Compte-rendu TP "Développement embarqué ARM sur cible nue"

Mewen Michel et Sander Ricou - MI11 UTC

TP 1

Edition de liens

Le but de ce TP était de prendre en main la chaîne de compilation afin de préparer une image mémoire correcte pour notre cible.
Il nous a fallu comprendre les bases du fonctionnement de l'édition de liens lors de la formation d'une image exécutable.

Nous avons donc préparé un programme C simple, test.c :

```c
int a;
int b = 1;
static int c;
char * s = "Salut";

int main(void){
    // Main qui ne quitte pas
    while(1){
        c = 42;
        a = a + b + c + s[1] + f();
    }
}
```

Nous avons besoin d'un compilateur tel que gcc pour compiler le programme en C ci dessus, cependant le gcc installé par défaut sur notre machine convient à l'hôte mais pas à la cible.
Il est nécessaire de compiler notre programme avec le compilateur suivant: `arm-none-eabi-gcc` avec l'option `-c`.

Après la compilation d'un programme en C (ou tout autre langage compilé), des fichiers objets sont générés. Ces fichiers peuvent ensuite être combinés pour former une image exécutable contenant le programme. La phase comprenant la création de l'image s'appelle l'édition de liens.

Après compilation, on peut "observer" le contenu des fichiers objet grâce à la commande `arm-none-eabi-objdump`. Ceci nous permet d'observer les différentes sections et leur contenu.
Avec les options `-ths`, on remarque la présence de 5 sections pour notre programme écrit en C : `.text`, `.data`, `.bss`, `.rodata` et `.comment`. On peut également vérifier dans quelle section se trouvent les variables utilisées. La valeur affectée à la chaîne de caractère ("Salut") est visible en section `.rodata`. De plus, la section`.data` a une taille de 8 octets et contient les variables initialées ici `b` et `s`. Sachant qu'en ARM un int (b) et un char* (s) valent tous les deux 4 octets, ceci correspond.
Il existe également les options `-rd` avec `objdump` afin d'afficher le désassemblage de la section `.text`.

Nous avons ensuite créé un second programme en C, test2.c:

```
extern int b;
int f(void){
    return (2*b);
}
```

On peut à nouveau inspecter le contenu du fichier objet associé en suivant les étapes comme précédemment.
Mais ce qui nous intéresse à présent, c'est d'exécuter la commande qui suit pour créer une image exécutable.
`arm-none-eabi-gcc -nostdlib test.c test2.c`

On se trouve alors avec le fichier `a.out`, l'image exécutable.
En exécutant la commande `arm-none-eabi-readelf -a a.out`, on observe dans le Header que ce ".out" est fait pour UNIX, ce qui ne nous convient pas. Cela provient du fait que le script d'édition de liens utilisé par défaut est pour UNIX (lors de l'exécution de `arm-none-eabi-gcc -nostdlib test.c test2.c`).
On peut accèder à l'ensemble des scripts d'édition de liens dans le dossier `/arm-none-eabi/lib/ldscripts/`.
Il nous faudra donc créer notre propre script afin que la compilation soit correcte.

Finalement nous avons créé un troisième programme cette fois-ci en c++, test3.cpp:

```
class A {
    public:
        A() {}
        int f() {return 2;}
};
int main() {
    A a;
    return a.f();
}
```

En le compilant (`arm-none-eabi-g++ -c test3.cpp`), puis en analysant son contenu on se rend compte que le fichier objet est beaucoup plus complexe qu'en C, avec un nombre de sections plus important. C'est pour cela que nous allons continuer les TPs en se cantonnant au langage C.

