

TP 1

I - Le makefile

I.1 - Partie configurable

C'est une partie du makefile où on peut configurer nos paramètres si nos besoins changent dans le futur. Voici la liste des configurations :

- **BOARD=versatile** : Variable qui définit la carte matérielle cible où les programmes vont s'exécuter dont la valeur est versatile. Cela correspond à versatileAB, une carte de développement pour ARM. Cette variable est utilisée par QEMU.
- **CPU=cortex-a8** : Indique le CPU pour pouvoir compiler le code pour un processeur cortex-a8. Cette variable est utilisée pour la compilation.
- **TOOLCHAIN=arm-none-eabi** : définit une chaîne de compilation. C'est un ensemble d'outils pour compiler, assembler, lier et générer un exécutable pour une architecture cible. Dans **arm-none-eabi**, **arm** désigne l'architecture ARM. **none** indique qu'il n'y a pas de système d'exploitation et **eabi** désigne l'ABI (Application Binary Interface). C'est une convention d'interface binaire utilisée pour les systèmes embarqués. Ça définit notamment comment on gère les appels de fonction avec le retour et le passage des arguments, mais aussi pour structurer les registres et la mémoire, la gestion de la pile ou des règles d'alignements. L'ordre des paramètres de la toolchain sont importants. Il peut avoir un 4 argument qui peut être par exemple un compilateur comme gcc, un linker -ld ou d'autres outils.
- **DEBUG?=yes** : La compilation, le linkage et pour transformer les fichiers assembleur en fichier objets auront les options de debug pour pouvoir faire fonctionner gdb correctement. Le ? signifie que DEBUG aura la valeur **yes** seulement si DEBUG n'a pas été définie avant dans le fichier ou lors de l'appel du make (par exemple : make DEBUG=no).
- **BUILD=build/** : indique le répertoire dans lequel seront placés les fichiers compilés.
- **objs= startup.o main.o exception.o uart.o** : indique de manière pas propre la liste des fichiers objets utilisée pour faire l'exécutable final. Petite note : cela n'indique pas nécessairement des fichiers existants, en effet, ils sont créés lors de la compilation ou de la transformation d'un fichier en assembleur en fichier objet.

I.2 - Partie non configurable

On se concentre sur la partie non configurable du makefile. Ici, toute la configuration concerne la carte matérielle versatile. On pourrait très bien avoir d'autres configurations pour d'autre type de carte.

- **VGA=-nographic** : permet de désactiver l'affichage graphique dans QEMU.
- **SERIAL=-serial mon :stdio** : indique à QEMU de rediriger les messages de la console vers l'entrée/sortie standard (stdio). -serial c'est pour ouvrir une connexion série, c'est un type de connexion pour envoyer des bits de manière séquentiel.
- **MEMSIZE=32** : sera la taille de la mémoire de QEMU
- **MEMORY="\$(MEMSIZE)K"** : ici, c'est juste transformer la valeur numérique en string avec le suffixe K pour kilooctets. Ainsi MEMORY est une variable qui sera utilisée pour définir la taille mémoire de QEMU, ici 32Ko.
- **MACHINE=versatileab** : définit un type de machine virtuel pour QEMU. À ne pas totalement confondre avec la variable BOARD. Ici la différence entre les deux C'est une question d'organisation du makefile, Ici, le makefile semble prévu pour changer de BOARD facilement, ce qui explique l'utilisation de BOARD seulement dans des conditions *ifeq \$(BOARD),versatile*). Ainsi si on veut programmer sur plusieurs cartes

différentes, on aurait juste à changer la valeur de BOARD qu'on donne en entrée du makefile sans devoir refaire, tous les arguments potentiellement différent dans la partie non configurable.

