Microprocesseur

TP3 : Interruptions

Cette séance permet une première approche des interruptions. Pour les mettre en pratique, nous allons réaliser un compteur à l'aide des afficheurs 7 segments.

Les étapes de cette séance seront :
- utilisation d'un formalisme plus haut niveau pour gérer l'afficheur 7 segments
- déclenchement d'une interruption sur le timer3
- gestion du balayage des afficheurs par interruption sur timer1
- gestion de l'avancée du compteur par interruption sur timer2
- quelques notions sur la gestion de l'énergie...

1) Appropriation du code de départ

La première étape du TP est de construire un projet à partir d'un programme abouti qui gère l'afficheur 7 segments en scrutation. Il s'agit de `led_counter_scrut.c` disponible sur la page du projet. Ce programme utilise un formalisme un peu plus abstrait pour permettre de gérer plus facilement le fait que les segments et afficheurs se situent sur des ports différents et des bits non contigus. En combinant les connaissances en C, la manipulation des afficheurs 7 segments du TP1 et la manipulation des timers du TP2, le programme de départ du TP3 est exigible à ce niveau de la formation. Il est fourni pour consacrer le temps de formation à des concepts plus utiles. Cependant, si ce programme contient des parties que vous ne comprenez pas, demandez de l'aide.

Créez un nouveau projet à partir du programme `compteur_scrut.c` fourni sur la page de ressources du TP.

Bien que ce programme soit déjà fonctionnel, nous prendrons le temps d'en détailler le contenu pour la culture. A la grande différence du TP1, des structures ont été mises en place pour pouvoir gérer les différents segments de façon indexée. La partie la plus évidente est constituée par les tableaux `SEG7_AFF_MASK` et `SEG7_SEG_MASK`, qui contiennent respectivement les masques de sélection des afficheurs et des segments. Comme ce sont des tableaux de masques, leur type de donnée est `const uint_32[]` (uint_32 étant le type *entier_non_signé_sur_32_bits* utilisé dans le microcontrôleur).
Les différentes sorties étant associées à des ports différents, accéder à leur masque par un index n'est pas suffisant, il faut également déterminer le registre d'affectation par ce même index. C'est le rôle des tableaux `SEG7_AFF_LAT` et `SEG7_SEG_LAT`. Pour les construire, nous utilisons des pointeurs vers les registres concernés, rangés dans un tableau. Un registre est de type `volatile uint_32`, un pointeur vers un registre est donc de type `volatile uint_32 *`. Les tableaux `SEG7_AFF_LAT` et `SEG7_SEG_LAT` sont alors de type `volatile uint_32 *[]`. Par exemple, `SEG7_SEG_LAT[0]` vaut `&LATG` (l'adresse du registre `LATG`, car le segment a est connecté au port G). Cela signifie que `*SEG7_SEG_LAT[0]` se comporte comme `LATG`. Cette technique permet donc d'oublier le bit et le port auquel est relié un segment. Il suffit de le manipuler grâce à son masque et à son registre accessibles dans les tableaux.

Ensuite, il faut convertir chaque nombre de la base 10 en représentation sur 7 segments. C'est le rôle du tableau `digit_7seg`, qui sert de table de correspondance. En effet, l'élément *i* de ce tableau contient la représentation de *i* sur 7 segments (le segment **a** est représenté par le bit 0, le segment **g** par le bit 6). Le tableau `seg_map` contient la valeur des segments pour chaque afficheur.

Le programme contient plusieurs fonctions déjà codées, notamment, l'initialisation des afficheurs 7 segments, des fonctions d'attente bloquantes pour chaque timer et une fonction de mise en sommeil du processeur (à vous de voir quelles fonction vous seront utiles).

Finalement, le programme effectue deux tâches :

- toutes les 2.5 ms (à 400 Hz), l'afficheur suivant est sélectionné, ce qui permet un balayage de tous les afficheurs à 100 Hz, et les segments sont allumés en fonction de `seg_map`.
- toutes les secondes (10 x 100ms) , le compteur est incrémenté, et `seg_map` est mis à jour en fonction de la nouvelle valeur du compteur.

Modifiez le programme pour ajouter un heartbeat sur la LED0 (Port A, bit 0) à partir du timer3 (déjà configuré). Utilisez d'abord la méthode simpliste :

- `wait_timer3()`
- inverser la LED 0

Cette méthode perturbe le fonctionnement de tout le programme car la fonction `wait_timer3()` est bloquante.

Utilisez une deuxième méthode basé sur un test non bloquant du bit 14 de IFS0 (au lieu d'une boucle bloquante), à la manière des tests sur les timers 1 et 2 (bits 4 et 9 de IFS0).

Que se passe-t-il si on essaye de mettre le processeur en sommeil ?

2) Configuration d'une interruption

Pour rappel, une interruption est une fonction qui est appelée par un évènement matériel. Elle bloque donc le programme en cours pour lui *voler* du temps de processeur afin de s'éxécuter. Lorsque la fonction se termine, l'état du processeur est restauré, et le programme continue sans se rendre compte qu'il n'a pas fonctionné pendant un certain temps. Plusieurs fonctions d'interruptions peuvent coexister sur un système, le mécanisme lié à chaque interruption est appelé **vecteur**. Etant donné qu'une interruption empêche l'avancée du programme principal, elle ne doit jamais faire d'attente bloquante.

Retour au TP : Pour mettre en œuvre une fonction d'interruption, il faut la déclarer avec une syntaxe spécifique et configurer le matériel pour qu'elle soit appelée suite à un evênement spécifique. A ce stade, il faut faire la différence entre le numéro de vecteur (la fonction à éxécuter) et le numéro d'IRQ (requête) qui identifie la raison du déclenchement. En effet, certains evènements sont connectés sur le même vecteur.