Revenons donc à nos programmes test.c et test2.c, nous allons les compiler puis générer à partir des deux fichiers objets notre image exécutable (`a.out`) pour avoir obtenir l'image mémoire (appelée "image"). Commandes à suivre:

```
arm-none-eabi-gcc -c test.c
arm-none-eabi-gcc -c test2.c

arm-none-eabi-ld armtd2.x test.o test2.o

arm-none-eabi-objdump -ths a.out

arm-none-eabi-objcopy -O binary -S a.out image
```

Note sur la génération de l'image exécutable (`a.out`), nous utilisons cette fois la commande `arm-none-eabi-ld` qui nous permet de spécifier un script (`armtd1.x`) correspondant à notre cible.

Contenu de `armtd1.x`:

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-bigarm",
          "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
  .text           : {
        *(.vector)
        *(.text)
  }
  .rodata           : {
        *(.rodata)
   }
  .bss           :
  {
    *(.bss)
    *(COMMON)
  }
  .hash           : { *(.hash) }
}
```

On peut comparer l'image mémoire générée avec a.out:
*`ls -la` pour afficher plus d'informations sur les fichiers et notamment comparer leur taille (image est beaucoup plus petit).
*`hexdump -C image`, visualiser le contenu de l'image en hexadécimal.

Pour résumé, la répartition en mémoire des différentes sections suit un schéma défini dans un script de liens. Les scripts par défaut du système prennent en charge la totalité des langages supportés, d'où leur longueur. Pour notre programme C comprenant 5 sections, un script beaucoup plus simple est suffisant (notre première version est armtd1.x).

Il est possible d'omettre les sections dont on peut se passer, comme par exemple les sections relatives aux commentaires. Cela permet de réduire drastiquement la taille des fichiers en sortie.

Nous verrons dans le prochain TP qu'au sein d'un tel fichier, il est possible de définir les adresses de début et de fin des sections. On peut également y utiliser des fonctions de base et définir des variables qui pourront être réutilisées plus tard dans du code assembleur.

OPENOCD, chargement de l'image sur la cible nue

Pour faire fonctionner la cible nous allons utiliser OPENOCD avec fichier de config `apf9328.cfg` à récupérer sur le moodle:

- `openocd -f ./apf9328.cfg`

Dans un nouveau terminal:

```
telnet localhost 4444
reset init
```

On ouvre ensuite putty et complète la configuration afin de se connecter au travers de la sonde JTAG à la cible.

Finalement, on revient au terminal d'OPENOCD:

```
resume// relance le programme qui était en init.
halt// arrête le programme.
reg
reg pc 0
load-image image 0x8000000
mdw 0x8000020// même résultat que `hexdump -C`
arm disasemble 0x8000020// même résultat qu'avec `-objdump -rd a.out`
resume 0x8000000// on précise la valeur de pc pour relancer le programme.
```

Au final, notre programme chargé à l'adresse `0x08000000` (début de la mémoire volatile SDRAM) plante à la première instruction (qui est un `push`) car le pointeur de pile `sp_svc` n'est pas initialisé.
Dans le prochain TP nous allons utiliser Eclipse pour configurer tout les manipulations précédentes dans le logiciel, c'est alors Eclipse qui s'en chargera.

TP 2

Paramétrage d'Eclipse

Afin de réaliser les opérations de la partie 1 plus facilement, nous avons suivi le [tutoriel sur Moodle (http://moodle.utc.fr/pluginfile.php/61465/mod_resource/content/4/eclipse_debug.pdf)](http://moodle.utc.fr/pluginfile.php/61465/mod_resource/content/4/eclipse_debug.pdf) afin de paramétrer Eclipse pour automatiser la compilation, l'édition des lien, le transfert de l'image mémoire et le débug sur cible depuis l'hôte.

Nous avons ensuite repris notre programme du TD1. Nous avons modifié le script d'édition de lien `armtd2.x` en ajoutant la ligne suivante :

