Fiche explicative : les interruptions

Periph'Team - INSA de Toulouse

1 Principe et fonctionnement

1.1 Principe

Supposons que vous devez vous lever impérativement à 08h00 pour assister à votre TP favoris (implicitement le TP de périph de 09h30). Vous avez deux possibilités : passer la nuit à regarder votre réveil, téléphone,... pour guetter l'heure à laquelle vous devez vous lever ou programmer l'alarme de votre réveil à l'heure à laquelle vous devez vous lever.

La première méthode consisterait à faire du **scrutatif** à savoir, lire continuellement l'heure sur le réveil. Cela peut tout à fait fonctionner si vous ne devez rien faire d'autre (en outre, vous ne pouvez pas dormir...).

L'autre solution est de programmer une demande d'**interruption** à savoir demander à vous faire réveiller lors d'un événement particulier : l'heure courante atteint l'heure de réveil. Cette demande vous **détournera** de votre activité (en l'occurrence ce que l'étudiant sait le mieux faire : dormir) pour que vous vous consacriez à l'activité que vous avez prévue de faire après le réveil (...vous rendormir!).

Tout microcontrôleur qui se respecte offre des mécanismes de gestion des interruptions. La nature de ce qui provoque une interruption (ou évènement) est très large et dépend du périphérique associé à l'interruption (*timer, ADC, GPIO,* unité de *capture/compare, DMA*,...). Par exemple, l'apparition d'un front montant sur une broche d'entrée, le débordement d'un timer, réception d'un caractère sur une *USART*, atteinte d'une valeur seuil sur une entrée analogique...

En utilisant les mécanismes de gestion des interruptions, la CPU est alors libre et si tout a été correctement con-figuré le déroulement séquentiel de votre programme sera **détourné** dès que l'événement interviendra pour aller exécuter une routine de traitement de l'interruption (**handler**).

A travers ce TP nous allons mettre en œuvre un détournement d'interruption sur un débordement d'un registre de comptage d'un timer mais le principe sera le même pour les autres périphériques et évènements. Le principe des interruptions est commun à tous les processeurs. La mise en œuvre peut présenter des différences mais on trouvera toujours la présence d'une **table des vecteurs d'interruption**. Cette table contient l'adresse où est placé en mémoire le point d'entrée de la routine à exécuter lorsqu'une interruption est déclenchée. Comme il existe plusieurs interruptions possibles, cette table est indexée et chaque entrée correspond à un numéro d'interruption. On parle donc de vecteur d'interruption pour chacune des entrées de cette table.

Le second grand principe des interruptions est la notion de priorisation de celles-ci. En effet comme elles peu-vent être multiples et que par essence elles sont asynchrones, que se passe-t-il quand une nouvelle demande d'interruption arrive alors que le processeur n'a pas encore fini de traiter une précédente ? Idem si deux de-mandes surviennent simultanément. On voit bien qu'il est nécessaire de définir une politique d'organisation de tout cela (et on aborde par ce biais une des problématiques majeures des systèmes informatiques dits à temps réel). Il en découle la nécessité au niveau matériel d'avoir une unité spécialisée, appelée gestionnaire d'interruption,

pour centraliser et traiter les demandes d'interruptions provenant de différentes sources. La figure 1 illustre ce principe, à noter qu'une unité périphérique peut parfois engendrer plusieurs vecteurs d'interruption (par exemple, l'unité périphérique c provoque les demandes n^0 3 et n^0 4 d'interruption). Il peut aussi arriver (et c'est même assez courant) qu'une unité ne possède qu'un seul vecteur mais que la cause du déclenchement soit multiple. Il conviendra alors à la routine d'interruption d'aller lire les registres spécifiques pour savoir quelle est la cause du déclenchement si le traitement doit être différencié.

Enfin sur le plan général, il est bien de savoir que le terme interruption est généralement lié à un événement extérieur au cœur du processeur (typiquement une entrée broche directe ou une unité périphérique) mais que le même mécanisme de détournement est mis en place lorsque le cœur rencontre un problème grave (reset, division par zéro, problème accès mémoire, ...). On parle alors plus communément d'exception ou de trappe même si la généralisation du terme interruption est assez commune (ne serait-ce que dans l'expression « table des vecteurs d'interruption » qui contient tous les cas possibles de détournement, extérieurs et intérieurs).

