPU et demande à interrompre son traitement en cours afin de lui affecter une nouvelle mission. Faut-il encore que le CPU y soit sensible !

3.3. Logique de démasquage d'interruption



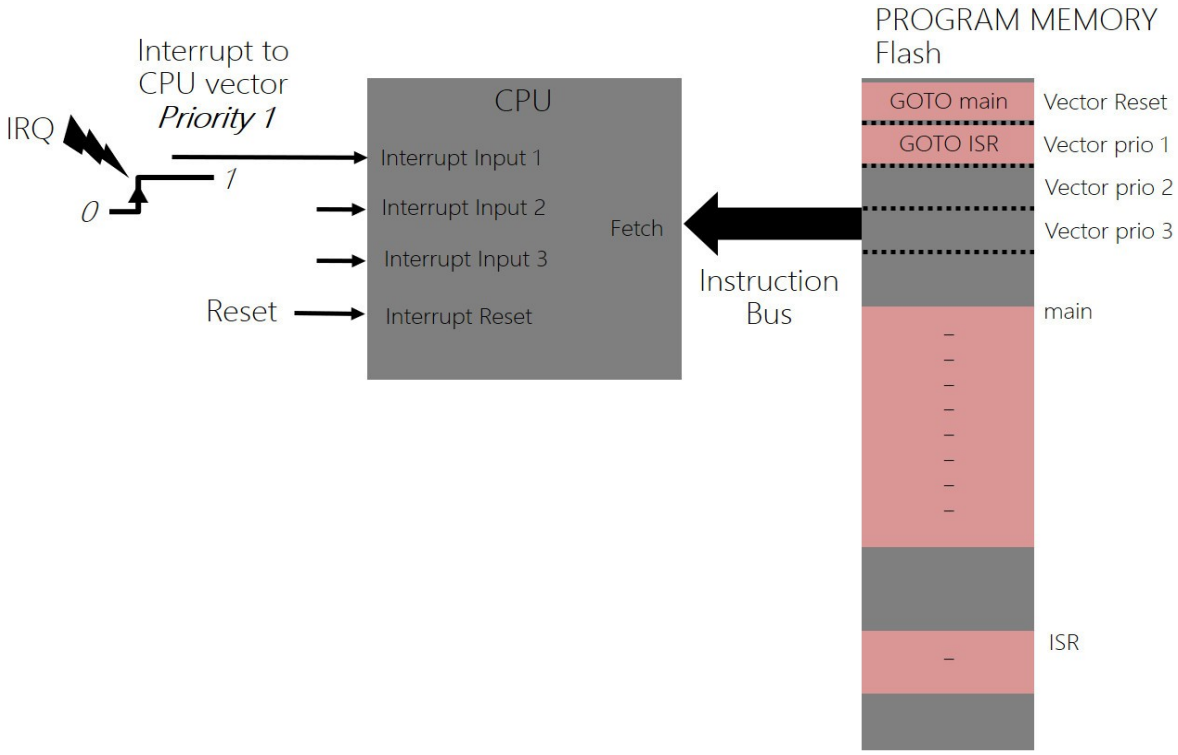
Il est à noter que par défaut à la mise sous tension, généralement le CPU n'est sensible à aucune source d'interruption, hormis celle du reset (bouton poussoir externe voire reset logiciel). Même si des périphériques devaient envoyer des IRQ (Interrupt Request), ils ne pourraient interrompre l'exécution du programme en cours. En effet, le développeur en charge du développement devra démasquer les sources d'interruption une à une en fonction des besoins spécifiques de l'application. Il s'agit d'une logique combinatoire de démasquage à réaliser afin de configurer le contrôleur d'interruption (cf. Interrupt Controller ci-dessus).

Comme pour tous les périphériques, cette configuration se fera par écriture dans des registres. Il y aura en général à minima la validation de la source d'interruption et la validation globale des interruptions pour l'ensemble du processeur. De même, les requêtes d'interruption ou IRQ seront mémorisées (cf. interrupt flag ci-dessus – simple bascule D) et auront à être acquittées explicitement par mise à zéro dans l'application par le développeur. Ces acquittements se feront dans les fonctions d'interruption (ou ISR) et seront présentés par la suite.

Si une IRQ parvient à traverser le contrôleur d'interruption, elle forcera le CPU à arrêter les traitements en cours et à exécuter un code spécifique (vecteur d'interruption). La commutation est matériellement câblée dans le CPU. Il est souvent associé un niveau de priorité (voire de sous priorités) à un vecteur d'interruption. Ceci permet à l'ingénieur de rendre plus au mois prioritaires des périphériques (Timer, UART, USB, Ethernet, etc) et donc les tâches à réaliser par l'application. Le problème se produira lorsque plusieurs périphériques enverront des requêtes simultanément.

3.4. Vecteur d'interruption

Un vecteur d'interruption est une ''petite'' zone en mémoire programme. Un vecteur d'interruption est donc chargé de mémoriser ''quelques'' instructions binaires. Ces instructions permettent une redirection vers la fonction d'interruption ou ISR (Interrupt Service/Software Routine). D'où le nom de vecteur. Les ISR sont quant à elles des fonctions logicielles accessibles depuis l'application et présentent dans le programme en cours de développement. Il est potentiellement possible d'avoir autant d'ISR que de vecteurs d'interruption dans une application. Rappelons que généralement il est associé un niveau de priorité à un vecteur d'interruption et donc aux ISR liées. Ces priorités sont également configurables par registres.