```
. = 0x08000000; // décale les instructions qui suivent à cette adresse mémoire
```

Nous avons trouvé cette valeur dans le [manuel utilisateur de la carte MC9328MX1 (http://moodle.utc.fr/pluginfile.php/38071/mod_resource/content/0/Sujets_TP/Etude_de_cas/MC9](http://moodle.utc.fr/pluginfile.php/38071/mod_resource/content/0/Sujets_TP/Etude_de_cas/MC9) ; elle correspond à la première adresse mémoire de SDRAM.

Attention :

- cette zone est volatile, à chaque redémarrage de la cible le code transféré sera perdu
- contrairement à ce qui est écrit dans le manuel utilisateur, il n'y a pas 128 MB de SDRAM mais 16 uniquement dans l'implémentation utilisée

Cependant, le programme ne fonctionne toujours pas, encore pour la même raison. Afin de le faire fonctionner, nous avons modifié le registre de pile `sp_svc` afin de l'initialiser quelque part dans la SDRAM (disons `0x08001000`). On peut maintenant observer l'exécution correcte de notre code sur la cible.

Il n'est cependant pas envisageable de modifier les registres à la main à chaque exécution, il va donc falloir faire quelques initialisations avant de lancer notre code C. De ce fait, nous allons coder cette partie en **assembleur**.

Initialisation de la cible en assembleur

Note :

- un fichier source assembleur a pour extension `.S` (s **majuscule**)
- contrairement au C, il faut préciser dans quelle section placer les lignes de codes suivantes

Grâce au programme en assembleur nous allons initialiser `sp` (le pointeur de pile), initialiser la zone `bss` à 0 en mémoire et enfin appeler notre fonction main pour commencer son exécution.

On ajoute a notre script de lien les variables `bss_start` et `bss_end`, on pourra récupérer ces valeurs depuis le code assembleur ce qui va nous permettre de mettre toutes les valeurs de bss à 0.

Code du script de liens:

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-bigarm",
          "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
  . = 0x08000000;
  .text           : {
        *(.vector)
        *(.text)
  }
  .rodata           : {
        *(.rodata)
   }
  .data     : {
        *(.data)
   }
  .bss           :
  {
    *(.bss)
    *(COMMON)
  }
  bss_start = ADDR(.bss);
  bss_end = bss_start + SIZEOF(.bss);
  .hash           : { *(.hash) }
}
```

Code assembleur:

```
.text
.arm

@ Set global variables (bss) to 0
ldr r0, =bss_start
ldr r2, =bss_end

mov r1, #0

.global bssToZeros
.func bssToZeros
bssToZeros:
    str r1, [r0], #4

    cmp r0, r2

    bne bssToZeros
.endfunc

@ Initialize the stack pointer
ldr sp , =0x08010000

@ Call main()
bl main

@ If main returns, loop
b .

.align 4
.end
```

TP 3

Dans ce TP, nous reprenons le code des semaines passées permettant de préparer la cible pour accueillir et exécuter notre code C cross-compilé.
Nous allons réaliser un petit programme pour faire clignoter à 2Hz la LED reliée au port GPIO D-31.

Allumage de la LED

Modifier les bits voulus en mémoire

Note : un bit n'est pas adressable directement, il faut accéder à un octet au minimum. Il est plus aisé sur ARM de travailler avec les mots mémoire de 32 bits.

Pour modifier un mot mémoire, plusieurs méthodes existent :

- Pour modifier un bit uniquement :
    - lire le mot mémoire
    - appliquer un masque en utilisant les opérateurs binaires suivants

        - & (et logique) : permet de passer des bits à 0,
        - | (ou inclusif) : permet de passer des bits à 1,
        - et ~ (non logique) : permet d'inverser le masque,
    - écrire le nouveau mot en mémoire
- Pour modifier l'octet ou le mot mémoire en entier. C'est possible de réaliser cela dans notre cas car nous travaillons sur une cible dédiée. Il faut simplement :
    - affecter la valeur voulue à la partie de la mémoire voulue

Paraméter notre LED

Les adresses de registre GPIO se trouvent de `0x0021C000` à `0x0021CFFF` (4KB). Dans notre cas, `i` vaut 31. Pour paramétrer notre pin GPIO, il faut réaliser les étapes suivantes :