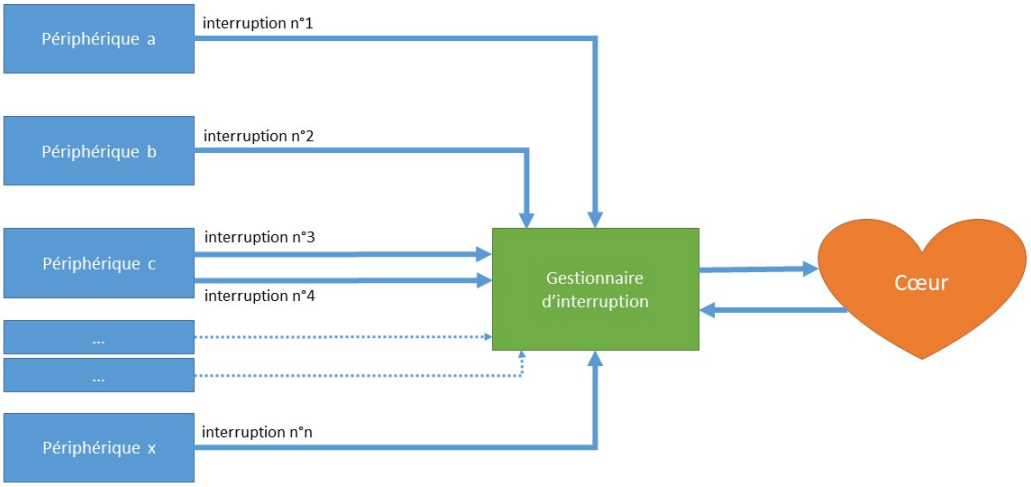


Figure 1: *Positionnement du gestionnaire d'interruption*

Chronologiquement, le processus de détournement correspond à la figure 2.

La grande nouveauté pour le programmeur est qu'une routine (*handler*) est exécutée sans qu'aucune ligne de code ne l'appelle explicitement. C'est un évènement matériel extérieur qui déclenche directement l'exécution de cette routine après traitement par le gestionnaire d'interruption. Si cela ne pose pas beaucoup de problème d'un point de vue de la conception, cela peut s'avérer beaucoup plus délicat pour la mise au point, notamment quand des considérations temporelles sont présentes.

2 Les interruptions sous le STM32

2.1 La TVI

La table des vecteurs d'interruption dans la spécification du cœur du STM32, appelé Cortex, est placée à partir de l'adresse $0x0000000$. Sa composition générale est donnée figure **??**.

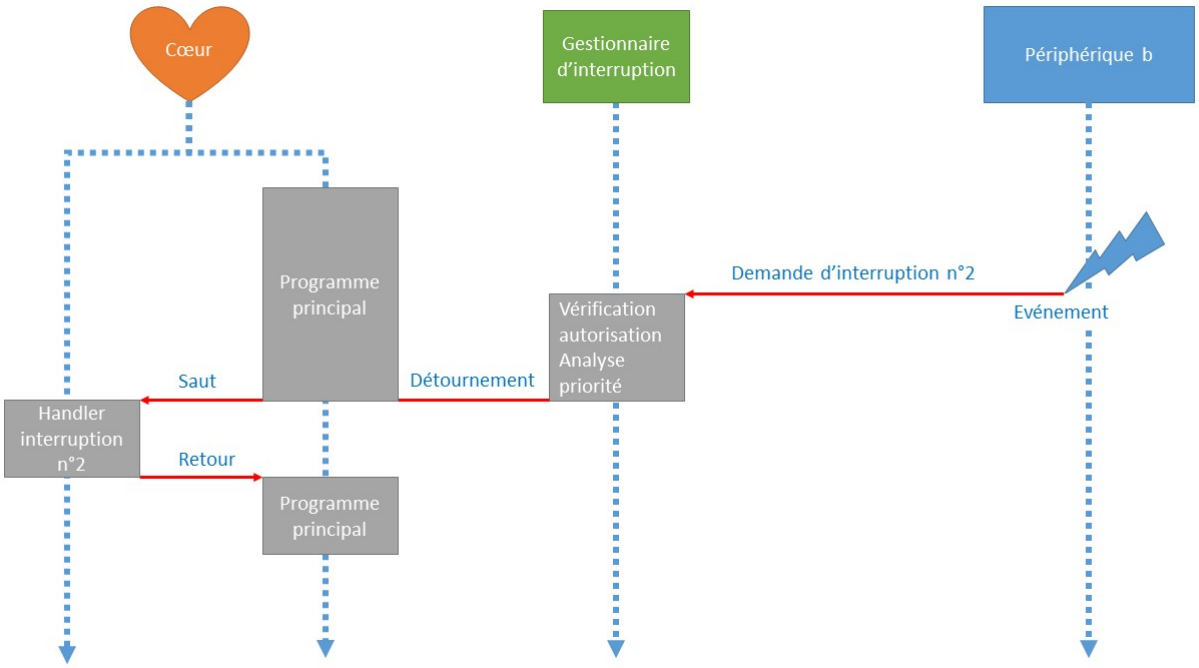
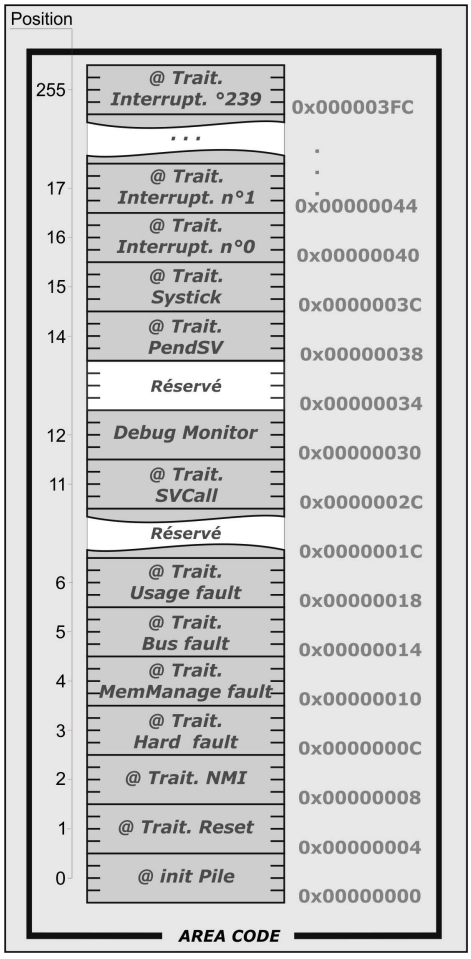