- **QEMU=qemu-system-arm** : définit le programme de QEMU qui permet d'émuler les systèmes ARM.
- **QEMU_ARGS=-M \$(MACHINE) -cpu \$(CPU) -m \$(MEMORY) \$(VGA) \$(SERIAL)** : La variable possède tous les arguments qui seront passés à QEMU grâce aux variables définies précédemment.
- **CFLAGS=-c -mcpu=\$(CPU) -nostdlib -ffreestanding -DCPU=\$(CPU) -DMEMORY="\$(MEMSIZE)*1024** : définit les arguments du compilateur, CFLAGS est d'ailleurs le nom de la variable utilisé par la règle implicite par Makefile pour compiler si on ne met aucune règle de compilation (ce qui rend cet outil incompréhensible pour les nouveaux qui apprennent à l'utiliser).
 - **-c** : indique à GCC de compiler les fichiers source en fichiers objet sans les lier.
 - **-mcpu=\$(CPU)** : permet d'optimiser le code et d'indiquer le processeur ciblé, ici cortex-a8.
 - **-nostdlib** : Pour ne pas utiliser la bibliothèque standard de C.
 - **-ffreestanding** : indique au compilateur qu'on n'utilise pas de librairie standard et qu'on est indépendant d'un OS.
 - **-DCPU=\$(CPU)** : C'est une macro pour le code C, c'est un define CPU cortex-a8.
 - **-DMEMORY="\$(MEMSIZE)*1024"** : définit une macro dans le code avec la taille mémoire en octets.
- **ASFLAGS= -mcpu=\$(CPU)** : Comme pour la compilation, mais c'est pour la transformation fichier assembleur en fichier objets.
- **LDFLAGS= -T kernel.ld -nostdlib -static** :
 - **-T kernel.ld** : spécifie un fichier de script de lien pour définir la disposition de la mémoire des différentes sections de code.
 - **-nostdlib** : n'utilise pas de bibliothèque standard libre.
 - **-static** : générer un fichier exécutable (binaire) sans bibliothèque dynamique, c'est-à-dire que les bibliothèques sont static et donc incluse directement dans le fichier exécutable.
- **ifeq \$(DEBUG),yes) CFLAGS+= -ggdb LDFLAGS+= -g ASFLAGS+= -g endif** : Rajoute les options de debug de gdb à la compilation, au linkage et à l'assembleur pour pouvoir faire fonctionner gdb lorsqu'on veut l'utiliser via la variable DEBUG (ou une macro avec make DEBUG=yes).
- **ifndef MACHINE \$(error Must choose a board (e.g. Versatile AB or PB)) endif** : Arrête la compilation si on n'a pas défini la carte cible (via la variable MACHINE). Je suppose que c'est pour avoir une erreur plus explicite au cas où MACHINE n'a pas été définie, mais je suppose que qemu pourrait très bien nous renvoyer l'erreur, donc c'est quoi l'intérêt ?

I.3 Règles de compilation

On se concentre sur les deux règles de compilations, on a deux types de fichier à transformer en fichier objet, du code en langage C et du code en langage assembleur ARM.

- **\$(BUILD).o : %.c**
\$(TOOLCHAIN)-gcc \$(CFLAGS) -o \$@ \$< : permet de compiler les programmes C des fichiers .c en fichiers objets en .o. Ici, on utilise le compilateur gcc (concaténer à la toolchain) pour compiler du code C.
- **\$(BUILD).o : %.s**
\$(TOOLCHAIN)-as \$(CFLAGS) -o \$@ \$< : permet de compiler les programmes assembleur des fichiers .s en fichiers objets Ici, on utilise l'outil -as pour Assembleur (concaténer à la toolchain) pour transformer le fichier assembleur en fichier objet.

Plusieurs choses à dire sur le Makefile :

- la structure d'une règle du make est de la forme :
target : dependances
command
 - **target** : La cible, c'est à dire le fichier ou une action que l'on souhaite construire
 - **dependances** : Ce sont toutes les dépendances nécessaires pour faire l'action ou pour la construction du fichier cible. Si la date de création de la dépendance est plus récente que la cible, Make réexécutera la règle. De plus, si la dépendance n'est pas encore créée, Make essaiera d'abord d'exécuter les règles nécessaires pour créer la dépendance.
 - **commands** : liste des commandes à exécuter sur une règle.
- le % est une partie d'un nom de fichier présent dans le répertoire courant (ou le make est exécuté), pour

chaque fichier trouvé correspondant à la cible, il exécutera la règle avec tous les % ayant la même valeur dans la règle.

- `$$` est une variable automatique qui correspond au nom du fichier cible, par exemple `build/main.o` avec % ayant `main` comme valeur
- `$$<` correspond à la première dépendance de la règle. C'est aussi une variable automatique de Make.

I.3 Construction et linkage du tout

Cette partie correspond aux règles de construction finaux de notre exécutable.

- **all : build \$(BUILD)kernel.elf \$(BUILD)kernel.bin** : c'est la première règle appelée par défaut par le make. Dans cette règle il faut faire d'abord construire les 3 dépendances qui sont créées par les règles suivantes :
- **\$(BUILD)kernel.elf : \$(OBJS)**
\$(TOOLCHAIN)-ld \$(LDFLAGS) \$(OBJS) -o \$(BUILD)kernel.elf
Création du fichier exécutable `kernel.elf` à partir des fichiers objets (`$(OBJS)`). Ici, on est à l'étape du linkage, la compilation a déjà été faite avant d'où l'utilisation des options de linkage (de `LDFLAGS`).
- **\$(BUILD)kernel.bin : \$(BUILD)kernel.elf** :
\$(TOOLCHAIN)-objcopy -O binary \$(BUILD)kernel.elf \$(BUILD)kernel.bin
Permet la création du fichier `kernel.bin` qui ne contient pas les informations de débogage de gdb. C'est un fichier exécutable en binaire sans les métadonnées grâce à l'option `-objcopy`. Ici la carte n'a pas besoin de toutes les métadonnées pour s'exécuter donc on les enlève.
- **build :**
@mkdir \$(BUILD)
Création du répertoire définit par la variable `BUILD`, c'est à dire `build`
- **clean :**
rm -rf \$(BUILD)
Pour nettoyer tout les fichiers générés lors de la compilation et du linkage afin de garder seulement le code non compilé.

I.5 Lancer QEMU

Cette partie s'occupe des règles pour lancer `qemu` avec ou sans debug, via `make run` ou `make debug`. On n'exécute un des 2 règles seulement quand la `BOARD` est une carte versatile.

