Journal n°1 IOT

COMBE Gaétan

2025

I - Le makefile

I.1 - Partie configurable

C'est une partie du makefile ou 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 débug 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 déactiver 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="$(MEMSIZE)K"** : ici, c'est juste transformer la valeur numérique en string avec le suffixe K pour kilooctets. Ainsi MEMORY est une variable qui sera utilisée pour définir la taille mémoire de QEMU, ici 32Ko.
- **MACHINE=versatileab** : définit un type de machine virtuel pour QEMU. À ne pas totalement confondre avec la variable BOARD. Ici la différence entre les deux C'est une question d'organisation du makefile, Ici, le makefile semble prévu pour changer de BOARD facilement, ce qui explique l'utilisation de BOARD seulement dans des conditions if *ifeq ($(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 définit précédemment.

- **CFLAGS= -c -mcpu=$(CPU) -nostdlib -ffreestanding -DCPU=$(CPU) -DMEMORY="($(MEMSIZE)*10**
  définit les arguments du compilateur, CFLAGS est d'ailleurs le nom de la variable utilisé par la règle implicite par Makefile pour compiler si on ne met aucune règle de compilation (ce qui rend cet outil incompréhensible pour les nouveaux qui apprennent à l'utiliser).
  - **-c** : indique à GCC de compiler les fichiers source en fichiers objet sans les lier.
  - **-mcpu=$(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=$(CPU)** : Comme pour la compilation, mais c'est pour la transformation fichier assembleur en fichier objets.
- **LDFLAGS= -T kernel.ld -nostdlib -static** :
  - **-T kernel.ld** : spécifie un fichier de script de lien pour définir la disposition de la mémoire des différentes sections de code.
  - **-nostdlib** : n'utilise pas de bibliothèque standard libre.
  - **-static** : générer un fichier executable (binaire) sans bibliothèque dynamique, c'est-à-dire que les bibliothèques sont static et donc incluse directement dans le fichier exécutable.
- **ifeq ($(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 compilier les programmes assembleur des fichiers .s en fichiers objets Ici, on utilise l'outils -as pour Asseubleur (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 réexecutera la règle. De plus, si la dépendance n'est pas encore créer, Make essayera d'abord d'executer les règles nécessaires pour créer la dépendance.
  - **commands** : liste des commandes à executer sur une règle.
- le % est une partie d'un nom de fichier présent dans le répertoire courant (ou le make est executé), pour chaque fichier trouvé correspondant à la cible, il executera 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 $(BUILD)kernel.bin** : c'est la première règle appelée par défaut par le make. Dans cette règle il faut faire d'abord construire les 3 dépendances qui sont crées par les règles suivantes :
- **$(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 déja été faite avant d'ou 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 metadonnées 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'execute 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 debogage. 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 executer avec le mode de debogage, 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 deboger. -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 execute 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 executer. Ainsi il va executer la fonction dans startup.s indiquer 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 déactive les interruptions IRQ et FIQ. les interruption IRQ (interrupt request) sont des interruption matérielle utilisés par les périphéries comme UART par exemple. FIQ (Fast interrupt request) sont un aute type d'interruption prioritaire pour des accès mémoire critique par exemple.
- **ldr sp,=stack_top** : permet de de charger le sommet de pile a 4KB après la dernière section bss. stack_pop est définie a 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 a 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,=_star
  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 ou 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 alloué, elle est appelé 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 executé. La fonction check_stacks utilise la macro MEMORY définit dans le makefile à la pré-compilation à l'aide de -DMEMORY définit a 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 une début de stack en dehors de la zone mémoire alloué 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 interruptionsmais on reparlera d'uart plus tard dans ce rapport. Ensuite on a une boucle infinie qui fait de l'attente active pour recevoir un character de l'utilisateur via la méthode **uart_receive(UART0, &c)** et une fois recu, le programme l'affiche avec la fonction **uart_send(UART0, c)**.

Dans notre header **main.h**, nous avons un ensemble de definition de fonction permettant de lire ou d'écrire dans la mémoire dans la zone MMIO (Memory-Mapped I/O) pour communiquer avec les periphéries. Toutes les fonctions définis ont plusieurs caractéristiques en commun :

- l'attribut **__inline** qui indique au compilateur d'inclure le code de la fonction directement la ou elle est utilisé sans faire d'appel de fonction. Cela permet de gagner en performance en nombre d'instruction assem-

bleur 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'ou 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 pret a 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'envoie de donnée. Si le bit est à 1, cela signifie que la donnée n'est pas prête a 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 a tout ce qui est dit au dessus. On peut lister les fonctions présent dnas 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 déactiver 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 chaine 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 a 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 ca 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 ca retourne un uint8_t dans l'implémentation pour une des fonctions.