Big loop et Interruptions

1 Application embarquée

Le principe d'une application embarquée consiste souvent à monitorer des évènement (capteurs, arrivée de données, échéance de timer, …) et à produire en réponse des actions (traitement d'une donnée, génération d'un signal, …). On peut considérer que monitorer un évènement en vu de produire une action sera une tâche. On peut distinguer 3 types d'implémentation pour faire tourner plusieurs 'tâches' ou traitement :

1. Big loop : une grande boucle exécute les tâches les unes après les autres
2. interruptions : chaque tâches est lancée à l'aide d'une interruption
3. multitâche : un exécutif indépendant partage le temps d'exécution des tâches sur le processeur.

Dans ce TP, on se concentrera sur les 2 premières implémentation, la troisième sera étudiée plus tard dans le semestre.

L'application étudiée dans ce TP utilise les programmes écrits lors des précédents TP. L'application souhaité se résume à 3 ou 4 opérations ou tâches principales :

- incrémentation de la luminosité d'une led tous les 500ms (ou clignotement)
- Mesure de la durée d'une impulsion (simulé par le bouton utilisateur de la carte)
- Envoi de la valeur mesurée sous forme de log vers la station hôte pour affichage
- Réception d'une donnée optionnelle (cette tache ne sera mise en place que dans la dernière partie du TP)

2 'Big loop'

La 'big loop' consiste à exécuter les 'tâches' de l'application les unes à la suite des autres dans une grande boucle. Pour un 'bon' fonctionnement chaque tâche doit pouvoir être exécutée 'suffisamment souvent' pour ne pas louper d'évènements et donc chaque tâche doit être 'suffisamment courte' en terme de temps d'exécution. La notion de 'courte' est dépendant de l'application. On comprend cependant que le temps de cycle (temps d'exécution de l'ensemble de la 'big loop') doit permettre de monitorer l'ensemble des évènements sans en perdre mais également que le temps de réaction à un évènement soit inférieur à un seuil tolérable (temps entre l'apparition d'un évènement et le début de sont traitement).
Le premier principe est d'éviter tout opération 'bloquante'. On entend ici par opération bloquante, une opération qui attend activement en boucle qu'un évènement se produise (exemple : while (! Évènement) ;)
En effet, si une tache attend indéfiniment qu'une donnée arrive, rien ne pourra être exécuté durant cette attente. Ainsi dans l'application exemple, si la tâche led attend activement l'échéance d'un timer, elle peut être bloquée durant 500ms. Les autres opérations ou tâches ne pourront pas s'exécuter.

Une application fonctionnant selon le principe de la grande boucle doit pour chaque tâche à réaliser

- Tester si l'évènement monitoré est survenu
- Traiter l'évènement le cas échéant
- Passer à la tache suivante

Ainsi le principe est qu'à chaque tour de boucle, l'état du système doit être testé (les différents évènements à monitorer) et lancer les actions uniquement lorsqu'une action est requise.

Exemple d'application

loop :
    test si timer led arrivé à échéance

→ oui, l'intensité de la led est changée
→ non, on passe au test suivant
test si le début d'une impulstion a été détectée
→ oui, on reset si besoin les paramètres du timer
→ non, on passe au test suivant
test si l'impulsion est terminée et enregistrée dans le timer
→ oui on effectue le traitement (ici on commence l'envoie de la valeur acquise)
→ non on passe au test suivant
test si un envoie est en cours et qu'il y a encore des données à envoyer
→ oui on envoie une nouvelle donnée
→ non test suivant
….

On voit sur cet exemple que les opérations à réaliser sont testées séquentiellement, sans ordre de priorité. L'ensemble de l'exécution de la boucle doit être suffisamment court pour ne pas pénaliser la prise en compte d'un évènement ou pire louper un évènement.

En reprenant le code du tp précédent, réalisez l'application 3 tâches décrite au tout début (ne pas s'occuper de la réception de donnée pour le moment). Votre application devra suive scrupuleusement la méthode décrite précédemment (test, action si évènement, test suivant sinon). Votre code ne devra plus comporter de boucle de scrutation (boucle du type while (! evenement) ;).