- **run : all**
\$(QEMU) \$(QEMU_ARGS) -device loader,file=\$(BUILD)kernel.elf
Lance `QEMU` avec le fichier `kernel.elf` sans le mode débogage. En réalité, il y a quand même la compilation, le linkage et l'assembleur avec gdb si on fait **make run** tout simplement. Pour avoir aucune trace de gdb, il faut faire **make run DEBUG=no**. `loader` est un chargeur de fichier binaire.
- **debug : all**
\$(QEMU) \$(QEMU_ARGS) -device loader,file=\$(BUILD)kernel.elf -gdb tcp : :1234 -S
C'est la règle pour exécuter avec le mode de débogage, ici on rajoute l'option **-gdb tcp : :1234** qui indique à `QEMU` d'attendre une connexion de GDB sur le port 1234 afin de pouvoir déboguer. `-S` permet à `QEMU` d'attendre gdb avant de lancer le programme.

II - Le programme

II.1 Le startup.s et exception.s

Lorsqu'on lance `QEMU`, le processeur exécute l'instruction à l'adresse `0x00000000`. Dans notre cas, on charge la table des vecteurs à l'adresse `0x0000 0000`, cette table des vecteurs se trouve dans le fichier `exception.s`. Mais alors pourquoi c'est cette table des vecteurs est chargée en premier et pas autre chose. La raison est que lors du linkage, on a utilisé un script appelé `kernel.ld`. Dans ce dernier, on indique comment on assigne les adresses des différentes sections du programme. Dans ce script on voit ces lignes de codes :

```
. = 0x0;  
.text : {  
build/exception.o(.text)
```

build/startup.o(.text)

} on voit que exception.o est mis en premier a l'adresse 0x00000000, et que les premières instructions du fichier est la table des vecteurs.

La première ligne de cette table de vecteur est **ldr pc, reset_handler_addr**, il charge donc dans le registre Program counter (PC) l'adresse de reset_handler. Pour rappel PC contient l'adresse de la prochaine instruction à exécuter. Ainsi il va exécuter la fonction dans startup.s indiquée par l'étiquette **_reset_handler**. Cette étiquette dans le fichier exception.s est connue car on a mis la directive **.global _reset_handler** qui permet de rendre visible aux autres fichiers l'étiquette lors l'édition de lien. Voici le code de **_reset_handler** :

- **msr cpsr_c, (CPSR_SYS_MODE | CPSR_IRQ_FLAG | CPSR_FIQ_FLAG)** : permet de mettre le processeur en mode système et désactive les interruptions IRQ et FIQ. les interruption IRQ (interrupt request) sont des interruption matérielle utilisés par les périphériques comme UART par exemple. FIQ (Fast interrupt request) sont un autre type d'interruption prioritaire pour des accès mémoire critique par exemple.
- **ldr sp, =stack_top** : permet de charger le sommet de pile à 4KB après la dernière section bss. **stack_pop** est définie à la fin du fichier kernel.ld utilisé pour configurer la mémoire. On a donc une pile de 4KB.
- **ldr r4, =_bss_start**
 ldr r9, =_bss_end
 mov r5, 0

 1 :
 stmia r4!, r5
 cmp r4, r9
 blo 1b

 Permet à l'aide d'une boucle de mettre toute la section bss à 0. Cette section contient tous les variables globales et statiques et toutes les variables doivent être initialisées à 0, c'est une norme de C. Or en RAM, il peut rester des anciennes valeurs qui ne sont pas 0.
- **eor r11, r11, r11** On efface le frame pointer, utilisé pour les retours de fonctions. Quand la valeur est null, ça indique le début de la pile, nécessaire pour GDB.
- **ldr r3, =_start**
 blx r3

 charge et saute au point d'entrée du programme C, ici c'est la fonction **_start** présent dans le fichier main.c.
- **halt :**
 b .

 C'est au cas où le programme C se termine, via GDB, on verra cette boucle infinie

II.2 Le main

On a une première fonction **void check_stacks()** qui vérifie si la pile dépasse la mémoire allouée, elle est appelée au tout début du **_start** qui pour faire simple est le "main()" du programme même si c'est pas tout à fait le premier code exécuté. La fonction **check_stacks** utilise la macro **MEMORY** définie dans le makefile à la pré-compilation à l'aide de **-DMEMORY** définit à **32*1024**, d'où la raison que le **define** n'apparaît pas dans le code des différents fichiers. Il regarde si l'adresse du sommet de la pile au tout début de l'exécution du programme C est toujours dans la zone mémoire allouée du programme (de 32KB). Si on est pas dans la zone, C'est qu'on a un début de stack en dehors de la zone mémoire allouée au programme. Cela peut arriver si le code compilé est beaucoup trop gros. Par conséquent, il faut arrêter le programme, on appelle ainsi **panic** qui est une boucle infinie. **Panic** permet donc lorsqu'on debug avec **gdb**, de comprendre qu'on a eu un problème.