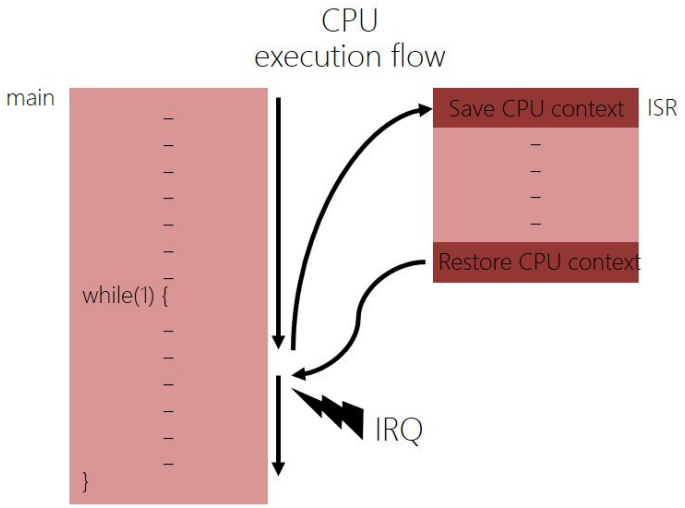
Attention, le schéma ci-dessus représente deux concepts radicalement différents (cf. version numérique du document) :

- Architecture matérielle (hardware) en gris pour les fonctions électroniques matérielles (CPU et mémoire programme seulement) et les flèches noires pour les bus et conducteurs physiques
- Micrologiciel (firmware) en rose correspondant au code binaire utile généré suite au processus de compilation et d'édition des liens sur ordinateur

Le firmware est donc mémorisé en mémoire programme non-volatile, souvent nommée mémoire flash sur MCU. Il s'agit de l'application logicielle embarquée dans le système matériel. Il s'agit de la mission du système embarqué répondant à un besoin. Sans système d'exploitation (scheduler), si aucun périphérique ne cherche à interrompre le CPU, alors celui-ci n'exécutera que du code de fonctions appelées depuis la fonction main. Dans l'exemple ci-dessus, dès que le CPU capte l'IRQ sur son entrée d'interruption de priorité 1, il commence à aller chercher puis exécuter le code présent dans le vecteur d'interruption de priorité 1 (simple instruction GOTO ISR). Puis par redirection le CPU ira exécuter le code de la fonction ISR associée. Dès que l'ISR est terminée, il reprend l'exécution de l'application exactement à l'instruction à laquelle il avait été interrompu par l'IRQ. Nous nommons ce phénomène commutation de contexte (sauvegarde et restauration). Il sera illustré sur PIC18 par la suite dans cet enseignement. De même, les vecteurs d'interruption sont généralement situés aux adresses les plus basses de la mémoire programme et sont sur certaines architectures translatables à d'autres adresses mémoire. L'ensemble des vecteurs d'interruption est nommée table des vecteurs d'interruption.

3.5. Fonction d'interruption ISR

Une ISR (Interrupt Service/Software Routine) ou fonction d'interruption est un fonction logicielle telle que vous avez pu en rencontrer en langage C dans toute application. A ceci près, qu'elle ne doit jamais être appelée de façon explicite depuis l'application. Il s'agit de programmation événementielle. Les ISR se réveilleront de façon asynchrone (non prédictible lorsqu'il s'agit de périphériques de communication) sur événement matériel en suivant la logique précédemment présentée (IRQ, démasquage puis vecteur d'interruption). Une ISR possédant le plus haut niveau de privilège d'exécution sur un processeur à CPU (MCU, AP, GPP, DSP, MPPA, etc), il faut toujours penser à passer le moins de temps possible à l'intérieur en déportant les traitements longs dans les fonctions appelées depuis le main (code applicatif). Sans système d'exploitation, utiliser alors des variables globales pour les échanges.



En respectant le code couleur précédemment utilisé (cf. version numérique pdf), les concepts maintenant présentés sont représentatif du logiciel développé par le développeur et du firmware embarqué. Sans système d'exploitation, les développements sont alors nommés *Baremetal,* soit nu directement sur le processeur sans système logiciel exploitant la machine (scheduler pour le CPU, gestionnaire mémoire par MMU/MPU pour la mémoire, drivers ou pilotes pour les périphériques, etc). L'application doit alors impérativement implémenter un while(1). Nous verrons en travaux pratiques comment développer une application en implémentant un *scheduler offline*.

