2 – GPIO

Objectifs : *gestion de périphériques (GPIO/LEDs), mise en œuvre d'interruptions simples (non vectorisées), programmation bas niveau à partir de code C.*

PREPARATION : LE CODE DES FONCTIONS `GPIO_Init, GPIO_Read, GPIO_Write` DE LA PARTIE **1.2** DEVRA ETRE PREPARE ET SERA VERIFIE AU DEBUT DE LA SEANCE DE TP.
REALISATION : LES PARTIES **1.2**, **1.3** et **1.4** DOIVENT ETRE VALIDEES PAR L'ENCADRANT AU COURS DE LA SEANCE.

Les interruptions matérielles constituent un moyen efficace pour les périphériques d'indiquer au processeur principal qu'il doit réagir (en temps réel) à un évènement (asynchrone). Par exemple, l'appui d'une touche sur un clavier génère une interruption auquel le processeur peut réagir par l'affichage d'un caractère. Il épargne ainsi au CPU le besoin d'interroger périodiquement les périphériques pour en obtenir des informations, en fournissant un moyen de l'interrompre à n'importe quel moment pour qu'il exécute une tâche particulière. De cette façon, le processeur n'arrête l'exécution d'un programme utilisateur qu'en cas de besoin.

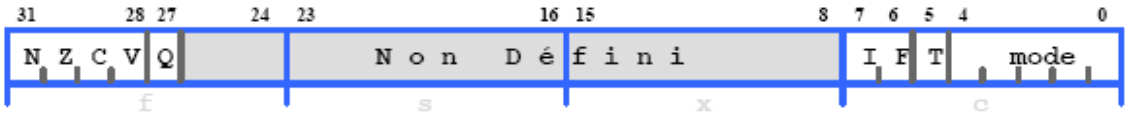
Pour réaliser une IRQ, le processeur dispose de lignes d'entrée-sortie spécifiques qui le relient aux différents périphériques susceptibles de déclencher une interruption. Le bloc diagramme de cette organisation pour la carte LPC2106 vous est donné page18 du User Manual disponible à l'adresse http://elec.polytech.unice.fr/~bilavarn/.

La carte étant composée de plusieurs périphériques, chacun est une source d'interruption potentielle. Pour éviter les conflits, les requêtes d'interruption sont gérées par un composant spécifique appelé le contrôleur d'interruption (Vectorized Interrupt Controller, VIC). Dans un premier temps, nous allons nous intéresser au cas des interruptions simples (non vectorisées, i.e. une seule source d'interruption possible, un seul sous programme exécuté en cas d'interruption).

Le principe de fonctionnement d'une interruption sur processeur ARM est le suivant :

- le périphérique génère une IRQ (Interrupt ReQuest).
- lorsque le processeur la reçoit et si l'interruption a été autorisée (I=0, bit 7 du registre d'état), le processeur :
  - passe en mode IRQ
  - sauvegarde l'adresse de l'instruction suivante dans R14 (LR)
  - sauvegarde le registre d'état (CPSR)
  - désactive IRQ (I=1)
  - charge R15 (PC) avec le vecteur d'interruption (placé à l'adresse 0x18)

Rappel : le registre d'état contient les indicateurs suivants :



Lorsqu'il a terminé de traiter l'interruption, il rétablit le contexte d'exécution avant interruption (CPSR), autorise à nouveau les interruptions (I=0) et reprend le programme interrompu à l'instruction sauvegardée dans R14.

1.1 Description du TP

L'objectif de cette séance est de mettre en œuvre une interruption simple (non vectorisée), sous la forme du clignotement d'une LED que l'on peut arrêter/remettre en route par une interruption générée par un switch de la carte (SW4). Dans un premier temps, nous allons écrire un programme C permettant de faire clignoter les LEDs. Puis nous ajouterons une interruption générée par le bouton SW4 qui arrêtera le clignotement.

Configuration du projet en C :

- Pour obtenir du code exécutable sur la carte à partir de code C, vous devez configurer votre projet IAR comme expliqué au TP1. Par la suite, nous utiliserons l'émulateur Seehau pour tester le code exécutable produit et vérifier ainsi le bon fonctionnement sur la carte (ou trouver les erreurs).
- D'autre part, les fichiers prototypes (.h) doivent être placés à la racine du projet. Ils n'ont pas besoin d'être ajoutés explicitement. En particulier, le fichier prototype `LPC210X.h` doit être placé à la racine du projet. Ce fichier contient des alias pour les adresses des registres de tous les périphériques disponibles sur la carte LPC2106 (ex : `GPIO_IOPIN → 0xE0028000`). Utilisez ces alias vous permettra d'éviter des erreurs de saisie des adresses registre. ***Ce fichier ainsi que la description détaillée des registres dans le manuel utilisateur vous seront indispensables pour la compréhension du fonctionnement des périphériques et des interruptions associées.***

1.2 Fonctions de gestion du GPIO

Reportez-vous au manuel utilisateur pour la description exhaustive du fonctionnement du GPIO. Vous devrez utiliser les alias définis dans le fichier prototype `LPC210x.h` qu'il faudra inclure dans chaque fichier C qui l'utilise.

Dans un fichier `gpio.c`, écrivez trois fonctions d'accès au port parallèle définies comme suit :

- `void GPIO_Init(int config)` : cette fonction permet de configurer le port parallèle au travers du registre `GPIO_IODIR`. Elle permettra d'initialiser le GPIO de façon à pouvoir utiliser les LEDs voulues (LED n° 1 sur port P0.8, LED n° 8 sur port P0.15). Exemple : une valeur de `config=0x00004700` configure les ports `P.0.8, P.0.9, P.0.10 et P.0.14` en sortie pour utilisation des LEDs 1, 2, 3 et 7.
- `short GPIO_Read()` : retourne la valeur du registre `GPIO_IOPIN`. ***Attention : la documentation précise qu'une LED est allumée si la sortie du port correspondant est à 0.***
- `void GPIO_Write(int mot)` : écrit l'état correspondant à `mot` dans le GPIO (en utilisant les registres `GPIO_IOSET` et `GPIO_IOCLR`). Exemple : l'appel à la fonction `GPIO_Write(0x00004700)`doit allumer les LEDs 1,2,3, 7 en mettant le registre `GPIO_IOPIN` à la valeur `0xXXXXB8XX`.