- Mettre le bit `i` du registre du port GPIO D dans le `use register` (`GIUS_D` à l'adresse `0x0021C320`) à 1 :

```
unsigned int mask_gius = 0x80000000;
ptr = (unsigned int*)0x0021C320;
*ptr = *ptr | mask_gius;
```

- Mettre les bits [2i-32+1] (= 31) et [2i-32] (= 30) du `configuration register` 1 (`OCR2_D` à l'adresse `0x0021C308`) à 1 tous les deux :

```
unsigned int mask_ocr2 = 0xC0000000;
ptr = (unsigned int*)0x0021C308;
*ptr = *ptr | mask_ocr2;
```

- Écrire les valeurs désirée sur le bit `i` du `port D data register` (`DR_D` à l'adresse `0x0021C31C`). Cette valeur sera 0 (`LOW`) dans notre cas car la borne de la LED qui n'est pas branchée à la pin GPIO est connectée au +xV :

```
unsigned int mask_dr = 0x80000000;
ptr = (unsigned int*)0x0021C31C;
*ptr = *ptr & ~mask_dr;
```

- Mettre le bit `i` du `data direction register` (`DDIR_D` à l'adresse `0x0021C3000`) à 0 pour définir la pin comme sortie :

```
unsigned int mask_ddir = 0x80000000;
ptr = (unsigned int*)0x0021C300;
*ptr = *ptr | mask_ddir;
```

La LED s'allume \o/

Clignotement de la LED (version approximative)

Afin d'éteindre la LED, il suffit de changer la valeur de la pin :

```
ptr = (unsigned int*)0x0021C31C;
*ptr = *ptr | mask_dr;
```

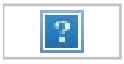
Pour pouvoir faire clignoter la LED, il va falloir attendre un peu une fois qu'on l'a allumée et une fois qu'on l'a éteinte. On peut faire ça en occupant notre processeur pendant un moment à ne rien faire :

```
for (b = 0; b<100000; b++){
    a *= 2;
    a /= 2;
}
```

Mais ça n'est pas très joli, utilisons plutôt l'horloge de la cible.

Clignotement de la LED (version précise)

Le fonctionnement du timer sur notre cible est le suivant :



Note : on n'utilise pas le mode capture qui sert à enregistrer l'état courant de l'horloge à l'arrivée d'un front montant d'un GPIO.

Pour pouvoir l'utiliser en tant que compteur par comparaison, il va faloir suivre les étapes suivantes :

- choisir l'horloge à considérer dans le `Timer ConTrol Register` (manuel (http://moodle.utc.fr/pluginfile.php/38071/mod_resource/content/0/Sujets_TP/Etude_de_
  p. 700 : OSC32, PERCLK1 (16 MHz) ou PERCLK1/16 (1 MHz)) :

```
ptr = (unsigned int*)0x00202000;
*ptr = 0x00000005;
```

- rediviser l'horloge grâce au `Timer PRescaler Register` (`0xFF` = division par 256 = 3906 Hz) :

```
ptr = (unsigned int*)0x00202004;
*ptr = 0x000000FF;
```

- définir le nombre de coups d'horloge avant de déclancher la comparaison (`Timer CoMpare Register`, `0x3E8` = déclanchement tout les 1000 coups) :

```
ptr = (unsigned int*)0x00202008;
*ptr = 0x000003E8;
```

- vérifier régulièrement le bit 0 du `Timer STATus Register` (1 = le timer a échu) :

```
ptr = (unsigned int*)0x00202014;
if (*ptr == 0x00000001)
    return 1; // true
else
    return 0; // false
}
```

- réinitialiser le `Timer STATus Register` à zéro pour permettre à nouveau une comparaison :

```
ptr = (unsigned int*)0x00202014;
*ptr = 0x00000000;
```

Notre LED clignote maintenant à 2Hz !

