Microcontrôleur 32 bits / STM32
Mineure Numérique
TP2 : Liaison SPI - Réseau d'interruptions

Y. Bornat J. Saussereau

April 2, 2025

1 Introduction

Dans ce TP, nous allons chercher à minimiser l'utilisation du processeur. C'est en effet le composant le plus polyvalent, mais le moins efficace énergétiquement. Plus on utilise les modules matériels plus on peut (au choix):

- Faire de calculs pour un même microcontrôleur (on optimise le budget économique et énregétique).
- Baisser la fréquence et donc la consommation.
- Passer du temps en mode sommeil, et donc réduire la consommation.
- Prendre des composants moins performants et donc moins coûteux (et accessoirement moins gourmants en énergie, ce dont on ne se privera pas de mettre en avant).

Bref, quelques soient les priorités du projet, on investit du temps de développement mais on y gagne sur tous les tableaux.

1.1 Rappel (rapide) sur les Interruptions

Une interruption est une fonction appelée par le matériel. Mais comme le matériel est lui-même piloté par le logiciel, les interruption font partie intégrante de l'architecture logicielle.

La seule solution pour échanger des informations avec une interruption est de passer par des variables globales.

Pour plus de détails, vous pouvez bien sur retourner lire le TP3 des enseignement de S7.

Certaines des activités demandées dans ce TP ne sont pas *utiles* en tant que tel, mais représentent des étapes décisives dans la compréhension de certains phénomènes, ou dans la mise en place d'un système final plus complet.

1.2 Plus loin que les interruptions

En fin de TP, nous verrons que malgré un gain en performances non négligeable les interruptions peuvent aussi montrer leurs limites. Il existe encore plus performant, mais, il n'y a pas de secrets ! Comme pour Faust et Icare, passé un certain niveau d'exigences, tout a un prix.

2 Sous-traiter l'envoi aux interruptions

Dans un premier temps, nous n'allons pas révolutionner les algorithmes du TP1. Nous allons nous contenter de gérer la succession des données en interruption, mais la fonction de communication restera bloquante.

2.1 L'essentiel, c'est que ça marche !

Le projet `speTP2_01_IT_base` vous donne un cadre de base. La fonction d'interruption associée au module SPI1 est `SPI1_Handler()`. La fonction de configuration du module SPI est maintenant `init_SPI_IT()`

Activité 1

Vérifiez que vous pouvez déclencher l'interruption du module SPI1. L'essentiel de la configuration est fait dans la fonction `SPI1_init_IT()`. Vous pouvez (devez) recopier la configuration fonctionnelle du TP1. Une partie de la configuration des interruptions est déjà faite. Il n'y a plus qu'à activer le bit qui va bien dans le registre qui va bien pour que le module SPI1 fasse des demandes d'interruption quand c'est nécessaire. Le mieux est de déclencher l'interruption, à la réception d'une donnée (cela signifie qu'il faut d'abord envoyer quelquechose).
Pour vérifier qu'elle se déclenche, il y a plusieurs possibilités :

- Allumer une LED alors qu'elle n'est allumée nulle part ailleurs dans le programme, et qu'elle est éteinte une seule fois avant d'activer les interruptions.
- Faire clignoter la LED dans une boucle infinie (c'est mal, mais c'est pas grave, c'est juste pour un premier test de validation)
- Lancer le debugger et mettre un point d'arrêt dans la fonction (il faut un debugger, ça tombe bien, il y en a un).
- Bouger un signal de sortie spécifique, et l'observer à l'oscilloscope (il faut un oscilloscope, ça tombe bien, il y en a).

Choisissez celle qui vous convient le mieux.

La fonction `SPI_exc_mult()` est laissée vide. Vous pouvez récupérer le code du TP1, a une différence près.