Ces trois fonctions constituent nos fonctions de base pour manipuler le GPIO, et en particulier les LEDs qui y sont connectées.

Créer ensuite un fichier `main.c`, dans lequel vous écrirez un petit programme permettant de tester ces trois fonctions en faisant clignoter les LEDs 1 à 4. Inclure le code des fichiers `gpio.c` et `main.c` dans vos compte-rendu avec des explications.

1.3 Fonction de gestion des LEDs

Créer un fichier `led.c` dans lequel vous écrirez les deux fonctions suivantes (qui devront bien sur réutiliser les fonctions précédentes) :

- **`void`** `led_Init(int led_config)` : cette fonction devra initialiser le GPIO de façon a pouvoir utiliser les LEDs voulues. Par exemple `led_Init(0x00004700)` configure les LEDs 1, 2, 3 et 7.
- **`void`** `led_Blink()`: cette fonction sera appelée dans une boucle infinie depuis la fonction *main*, et devra successivement allumer ou éteindre les LEDs configurées par `led_Init`. Il faudra prévoir une temporisation dans la boucle infinie pour le clignotement, qui ne sera pas visible autrement. Vous pouvez utiliser une temporisation par une simple boucle for (e.g. `for (i=0; i<300000; i++){}`).

Modifiez votre programme principal `main()` de manière à tester ces deux fonctions par un clignotement infini de la LED n° 1. Inclure le code des fichiers `led.c` et `main.c` dans vos comptes-rendus avec des explications.

- Modifier ensuite le programme principal pour que le clignotement s'exécute en fonction de la valeur d'une variable globale `run`. (si `run=1` → clignotement, si `run=0` → pas de clignotement).
- Par la suite, la valeur de cette variable sera modifiée par une routine d'interruption (toujours dans le programme principal) qui ira modifier la valeur de cette variable pour contrôler le clignotement. Ce sera le rôle de la fonction :
  ```
  __irq __arm void IRQ_Handler(void) { }
  ```
  que vous rajouterez dans le `main()` et ne contiendra rien pour l'instant.

Vous mettrez une copie de vos listings commentés à ce stade dans votre compte-rendu.

1.4 Utilisation d'une interruption extérieure (EXTINT2)

Le principe de fonctionnement de l'interruption dans notre programme est le suivant : lorsqu'une interruption se produit et qu'elle a été autorisée, le processeur interrompt son traitement en cours et exécute une fonction prédéfinie appelée routine de gestion des interruptions. Il existe deux types d'interruptions : les interruptions standards (IRQ) et les interruptions rapides (FIQ). Dans notre cas nous utiliserons les interruptions standards (IRQ), la routine de gestion des interruptions correspondante s'appelle `IRQ_Handler`.

Configuration du projet :
Complétez la fonction `IRQ_Handler(void)` dans le `main()` (voir **1.3**).
Ajoutez dans le `main()` une fonction :

- `void isr_Init(void)`

Vérifier que le programme se compile correctement à ce stade.

a) Pour pouvoir mettre en œuvre notre interruption, il faut ensuite compléter la fonction `isr_Init` qui a la charge de configurer le switch d'entrée SW4, qui servira à générer l'interruption, et le *contrôleur d'interruption* (se référer au manuel utilisateur, rubrique VIC et VIC usage note). Les étapes à suivre sont les suivantes :

- Désactiver les interruptions (fonction `__disable_interrupt()`). Vous devez inclure le fichier prototype `intrinsics.h` pour utiliser cette fonction.

- Chercher dans le manuel utilisateur la configuration du registre `PINSEL0` pour que le port P0.15 (relié à SW4) génère une interruption `EINT2`.
- Sélectionner les interruptions de type `IRQ` (registre `VICIntSelect`).
- Activer les interruptions en fonction de la source (voir table 44 et registre `VICIntEnable`).
- Définition de l'adresse de la routine de gestion des interruptions (registre `VICDefVectAddr`), qui est dans notre cas `IRQ_Handler`.
- Autoriser les interruptions (fonction `__enable_interrupt()`). Cette fonction est présente dans le fichier prototype `intrinsics.h`.

b) Ainsi configuré, notre programme doit être capable de générer une interruption. Vérifier que le programme se compile correctement à ce stade. Vous pouvez également vérifier le fonctionnement de l'interruption en plaçant un point d'arrêt sur la fonction `IRQ_Handler` et en vérifiant que l'exécution s'y arrête lorsque l'on appuie sur SW4.

c) Pour que l'interruption arrête le clignotement maintenant, il faut indiquer à la routine `IRQ_Handler` qu'elle doit simplement modifier la valeur de la variable globale `run` (`0` ou `1`) pour arrêter/reprendre le clignotement.

Ajouter le code nécessaire, compiler et vérifier cette fois que l'interruption arrête le clignotement.

d) Enfin, vous remarquerez que l'interruption ne fonctionne qu'une seule fois. Il faut en effet réinitialiser l'indicateur d'interruption `EINT2` afin de pouvoir en générer à nouveau. Rajouter le code nécessaire, tester votre programme et inclure le code source de vos programmes dans vos comptes-rendus, avec un petit paragraphe d'explication du fonctionnement des interruptions dans votre programme.