La fonction **void _start(void)** est essentiellement le **main** de notre programme. Il est appelé par **_reset_handler** dans **startup.s**, c'est notre point d'entrée de notre programme C, il vérifie si la pile est bien dans la zone allouée. Puis il initialise les UARTs avec la fonction **wart_init()**. et on active UART0, c'est pour les interruptions mais on reparlera d'uart plus tard dans ce rapport. Ensuite on a une boucle infinie qui fait de l'attente active pour recevoir un caractère de l'utilisateur via la méthode **uart_receive(UART0, &c)** et une fois reçu, le programme l'affiche avec la fonction **uart_send(UART0, c)**.

Dans notre header **main.h**, nous avons un ensemble de définition de fonction permettant de lire ou d'écrire dans la

mémoire dans la zone MMIO (Memory-Mapped I/O) pour communiquer avec les périphéries. Toutes les fonctions définies ont plusieurs caractéristiques en commun :

- l'attribut **__inline** qui indique au compilateur d'inclure le code de la fonction directement là où elle est utilisé sans faire d'appel de fonction. Cela permet de gagner en performance en nombre d'instruction assembleur mais aussi peut dans certains cas éviter de faire grossir la pile d'appel de fonction qui pourrait faire un stack overflow.
- l'attribut **__attribute__((always_inline))** permet de forcer l'utilisation du inline par le compilateur, même si la fonction est potentiellement grosse.
- les paramètres de fonction **bar** et **offset** qui sont respectivement l'adresse de base de la zone MMIO et offset le déplacement en octet dans cette zone. Ce qui permet de lire ou d'écrire à une adresse très précise.

Dans cette ensemble de fonction on retrouve :

- 3 fonctions de lecture appelés **uintX_t mmio_readX(void *bar, uintX_t offset)** de 8, 16 ou 32 bits qui a lit à partir de l'adresse bar+offset et renvoie la valeur lu.
- 3 fonctions d'écriture appelés **'void mmio_writeX(void *bar, uintX_t offset, uintX_t value)** de 8, 16 ou 32 bits qui écrit à partir de l'adresse bar+offset.
- La fonction **void mmio_set(void* bar, uint32_t offset, uint32_t bits)** : Permet de passer certains bits à 1 à l'adresse mémoire bar+offset à l'aide du paramètre bits en utilisant un mask.
- **void mmio_clear(void* bar, uint32_t offset, uint32_t bits)** Permet d'effacer certains bits à l'adresse mémoire bar+offset à l'aide du paramètre bits en utilisant un mask
- La définition de la fonction **void panic()** qui comme indiqué plus haut savoir lorsqu'on a eu un problème avec la pile au démarrage du programme C.
- La définition de la fonction **void kprintf(const char *fmt, ...)** dont je n'ai aucune idée de ce qu'elle fait.

II.3 UART et la MMIO

UART est un protocole de communication série qui permet d'échanger des informations entre le micro-contrôleur et un périphérique comme un ordinateur. Dans mon cas, ici, on communique entre QEMU et le terminal. On définit une structure uart comme suit :

```
struct uart {
    uint8_t uartno;
    void* bar};
```

Avec **uartno** une constante différente pour chaque uart pour les différencier. C'est un index dans le tableau uarts. On définit le uartno de chaque uart par une macro présent dans le fichier **uart.h**. Pour Bar (Base adresse register) qui est l'adresse de base du uart, il est unique pour chaque uart. On a aussi une macro définie dans le fichier **uart-mmio.h** On retrouve toutes les adresses dans la documentation :

Table 4-1 Memory map (continued)

Peripheral	Location	Interrupt ^a PIC and SIC	Address	Region size
Smart Card 0 Interface	Dev. chip	PIC 15	0x101F0000– 0x101F0FFF	4KB
UART 0 Interface	Dev. chip	PIC 12	0x101F1000– 0x101F1FFF	4KB
UART 1 Interface	Dev. chip	PIC 13	0x101F2000– 0x101F2FFF	4KB
UART 2 Interface	Dev. chip	PIC 14	0x101F3000– 0x101F3FFF	4KB

FIGURE 1 – Versatile Application Baseboard ARM926EJ-S User GUIDE, p.140

On voit les plages d'adresse de chaque uart, pour définir nos macro pour la BAR de chaque uart, on récupère la première adresse de chaque plage. d'où les macros suivantes dans le code :

```

define UART0_BASE_ADDRESS ((void*)0x101F1000)
define UART1_BASE_ADDRESS ((void*)0x101F2000)
define UART2_BASE_ADDRESS ((void*)0x101F3000)

```

Dans ces plages d'adresse, on a 2 adresses spécifiques :

- Data register à l'offset 0x00 qui stock la donnée à envoyer ou recut.
- Flag Register à l'offset 0x18 indique le statut de l'uart pour savoir s'il est prêt à recevoir ou à envoyer des données ou à recevoir des données. Dans ce même registre, au bit à la position 4 (en partant de droite), on a l'information sur l'envoi de donnée. Si le bit est à 1, cela signifie que la donnée n'est pas prête à être lu. Lorsque le bit vaut 0, il peut être lu. Un autre bit nous intéresse, celui à la position 1 qui indique si on peut envoyer la donnée ou pas.

