Architecture ARM v5 : modes de fonctionnement et exceptions

Stéphane Bonnet

16 avril 2013

Table des matières

1 Introduction

Comme la plupart des processeurs, l'architecture ARM est capable de réagir à des événements exceptionnels ou anormaux indépendants du déroulement normal du programme en exécutant des traitements spécifiques fournis par le programmeur. Ces événements sont appelés *exceptions* et peuvent être de deux types :

Les erreurs – Aussi appelées *fautes* sur d'autres architectures, il s'agit d'erreurs lors de l'exécution d'une instruction qui empêchent son aboutissement (comme l'accès à une donnée dans une zone de mémoire non implémentée ou la tentative d'exécution d'une instruction inconnue). Ces exceptions sont synchrones : elles surviennent à un moment "prévisible" lors de l'exécution d'une instruction et sont dues à des erreurs de logique dans le programme

Les interruptions Il s'agit d'événements asynchrones d'origine externe au processeur. Cette classe d'exceptions permet au programme de traiter des événements urgents ou imprévisibles d'origine externe (c'est à dire indépendants de l'instruction en cours d'exécution) lorsqu'ils surviennent. Cette catégorie inclut aussi l'exception de *reset*, qui survient systématiquement à la mise sous tension du processeur et qui provoque le démarrage de l'exécution à une adresse mémoire conventionnelle, `0x00000000` par défaut sur ARM.

Cette deuxième catégorie est la plus importante dans le développement bas niveau de système embarqués temps-réel ou non : les interruptions permettent au processeur de réagir très rapidement à des changements d'état de périphériques par un traitement spécifique et de reprendre le programme interrompu ensuite. De plus, puisque c'est le processeur (et donc le matériel) qui est chargé de déterminer si un événement externe s'est produit ou non, le programme utilisateur n'a pas à faire cette vérification au travers d'un système de scrutation par attente active. L'écriture des programmes est plus simple (dans le cas ou on doit vérifier l'occurrence de plusieurs événements indépendants par exemple) et la charge du processeur plus faible.

Toutefois, les interruptions ne garantissent pas en général que la latence de traitement d'un événement, c'est à dire le temps qui s'écoule entre sa survenue et le début de son traitement, est plus courte. Lors du traitement d'une interruption, le processeur doit en effet sauvegarder le contexte de l'exécution en cours afin de pouvoir la reprende ensuite, ce qui prend du temps. Aussi, dans des programmes simples, une attente active peut être préférable. Utiliser le programme suivant sera plus rapide dans certains cas que le traitement d'une interruption :

```
    ldr r0, =status @ Charger r0 avec l'adresse
                    @ du registre d'état
attendre:
    ldr r1, [r0]    @ Charger la valeur du registre
                    @ d'état dans r1
    tst r1, #0x80   @ Tester le bit 7 (par exemple)
    beq attendre    @ Attente active tant qu'il vaut 0
    ...
```

Mode		Description
User	usr	Mode d'exécution normal des programmes
System	sys	Mode d'exécution privilégié système
Supervisor	svc	Mode superviseur
Abort	abt	Mode d'erreur de donnée
Undefined	und	Mode d'erreur d'instruction
Interrupt	irq	Mode d'interruption externe générale
Fast interrupt	fiq	Mode d'interruption externe rapide

TABLE 1 – Modes de fonctionnement

2 Modes de fonctionnement

Sur l'architecture ARM v5, le processeur dispose de plusieurs modes de fonctionnement, plus ou moins privilégiés, destinés à établir une hiérarchie entre les différents programmes en exécution (système, application, pilotes de périphériques...) Le mode de fonctionnement se modifie soit manuellement en programmant un changement de mode au travers du registre de contrôle et d'état `CPSR` (*Current Program Status Register*), soit automatiquement lors de la survenue d'une exception.