Ecriture d'une fonction d'interruption :
La fonction ne doit recevoir aucun argument et n'en renvoyer aucun (le matériel n'envoie pas d'arguments et n'en récupère pas non plus). Il faut également configurer le vecteur auquel la fonction sera associée. La syntaxe est la suivante :

```
void __attribute__((interrupt(ipl1soft), vector(numvect))) nom ( void ) {
…
}
```

- le mot-clé `ipl1soft` désigne la priorité et le mode de sauvegarde/restauration de l'état du processeur. Sa connaissance approfondie dépasse les objectifs de cet enseignement. Il suffit simplement d'utiliser ce paramètre avec la bonne priorité pour que l'interruption fonctionne (`ipl1soft` pour le premier niveau, `ipl7soft` pour le niveau le plus élevé). Ce niveau DOIT être cohérent avec la configuration des interruptions.
- Le mot-clé `numvect` désigne le numéro de vecteur à utiliser. L'ensemble des valeurs de vecteurs pour chaque source d'interruption est donnée en pages 66 et 67 de la datasheet. La liste des constantes est accessible aux lignes 18490 à 18534 du fichier p32mx370f512l.h qui fourni l'ensemble des définitions du microcontrôleur.
- Le `nom` de la fonction d'interruption n'a pas d'importance mais doit être unique (et de préférence explicite). La fonction sera appelée par son adresse mémoire au niveau matériel. Le compilateur se chargera de la placer au bon endroit en fonction du numéro de vecteur qui a été configuré.

Configuration d'une fonction d'interruption :
Pour que la fonction d'interruption puisse se déclencher suite aux évènements matériels, il faut également configurer certains registres :

- INTCON : il s'agit du registre de configuration des interruptions. Il est décrit page 70 de la datasheet. Pour un bon fonctionnement, il faut activer le mode *multivector*. SSO est uniquement utile en mode monovecteur, et nous n'utilisons pas les sources d'interruptions externes.
- Les registres IPCx contiennent la priorité de chaque vecteur d'interruption. Ils sont décrits page 73, mais l'affectation de chaque vecteur est réellement documentée pages 68 et 69. Les priorités de vecteurs sont organisées par ordre croissant à raison de 4 par registre. Pour l'instant, n'importe quelle priorité non nulle fera l'affaire (une priorité nulle signifie que l'interruption ne doit pas se déclencher).
- Les registres IECx permettent de décider quels évènements produiront des demandes d'interruptions. Les autorisations sont organisées par ordre croissant du numéro d'IRQ (et non de vecteur). Donc les IRQs 0 à 31 se situent sur les bits du registre IEC0, les 32 suivantes sur IEC1, *etc*. Les numéros d'IRQ associés à chaque evènement sont également disponibles pages 66 et 67.
- *IE* : un bit spécifique du processeur doit également être mis à '1'. La seul façon de le faire est de passer par l'instruction assembleur *ei*. Cela se fait grâce à la ligne suivante :

```
__asm__ ("ei");
```

A ce niveau, et si vous n'avez pas oublié d'étape, votre/vos interruption(s) peuvent se déclencher, et ne manqueront pas de le faire dès que leur *flag* passera à '1'. Ce *flag* est le bit de déclenchement associé à chaque IRQ dans les registres IFSx. Par exemple, et pour rappel, le bit 4 de IFS0 est associé au timer1, à chaque fois que le timer 1 déborde, ce bit passera donc à '1'. L'interruption se déclenchera **TANT QUE** ce *flag* sera à '1', il faudra donc le remettre à '0' pour eviter de rester bloqué dans une interruption.

Récupérez les élements syntaxiques de base dans le fichier `empty_skeleton_it.c`. Modifiez le programme du compteur pour déclencher une interruption sur le timer3. Allumez la LED1 dans l'interruption (et nulle part ailleurs) pour vérifier qu'elle se déclenche.

Une fois que vous aurez démontré que votre interruption se déclenche, modifiez-la pour qu'elle fasse clignoter la LED1. Pour cela, il suffit de changer l'état de La LED1 à chaque déclenchement de l'interruption.

Cette source d'interruption ne sera plus utilisée pour la suite du TP, il ne s'agissait que d'une mise en bouche...

3) Retour au compteur...

Ecrivez et configurez une fonction d'interruption pour qu'elle se déclenche sur le timer1. Cette fonction doit assurer le balayage sur l'afficheur 7 segments. Ce qui signifie :

- incrémenter l'indice désignant l'afficheur sélectionné
- mettre à jour les sorties vers les anodes
- mettre à jour les valeurs des segments en fonction de `seg_map` (ou d'une structure équivalente que vous aurez définie)
- probablement une ou deux autres bricoles qu'il vous faudra trouver (en fonction de vos choix d'implémentation)

Si cette étape fonctionne, la boucle infinie dans la fonction `main()`, peut se contenter de mettre à jour `seg_map` à chaque seconde. Le balayage est assuré par la fonction d'interruption, il n'y a plus à s'en soucier à l'avenir.

4) Encore un peu plus loin...

A ce niveau, il ne reste qu'à incrémenter le compteur et mettre à jour `seg_map` dans une interruption sur le timer2.

Vous remarquerez alors qu'il ne reste que la fonction *heartbeat* dans la boucle infinie. Modifiez votre code pour revenir à l'algorithme basique du début basé sur la fonction d'attente liée au timer3. Il n'y a plus d'interférences entre l'attente et le fonctionnement du compteur, c'est beau la vie !

Que se passe-t-il si on met le processeur en sommeil ? Quel est l'intérêt ?