TP 4

Pour ce TP, nous voulons pouvoir débrancher et rebrancher la cible et que la LED clignote d'elle même. Pour permettre cela, nous allons donc charger notre code en mémoire Flash. Ensuite, nous allons utiliser les interruptions pour avoir un timer propre pour notre LED. Cette idée permettrait en effet à notre programme de pouvoir faire autre chose ou d'endormir le processeur le temps que le timer tombe.

Installation du code en Flash

Placer notre code en flash n'est pas très compliqué ; il suffit de changer l'adresse de début des sections :

```
SECTIONS
{
    . = 0x0;    @ au lieu de . = 0x08000000;
    .text : { *(.text) }
    [...]
}
```

La partie plus chatouilleuse est qu'il va falloir charger en mémoire volatile la valeur des variables initialisées et permettre aux symboles correspondants de pointer vers ces cases mémoires désormais initialisées.

Pour la section `.data` de notre code du TP1 par exemple, il va falloir séparer l'adresse du symbole b et de la constante 1 :

```
int a;
int b = 1;
static int c;
char * s = "Salut";

int main(void){
    // Main qui ne quitte pas
    while(1){
        c = 42;
        a = a + b + c + s[1] + f();
    }
}
```

On va donc assigner à nos sections contenant des variables deux adresses :

- une **adresse de chargement** (**LMA**), en flash pour mettre la totalité du contenu de nos sections non volatiles, dont nos symboles.
- une **adresse virtuelle** (ou **d'exécution**, **VMA**), en RAM pour les valeurs variables.

On va préciser la VMA et la LMA dans le script de l'éditeur de lien de la façon suivante :

```
SECTIONS {
    [...]
    .data 0x12 : AT(0x12000)
    {
        [...]
    }
    [...]
}
```

Ici, `0x12` est la VMA et `0x12000` est la LMA de la section `.data`. Dans notre cas, cela donne le résultat suivant :

```
.rodata           : {
    *(.rodata)
    @ ???
}
rodata_end = LOADADDR(.rodata) + SIZEOF(.rodata);
.data 0x008000000 : AT(rodata_end) {
    *(.data)
}
data_VMA_start = ADDR(.data);
data_start = LOADADDR(.data);
data_end = data_start + SIZEOF(.data);
```

On utilise les nouvelles macros suivantes :

- `ADDR(.data)` donne l'adresse virtuelle du premier octet de la section
- `LOADADDR(.data)` donne l'adresse de chargement de la section

Note : Par défaut, l'éditeur de lien suppose que VMA et LMA valent le location counter `.`. C'est la raison pour laquelle, sans les préciser auparavant, notre code fonctionnait quand même.

Ensuite, il va falloir copier depuis la flash les valeurs initiales des données à l'emplacement où elles sont censées se trouver.

```
@ Copy variables to their virtual address
ldr r0, =data_VMA_start
ldr r1, =data_start
ldr r2, =data_end

.global dataToVMA
.func dataToVMA
dataToVMA:
    ldr r3, [r1]
    str r3, [r0]
    add r0, #4
    add r1, #4

    cmp r1, r2

    bne dataToVMA
.endfunc
```

Mais ça ne fonctionne toujours pas... On nous a souflé l'astuce suivante : notre section `.rodata` ne faisant que 6 octets (soit 1,5 mots mémoire ARm de 32 bits), nous avons placé la section `.data` en flash de manière décalée ! Pour recaler le location counter correctement, il faut rajouter, à la place du `@ ???` dans le code plus haut :

```
    . = ALIGN(4);
```

Vecteur d'exception

Malgré cet éclair de génie, on a toujours une erreur... Essayer de démarrer avec notre stack pointer `sp` initialisé à `0x08000000` alors que notre code se trouve en flash n'a jamais été fructueux !

