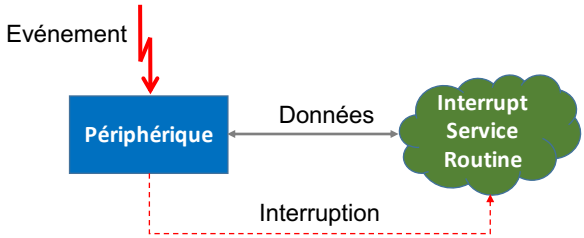
Systèmes embarqués 2 – TP.10 – Mini Projet OS

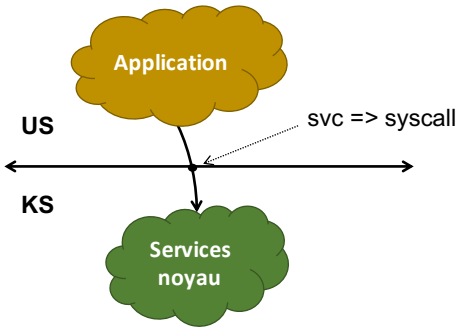
Réalisation d'un système d'exploitation

Interrupt Handling – FLIH

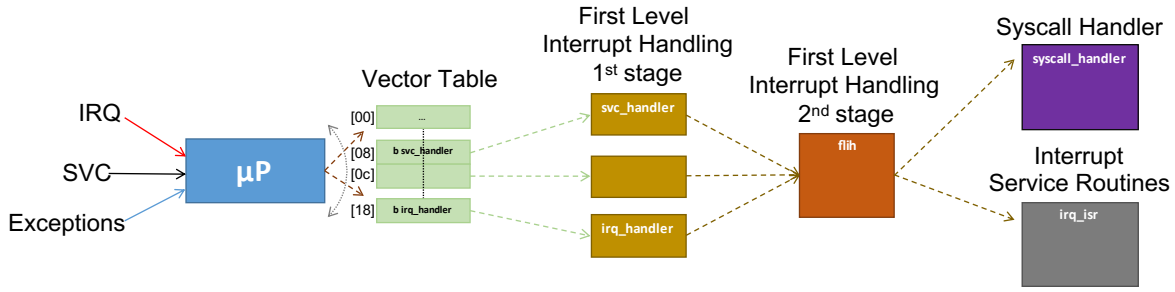
Le traitement des interruptions (Interrupt Handling) est un aspect incontournable d'un système d'exploitation, aussi élémentaire soit-il. Il procure à l'OS la capacité de traiter de façon asynchrone des événements levés par le matériel. Cette capacité à traiter des événements spontanés procure une grande réactivité au système et à ses applications en offrant un traitement rapide et approprié de l'événement.



Hormis le traitement d'événements matériels, le traitement d'interruptions permet également de traiter des événements logiciels. Le principal d'entre eux est le traitement d'appels système (syscall). Les interruptions logicielles permettent à un processus en espace utilisateur de franchir la barrière de protection en quittant son mode non privilégié pour accéder au mode privilégié du µP et ainsi aux services fournis par l'OS dans l'espace noyau.



Il est intéressant de noter qu'au niveau du µP, ces deux types d'interruptions sont traités de manière similaire.



Sur la figure ci-dessus, IRQ (Interrupt ReQuest) représente la ligne d'entrée du processeur pour toutes les interruptions matérielles. L'infrastructure mise en place dans cette phase du mini-projet, offrira le traitement de base, traitement au niveau du µP, des interruptions matérielles. Le traitement complet des interruptions matérielles sera réalisé dans une 2e phase du mini-projet. Il est important de noter que sans l'aide du contrôleur d'interruptions, il n'est pas possible de simuler cette interruption.

Systèmes embarqués 2 – TP.10 – Mini Projet OS

Réalisation d'un système d'exploitation

Interrupt Handling – FLIH

SVC représente l'entrée du processeur pour les interruptions logicielles. Dans une 1ère phase, les interruptions logicielles seront traitées comme une exception. Ce n'est que dans une 2e phase du projet qu'elles seront adaptées pour le traitement d'appels système (syscall).

Exceptions représentent les entrées du processeur pour toutes les interruptions levées par des événements d'exception que le processeur peut détecter. Le processeur est capable de détecter 3 exceptions distinctes :

- les instructions non définies (undefined instruction)
  Cette exception est levée chaque fois que le µP détecte une instruction qu'il ne fait pas partie de son jeu d'instruction. Certains OS utilisent cette fonctionnalité pour émuler des instructions supportées par d'autres processeurs.