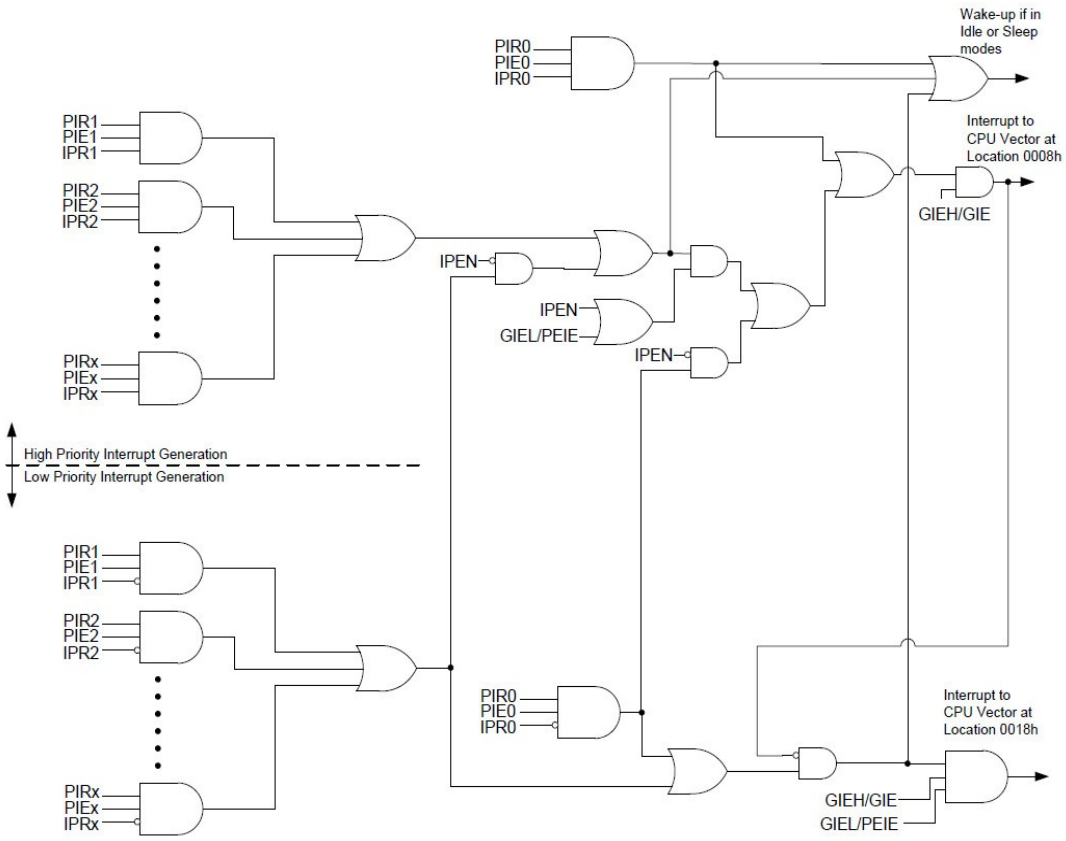
Les ISR étant spécifiques à une architecture, leur déclaration sera différente d'un processeur à un autre et donc d'une chaîne de compilation à une autre. Le plus souvent, un qualificateur de fonction lui est associé. Le compilateur a besoin de ce qualificateur afin d'implémenter les sauvegardes et restaurations de contexte en en-tête et pied de fonction. L'exemple ci-dessous illustre une déclaration d'ISR de priorité haute sur PIC18 en langage C sur toolchain XC8. Pour information, il y a seulement deux niveaux de priorité (haute et basse) et donc deux vecteurs d'interruption sur PIC18 (en plus de celui du reset).

```
void main (void)
{
   _
   while(1) {
      _
      _
   }
}
```

```
void __interrupt(high_priority) ISR (void)
{
   _
   _
   _
}
```

Vous constaterez que le code C ainsi que le schéma ci-dessus utilisent deux illustrations différentes pour présenter un même concept (texte vs graphique). L'une est une implémentation technologique en langage C et l'autre une représentation conceptuelle sous forme de dessin. La majorité des exercices demandés dans la trame de travaux pratiques seront représentés graphiquement de façon générique. Il sera à votre charge de réaliser les implémentations technologiques C ou assembleur sur PIC18 sous environnement de développement MPLABX et chaîne de compilation XC8.

3.6. Gestion des interruptions sur PIC18



Les MCU 8bits PIC18 de Microchip ont tous en commun le même CPU, les mêmes bus, et donc le même jeu d'instructions assembleur (ISA) ainsi que la même chaîne de compilation (XC8). Il existe néanmoins un très large choix de PIC18 différents au catalogue de Microchip. Il sont tous différenciés par leurs ressources mémoire (Flash et SRAM) ainsi que par leur jeu de périphériques (UART, SPI, I2C, USB, CAN, etc). Un PIC18 intègre en général un large jeu de périphériques et chaque périphérique possède une voire plusieurs sources d'interruption. Il existe donc un certain nombre de registres de configuration pour les interruptions. Nous pouvons classer tous ces registres 8bits de configuration des interruptions sur PIC18 en 4 sous familles :

- INTCON : *Configuration globale pour l'ensemble du système (bits GIEH, GIEL, IPEN, etc)*
- PIEx (x=0 à 7) : *Configuration des bits de validation ou démasquage (interrupt enable)*
- PIRx (x=0 à 7) : *Lecture et acquittement des bits d'interruption IRQ (interrupt flag)*
- IPRx (x=0 à 7) : *Configuration des bits de priorité (ISR priorité basse ou haute)*

De même, pour chaque périphérique, 3 bits seront alors à configurer (xxx dépend du périphérique à configurer). Ces bits correspondent à des champs de bits dans les registres précédemment cités.

- xxxIE : *Interrupt Enable (validation de l'interruption afin de rendre le CPU sensible à l'IRQ)*
- xxxIF : *Interrupt Flag (mémorisation de l'IRQ pour acquittement dans l'ISR)*
- xxxIP : *Interrupt Priority (configuration du vecteur d'interruption de priorité basse ou haute)*

L'exemple suivant présente une configuration d'interruption en assembleur pour le Timer0 :

```
; clear flag, enable and set low priority interrupt

BCF        PIR0, TMR0IF
BSF        PIE0, TMR0IE
BCF        IPR0, TMR0IP
```

```
; global interrupt enable for all MCU system

BSF        INTCON, IPEN
BSF        INTCON, GIEL
BSF        INTCON, GIEH
```

Program memory mapping of PIC18F27K40

Observons ci-dessous un extrait de datasheet présentant l'organisation de la mémoire programme d'un PIC18F27K40, soit le MCU utilisé en TP. Nous pouvons également constater les emplacements et tailles des 3 vecteurs d'interruption des MCU PIC18 :