Il existe sept modes, repris dans le tableau 1. Parmi ceux-ci, on distingue un mode *non privilégié*, qui restreint l'utilisation de certaines instructions comme celles permettant de changer de mode : le mode **User**. Ce mode est utilisé par exemple dans les systèmes multi-utilisateurs et multi-tâches pour empêcher un processus de réaliser des opérations affectant les autres processus.

Tous les autres modes sont *privilégiés* et ont accès à toutes les fonctions du processeur. Ils sont destinés à être mis en œuvre par le noyau d'un système d'exploitation. Dans ces derniers, on distingue le mode d'exécution priviégié normal **System**, qui est supposé être le mode de fonctionnement du système d'exploitation lorsqu'il ne traite pas d'exceptions, et les modes d'exception **Supervisor**, **Abort**, **Undefined**, **IRQ** et **FIQ**. Au démarrage, le mode de fonctionnement est **Supervisor**.

2.1 Changements de mode par programme

Le registre CPSR

Le mode de fonctionnement actuel est programmé par la valeur du champ de cinq bits`M` du registre `CPSR` (figure 1). Il suffit d'écrire une nouvelle valeur dans ce champ pour changer de mode de fonctionnement. Son interprétation est donnée dans le tableau 2.

FIGURE 1 – Structure du registre `CPSR`

M[4 :0]	Mode
0b10000	User
0b11111	System
0b10011	Supervisor
0b10111	Abort
0b11011	Undefined
0b10010	Interrupt
0b10001	Fast interrupt

TABLE 2 – Programmation du mode

Modification de CPSR

Le registre CPSR se modifie au moyen des instructions assembleur `mrs` et `msr`. Ces instructions sont privilégiées.

- L'instruction `mrs` (*Move from PSR*) permet de copier la valeur courante de `CPSR` dans un registre général.
- L'instruction `msr` (*Move to PSR*) permet de copier la valeur d'une partie d'un registre général dans `CPSR`. La partie à copier est donnée par un suffixe ajouté à CPSR dans l'instruction :
  - `CPSR_c` accède aux bits de contrôle (bits 0...7)
  - `CPSR_x` accède aux bits d'extension (bits 8...15)
  - `CPSR_s` accède aux bits d'état (bits 16...23)
  - `CPSR_f` accède aux bits de condition (bits 24 ...31)

On peut combiner les suffixes. Ainsi, écrire `CPSR_cxsf` est équivalent à sélectionner tous les champs.

Pour changer le mode courant, on peut utiliser un programme comme ci-dessous (dans l'exemple on passe dans le mode **Supervisor**) :

```
mrs r0, CPSR      @ Copie CPSR dans r0
bic r0, r0, #0x1f @ Met à 0 les 5 bits M
orr r0, r0, #0x13 @ et change vers le mode superviseur
msr CPSR_c, r0    @ Recharge les bits de contrôle
nop               @ de CPSR
```

3 Banques de registres

Du point de vue d'un programme, un processeur ARM dispose de 16 registres entiers 32 bits (`r0` à `r15`) et d'un registre d'état (`CPSR`). Certains de ces registres sont spécialisés : `r13` est le registre pointeur de pile `sp`, `r14` est le registre de lien `lr` destiné à recevoir l'adresse de retour lors des appels de sous-programmes ou de gestionnaires d'exception et `r15` est le compteur de programme `pc`.

Cependant, certains de ces registres existent en plusieurs exemplaires dans le processeur : les registres physiques sont organisés sous forme de banques de registres dont seuls 16 sont visibles à chaque instant. Ils gardent les mêmes noms, mais ils changent lors des commutations de mode de fonctionnement. Les registres communs à tous les modes et ceux qui sont dépendants du mode de fonctionnement sont repris figure 2.