- les échecs lors d'accès à des données (data abort)
  Il est intéressant de noter que les « data abort » peuvent être utilisés par les OS soit pour l'allocation de mémoire pour la pile des processus, soit pour détecter une violation de l'espace mémoire adressable du processus.

- les échecs lors d'accès à des instructions (prefetch abort).
  Cette exception est normalement fatale pour le µP. En effet, cette exception indique que le µP n'a pas réussi à lire une instruction et qu'il n'est plus possible de poursuivre l'exécution du programme.

Table des vecteurs

La table des vecteurs est un composant clef du système d'interruptions. Elle permet au µP d'appeler une routine de traitement lorsqu'une interruption ou une exception est levée. Elle fournit l'interface indispensable entre le matériel et le logiciel.

Sur les processeurs ARM Cortex-A8, cette table contient 8 entrées distinctes pour les 7 vecteurs d'interruptions que le processeur est capable de traiter. Chaque entrée de cette table contient une instruction. Cette instruction doit permettre d'appeler une 1ère routine de traitement de l'interruption.

```
vector_table:
1:    b     1b              // reset
      b     undef_handler
      b     svc_handler
      b     prefetch_handler
      b     data_handler
1:    b     1b              // reserved
      b     irq_handler
      b     fiq_handler
```

Systèmes embarqués 2 – TP.10 – Mini Projet OS

Réalisation d'un système d'exploitation

Interrupt Handling – FLIH

Les routines de traitement d'interruption de 1er niveau (First Level Interrupt Handler – FLIH) sont chargées de sauvegarder le contexte du processeur, d'appeler une routine de traitement d'interruption C (Interrupt Service Routine – ISR) et de restaurer le contexte avant de se terminer.

L'implémentation de la FLIH se décompose en 2 routines distinctes. La 1ère routine, écrite en assembleur, se charge tout d'abord de corriger la valeur de l'adresse de retour si nécessaire (correction du registre LR) avant de sauvegarder le contexte en plaçant sur la pile les registres du µP. Il est important de se rappeler que préalablement le µP a sauvé l'adresse de retour dans le registre LR et le registre de statut (CPSR) dans le registre SPSR du mode correspondant à l'interruption en cours de traitement. Cette 1ère routine appellera ensuite la 2e routine, écrite en C, chargée d'appeler la routine de traitement d'interruption (ISR). Une fois le traitement terminé, la 1ère routine restaurera le contexte du µP et retournera au programme interrompu par l'événement. Il est important de se rappeler que la restauration ne consiste pas seulement à charger les registres du processeur depuis la pile, mais également à restaurer le contenu du registre CPSR et à retourner là où le programme avait été interrompu. Voici deux exemples :

```
undef_handler:
      stmfd sp!, {r0-r12,lr}
      mov   r0, #UNDEF
      mov   r1, lr
      ldr   r2, flih
      blx   r2
      ldmfd sp!, {r0-r12,pc}^

irq_handler:
      sub   lr, #4
      stmfd sp!, {r0-r12,lr}
      mov   r0, #IRQ
      mov   r1, lr
      ldr   r2, flih
      blx   r2
      ldmfd sp!, {r0-r12,pc}^

flih: .long       // adresse de la FLIH en C
```

Dans les exemples ci-dessus, on remarque que l'appel de la routine de traitement d'interruption de 1er niveau écrite en C se fait de manière indirecte. Cette technique, un peu plus complexe, offre une grande souplesse d'implémentation. En effet, lors de l'initialisation de l'infrastructure de traitement des interruptions, l'application développée en C pourra ainsi choisir la routine FLIH qu'elle souhaite, ceci sans devoir manipuler du code en assembleur.