3 Utilisation des interruptions

La méthode précédente n'est pas toujours simple à mettre en œuvre et manque de flexibilité. L'utilisation des interruptions va permettre d'obtenir une application qui réagira lorsque les évènements se produiront.
Le principe consiste à ce que chaque élément monitoré conduisent à la génération de requêtes d'interruption lorsqu'ils change d'état. Ainsi chaque évènement lancera une routine de traitement d'interruption, dans laquelle sera traité l'évènement. Cette méthode présente plusieurs avantages :

- prise en compte quasi immédiate de l'évènement (pas d'attente que ce soit le moment du test)
- possibilité de définir des priorité de prise en compte (au lieu des faire les test séquentiel) en définissant des priorités sur les requêtes d'interruptions et leur traitement.
- Le temps de réaction à un évènement (temps entre l'apparition d'un évènement et l'action associée) peut être déterminé dans le 'pire cas' (= temps réel)

Il est possible de développer une application qui fonctionne entièrement à l'aide d'interruption ou de faire un mixte entre la grande boucle et l'utilisation des interruptions. Il peut être pas exemple intéressant de n'effectuer que les traitements urgents au sein des routine de traitements des interruptions (handler) et de laisser les traitements plus lourds et moins urgent au sein de la grande boucle.

Dans les exercices suivant vous allez mettre en œuvre les périphérique en utilisant les interruptions. Il peut être judicieux de tester chaque périphérique sous interruption indépendamment.

3.1 TIM5

Les drapeaux du registre d'état des timer du STM32 peuvent être configurés pour générer des requêtes

d'interruption. Lorsqu'une requête d'interruption est générée par un périphérique, celle-ce est détectée par le contrôleur d'interruption NVIC. Si la requête d'interruption est autorisée au niveau du contrôleur NVIC, l'application en cours d'exécution sera déroutée vers la routine de traitement d'interruption associée (fonction aussi appelée handler d'interruption).

Pour le tim5 La routine de traitement appelée est **void TIM5_IRQHandler(void)**. La routine est déjà enregistrées dans la table des vecteurs d'interruption (une fonction TIM5_IRQHandler() définie avec un attribut 'weak' est présente dans le code de démarrage écrit en asm. L'attribut 'weak' permet à l'utilisateur de redéfinir la fonction.

Pour réaliser la première tâche à l'aide des interruptions (changement régulier de l'intensité de la led) vous devez après avoir configuré le TIM5 :

- définir une priorité de prise en compte de l'interruption au niveau du NVIC (cf. annexe)
- autoriser les interruptions au niveau du NVIC
- configurer le timer pour définir quels drapeaux peuvent générer une requête d'interruption (ici UIF est suffisant mais d'autres drapeaux peuvent aussi générer des requêtes)

A chaque interruption la routine TIM5_IRQHandler() est appelé par le système. Dans cette routine vous devez :

- déterminer quel drapeau à généré la requête d'interruption (nécessaire si vous avez configuré l'interface pour que plusieurs drapeaux génèrent des requêtes d'interruption)
- acquitter l'interruption : comme pour le polling il est nécessaire par exemple de remettre UIF à 0 sinon la requête d'interruption sera maintenue.

Ensuite vous choisissez les opérations que vous souhaitez implémenter dans cette fonction. Ici vous pouvez changer uniquement la valeur du PWM.

En phase de test vous pouvez placer un breakpoint sur la routine de traitement des interruptions.

Remarque contrairement à d'autre processeur, avec les cortex-M il n'est pas nécessaire d'acquitter le contrôleur d'interruption ou d'utiliser des instructions spéciales pour le retour d'interruption.

3.2 TIM4

Pour le TIM4, il semble judicieux de générer des requêtes d'interruption pour les 3 drapeaux : UIF, CC1F et CC2F, de manière à détecter les débuts et fin d'impulsion ainsi que les 'overflow' dans les cas d'impulsions qui dureraient plus que 65535 ticks.
Dans cette partie vous mettrez juste à jour dans une variable globale la valeur de la dernière durée d'impulsion. Testez au débogueur.