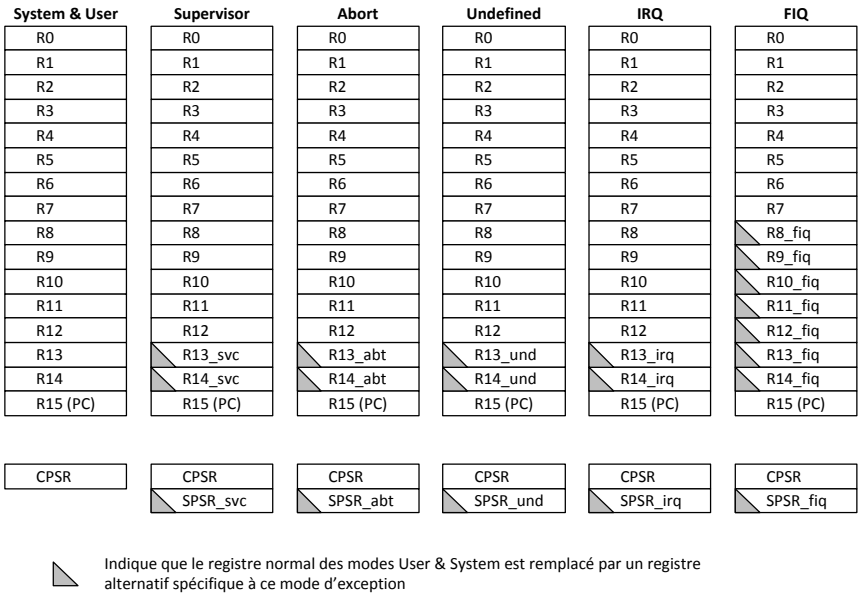

FIGURE 2 – Organisation des registres

Ainsi, si en mode **Supervisor**, on exécute l'instruction `mov sp, #0x4000` et qu'on passe en mode **User**, la valeur du registre `sp` dans ce nouveau mode n'aura pas été affectée par le chargement précédent. L'intérêt essentiel de ce principe est de permette de réduire l'effort nécessaire aux commutations entre modes : puisque ces registres existent en plusieurs exemplaires, il n'est pas nécessaire de les sauvegarder ou de les restituer.

Il existe aussi un registre appelé `SPSR` dans chaque mode d'exception. Il est destiné à recevoir la sauvegarde du registre `CPSR` du programme interrompu. Les instructions `mrs` et `msr` peuvent aussi modifier `SPSR`.

On notera que les modes `User` et `System` partagent la même banque.

3.1 Accès à la banque de registres d'un mode non actif

L'inconvénient de ce mécanisme est que certains registres des modes `usr` et `sys` sont inaccessibles lorsqu'un mode d'exception est actif. Il est toutefois possible de les copier en mémoire en utilisant une forme spéciale des instructions `ldm` (*load multiple*) et `stm` (*store multiple*).

Cette forme, dite *force user mode*, permet d'accéder à la banque `usr` quelque soit le mode d'exception actif[1] et s'indique en postfixant l'instruction de chargement par le caractère "`^`". Par exemple, le programme suivant empile la valeur du registre `lr` du mode `usr` sur la pile active pour le mode d'exception courant :

```
stmfd sp, {lr}^
nop
```

1. bien sûr, ceci n'a pas de sens si le mode est déjà `usr` ou `sys`

Le programme suivant restaure tous les registres `usr` (à l'exception de `pc`) depuis la pile du mode d'exception courant :

```
ldmfd sp, {r0-r14}^
nop
```

Ces instructions provoquent une commuation de la banque de registres active vers la banque `usr` au début de leur exécution. Tout accès en écriture au registre d'index (`sp` dans l'exemple) pendant l'exécution se fait donc *dans la banque* ***usr*** et non dans la banque du mode d'exception actif. Par exemple, en supposant que le mode actuel est `svc`, le programme erroné suivant :

```
ldmfd sp!,{r0-r14}^   @ Comportement indéfini
nop
```

utiliserait `sp_svc` comme adresse où chercher les valeurs des registes `r0` à `r14` mais mettrait à jour le registre `sp_usr` à la fin du transfert, d'où incohérence des valeurs des pointeurs de pile ensuite.