Activité 2

Écrivez le contenu de `SPI_exc_mult()` pour que l'essentiel de l'envoi du tableau `data` se fasse en interruption. Pour cela, vous aurez besoin :

- d'une variable globale de type `unsigned char *` pour pointer sur la donnée qui vient d'être envoyée
- d'une variable globale dont la valeur sera le nombre de données qu'il reste à recevoir.
- d'une fonction d'interruption qui se déclenche à chaque réception de donnée et envoie la donnée suivante (si nécessaire)

Les seules parties qui ne se font pas en interruption doivent être :

- la mise à 0 de SSn
- L'envoi de la première valeur
- l'attente avant de remonter SSn
- la mise à 1 de SSn
- une attente avec SSn en haut pour éviter que la commande suivante n'arrive trop tôt.

ATTENTION: ce n'est pas parce qu'on utilise des interruptions que l'envoi est instantanné. avant de remonter SSn, il faut attendre que toutes les données soient envoyées.

Si ça marche, félicitations ! Vous avez réussi a faire pareil qu'avant, mais en plus compliqué, et en consommant plus d'énergie (ben oui, le module SPI ne travaille pas gratuitement)

2.2 Réduire la consommation

L'utilisation du module SPI ne réduit la consommation *que* s'il fonctionne *à la place* du processeur. S'il fonctionne en même temps, il permet simplement de faire du multitâche (envoi et calcul simultané). Ici, on ne fait ni l'un, ni l'autre : le processeur fonctionne, mais ne fait aucun calcul !

Le plus simple est de mettre le processeur en sommeil pendant l'envoi. Pour cela, il y a la fonction `__WFI()` (Wait For Interrupt). Nous ne rentrerons pas plus dans les détails des niveaux de consommation du système, mais arrêter le processeur, c'est déjà pas mal.

Activité 3

Prouvez que la fonction `__WFI()` met le processeur en sommeil. Utilisez la à un endroit quelconque (mais judicieux) de votre programme.
Si vous utilisez le debugger, vous pourrez relancer le processeur, sinon, seul le *reset* peut le relancer.

Pour réduire la consommation, il suffit donc d'appeler `__WFI()` juste après l'envoi de la première donnée. Le processeur sera réveillé au déclenchement de l'interruption suivante pour l'éxécuter.

Activité 4

Arrêtez le processeur pendant l'échange de données SPI (C'est con, on ne peut pas le constater :/). On partira du principe que c'est réussi si vous appelez `__WFI()` sans perturber le fonctionnement du programme.

2.3 Gestion du SSn

On a beau avoir fait ce qu'on peut, il reste quand même des moments d'inactivité qui occupent le proceseur. C'est le cas des attentes entre l'envoi de la dernière donnée et la remontée de SSn, ou entre la remontée de SSn et son retour à 0.

Bien sur, vous pouvez utiliser la fonction `delay_us_timer7` au lieu de `delay_us_timer6`. Vous aurez droit à une attente éco-responsable qui éteint le processeur quand nécessaire. La technique consiste à régler le timer pour qu'il déclenche une interruption après la durée voulue, et domir en attendant. L'interruption réveillera le processeur.

Activité 5

Dans les faits, lorsqu'il arrive à la fin, le timer 7 fait passer le registre `TIM7->SR` à 1. Pourquoi est-il impossible de voir ce bit passer à 1 depuis la fonction `delay_us_timer7` ?

2.4 Le retour de la machine d'états

L'objectif va maintenant être de gagner un peu de puissance de calcul. Il n'y a toujours pas de secret, si on fait plus de calculs, on consommera plus d'énergie, mais l'objectif est de ne pas ralentir le système si on a besoin de puissance de calcul.

En effet, lors de l'attente avant et après la remontée de SSn, les données reçues sont déjà prêtes. La fonction `SPI_exc_mult()` ne continue que pour gérer une séquence qui peut être réalisée en parallèle de la boucle principale.