Ainsi, grâce à tout ce qui est dit au dessus. On peut lister les fonctions présentes dans le fichier `uart.c` :

- **`void uart_init(uint32_t uartno, void* bar)`** : Permet d'initialiser un seul uart avec son index dans le tableau et son adresse de base de sa plage d'adresses.
- **`void uarts_init()`** : initialise les 3 uarts avec leurs index et leurs BARs
- **`void uart_enable(uint32_t uartno)`** : Devrait activer les interruptions du uart spécifié mais cela reste légèrement flou pour moi.
- **`void uart_disable(uint32_t uartno)`** : Devrait désactiver les interruptions du art spécifié, mais comme pour la fonction précédente, cela reste encore flou pour moi.
- **`void uart_receive(uint8_t uartno, char *pt)`** Fait une attente active pour recevoir la donnée à l'adresse prévu du uart comme vu précédemment. L'attente se fait sur le bit prévu du registre flag.
- **`void uart_send(uint8_t uartno, char s)`** Fait une attente active pour envoyer la donnée à l'adresse prévu du uart, et comme pour recevoir, on attend un bit spécifique du registre de flag comme vu précédemment
- **`void uart_send_string(uint8_t uartno, const char *s)`** Permet d'envoyer une chaîne de caractères complète qui se finit par le caractère de fin .

Les questions

- Est-ce normal que lorsqu'on fait `make run`, on a quand même les options de `gdb` à la compilation, le `linkage` et à la conversion de l'assembleur en objet. ?
- pourquoi avoir mis des `ifeq` pour vérifier que le board est bien en versatile, si on a aucun autre type de board avec lequel on travaille ?
- A quoi sert la fonction définie dans le `main.h` `kprintf(const char * fmt,...)???`
- A quoi ça sert d'avoir des plages d'adresse pour les uarts aussi énorme ?
- clarifier le lien avec les uart et les interruptions, cela reste encore flou.
- pk dans le code par défaut dans le `main.h` on envoie un `uint32_t` alors que ça retourne un `uint8_t` dans l'implémentation pour une des fonctions.

TP 2

Le problème actuel du projet à la fin de l'étape 1 est l'utilisation d'attente active, c'est à dire l'utilisation une boucle while/for qui consomme du temps processeur alors qu'il ne fait rien. L'objectif avec les interruptions est d'endormir le processeur jusqu'à ce que le périphérie envoie une donnée envoie une interruption qui réveillera le processeur.

I - Les interruptions

Pour pouvoir utiliser les interruptions, il faut les activer a différent niveaux :

- Au niveau du périphérie (device) au niveau du UART dans des registres spécifiques.
- Au niveau du VIC qui est un contrôleur central des interruptions de la carte.
- Au niveau du CPU, c'est un composant différent du CPU sur la carte ou le code est executé.

I.1 - Interruption au niveau du UART

Pour activer les périphéries, notamment lorsqu'on recois des données du terminal, il faut connaitre ou modifier dans les registres.

- **UART_IMSC** est un registre ou chaque bit correspond à l'activation d'un type d'interruption sur l'interface d'un UART. Il est défini a l'offset **0x038** à partir de la base adress de l'uart concerné.
- Le bit numéro 4 : Lorsqu'il est à 1, il permet d'activer la réception d'interruption lorsqu'une donnée est reçu dans la FIFO de reception dont la macro est **UART_IMSC_RXIM**
- Le bit numéro 5 : (non utilisé dans le code du step2) active l'interruption d'émission (TX)

Par conséquence, dans le code, on rajoute la fonction **void uar_enable(const uint32_t uartno)** qui permet d'activer les interruptions au niveau du device et qui va donc écrire a cette endroit : **mmio_write32(uart->bar, UART_IMSC, UART_IMSC_RXIM);**.

I.2 - Interruption au niveau du VIC

Pour comprendre comment fonctionne les interruptions au niveau du VIC, Voici une explication rapide et utile des registres et information nécessaires :

- **VIC_BASE_ADDR** Tout comme les UARST, Il y a une zone mémoire avec différents registres pour controler le VIC. Et donc il y a une base adresse mémoire. Tous les points suivant sont des registres.
- **VICRAWSTATUS** registre contenant les interruptions actives. Si le VIC recoit une demande d'interruption d'un device. C'est ici que le bit passera à 1 pour indiquer d'une irq. (autorisé ou non)
- registre indiquant les interruptions autorisés par des bits à 1.
- **VICIRQSTATUS** Il indique quelles interruptions sont actives **ET** autorisé en faisant un & entre les 2 registres précédant. Il permet de transmettre au CPU. La macro définit ne sert a rien hormis a nous faire chercher dans la documentation. Le signal est effectué au niveau matériel et la fonction **_isr_handler** est appelé automatiquement.
- **VICINTSELECT** Permet de selectionner le type d'interruption, IRQ ou IFQ qui est une interruption plus rapide.
- **VICINTCLEAR** registre pour indiquer que l'interruption a été traité. Lorsque le bit passe à 1, ca effacera le bit dans le registre **VICINTENABLE**.