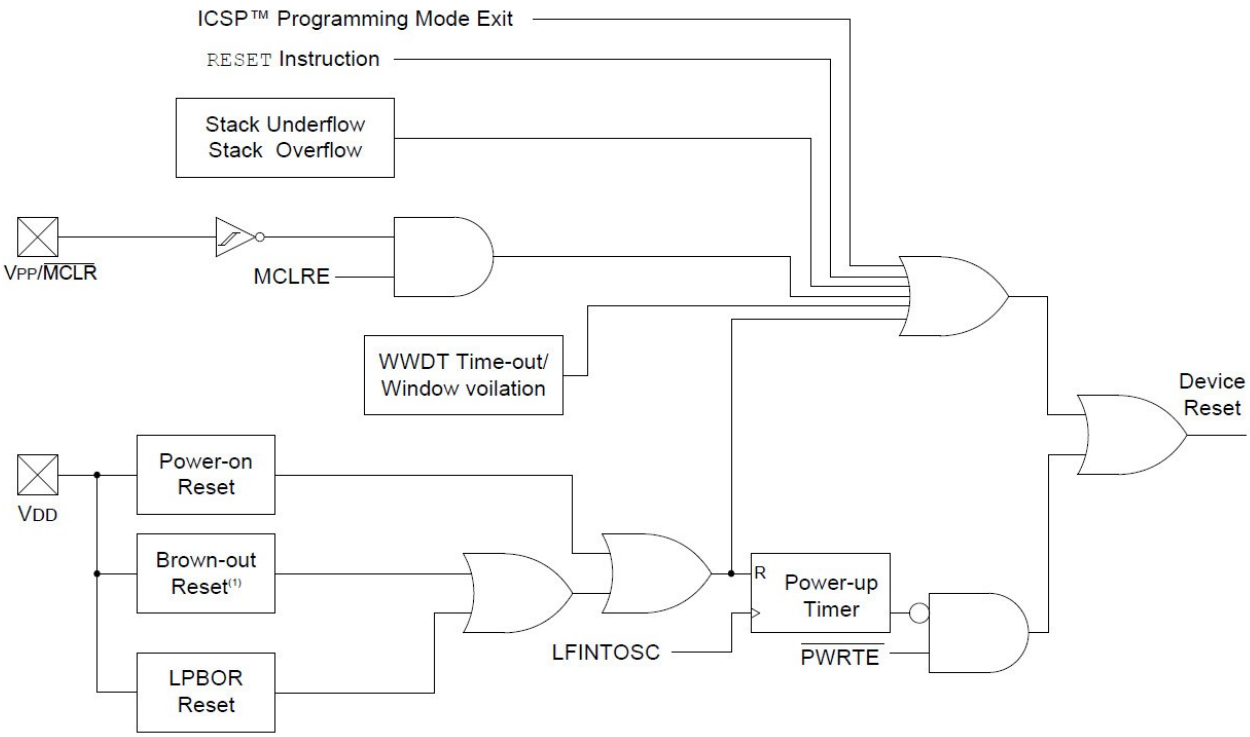
- Adresse 0x000000 : *reset*
- Adresse 0x000008 : *vecteur de priorité haute*
- Adresse 0x000018 : *vecteur de priorité basse*

Ces emplacements sont les mêmes pour tous les PIC18 de Microchip du marché. Chaque vecteur d'interruption ne propose que quelques octets de stockage. Pour information, un MCU PIC18F27K40 possède 128ko de mémoire programme Flash (ci-dessous 64KW soit 64KWords - 1Word=2octets sur PIC18) et 1ko de mémoire donnée non-volatile EEPROM. Ces deux technologies de mémoire sont non-volatiles et à ne pas confondre avec la mémoire donnée volatile de technologie SRAM (4Ko sur PIC18F27K40)

<table>
<tr><th rowspan="2">Address</th><th colspan="5">Device</th></tr>
<tr><th>PIC18(L)Fx4K40</th><th>PIC18(L)F25/45K40</th><th>PIC18(L)F65K40</th><th>PIC18(L)Fx6K40</th><th>PIC18(L)Fx7K40</th></tr>
<tr><td>Note 1</td><td colspan="5">Stack (31 Levels)</td></tr>
<tr><td>00 0000h</td><td colspan="5">Reset Vecor</td></tr>
<tr><td>...</td><td colspan="5">...</td></tr>
<tr><td>00 0008h</td><td colspan="5">Interrupt Vecor High</td></tr>
<tr><td>...</td><td colspan="5">...</td></tr>
<tr><td>00 0018h</td><td colspan="5">Interrupt Vecor Low</td></tr>
<tr><td>...</td><td colspan="5">...</td></tr>
<tr><td>00 001Ah to 00 3FFFh</td><td>Program Flash Memory (8 KW)</td><td>Program Flash Memory (16 KW)</td><td>Program Flash Memory (16 KW)</td><td>Program Flash Memory (32 KW)</td><td rowspan="3">Program Flash Memory (64 KW)</td></tr>
<tr><td>00 4000h to 00 7FFFh</td><td rowspan="4">Not Present(2)</td><td rowspan="4">Not Present(2)</td><td rowspan="4">Not Present(2)</td><td rowspan="3">Not Present(2)</td></tr>
<tr><td>00 8000h to 00 FFFFh</td></tr>
<tr><td>01 0000h to 01 FFFFh</td></tr>
<tr><td>02 0000h to 1F FFFFh</td><td>Not Present(2)</td></tr>
<tr><td>20 0000h to 20 000Fh</td><td colspan="5">User IDs (8 Words)(3)</td></tr>
<tr><td>20 0010h to 2F FFFFh</td><td colspan="5">Reserved</td></tr>
<tr><td>30 0000h to 30 000Bh</td><td colspan="5">Configuration Words (6 Words)(3)</td></tr>
<tr><td>30 000Ch to 30 FFFFh</td><td colspan="5">Reserved</td></tr>
<tr><td>31 0000h to 31 00FFh</td><td colspan="2">Data EEPROM (256 Bytes)</td><td colspan="3" rowspan="2">Data EEPROM (1024 Bytes)</td></tr>
<tr><td>31 0100h to 31 01FFh</td><td colspan="2">Unimplemented</td></tr>
<tr><td>30 000Ch to 30 FFFFh</td><td colspan="5">Reserved</td></tr>
<tr><td>3F FFFCh to 3F FFFDh</td><td colspan="5">Revision ID (1 Word)(4)</td></tr>
<tr><td>3F FFFEh to 3F FFFFh</td><td colspan="5">Device ID (1 Word)(4)</td></tr>
</table>