La réactivation de la banque de registres du mode actif ne se produit qu'au premier cycle de l'instruction qui suit le chargement, ce qui rend impossible l'accès aux registres de cette banque dans cette instruction. Pour éviter tout problème, il suffit de faire suivre le chargement d'une instruction *no operation* `nop`, qui est équivalente à `mov r0,r0`.

4 Exceptions

Il existe sept exceptions sur l'architecture ARM (tableau 3). Lorsqu'un des événements correspondants survient, le processeur change de mode d'exécution pour passer dans le mode associé à l'exception et poursuit le traitement en exécutant un gestionnaire d'exception spécifique.

Exception	**Mode associé**	**Adresse vecteur**
Reset	Supervisor	`0x00000000`
Instruction indéfinie	Undefined	`0x00000004`
Interruption logicelle (SWI)	Supervisor	`0x00000008`
Erreur de recherche d'inst.	Abort	`0x0000000C`
Erreur de chargement mémoire	Abort	`0x00000010`
Interruption externe normale (IRQ)	IRQ	`0x00000018`
Interruption externe rapide (FIQ)	FIQ	`0x0000001C`

TABLE 3 – Les sept exceptions

4.1 Table des vecteurs d'exception

Puisque le but des exceptions est de provoquer l'exécution d'un traitement spécifique quand survient un événement inattendu, il faut pouvoir indiquer au processeur à quelle adresse en mémoire se trouve ce programme de traitement, le *gestionnaire d'exception*. A cette fin, le processeur utilise une table en mémoire, conventionnellement placée à l'adresse `0x00000000`, comportant huit entrées

de 32 bits chacune, appelées *vecteurs d'exception*. La sixième entrée n'est pas utilisée. Chaque vecteur est assez grand pour contenir une instruction : lors de la survenue d'un événement, le processeur exécute l'instruction contenue dans le vecteur d'exception associé.

Pour définir cette table, on peut utiliser la structure suivante [2] en assembleur GNU, en prenant soin de faire en sorte que l'étiquette `vectors` soit placée à l'adresse `0x00000000`.

```
vectors:
      b reset          @ Reset vector
      b undefined      @ Undefined instruction
      b soft_interrupt @ Software interrupt
      b prefetch_abort @ Prefetch abort exception
      b data_abort     @ Data abort exception
      b .              @ Reserved vector, not used
      b irq_handler    @ Normal interrupt
      b fiq_handler    @ Fast interrupt
```

Les branchements ARM ne peuvent pas atteindre une instruction située à plus de 32 Mo de l'emplacement de l'instruction de branchement. On peut utiliser des instructions du type `ldr pc,=irq_handler` quand c'est un problème.

4.2 Déroulement d'une exception

Le traitement d'une exception se déroule en trois étapes :

1. Sauvegarde du contexte partiel d'exécution courant par commutation de banque de registres et branchement au vecteur approprié. Ces opérations sont réalisées *automatiquement* par le processeur.
2. Exécution du code du gestionnaire d'interruption (y compris sauvegarde et restauration du reste du contexte d'exécution si nécessaire) fourni par le programmeur
3. Restauration du contexte partiel d'exécution et poursuite de l'exécution à l'instruction interrompue par l'exception.

Il peut y avoir des petites variations entre le traitement des diverses exceptions (en particulier si les interruptions externes IRQ et FIQ restent autorisées ou non lors de l'entrée dans ue gestionnaire d'exception). De plus, les exceptions logicielles SWI diffèrent de ce qui est décrit ci-dessous.

Appel du gestionnaire d'exception

Lors de la survenue d'une exception, le processeur réalise automatiquement les actions suivantes :

- Sauvegarde de `pc` dans `lr_mode`
- Sauvegarde de `CPSR` dans `SPSR_mode`
- Changement de `CPSR` pour commuter dans le mode d'exception approprié
- Chargement de `pc` avec l'adresse du vecteur d'exception approprié.