II - Traitement d'une interruption et modification du code

II.1 Modification du code

Il y a beaucoup de modification à faire pour pouvoir traiter des interruptions. Premièrement dans le makefile, on rajoute les nouveaux fichiers objets **irq.o isr.o** à la liste déjà existante.

Deuxièmement, dans le script de linkage, on doit rajouter une pile pour les interruptions dont le nom du sommet de cette pile est **irq_stack_top**. C'est une nouvelle pile qui permet d'enregistrer tous les registres du CPU qui peuvent potentiellement être modifiés lors de l'interruption. Dans mon cas, j'ai rajouté la pile d'interruption 4Kb après la pile d'exécution du programme ce qui probablement beaucoup. Il faudrait aussi modifier la fonction **void check_stacks()** pour vérifier que le sommet de pile au début du programme reste dans les 32Kb alloués au programme. (ce que je n'ai pas fait dans le code).

Troisièmement, dans la fonction du point d'entrée du programme C **_start(void)** :

- on autorise les interruptions au niveau des périphériques (seulement le uart0) via la fonction **uart_enable(UART0)** ;
- et on configure les interruptions via **vic_setup_irqs()** ;{. avec le code suivant : **void vic_setup_irqs()**

```
for (uint32_t i = 0; i < NIRQS; i++) {  
    vic_disable_irq(i);  
    _irqs_setup();
```

On désactive les interruptions puis on appelle **_irqs_setup()**. (Note importante : Oui elles sont déjà activées, ce qui semble être bizarre, en fait j'ai enlevé le flag qui désactive les interruptions dans **_reset_handler** ce qui est très probablement pas du tout une bonne idée. Je devrais laisser désactiver les IRQ dans le **_reset_handler** et les activer seulement lorsqu'on appelle la fonction **_irqs_setup**.) Voici le code de la fonction **_irqs_setup** qui permet de setup la pile des interruptions :

```
mrs r0, cpsr Save le mode du processeur  
bic r1, r0, CPSR_SYS_MODE Supprimer le mode système du CPU  
orr r1, r1, CPSR_IRQ_MODE  
msr cpsr, r1 Passer en mode IRQ du CPU  
ldr sp, =irq_stack_top Charger l'adresse de pile des IRQ dans sp  
msr cpsr, r0 Recharger le mode du CPU du début de fct  
mov pc, lr Revenir à la fonction appelante.  
.size _irqs_setup, . - _irqs_setup  
.endfunc
```

- On configure une fonction **receive_handler** à appeler lorsqu'on reçoit une interruption sur UART0. On envoie aussi un cookie qui possède 2 informations, l'index dans le tableau des UART pour que dans **receive_handler** on puisse accéder à la base adresse du UART utilisé et une adresse d'un caractère pour qu'on puisse utiliser le char reçu plus tard même si ce dernier n'est pas utile vu qu'on ne l'utilise pas dans **_start**.

II.2 Traitement d'une interruption

Une fois tout configuré, que se passe-t-il lorsqu'on envoie une requête d'interruption au CPU ?

Pour information, l'interruption se fait au niveau matériel, lorsque le CPU en reçoit une, il exécute cette instruction **ldr pc, irq_handler_addr** qui permet de charger en prochaine instruction l'adresse d'une fonction appelée **_isr_handler** dont voici le code :

```
sub lr, lr, #4  
stmfd sp!, {r0-r12, lr}  
bl isr  
ldmfd sp!, {r0-r12, pc}^
```

On commence par sauvegarder l'adresse de retour pour savoir une fois l'interruption finie d'exécuter, pour retourner à l'endroit où le programme était. Ensuite on enregistre tous les registres r0 à r12 désignée par Stack Pointer. On exécute **isr()** puis on recharge tous les registres enregistrés avant de finir l'interruption.

Dans la fonction **void isr()** :

- On désactive les interruptions au niveau du CPU dans le registre CPSR via la fonction **_irqs_disable**.
- On récupère le registre VICIRQSTATUS où chaque bit indique si y'a une interruption ou non.

- Pour chaque bit à 1 du registre précédant, on récupère la structure **handler_t** associé au bit ou on a le cookie et la fonction à appeler. Puis on appelle la fonction de cette structure.
- On réactive les interruptions du processeur une fois toutes les IRQ traité avec la fonction **_irqs_disable()** qui touche au registre CPSR.

Dans mon cas, sur le bit numéro 12 qui correspond au interruption du UART0, j'appelle ma fonction **void receive_handler(uint8_t id, void* cookie_void)** qui récupère le caractère reçu et le renvoie directement.

III - Les caractères

Certains touches du clavier semble être mal interpréter par qemu, Ainsi la flèche de gauche semble faire un signal de reset et reexecuter le programme. Donc l'apparition de mon message présent dans la fonction **_start**. Les accents ne semble pas poser de problèmes.