3.3 USART2

De même que pour les timers, il est possible pour le périphérique USART de générer des requêtes d'interruption sur les drapeaux du registre d'état (SR).

Pour la transmission, un requête d'interruption peut être générée à chaque fois que le registre tampon d'émission est vide (TDR). La routine d'interruption se contente de placer une nouvelle données dans le registre de donnée. Il et à noter qu'une requête d'interruption est générée, dès que TXE est à 1. Si aucune donnée n'est à transmettre, il faut désactiver les interruptions au niveau de l'USART.

Il vous est demandé ici d'écrire une fonction dite 'utilisateur' qui aura le prototype suivant :

int32_t USART2_transmit_IRQ (uint8_t buffer, uint32_t len) ;

Cette fonction sera non bloquante. Elle est uniquement chargée d'initier une émission de len données 8 bits et devra revenir immédiatement après la mise en place de cette émission. L'essentiel de la transmission sera ensuite gérée dans la routine de traitement des interruptions de l'USART2.

Accès multiple : il est possible qu'une tâche souhaite envoyer des données sur la liaison série alors qu'une transmission est en cours. Par exemple si vous souhaiter envoyer un message lorsqu'une impulsion est mesurée et que 2 impulsions consécutives sont mesurées. Il se peut que le premier envoi soit encore en cours alors que votre application souhaite initier un deuxième envoi. Il existe plusieurs technique pour gérer ce type de problème (accès a une ressource unique) ici le second envoi sera purement et simplement abandonné.

Ainsi la fonction USART2_transmit_IRQ () renverra -1 si une transmission est en cours et 0 si l'émission à pu être mise en place.

Testez votre fonction depuis la boucle principale de l'application dans un premier temps.

3.4 Application complète entièrement sous interruption avec endormissement

L'envoi des données est dépendant de la réception d'une impulsion. L'initiation d'un envoi de données peut se faire directement depuis l'ISR de TIM4 quand la durée d'une impulsion et acquise. Dans ce cas la boucle principale est entièrement vide. Toutes les actions s'effectuent depuis les routine de traitements des interruptions.

Il est possible d'endormir le SoC lorsqu'il n'a aucun traitement à faire. L'endormissement doit être un endormissement léger parce que es périphériques doivent continuer à fonctionner. Seul le cœur sera endormi. Ce dernier sera réveillé dès qu'une requête d'interruption sera reçue.
Sur les cortex-M l'endormissement s'effectue en appelant la fonction CMSIS ___WFI(). L'appel de cette fonction endort le cœur, qui reprendra son exécution lorsqu'une requête d'interruption non masquée par le NVIC sera reçue.
Sur les Cortex-M il est également possible de paramétrer le système pour que le cœur se ré-endorme au retour d'interruption. Dans ce cas seul les routine d'interruptions s'exécuteront. Le réglage s'effectue au niveau du bit CB_SCR_SLEEPONEXIT dans le registre SCR du périphérique CB (control Block).

Ainsi il est possible de développer 2 types d'application :

1. une application standard qui effectue une partie de traitement hors interruption (big loop, mutitache) et qui s'endort lorsque rien n'est à faire par un appel à ___WFI(), bit CB_SCR_SLEEPONEXIT à 0
2. une application qui ne fonctionne qu'avec des interruptions et dort le reste du temps, bit CB_SCR_SLEEPONEXIT à 1

Dans cette partie tout pouvant être géré par interruption vous pouvez mettrez le bit CB_SCR_SLEEPONEXIT à 1. Ainsi après initialisation, toutes les tâches seront réalisées dans les routines de traitement des interruptions, la boucle principale sera vide.

3.5 Mixage ‘big loop’ et interruptions

Dans cette partie, une partie de l’application sera gérée dans la boucle principale. L’intérêt de ce type d’application consiste à laisser dans la boucle principale les traitements ‘lourd’ afin de laisser dans les routines de traitement des interruptions que les traitements urgents. L’ensemble doit permettre au système d’être plus ‘reactif’. Par exemple, on peut imaginer une application qui a un gros traitement de données à faire, en plus de la gestion des capteurs/actionneur. Le traitement lourd est alors laissé dans la boucle principale afin de laisser la possibilité d’être interrompu par les traitements plus urgent.