Dans les exemples ci-dessus, on remarque également que l'implémentation des deux routines est très similaire. Pour simplifier l'implémentation de telles routines, les assembleurs proposent l'utilisation de macros. La macro permet au développeur d'écrire une seule fois le code et de le customiser lors de l'assemblage par l'utilisation d'arguments.

```
.macro flih_1st_stage offset, vector
      .if \offset != 0
      sub   lr, #\offset
      .endif
      stmfd sp!, {r0-r12,lr}
      mov   r0, #\vector
      mov   r1, lr
      ldr   r2, flih
      blx   r2
      ldmfd sp!, {r0-r12,pc}^
.endm
```

Grâce à cette ma macro, l'implémentation des routines de 1er niveau est fortement simplifiée

```
undef_handler:    flih_1st_stage 0, INT_UNDEF
svc_handler:      flih_1st_stage 0, INT_SWI
prefetch_handler: flih_1st_stage 4, INT_PREFETCH
data_handler:     flih_1st_stage 4, INT_DATA
irq_handler:      flih_1st_stage 4, INT_IRQ
fiq_handler:      flih_1st_stage 4, INT_FIQ
```

Initialisation de l'infrastructure

Afin que le traitement des interruptions fonctionne, il faut l'initialiser. Cette tâche peut être divisée en 3 parties distinctes :

- L'initialisation des pointeurs de piles (stack pointeur)

```
mrs   r3, cpsr                // save mode
msr   cpsr_c, #0xd1           // switch to fiq mode
ldr   sp, =FIQ_STACK_TOP
msr   cpsr_c, #0xd2           // switch to irq mode
ldr   sp, =IRQ_STACK_TOP
msr   cpsr_c, #0xd7           // switch to abort mode
ldr   sp, =ABORT_STACK_TOP
msr   cpsr_c, #0xdb           // switch to undef mode
ldr   sp, =UNDEF_STACK_TOP
msr   cpsr_c, r3              // restore mode
```

- La configuration du registre du µP indiquant où se trouve la table des vecteurs

```
ldr   r2, =VECTOR_BASE_ADDR
mcr   p15, #0, r2, c12, c0, #0
```

- L'installation de la variable flih contenant l'adresse de la routine FLIH écrite en C

```
ldr   r2, =flih
str   r0, [r2]    // r0
```

Utilisation de la SRAM interne

Le µP TI AM335x dispose de plusieurs mémoires SRAM internes à son chip. Ces mémoires SRAM ont des temps d'accès très court à comparer à ceux de la mémoire DDR externe au µP. Leur utilisation permet d'améliorer les performances de notre système de traitement d'interruptions.

La mémoire L3 OCMC de 64KiB située à l'adresse 0x4030'0000 est parfaitement adaptée pour stocker

- la table des vecteurs
- les routines de traitement des interruptions de 1er niveau
- les piles pour les différents modes d'opération du µP

Afin de l'utiliser pour les piles des différents modes d'opération du µP, il suffit d'allouer des piles de 8KiB comme suit :

```
#define AM335X_OCMC_SRAM_BASE_ADDR  (0x40300000)
#define STACK_BASE                  (AM335X_OCMC_SRAM_BASE_ADDR + 0x1000)
#define IRQ_STACK_TOP               (STACK_BASE + 0x8000)
#define FIQ_STACK_TOP               (STACK_BASE + 0x6000)
#define ABORT_STACK_TOP             (STACK_BASE + 0x4000)
#define UNDEF_STACK_TOP             (STACK_BASE + 0x2000)
```

Pour placer la table des vecteurs et les routines de traitement des interruptions de 1er niveau dans cette SRAM, il suffit de copier les contenue de la DDR dans la SRAM avec la fonction « memcpy », soit :

```
#define VECTOR_BASE_ADDR      (AM335X_OCMC_SRAM_BASE_ADDR)

ldr   r0, =VECTOR_BASE_ADDR
ldr   r1, =vector_table_start
ldr   r2, =( vector_table_end - vector_table_start)
bl    memcpy
```

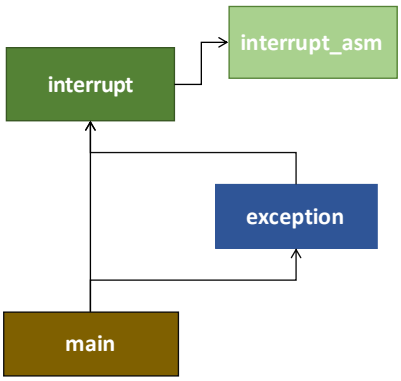
Systèmes embarqués 2 – TP.10 – Mini Projet OS

Réalisation d'un système d'exploitation

Interrupt Handling – FLIH

Diagramme des modules du FLIH

Le diagramme ci-dessous représente les différents modules nécessaires à l'implémentation du système de traitement des interruptions de 1er niveau ainsi que du traitement des exceptions.