Les questions :

- A quoi sert IFQ par rapport a une IRQ, et comment elle fait pour être "plus rapide". Est ce qu'elle a juste une priorité par rapport a IRQ lorsqu'il y a 2 interruptions.
- Qu'est ce qu'il se passe si y'a une interruption dans le **_reset_handler**? je suppose que rien n'est transmis et donc y'a aucun effet puisque le vic etc n'est pas configuré. Pourtant le control C semble marchait (a reverif en testant)

I - Séparation du mode interruption du reste du code

Les interruptions doivent être courtes, elles ne doivent pas prendre beaucoup de temps à exécuter sous peine de perdre des informations envoyés par des périphéries externes car les files FIFO sont pleines ou même de perdre l'interruption de la périphérie. Ainsi le but est de rendre les interruptions bien plus courtes en évitant par exemple d'exécuter un shell qui peut potentiellement être long. De plus cela permet de structurer le code et de bien séparer les différentes tâches.

Comme première idée, on fait une "barrière" entre la fonction **isr()** exécutée lors d'une interruption et le programme exécuté en mode "normal". Pour que **isr()** fasse sa tâche rapidement dans la fonction de callback **void uart_rx_handler(uint32_t irq, void* cookie)**, on lit seulement les données dans la FIFO et on les écrit dans un buffer circulaire qu'on appellera un **ring** dont on reparlera plus tard de son importance. Lorsqu'on a lu toutes les données de la FIFO, on appelle une fonction **read_listener(void *cookie)** qui lit les données du ring pour les placer dans un buffer présent dans le cookie. Ici, malgré que c'est ce qu'il y a dans mon code, ce n'est pas une bonne méthode de faire. En effet, les données doivent être lues au niveau du main (ici c'est **__start()**) pas du côté de **isr()**.

Or pendant qu'on lit ces données dans **read_listener()** les interruptions sont toujours coupées ce qui est une mauvaise chose.

Revenons au ring, ce dernier est très important, les lectures et écritures sont faites de telle manière que même s'il y a 2 programmes qui accèdent aux données en même temps, il n'y a pas de problème d'incohérence. Cela est garanti par la manière où est construit le code du ring et surtout par des méthodes au niveau matériel pour rendre le code "free lock". Ça peut être soit via des registres dédiés ou des instructions dites atomiques. Cela veut dire qu'il ne peut pas avoir un programme qui interrompt l'instruction. Ainsi le ring nous permet d'être utilisé par 2 p le mode d'interruption via la fonction **isr** et l'exécution normale du programme via **__start()**.

Pour corriger le problème de la fonction **read_listener()** appelée par **isr** et qui lit toutes les données. Il faudrait que ce **read_listener** modifie juste une variable globale pour indiquer qu'on a des données à lire et faire une fonction supplémentaire **read()**. Cette dernière sera appelée côté **__start()** dès que le programme se réveille après une interruption et si la variable globale indique qu'il y a des données à lire (ce que je n'ai pas fait dans le code).

Dans notre cas, on va supposer qu'on veut faire un shell et qu'on veut exécuter un programme dès qu'on voit le caractère **\r**. Pourquoi ce caractère et pas **\n**? car lorsqu'on appuie sur entrée dans un terminal, c'est ce caractère qui est envoyé et non **\n**. Cela dépend de la configuration du terminal. Le shell est exécuté en désactivant les interruptions. Une fois l'exécution finie, on réactive les interruptions via la fonction **core_halt()** qui appelle l'instruction assembleur **wfi** qui dans mon cas ne semble pas réactiver les interruptions et je ne sais pas pourquoi. De plus, je me demande pourquoi on désactive les interruptions à ce moment-là. On n'utilise aucune structure ou ressources en commun avec **isr** hormis le ring mais ce dernier est construit pour fonctionner de manière synchronisée. Du coup, je désactive pas les interruptions avant d'exécuter le shell.

I - Les événements

Au TP précédent, j'ai eu beaucoup de problème sur comment appeler le listener. Grâce à la nouvelle file d'évènement. On rajoute désormais un événement à la file d'évènement pour dire au programme principal d'appeler le listener. C'est une file disponible partout dans le programme (en mode interruption comme en mode normal).