2. L'assembleur GNU définit le symbole "." comme valant en permanence l'adresse de l'instruction courante

Retour d'exception

Les exceptions sont reconnues après que l'instruction en cours ait été recherchée en mémoire, mais avant qu'elle ne s'exécute. Il faudra faire en sorte qu'elle soit recherchée à nouveau pour être exécutée lors de la sortie du gestionnaire d'exception, sans quoi on sauterait une instruction dans le programme interrompu. Il faut donc que le retour se fasse à l'adresse sauvegardée *moins 4*.

Pour provoquer un retour d'exception, il faut réaliser les opérations suivantes :

- Restauration de `CPSR` à partir de `SPSR_mode`
- Affectation à `pc` de la valeur contenue dans `lr_mode`

Pour réaliser le retour, il faut utiliser l'instruction suivante :

```
subs pc, lr, #4          @ Retour de routine d'exception
```

En règle générale, le suffixe "s" ajouté aux instructions arithmétiques signifie qu'elles doivent mettre à jour les drapeaux de condition de CPSR en fonction du résultat (ce qui n'est pas réalisé par défaut). Dans le cas précis où la destination est `pc`, ce suffixe indique qu'il faut resaurer `CPSR` à partir de `SPSR_mode`.

Une autre méthode peut être utilisée pour effectuer un retour d'exception, dans le cas où le gestionnaire à utilisé la pile pour compléter la sauvegarde de contexte :

```
irq_handler:
      sub   lr, lr, #4       @ Ajustement de l'adresse de retour
      stmfd sp!,{r0-r12,lr}  @ Sauvegarde du contexte restant

            ...
            ...

      ldmfd sp!,{r0-r12,pc}^ @ Restauration de r0,r1,...r12
                             @ et retour
```

Dans le cas de cette dernière instruction, le symbole "`^`" indique qu'on veut en plus que l'instruction restaure `CPRS` à partir de `SPSR` dans le cas où `pc` fait partie des registres restaurés à partir de la pile [3].

4.3 Exception logicielle SWI

Il s'agit d'une exception particulière car elle est explicitement provoquée par l'exécution d'une instruction spécifique : `svc`. Un programme qui fonctionne en mode **User** ne peut changer explicitement le mode de fonctionnement, puisque `mrs` et `mrs` sont des instructions privilégiées. La seule possibilité qu'il a est de provoquer une exception : c'est le rôle de l'exception SWI. Cette instruction permet donc d'implémenter facilement un changement de niveau de privilèges pour réaliser des appels systèmes.

Puisque dans ce cas, l'instruction qui a provoqué l'exception est l'instruction interrompue elle-même (`svc`), il ne faut pas qu'au retour du traitement

3. En fait, ce suffixe indique le positionnement du bit S dans l'encodage de l'instruction. Ce bit est interprété comme *force user mode* si `pc` ne fait pas partie des registres à charger et comme *restore SPSR* s'il en fait partie. Dans les autres instructions, il indique que l'instruction doit mettre à jour les drapeaux d'état

d'exception elle soit exécutée à nouveau (sans quoi l'exception SWI serait immédiatement redéclenchée). Contrairement aux autres exceptions, il ne faut pas décrémenter de 4 l'adresse de retour sauvegardée. On utilise le code suivant pour le retour d'exception :

```
movs pc, lr
```

Enfin, l'instruction `svc` admet un opérande immédiat encodé sur 24 bits. Il suffit d'examiner le code de l'instruction en mémoire pour extraire cet opérande, ce qui permet de différencier 2^{24} appels système différents.

4.4 Interruptions externes IRQ et FIQ

Ces exceptions sont provoquées par l'affirmation d'un signal externe par un périphérique et sont traitées comme toutes les autres exceptions.