Voici une description sommaire des modules :

- main : module principale de l'application
- `exception` : module implémentant les routines de traitement des exceptions
- `interrupt` : module fournissant les services permettant à l'application d'attacher les routines de traitement des interruptions (ISR) à un vecteur d'interruption
- `interrupt_asm` : module en assembleur implémentant le traitement des interruptions de 1er niveau. Ce module réalise également l'implémentation de services de bas niveau en assembleur.

Travail à réaliser

Dans le cadre de ce laboratoire, les étudiant-e-s devront implémenter la fonctionnalité des modules interrupt et exception. Pour le module interrupt, les services suivants devront être réalisés :

- Méthode « interrupt_init »
  Cette méthode permet d'initialiser le module et de configurer le µP pour le traitement de tous les vecteurs d'interruptions et exceptions disponible sur le µP
- Méthode « interrupt_attach »
  Cette méthode permet d'attacher une routine de traitement des interruptions (ISR) à un vecteur donné. Lors de l'appel de cette méthode, on pourra donner un paramètre, lequel sera passé comme argument avec l'identifiant du vecteur de l'interruption lors de l'appel de la méthode de traitement d'interruption (ISR). Cette méthode n'autorisera d'attacher qu'une seule ISR à un vecteur d'interruption. En cas d'erreur la valeur -1 sera retournée, en cas de succès 0.
- Méthode « interrupt_detach »
  Cette méthode permet de détacher une ISR de son vecteur d'interruption.

- Méthode « interrupt_enable »
  Cette méthode autorise la levée d'interruptions matérielles.
- Méthode « interrupt_disable »
  Cette méthode bloque la levée de toutes les interruptions matérielles.

Pour le module exception, un seul service doit être réalisé, soit :

- Méthode « exception_init »
  Cette méthode initialise le module devra attacher une routine de traitement d'interruption (ISR) par défaut pour chaque vecteur que le programme principal pourra générer.
  On limitera la fonctionnalité de cette ISR par défaut à l'affichage d'un message approprié sur la console de la cible lorsqu'une exception est levée, p. ex.

```
void exception_handler (enum interrupt_vectors vector, void* param)
{
      printf ("ARM Exception(%d): %s\n", vector, (char*)param);
}
```

Il faut étendre la fonctionnalité du programme principal contenue dans le fichier « main.c » afin de permettre la validation de l'implémentation. Les exemples de code permettent de générer une partie des exceptions et interruptions que le µP peut traiter, soit

- Pour générer une « software interrupt » :

```
__asm__ ("svc #1;");
```

- Pour générer une « undefined instruction » :

```
__asm__ (".word 0xffffffff;");
```

- Pour générer un « data abort » dû à un mauvais alignement d'une donnée :

```
long l = 0;
long* pl = (long*)((char*)&l+1);
*pl = 2;
```

- Pour générer un « prefetch abort » :

```
__asm__ ("mov pc,#0x00000000;");
```

  Il est intéressant de noter, que si l'on génère un « prefetch abort » avec l'instruction ci-dessus, le µP n'a plus la possibilité de connaître l'adresse de l'instruction ayant généré cette exception.

Conditions d'exécution

Pour télécharger le squelette du FLIH, mettez simple à jour votre workspace depuis le dépôt centralisé :

```
$ cd ~/workspace/se12/tp/tp.10
```

Systèmes embarqués 2 – TP.10 – Mini Projet OS

Réalisation d'un système d'exploitation

Interrupt Handling – FLIH

```
$ git pull upstream master
```

Cette mise à jour crée un nouveau répertoire « interrupt » et place à l'intérieur le squelette des fichiers pour le traitement des interruptions de bas niveau (First Level Interrupt Handling – FLIH).

Une fois la mise à jour faite, il faudra encore mettre à jour le Makefile situé à la racine du projet (../tp/tp.10/Makefile). Il faut simplement ajouter à la ligne no 5, le répertoire « interrupt », soit :

```
LIBDIRS+=vfs shell io interrupt
```

Sans cette mise à jour, les fichiers placés dans le répertoire « interrupt » ne seront pas pris encore dans le processus de génération de l'application et par conséquent ni compilés ni linké avec l'application.

Le code et le rapport seront rendus au travers du dépôt Git centralisé :

- Sources : *.../tp/tp.10*
- Rapport : *.../tp/tp.10/doc/report_flih.pdf*

Délai

- Le journal et le code doivent être rendus au plus tard 6 jours après le TP à 23h59