PIC18F27K40

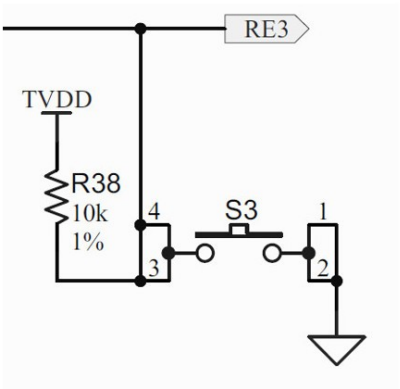
3.7. Gestion du RESET sur PIC18

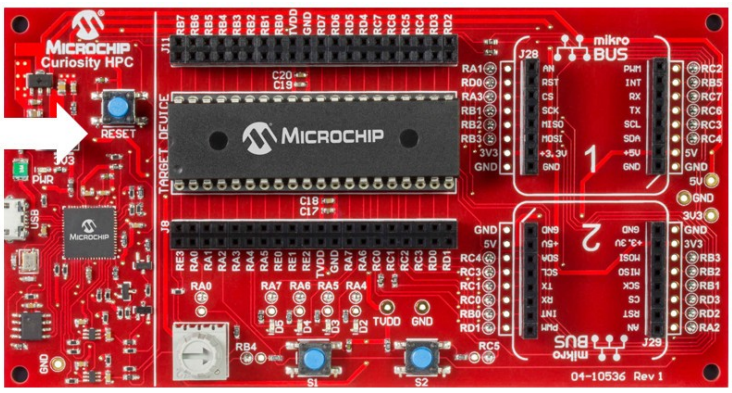


Nous pouvons observer ci-dessus la logique de démasquage du RESET sur PIC18F27K40. Par exemple, si un niveau logique bas est appliqué sur la broche externe RE3 nommée Vpp/$\overline{\text{MCLR}}$ (Master Clear) et si le bit de configuration MCLRE (MCLR Enable) est actif (fait par défaut à la mise sous tension), alors un RESET processeur est forcé.

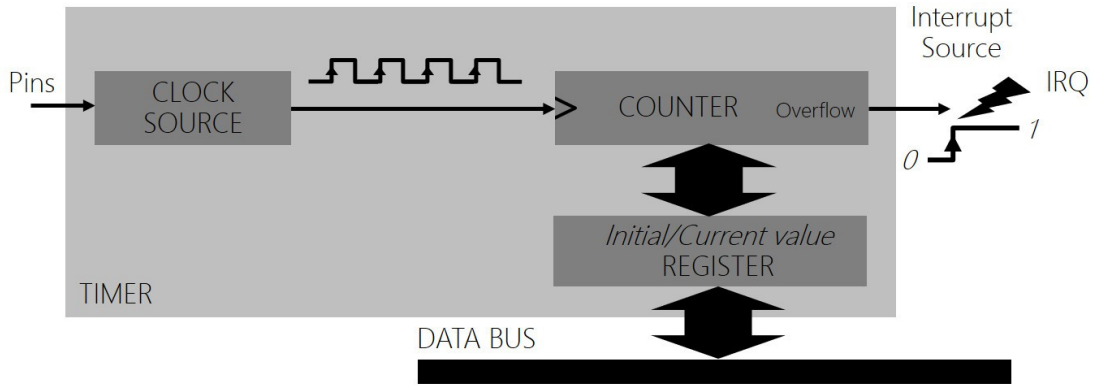
Le CPU exécute alors le code présent dans le vecteur d'interruption du RESET à l'adresse 0x000000 de la mémoire programme. En général, celui-ci appel la fonction main (en assembleur) ou la fonction de startup (en langage C) utilisée par défaut par la chaîne de compilation.

Observons par exemple ci-dessous le schéma électrique de câblage du bouton poussoir du RESET sur la carte Curiosity HPC de Microchip utilisée en TP. La broche Vpp/$\overline{\text{MCLR}}$ est la broche RE3 sur le boîtier du processeur. Il est courant sur un processeur qu'une broche ait différents noms et donc différents rôles possibles. Cela dépend de certaines fonctionnalités additionnelles associées à la plupart des broches. Chaque rôle associé à une broche aura à être configuré par le développeur en fonction des besoins spécifiques de chaque application. Ces aspects seront abordés dans la suite des exercices. Toute fonctionnalité matérielle du MCU est bien entendu documentée dans la documentation technique du processeur.





3.8. Module périphérique de comptage Timer



Un Timer sera toujours conçu autour d'un compteur ou décompteur numérique. Il est donc dédié aux opérations de comptage et le plus souvent à la gestion de bases de temps dans les applications (acquisitions de mesures physiques à intervalles de temps régulier, tâches périodiques, etc). Quelque soit la technologie du Timer, nous pouvons jouer sur 3 éléments afin de configurer une référence temporelle :

- Fréquence/Période de comptage (horloge de référence du Timer – fonction matérielle Clock Source ci-dessus)
- Valeur initiale de comptage (fonction matérielle Initial/Current Value Register ci-dessus)
- Nombre de bits utilisés par le compteur 8-16-32-64 bits avant débordement (fonction matérielle Counter et overflow ci-dessus)