Pour cela, il est possible de faire en sorte que la remontée de SSn se fasse après la fin de `SPI_exc_mult()`, dans une interruption déclenchée par le timer 7. Pour éviter tout conflit, il faut une variable globale qui permet de définir si le système est en attente pour relever SSn, en attente entre deux commandes ou pas en attente. Selon les actions faites ou à faire, cette variable changera de valeur. Bref, nous avons besoin d'une machine d'états. Ça tombe bien, regardez la définition de la variable globale `SPI_FSM`...

Voici comment se déroule le mécanisme :

- Lors d'un appel de `SPI_exc_mult`. On modiife le pointeur de données et la quantité à recevoir pour préparer le premier envoi. Si les attentes sur timer7 sont inactives (`SPI_FSM==SPI_FSM_IDLE`), il faut baisser SSn et envoyer la première donnée. Sinon, on laisse faire les interruptions.

- La fonction `SPI_exc_mult` attend que la quantité de données à recevoir soit nulle, ou que `SPI_FSM==SPI_FSM_WSSnLOW` en utilisant le mode de mise en sommeil.
- Pour `SPI_exc_mult`, le job est fini, il n'y a qu'a terminer proprement pour que l'appelant récupère ses données.
- Si l'interruption SPI1 se déclenche, c'est qu'on a reçu des données. il faut mettre à jour les données et le nombre à recevoir. Si c'était la dernière donnée, il faut mettre à jour `SPI_FSM` pour que `SPI_exc_mult()` se termine et configurer le timer 7 pour avoir une interruption quand nécessaire.
- Si l'interruption du timer 7 se déclenche, et que `SPI_FSM==SPI_FSM_WSSnLOW`, il faut relever SSn et programmer une nouvelle interruption pour savoir quand on pourra rebaisser SSn. En attendant, `SPI_FSM` passe à `SPI_FSM_WSSnHIGH`
- Si l'interruption du timer 7 se déclenche, et que `SPI_FSM==SPI_FSM_WSSnHIGH`, Le transfert précédent est terminé.
  - S'il reste des données à envoyer, c'est qu'une nouvelle commande est en attente. Il faut baisser SSn et envoyer la prochaine donnée.
  - S'il ne reste plus de données, on peut se reposer.

  Dans tous les cas, `SPI_FSM` passe à `SPI_FSM_IDLE`

Avec cette organisation, la fonction `SPI_exc_mult()` bloque moins de temps. Poru la propreté du code, il est aussi possible d'identifier par `SPI_FSM_SENDING` le temps pendant lequel le module SPI1 est occupé à tranférer les données.

Activité 6

Mettez en place le système décrit précédemment.

On a un réseau d'interruptions. Plusieurs interruptions programment leurs déclenchements respectifs selon les besoins, et de façon (presque) autonome. C'est beau, non ?

Bon ! Il reste un souci : le déclenchement d'une interruption demande environ 1 µs, ce temps peut paraître bref pour une liaison type UART, mais il devient handicapant pour une liaison rapide. Imaginez : si la lisaison depasse 10 MHz, les données arrivent plus rapidement que le temps de déclenchement d'une seule interruption.

Pour gérer ce genre de situation, il y a un module matériel qui peut faire des copies de données sans passer par le processeur, le contrôleur DMA (Direct Memory Access). Malheureusement, c'est une hitoire trop longue à raconter pour un TP de quelques heures :/

3 conclusion

A ce niveau, vous avez manipulé tout un ensemble d'interruptions pour construuire une application cohérente. Nous sommes encore loin d'un système complet, mais les principes sont posés, le reste n'est qu'une question de temps de travail.

Nous avons également abordé un aspect de la réduction de la consommation en mettant le processeur en sommeil. Techniquement, il y a de nombreux états de réduction de la consommation selon que le processeur soit alimenté, que les périphériques recoivent l'horloge, ou soient alimentés. Le mode le plus profond étant celui ou même la mémoire n'est plus alimentée et perd donc ses données. Moins la consommation est élevée, plus il faut de temps pour revenir à un état fonctionnel.