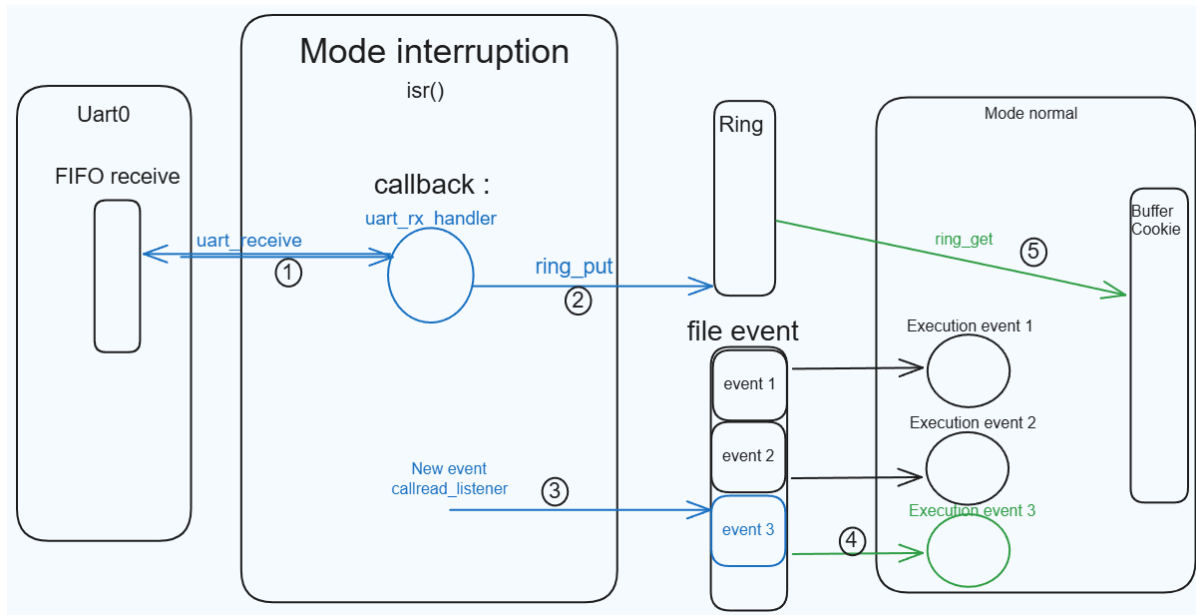


FIGURE 2 – Schéma simplifié de la gestion des événements

La file d'évènement est un tableau circulaire d'évènement très fortement inspiré du ring, Attention, le ring est lock free. Ici vu qu'on ne manipule pas des entiers, mais des structures de type **event_t**. Je ne suis pas certain que ça soit synchronisé. Ainsi, à chaque appel à la file est synchronisé par des désactivations des interruptions. Pour information, ma file d'évènement est codé par les fichiers **queue.c** et **queue.h**.

Comment marche le code ? Lorsqu'on lance le programme, on commence toujours par vérifier la pile via **stacks()** puis d'initialiser les interruptions via le VIC et sur le device uart0 pour la lecture. On passe en paramètre de callback la fonction **read_listener** et le cookie. Maintenant, mon main (**_start()**) rajoute un événement à la file d'évènement pour exécuter une fonction pour afficher du texte. Puis, on boucle à l'infinie sur récupérer le prochain événement de ma file. Pour cela, on vérifie qu'elle n'est pas vide, on exécute le plus vieil événement arrivé dans la file. Sinon, on s'endort en attendant un réveil via la fin d'une interruption. Sur le schéma, par exemple, l'événement 3 est rajouté à la fin de la file, et sera exécuté après tous les événements (1 et 2) déjà présent dans la file.

Lorsqu'on a une interruption sur uart0 pour des données reçues. On lit dans la FIFO receive du uart0 et on met la donnée dans le ring tant qu'on a des données dans la FIFO. On lit la donnée octet par octet. Ceci est réalisé par la fonction de callback donné en paramètre, ici, c'est **uart_rx_handler()** dans notre schéma, cela correspond à une boucle de l'action 1 et 2. Une fois la fonction de callback fini. On désactive les interruptions pour rajouter dans la file un événement qui contient la fonction **read_listener()** à appeler (action numéro 3 sur le schéma) puis réactive les interruptions.

Ici, contrairement à mon TP précédent, **read_listener()** n'est pas appelé coté interruption, mais bien coté mode normal. Lors du réveil du programme principal, on boucle sur tous les événements et on exécute chaque événement en appelant la fonction définit dans sa structure. Lorsqu'on récupère un événement dans la file d'évènement, on interdit les interruptions pour empêcher la file d'être modifiée par le mode interruption en même temps que le mode normal. Lorsque notre événement 3 (sur mon schéma) est en sommet de file. On le récupère (action 4 sur le schéma) et on exécute la fonction, dans notre cas, c'est notre **read_listener** qui va lire toutes les données du ring (action 5 sur

le schéma) et les placer dans le buffer du cookie et puis agir en conséquence. Dans notre cas, lorsqu'il reçoit le char `\r`, on renvoie "EXE command" à l'utilisateur.

II - Pour aller plus loin

Même si les événements semblent fonctionner dans mon cas, voici quelques pistes d'améliorations possibles dont certaine aurait dû être fait :

- Une spécification et une implémentation des fonctions **read write** pour un utilisateur. On peut aussi rajouter des appels système.
- Rajouter l'activation des interruptions pour le write avec un ring spécialement pour lui.
- On peut envisager une file d'événement avec des priorités, notamment lorsqu'on exécute des longs événements. Cela a pour effet d'empêcher que la lecture de la FIFO du uart0 soit fait longtemps après et donc avec possible perte de données. Une autre solution aurait été de rajouter l'événement en début de file plutôt qu'à la fin, mais cela n'empêche pas que l'événement exécuté en cours par le main soit long. Ainsi, il faudrait arrêter d'exécuter l'événement en cours et exécuter le `read_listener` d'abord. Mais cela complique beaucoup la gestion et il y a des risques de famine dans des cas extrême.
- On aurait pu coder les threads, mais cela aurait nécessité une scheduler ainsi que des espaces mémoires réservés pour le processus. Cela prend du temps et on a déjà codé les threads et les processus lors du projet Nachos.
- Faire une fonction shell plutôt qu'un envoi de "EXE command".