Lors de la survenue d'une interruption IRQ, le processeur désactive la prise en compte de ces interruptions afin d'éviter qu'un nouveau traitement d'interruption vienne interrompre celui en cours : par défaut, les IRQ ne peuvent pas être imbriquées. En revanche, les interruptions FIQ restent autorisées si elles l'étaient au départ.

Dans le cas des interruptions FIQ, IRQ et FIQ sont toutes deux interdites automatiquement lors du traitement d'exception.

La différence entre IRQ et FIQ tient au nombre de registres automatiquement suabegardés dans chaque mode : puisque en mode **FIQ** plus de registres sont sauvegardés, le gestionnaire d'interruption a moins de travail a réaliser, ce qui minimise le temps nécessaire au traitement de l'exception.

Autorisation et interdiction des IRQ et FIQ

Deux bits du registe `CPSR` (figure 1) contrôlent la prise en compte des interruptions externes : les bits `I` et `F`.

Bit `I` – Indique si les interruptions IRQ sont autorisées.
- 0 : irq autorisées
- 1 : irq interdites

Bit `F` – Indique si les interruptions FIQ sont autorisées.
- 0 : fiq autorisées
- 1 : fiq interdites

Ces bits sont positionnés automatiquement lors d'un reset : les interruptions externes sont interdites par défaut.

5 Support des exceptions par gcc

5.1 Gestionnaires d'interruption

GCC permet l'écriture directe de gestionnaires d'exception en C. Il suffit de déclarer une fonction comme telle :

```
void __attribute__((interrupt("IRQ"))) irq_handler(void) {
 /* Code du gestionnaire d'interruption IRQ */
}

void __attribute__((interrupt("FIQ"))) fiq_handler(void) {
 /* Code du gestionnaire d'interruption FIQ */
}
```

Dans le cas d'un gestionnaire d'IRQ par exemple, le code généré [4] est le suivant :

```
irq_handler:
   @ Interrupt Service Routine
   @ args = 0, pretend = 0, frame = 8
   @ frame_needed = 1, uses_anonymous_args = 0
   sub lr, lr, #4
   stmfd sp!, {r2, r3, lr}

   ldr    r3, =var
   ldr r2, [r3]
   add r2, r2, #1
   str r2, [r3]

   ldmfd sp!, {r2, r3, pc}^
```

Le fait de déclarer une fonction comme gestionnaire d'interruption indique à gcc qu'il doit sauvegarder les registres utilisés sur la pile et générer une instruction de retour d'exception à la place du traditionnel `bx lr`.

Puisque ce gestionnaire s'exécute en mode **IRQ** ou **FIQ**, toutes les opérations sur la pile se font en utilisant le registre `r13_irq` ou `r13_fiq` : il faut que leur valeur initiale soit correcte pour que l'exécution se déroule sans problèmes.

5.2 Fonctions nues

Une autre technique peut être utilisée pour réaliser un gestionnaire d'interruption. Au lieu de laisser gcc générer le code approprié, on peut lui imposer de ne générer que le code lié aux instructions fournies par l'utilisateur : il s'agit d'une fonction nue, dénuée de prologue et d'épilogue.

La déclaration d'une fonction nue de fait comme suit :

```
void __attribute__((naked)) irq_handler(void) {
  /* Code du gestionnaire d'interruption */
}
```

4. Ce code peut différer en fonction du contenu du gestionnaire

Le code généré est le suivant :

```
irq_handler:
   @ Naked function: prologue and epilogue provided by the
   @ programmer
   @ args = 0, pretend = 0, frame = 8
   @ frame_needed = 1, uses_anonymous_args = 0

   ldr    r3, =var
   ldr    r2, [r3]
   add    r2, r2, #1
   str    r2, [r3]
```

On constate que gcc n'a généré que le code lié à l'algorithme implémenté. Il n'y a aucune instruction de sauvegarde de contexte ou de retour de fonction. L'avantage de ce type de fonction est de permettre au programmeur de contrôler intégralement le code émis. L'inconvénient est que les opérations ajoutées habituelleement par gcc de manière automatique doivent être définies explicitement.