Dans cet exercice, vous laisserez dans la boucle principale le traitement de la dernière impulsion mesurée (ici il s’agira juste de convertir le nombre tick en ms, puis en chaîne de caractères) ainsi que sont envoi vers la station hôte. Vous ajouterez également dans cette boucle principale la modification de la période de changement d’intensité de la led. La grande boucle sera sous la forme suivante :

loop :

- test si une nouvelle impulsion a été mesurées
  - si oui : récupération, conversion éventuelle en milliseconde, envoi du log à la station hôte
- test si une nouvelle période de changement d’intensité de la led a été reçue
  - si oui : récupération de la chaine de caractère, transformation en entier et modification de la période de changement d’intensité de la led
- endormissement

L’envoie et la réception de donnée s’effectue par interruption afin de pouvoir endormir le système. Sous interruption la détection de fin de trame série permettant de déterminer que toute la chaîne de caractère a été reçue s’effectue à l’aide du drapeau ‘idle’ su state register. L’USART devra être configurée pour que ce drapeau ‘idle’ génère une requête d’interruption.

Remarque : les routines de traitement d’interruption (ISR) et la boucle principale fonctionnent indépendamment, la communication entre ces fonctions se fait à l’aide de variables globales partagées

4 annexe : interruption sur Cortex-M, rappels

Les handler d'interruption sur les Cortex-M peuvent être écrit comme des fonctions C standard (ce qui est rarement le cas pour d'autres processeurs). Le processeur sauve les registres qui peuvent être corrompus par un appel de fonction : registres 'scrtach' (r0-r3, r12) et le registre de lien r14 dans la pile courante. L'adresse de retour (adresse de l'instruction à exécuter au retour d'interruption) est également copiée dans la pile courante.
Une fois ces sauvegarde réalisées, le registre de lien r14 est chargé avec une valeur spécifique (EXEC_RETURN) et le processeur se branche à l'adresse de la routine de traitement des interruption (récupérée dans la tables de vecteurs d'interruptions). La valeur EXEC_RETURN permet au processeur au moment du retour, d'une part de déterminer qu'il exécutait une routine de traitement d'interruption et d'autre part de déterminer quoi restaurer et dans quel mode revenir.
Les adresses des handler d'interruption sont stockées à partir de l'adresse 4. L'adresse 0 est réservée pour stocker l'adresse du pointeur de pile initial (MSP). Vous pouvez observer la section *.isr_vector* dans le fichier de startup *stm32f401xe.s*. Vous pourrez notez aussi que tous les adresses pointent vers des routines avec attribut weak et sont 'aliasées' vers une routine de traitement par défaut qui effectue une boucle infinie.

Pour programmer le contrôleur d'interruption NVIC, vous pouvez utiliser les fonctions C de la librairie CMSIS suivante :

- `void NVIC_SetPriorityGrouping(uint32_t PriorityGroup)  // définit les priorités`
- `void NVIC_SetPriority (IRQn_Type IRQn, uint32_t priority)`
- `void NVIC_EnableIRQ (IRQn_Type IRQn)  // autorise l'IRQ numéro IRQn (NVIC)`
- `void NVIC_DisableIRQ (IRQn_Type IRQn) // masque l'IRQ numéro IRQn (NVIC)`
- `void __enable_irq(void)        // autorise les IRQ au niveau du cœur Cortex-M`
- `void __disable_irq(void)       // masque les IRQ au niveau du cœur Cortex-M`

Les numéros d'interruption (IRQn) sont définie dans le fidhier stm32f401xe.h (e.g. TIM5_IRQn).
Il est à noter que le STM32 n'implémente que 4 bits de priorité donc 16 niveaux de 0 (le plus prioritaire à 15 (le moins prioritaire). Ces 4 bits peuvent être divisés en 2 parties : priorité et sous-priorité. Cette subdivision peut être fixée en appelant la fonction `NVIC_SetPriorityGrouping()` avec comme argument 3 (4 bits de priorité, pas de sous priorité) à 7 (pas de priorité et 4 bits de sous-priorité).