Figure 2: *Positionnement du gestionnaire d'interruption*



La table des vecteurs d'interruption dans la spécification du cœur du *STM32*, appelé Cortex, est placée à partir de l'adresse $0x0000000$. Sa composition générale est donnée figure ci-contre. Chaque vecteur contient une adresse et fait 4 octets. La première valeur (l'entrée 0 à l'adresse $0x0000000$) n'est pas réellement un vecteur mais l'adresse RAM qui initialisera le pointeur de pile système après un *Reset*. La seconde valeur (l'entrée 1 à l'adresse $0x00000004$) n'est pas non plus réellement un vecteur mais est l'adresse ROM qui initialisera le pointeur de programme après un *Reset*, donc le point d'entrée du programme au démarrage. Les valeurs 2 à 14 (les entrées 2 à 14 aux adresses $0x00000008$ à $0x00000038$) sont réellement des vecteurs et correspondent à des exceptions, donc des interruptions internes. Le vecteur 15 (l'entrée 15 à l'adresse $0x0000003C$) est le vecteur d'interruption pour le timer interne au Cortex (appelé *Systick*). Les vecteurs 16 à 255 correspondent aux vecteurs d'interruption possibles pour les périphériques. Il est important de noter que ceci correspond à la spécification d'ARM. STMicro (le fabricant du microcontrôleur) n'utilise pas l'ensemble de ces vecteurs. Le chapitre 10 du *Reference Manual (RM0008)* vous renseignera sur le nombre d'interruption qu'utilise le STM32. Une question que vous aurez sûrement à vous poser est des savoir à quelle entrée et à quelle adresse de la table se trouve le vecteur de l'interruption n^i du périphérique p?

2.2 Le NVIC

Le ***N**ested **V**ector **I**nterrupt **C**ontroler* est le gestionnaire d'interruption des Cortex et donc des STM32. Comme il est générique au Cortex, il faut consulter la documentation du Cortex (*Programming Reference PM0056*) pour en comprendre le fonctionnement. Pour chacune des interruptions possibles, le NVIC possède :

- une autorisation de déclenchement (voir les registres *ISER* et *ICER*) ;
- un statut : inactive, en attente ou active (voir les registres ICPR et *IABR*) ;
- un niveau de priorité (voir les registres 32 bits *IPR* et leur déclinaison par octet *IP*).