A cette fin, on peut utiliser la possibilités d'inclure directement du code assembleur dans le code C en utilisant l'assembleur *inline*.

5.3 Assembleur *inline*

Il s'agit d'une fonctionnalité de gcc (et d'autres compilateurs), qui permet d'insérer directement au sein des instructions C/C++ des instructions assembleur au moyen de l'instruction `asm` ou `__asm__`, qui sont strictement équivalentes.

Un exemple simple :

```
void do_nothing(void) {
   asm("mov  r0, r0");
  }
```

Un exemple un peu plus compliqué :

```
void do_nothing_4_times(void) {
   asm("mov  r0, r0\t\n"
       "mov  r0, r0\t\n"
       "mov  r0, r0\t\n"
       "mov  r0, r0\t\n");
  }
```

Le format général une instruction assembleur inline est :

```
asm(instructions : sorties : entrées : registres écrasés);
```

Elle comprend les quatre éléments suivants :

- Les instructions assembleur, qui sont données dans une chaine unique.
- Une liste optionnelle d'opérandes d'entrée
- Une liste optionnelle d'opérandes de sortie
- Une liste optionnelle de registres qui sont écrasés par le code assembleur, séparés par des virgules

Un exemple encore plus compliqué :

```
int do_something(int x) {
  int y;

  asm("mov  r0, %[input], lsl #1\t\n"
      "add  %[output], %[input], r0\t\n"
      : [output]"=r" (y)
      : [input]"r" (x)
      : "r0");
  return(y);
}
```

Ici, on indique que la variable `x` est une entrée, qu'on veut qu'elle soit placée dans un registre libre par le compilateur (`"r"`) et que ce registre est représenté par `%[input]` dans le code assembleur. De même, on indique que la variable `y` est une sortie, qu'on veut qu'elle soit placée dans un registre libre par le compilateur qui ne sera utilisé qu'en écriture (`"=r"`) et que ce registre est représenté par `%[output]` dans le code assembleur. Enfin, on indique que le code assembleur utilise le registre `r0`, ce qui suggère vivement au compilateur de ne pas l'utiliser, ou de le sauvegarder avant le code assembleur.

Il existe bien d'autres possiblités. Vous pouvez par exemple vous rendre sur la page `http://www.ethernut.de/en/documents/arm-inline-asm.html` pour plus de détails.

Quelques exemples utiles

Si on veut réutiliser facilement un morceau de code assembleur qui fait quelque chose d'utile, on peut l'inclure dans une macro. Par exemple, les macros ci-dessous permettent d'activer / désactiver les interruptions :

```
#define irq_enable()           \
   asm volatile(               \
      "mrs    r0,cpsr\t\n"     \
      "bic    r0,r0,#0x80\t\n" \
      "msr    cpsr_c,r0\t\n"   \
      :                        \
      :                        \
      : "r0")

#define irq_disable()          \
   asm volatile(               \
      "mrs    r0,cpsr\t\n"     \
      "orr    r0,r0,#0x80\t\n" \
      "msr    cpsr_c,r0\t\n"   \
      :                        \
      :                        \
      : "r0")
```

Les caractères "tabulation" et "saut de ligne" `\t` et `\r` présents à chaque ligne sont indispensables pour éviter des erreurs de compilation. Le mot-clé

`volatile` empêche le compilateur de tenter d'optimiser le code assembleur s'il n'a pas d'effet sur les données du programme.

6 Remarques

Ce document n'est qu'une brève introduction. Pour plus de détails, vous pouvez vous référer à la documentation technique "ARM Architecture Reference Manual" que vous trouverez sur le site web de l'UV. Le site `http://infocenter.arm.com/help/index.jsp` est aussi une mine d'informations qui regroupe toutes les documentations, notes d'applications, références, ...publiées par ARM (les parties croustillantes nécessitent un enregistrement gratuit).