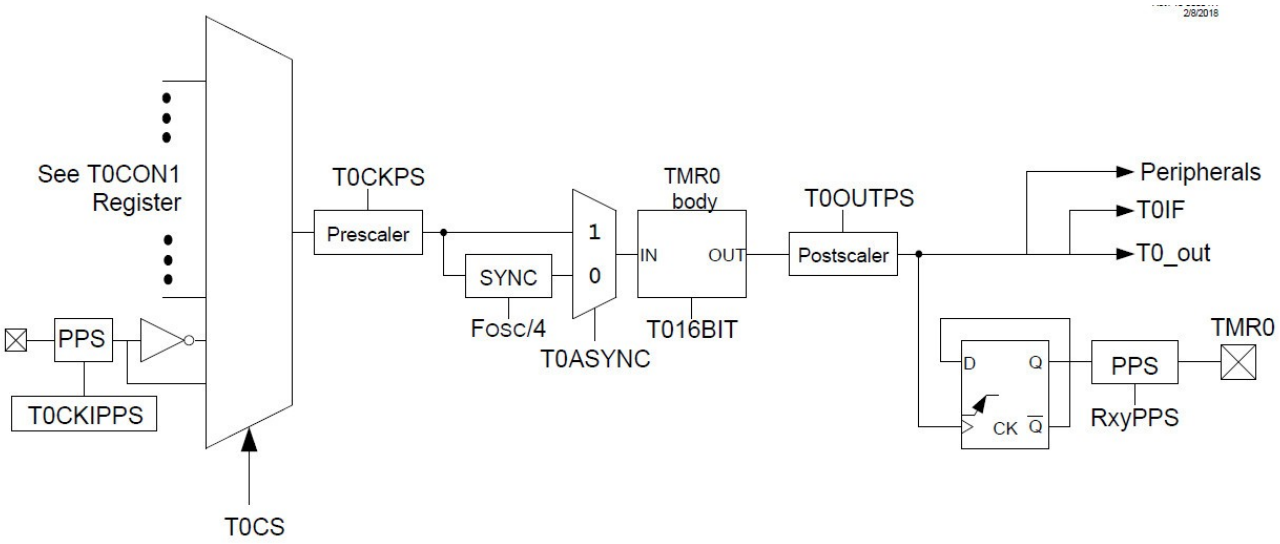
L'ensemble de ces trois éléments constituent en général un tout nommé Timer. Néanmoins, les Timers les plus élémentaires rencontrés sur processeur ne possèdent même pas de référence initiale de comptage ni d'horloge de référence configurable. Ils démarrent en partant de 0 à la mise sous tension et comptent à la fréquence de travail du CPU. Il sont souvent nommés TSC (Time Stamp Counter) et sont présents dans chaque CPU d'un grand nombre de processeurs du marché (Intel, AMD, TI C6000, ARM, etc). Ils sont souvent utilisés pour réaliser de la mesure de temps d'exécution de programme. Néanmoins, un Timer applicatif est plus riche en services matériels et fonctionnalités (comparateur numérique, rechargement automatique, module PWM, etc). Leur configuration peut sur certaines technologies être même relativement ardue.

Observons ci-dessous la configuration d'un Timer 8bits générique et calculons une valeur initiale de comptage à recharger après débordement afin d'obtenir une base de temps. Après comptage puis mise au niveau haut du bit de débordement (bit d'overflow), le Timer reprend le comptage à 0 par défaut (sans chargement de la valeur initiale). Par exemple sur 8bits ($255_{10} + 1_{10} = 1111\ 1111_2 + 1_2 = 1\ 0000\ 0000_2 = 256_{10}$ soit un résultat sur 9bits), le bit de débordement est le $9^{ième}$ bit actif. Exemple de calcul d'une valeur initiale de comptage sur Timer 8bits :

- Timer 8bis, comptage de 0 à 255 soit de 0x00 à 0xFF
- Hypothèse d'une horloge de référence de période 1ms
- Quelle serait la valeur initiale de comptage à charger au compteur afin d'obtenir un débordement après 100ms ?
- Réponse : *155 ou 0x9B (valeur à charger dans le registre initial de comptage)*
- Pourquoi : *Le Timer aura à compter 100 cycles de référence (100 x 1ms = 100ms) avant débordement au $256^{ième}$ cycle. Nous devons donc débuter le comptage à la valeur 155*

Sur un Timer, le signal à la source de l'interruption ou IRQ n'est autre que le bit de débordement (overflow flag). Ce signal électrique est alors envoyé au CPU (cf. chapitre Interruption).

3.9. Module périphérique Timer0 sur PIC18



Le MCU PIC18F27K40 utilisé à l'école intègre 4 Timers 16bits (Timer0, Timer1, Timer3 et Timer5) ainsi que 3 Timers 8bits (Timers2, Timer4 et Timer6). Il intègre donc 7 Timers pouvant être utilisés pour différents besoins dans une application.

Le schéma bloc ci-dessus ne présente que la structure interne du Timer0. Les autres Timers du PIC18F27K40 offrent d'autres services matériels complémentaires et sont donc différents. Le Timer0 est configurable en mode 16bits ou 8bits. Seul le mode 16bits est présenté ci-dessus et sera utilisé en TP. Petit exercice, entourer sur le schéma les fonctions matérielles suivantes :

- *Sous module compteur 16bits (COUNTER) ?*
- *Sous module de référence d'horloge (CLOCK SOURCE) ?*
- *Signal d'interruption IRQ envoyé au CPU par le Timer0 ?*

Ouvrir la datasheet des processeurs PIC18Fx7K40 et parcourir le chapitre 18 relatif au Timer0 (Timer0 Module) afin d'observer la configuration des registres. Le Timer0 possède deux registres 8bits de configuration (T0CON0 et T0CON1) et deux registres 8bits de chargement de la valeur initiale de comptage (TMR0L et TMR0H). Analysons le registre 8bits T0CON1 chargé de configurer l'horloge de référence.