La règle de fonctionnement est simple : lorsqu'une demande d'interruption arrive au *NVIC*, celui-ci récupère son numéro sur le bus de donnée. Il la traite si elle est autorisée et la place en attente. Si son niveau de priorité est plus petit (attention plus le niveau est faible plus la priorité est importante) que le niveau de priorité courant, elle devient instantanément active et le détournement est alors enclenché.

> Remarque : sur le STM32, le niveau de priorité programmable est codé sur 16 niveaux (de 0 le plus important à 15 le plus faible). Ce niveau est codé dans les 4 bits de poids fort de chacun des octets composant les registres *IPR*. Chacun de ces octets est accessible avec le champ *IP* de *type unit8* de la structure *NVIC*.

2.3 Au niveau des unités périphériques

Pour une unité périphérique classique, lorsque qu'il existe un événement qui déclenche une demande d'interruption il y a obligatoirement un bit d'un registre dédié qui traduira la survenue de cet événement et donc potentiellement l'envoi d'une demande d'interruption. Cette demande ne sera transmise au NVIC que si elle est également autorisée en local (au niveau de l'unité périphérique). Il existe donc également pour chacune de cette demande un bit *enable* correspondant dans un registre du périphérique. Exemple pour l'unité *timer 2* Il existe jusque 6 événements distincts qui peuvent provoquer une demande d'interruption. Les bits de poids $0, 1, 2, 3, 4$ et 6 du registre *TIM2_DIER* (voir figure 3) permettent d'autoriser la transmission de la demande d'interruption. Lorsqu'un de ces événements advient, le bit d'activation correspondant (qui est un des bits de poids $8, 9, 10, 11, 12$ ou 14 du même registre) passe à un.

15.4.4 TIMx DMA/Interrupt enable register (TIMx_DIER)

Address offset: 0x0C

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	TDE	Res	CC4DE	CC3DE	CC2DE	CC1DE	UDE	Res.	TIE	Res	CC4IE	CC3IE	CC2IE	CC1IE	UIE
	rw		rw	rw	rw	rw	rw		rw		rw	rw	rw	rw	rw

Figure 3: *Exemple de bits d'autorisation et de demande d'interruption pour les timers 2 à 6.*

2.4 Les handler

Un *handler* (ou routine d'interruption) est une fonction C classique de type *void/void*. Elle ne peut en effet ni prendre ni retourner d'arguments dans la mesure où elle n'est pas explicitement appelée dans le code.

Comment faire pour que l'adresse d'implémentation de cette routine corresponde à l'adresse indiquée par la table des vecteurs d'interruption?

La solution retenue par STmicro est de figer le nom de ces *handler*, d'en déclarer une version minimale dans le fichier d'amorce (*startup_stm32f10x_md.s*) et de créer une table ad hoc (nommée *_Vectors* dans ce même fichier), reconnue par le linker pour qu'au chargement du programme en mémoire flash la table soit correctement initialisée. L'astuce dans la création de ces routines minimales est de les qualifier comme *[WEAK]*. Ainsi, si dans le reste du projet, le compilateur rencontre une autre routine portant exactement le même nom, il utilisera cette nouvelle routine en lieu et place de celle préexistante et créera la table *_Vectors* avec cette nouvelle adresse.

Exemple pour l'unité timer 2 : La routine d'interruption s'appelle *TIM2_IRQHandler*. Il suffit donc de créer quelque part dans votre projet les quelques lignes de code suivantes :

```
void TIM2_IRQHandler(void)
{
 /* completer avec le code de la routine du traitement de l'interruption */
}
```

pour qu'automatiquement le détournement de l'interruption n°°8 (*TIM2*) provoque l'exécution de ce code.

3 Procédure à suivre pour programmer une interruption

De façon générale, lorsque l'on veut programmer une interruption du STM32, les étapes à suivre sont :

1. Analyser le périphérique et repérer l'événement à prendre en compte
2. Trouver le numéro (et donc le vecteur associé) de l'interruption concernée
3. Configurer les registres du périphérique pour valider en local l'envoi d'une demande d'interruption
4. Fixer dans le NVIC la priorité de l'interruption
5. Autoriser dans le NVIC la prise en compte de l'interruption
6. Trouver le nom du Handler de l'interruption et écrire une routine avec le même prototype qui ne sera pas weak. Dans cette routine **il est impératif d'effacer le bit du périphérique qui provoque le déclenchement de l'interruption** faute de quoi celle-ci sera derechef activée au retour (même si dans cer-tains rares cas cela est fait automatiquement...).