Lors du reset, la cible vient normalement (si on a prévenu Eclipse de ne pas toucher à notre `sp`) se placer à l'adresse `0x0`. En fait, les 8 premières instructions constituent le vecteur d'exception qui donne l'instruction à réaliser en cas d'exception.

```
.section ".vector", "x", %progbits
vectors:
    b resetHandler      @ Reset vector
    b .                 @ Undefined instruction
    b .                 @ Software interrupt
    b .                 @ Prefetch abort exception
    b .                 @ Data abort exception
    b .                 @ Reserved vector, not used
    b irq_handler       @ Normal interrupt
    b .                 @ Fast interrupt
```

Et on déclare l'étiquette `resetHandler` juste après, avant le code assembleur déjà réalisé :

```
.global resetHandler
resetHandler:
```

Afin d'être sûr que ce vecteur se trouve à l'adresse `0x0`, on modifie notre script d'édition de lien de la manière suivante :

```
. = 0x0;
.text : {
    *(.vector)
    *(.text)
}
```

Un dernier détail : pour pouvoir utiliser la cible sans qu'elle ait été initialisée par gdb, il faut appeler en premier la macro suivante (une fois ajoutée au projet, forcément) :

```
@ Configure target
bl _lowlevel_init
```

Magie, ça fonctionne ! Notre la LED de notre cible clignote bien si on coupe et qu'on rallume l'alimentation.

Timer par interruption

On souhaite maintenant que notre programme soit capable de faire autre chose le temps du timer et puisse réagir à la fin du timer. On va donc utiliser une exception, générée par le timer lorsque que la comparaison survient. Lorsque le processeur va observer un front montant sur son entrée IRQ, il devra appeler notre fonction gestionnaire d'exception. Pour réaliser tout cela il faut donc :

- faire une fonction handler d'IRQ dans le C :

```
/* Gestionnaire d’interruption IRQ */
void __attribute__((interrupt("IRQ"))) irq_handler(void) {
    ptr = (unsigned int*)0x00202014; // TSTAT
    if (*ptr == 0x00000001) {       // lis juste une fois TSTAT pour pouvoir
d'en modifier la valeur
        resetUglyTimer();
        toggleLed();
    }
}
```

- configurer le timer pour qu'il émette des interruptions :

```
// Configuration du timer
unsigned int mask_IRQEN = 0x00000011;
ptr = (unsigned int*)0x00202000;
*ptr = *ptr | mask_IRQEN;
```

- configurer l'entrée du timer dans le gestionnaire d'exception matériel AITC

```
// Routage de l'interruption par l'AITC
ptr = (unsigned int*)0x00223008;
*ptr = 59;
```

- activer les deux bits utiles du registre CPSR pour que le front montant soit perçu par le processeur ou bien utiliser les macros C `IRQ_enable`/`IRQ_disable`
- initialiser le pointeur de pile en mode IRQ :

```
@ Initialise le pointeur de pile SVC
ldr sp , =0x08010000

@ Passe en mode IRQ
mrs r0, CPSR      @ Copie CPSR dans r0
bic r0, r0, #0x1f @ Met à 0 les 5 bits M
orr r0, r0, #0x12 @ et change vers le mode interrupt
msr CPSR_c, r0    @ Recharge les bits de contrôle de CPSR
nop

@ Initialise le pointeur de pile IRQ
ldr sp , =0x08020000

@ Passe en mode SVC
mrs r0, CPSR
bic r0, r0, #0x1f
orr r0, r0, #0x13 @ Change vers le mode superviseur
msr CPSR_c, r0
nop
```

Au final, notre configuration des exceptions dans le C est donc :

```
void configureIRQ(){
    irq_enable();

    ptr = (unsigned int*)0x00223008;
    *ptr = 59;

    ptr = (unsigned int*)0x00202000;
    *ptr = *ptr | mask_IRQEN;
}
```

Et notre main contient uniquement :

```
configureIRQ();
configureTimer();
configureLed();
```

... et retourne une fois cela terminé. En attendant le timer, on va donc rester bloqués dans l'instruction :

```
@ If main returns, loop
b .
```

Voici un schéma résumant le chemin d'une interruption :