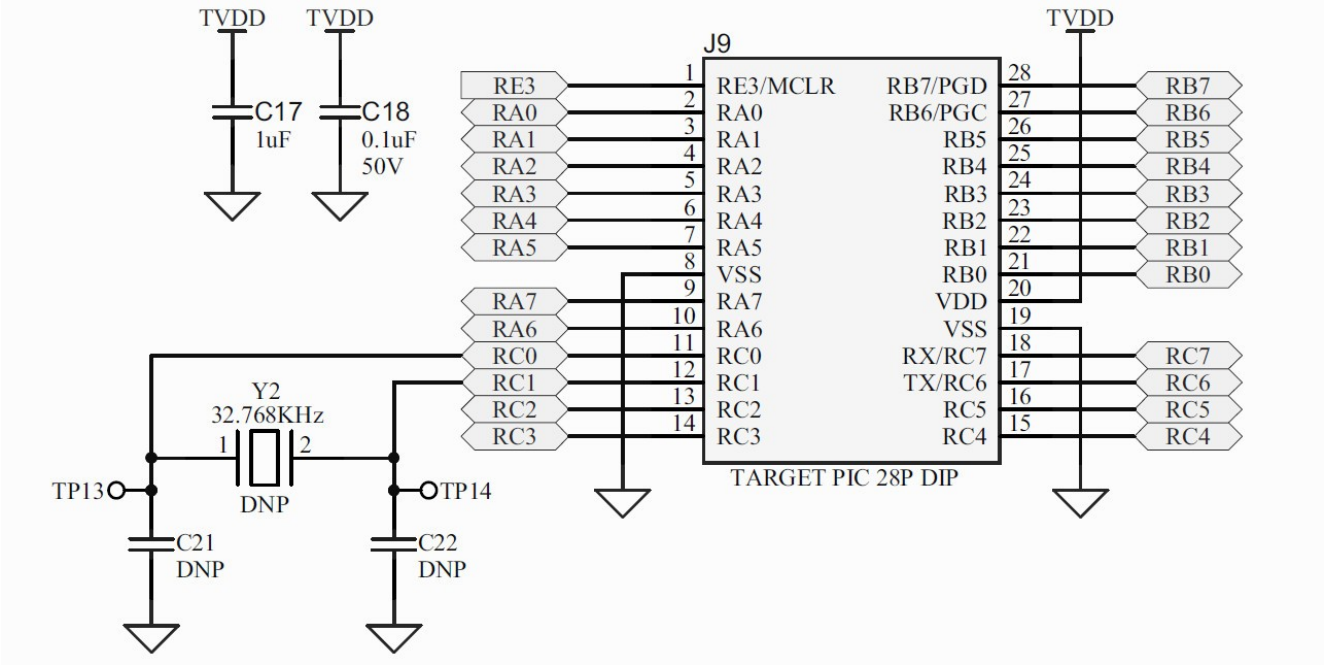
Nous pouvons observer sur le schéma ci-dessus que les champs T0CS (Timer0 Clock Source Select) et T0ASYNC (Timer0 Input Synchronization) permettent de piloter deux multiplexeurs d'aiguillage chargés de router la référence interne ou externe d'horloge. De même, le champ T0CKPS (Timer0 Clock Prescaler Select) permet de configurer un diviseur de fréquence (1:1, 1:2, 1:4, etc, 1:32768) avant d'entrer sur le compteur 16bits.

Name: T0CON1
Offset: 0xFD6

Timer0 Control Register 1

Bit	7	6	5	4	3	2	1	0
	T0CS[2:0]			T0ASYNC	T0CKPS[3:0]			
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	0	0	0

3.10. Configuration du Timer0 sur PIC18



Attention, la configuration proposée n'est pas celle demandée dans les TP. Elle sera donc à adapter. L'exemple suivant présente comment réaliser une base de temps très précise de 1 seconde. Il s'agirait d'une configuration de Timer si nous souhaitions réaliser une montre à quartz ou un réveil par exemple. En effet, les quartz 32.768KHz comme présenté ci-dessus sont typiquement ceux rencontrés dans les montres à Quartz. Les quartz possèdent une très faible dérive en fréquence par rapport à d'autres technologies de résonateurs (RC, MEMS, etc), typiquement proche de +/-10ppm (Particules Par Millions) soit +/- 0,00001% d'erreur et donc de précision.

Cet exercice pourrait d'ailleurs être réalisé sur la carte Curiosity HPC utilisée en TP à l'image de la capture du schéma électrique ci-dessus mais en utilisant néanmoins le Timer3 (broches utilisées par défaut sur la carte Curiosity HPC afin de relier le MCU au Quartz 32.768KHz).

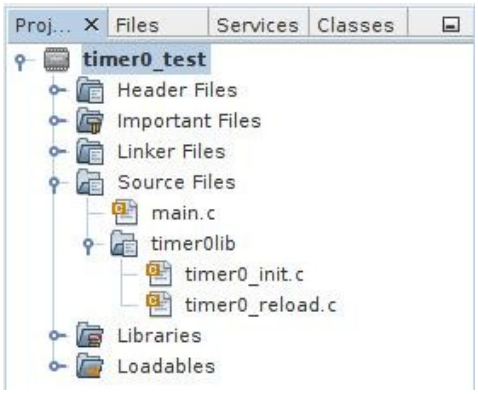
```
; Timer configuration :
; Timer disable, 16bits mode, post-scaler 1:1

MOVLW       0b00010000
MOVWF       T0CON0

; Timer configuration :
; Pins selection RC0 and RC1
; Extern 32.768KHz Quartz resonator
; No CPU synchro, pre-scaler 1:1

MOVLW       0b00110000
MOVWF       T0CON1

; Timer initialization :
; Initial decimal value 32767 (0x7FFF)
; Always write TMR0H before TMR0L

MOVLW       0x7F
MOVWF       TMR0H
MOVLW       0xFF
MOVWF       TMR0L
```

```
; Interrupt configuration :
; Clear flag, enable
; Set low priority interrupt

BCF         PIR0, TMR0IF
BSF         PIE0, TMR0IE
BCF         IPR0, TMR0IP

; Interrupt configuration :
; Global interrupt enable

BSF         INTCON, IPEN
BSF         INTCON, GIEL
BSF         INTCON, GIEH

; Timer enable
; The Timer start to count only now !

BSF         T0CON0, T0EN

; … This program generate an
interruption after 1s !
```

3.11. Configuration du Timer0 et interruption

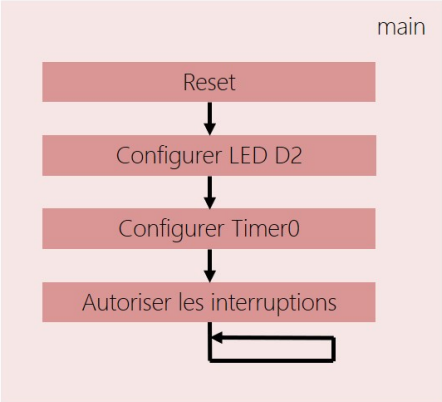
Nous allons nous intéresser au mécanisme de gestion des interruptions et de configuration de Timer. Nous nous efforcerons de gérer une base de temps en soulageant au maximum le CPU. Objectif, ne travailler qu'en cas d'absolue nécessité puis mettre le CPU en veille le reste du temps.

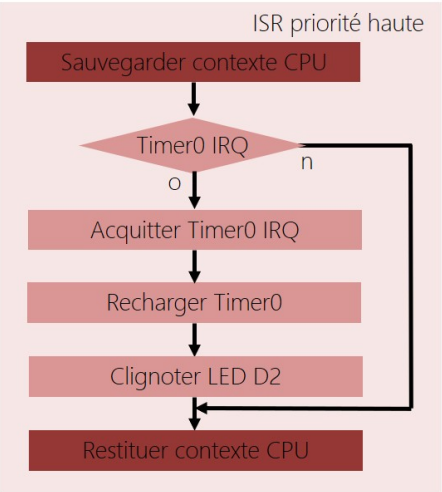
- Créer un projet MPLABX nommé *timer0_test* dans le répertoire *disco/bsp/timer0/test/pjct*. Inclure les fichiers *bsp/timer0/test/main.c, bsp/timer0/src/timer0_initc, bsp/timer0/src/timer0_reload.c* et s'assurer de la bonne compilation du projet. S'aider des tutoriels dans *mcu/tp/doc/tutos*



- Modifier les sources du projet afin de configurer le Timer0 pour qu'il puisse lever une interruption de priorité haute toutes les 20ms (horloge CPU Fosc à 64MHz donc Fosc/4 = 12MHz). *S'aider du fichier d'en-tête bsp/timer0/include/timer0.h et de la documentation (cartouche doxygen) de la fonction d'initialisation dans ce même header. Prendre l'habitude de parcourir les fichiers d'en-têtes durant la trame de TP et garder cette habitude pour le reste de votre vie dans le monde du développement logiciel. Les fichiers d'en-tête (headers) servent à générer les documentations techniques des bibliothèques logicielles. Ils précisent l'ensemble de l'API et services disponibles (Application Programming Interface).*

- Développer une application de test implémentant le diagramme de séquence suivant :





- Valider la base de temps à l'oscilloscope (période 40ms, 20ms LED allumée et 20ms LED éteinte)
- Estimer la charge CPU (durée de travail du CPU par unité de temps) ?

3.12. Analyse assembleur et commutation de contexte

- Compiler puis exécuter le programme en mode *debug* sur carte physique ou simulateur MPLABX
- Ouvrir et déplacer dans l'environnement graphique de l'IDE une fenêtre permettant d'observer le code binaire désassemblé
  - Window → PIC Memory Views → Program Memory
- Parcourir la totalité de la mémoire flash, identifier l'emplacement des fonctions (*main, timer0_init, timer0_reload et l'ISR*) et retranscrire en partie le code assembleur de l'ISR (seulement) ci-dessous. L'analyse peut commencer à être subtile, c'est normal.

Label	Disassembly Listing

- Comment se nomment les traitements observés en en-tête et pied d'ISR ? Quels sont leurs rôles ?

- Déclarer l'ISR comme une fonction C classique (sans qualificateur de fonction *interrupt*). Réitérer l'exercice d'analyse et observer le code binaire généré. Quels problèmes pourraient survenir dans d'autres applications si nous devions garder cette implémentation ?

3.13. Mise en veille du CPU

- Appeler l'instruction assembleur *sleep* dans la boucle infinie de la fonction principale. Configurer dans la fonction d'initialisation du Timer0 le champ IDLEN du registre CPUDOZE afin de valider le mode veille du CPU. Vérifier le bon fonctionnement de l'application.

```
asm(''sleep'');
```

- Expliquer à l'aide d'un chronogramme le fonctionnement de l'application. Quand le CPU exécute-t-il le code du main, le code de l'ISR et est-il en veille ?

- Estimer par le calcul la charge CPU (durée de travail du CPU par unité de temps) ?

- Quelles sont vos conclusions entre la fin de l'exercice 2 (avec délais logiciels) et maintenant (avec Timer, interruption et mise en veille), sachant que l'application réalisée d'un point de vu utilisateur est la même (clignotement d'une LED) ?

Une mise en veille correspond à une inactivité du CPU. Le CPU arrête donc sa machine séquentielle d'état (Fetch/Decode/Execute/WriteBack) et stoppe donc le processus d'exécution d'instructions. Seul un périphérique, et donc un événement physique interne ou externe, peut réveiller un CPU de sa veille. Dans cet exercice, il s'agit d'une IRQ envoyée par le Timer0 précédemment configuré. Mais par exemple sur votre ordinateur, en cas de mise en veille CPU, il s'agit le plus souvent d'une IRQ venant du contrôleur périphérique USB auquel est relié le clavier ou la souris.

Il est à noter qu'une mise en veille est toujours explicitement (directement ou indirectement) demandée depuis l'application ou le système d'exploitation. C'est donc au développeur d'estimer et de demander au système de se mettre au repos, comme il est de sa responsabilité de prévoir le mécanisme de réveil.


notes