MI11 – Développement sur cible nue

Stéphane Bonnet

Université de Technologie de Compiègne bonnetst@utc.fr

Printemps 2022

Plan du cours

- 1 Introduction
- 2 Développement embarqué
- 3 Une chaîne de bas niveau
- 4 Communication hôte / cible
- 5 Débogage HW
- 6 Conclusion

Objectifs

- Maîtriser les outils et techniques liées à la construction d'applications embarquées sans support d'un système d'exploitation
 - Développement de firmwares
 - Développement de chargeurs de démarrage (bootloaders)
 - Développement de noyaux, d'exécutifs, de systèmes d'exploitation
- Mettre en œuvre ces techniques sur une cible type ARM

Mise en pratique

- Ce qu'on va faire :
 - Étudier (rapidement!) l'architecture ARM et son fonctionnement
 - Construire et configurer une chaîne de développement pour la cible
 - Étudier les codes nécessaires au démarrage d'un programme applicatif sur la cible
 - Contrôler la cible à distance au moyen d'une sonde JTAG
 - Suivre et analyser l'exécution d'un programme sur la cible (débogage matériel)
 - Écrire quelques programmes de base pour la cible

- 2 Développement embarqué

- 5 Débogage HW
- 6 Conclusion

Outils de développement

Généralités

Processus de développement similaire à celui employé pour développer des logiciels classiques. Différence essentielle : le logiciel développé s'exécute sur la *cible* et non sur *l'hôte* qui a servi à le construire.

Hôte

Machine sur laquelle les outils de développement fonctionnent.

Cible

Machine sur laquelle les logiciels développés doivent s'exécuter. Architecture possiblement différente de celle de l'hôte. Outils de développement

Processus de développement

Plus ou moins de contacts avec la cible en fonction des outils mis en œuvre. Elle peut être complètement abstraite ou au contraire devoir être considérée dans tous ses détails. On distingue :

- Outils de bas niveau. Supposent une connaissance détaillée du fonctionnement de la cible et du processus de programmation
- Outils de niveau intermédiaire. Automatisent les tâches liées aux outils de bas niveau
- Outils de haut niveau. Basés sur des modèles, permettent de générer des applications sans que l'ingénieur n'aie à connaître les détails ni de la cible, ni des outils de bas niveau.

Outils de bas niveau

Destinés à produire des logiciels *natifs* pour la cible. En fonction de la nature de la cible, ils créent :

- des images mémoire avec lesquelles la cible sera programmée
- des images exécutables par le système d'exploitation (OS, Operating System) installé sur la cible.

Image mémoire

Fichier contenant la copie exacte du contenu de la mémoire de la cible, telle qu'elle doit être au moment de l'exécution du programme.

Image exécutable

Fichier structuré contenant les informations nécessaires au chargeur (loader) d'un OS pour créer un nouveau processus pour le programme.

Ils comprennent aussi des logiciels permettant d'interagir avec la cible (prise de contrôle, chargement des images)

Outils de niveau intermédiaire

Automatisent l'utilisation des outils de bas niveau. En général, environnements de développement intégrés fournis par le fabricant de la cible ou des tiers. Prennent en charge la communication avec la cible, isolant le développeur des détails de fonctionnement. Quelques exemples :

- Xcode d'Apple (développement pour iOS) ou Android Studio (développement pour Android)
- KIEL, CodeSourcery pour architecture ARM,
- Code Composer Studio pour les DSP / processeurs de Texas Instruments,
- MPLAB X pour microcontrôleurs PIC de Microchip,
- Arduino IDE, ...

Peuvent être adaptés au développement bas niveau (OS, bootloaders...) ou être limités au développement d'applications pour un OS ou un noyau (temps-réel) spécifique.

Outils de haut niveau

Développement basé modèle d'applications embarquées complexes. Retrouvés dans les processus d'ingénierie système adoptés par

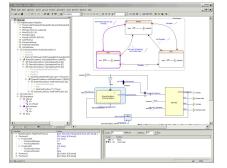
- l'industrie aéronautique
- l'industrie automobile
- l'industrie ferroviaire
- les systèmes médicaux
- et bien d'autres secteurs industriels (nucléaire...)

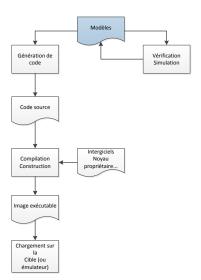
Quelques exemples:

- Matlab/Simulink Embedded Coder
- ANSYS/Esterel SCADE (Aéronautique)
- ETAS Ascet (Automobile)
- dSpace, ...

Exemple d'outil de haut niveau - SCADE

- Outil graphique
- Code certifiable
- Génération adaptée à différents intergiciels (middleware) ou noyaux (VxWorks, PikeOS, FreeRTOS...)





Tests et interaction avec la cible

Le développement logiciel comprend systématiquement des phases de test, de vérification et de validation des programmes.

Dans le cas des systèmes embarqués, ces activités peuvent être conduites soit :

- directement sur la cible (émulation)
- en utilisant un simulateur logiciel de la cible

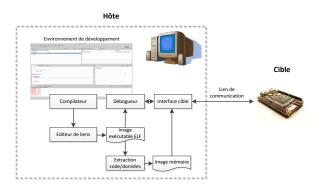
Vérification

S'assurer que le système a été développé correctement

Validation

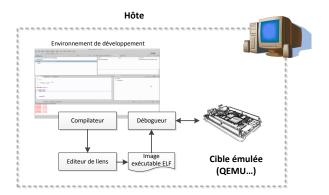
S'assurer que le système répond au besoin (ou au moins au cahier des charges)

Exploitation de la cible réelle



- Plus complexe à mettre en œuvre
- Le matériel doit être disponible
- Performances et comportement conformes à la réalité

Simulation



- Plus simple d'utilisation
- Permet de tester les logiciels avant disponibilité du matériel
- Approximation plus ou moins exacte du matériel réel, on est pas à l'abri de surprises...



Tests et implantation sur la cible

Émulation vs. Simulation (1/2)

En général

- Émulation : reproduction exacte du comportement d'un modèle réel jusque dans ses moindre détails physiques.
- Simulation : reproduction d'un comportement similaire à un système réel, mais implémenté de manière totalement différente à la réalité (p. ex simulateur de vol)

Émulation vs. Simulation (2/2)

Pour l'informatique embarquée bas niveau

- Emulation: remplacement physique d'un processeur / « System-on-Chip » par un émulateur matériel qui le reproduit exactement d'un point de vue électrique / numérique, piloté au travers d'une sonde par un logiciel
- Simulation: utilisation d'un logiciel qui émule (au sens général) le processeur / « System-on-Chip » sans aucune interaction avec la cible réelle (par exemple QEMU).

Les systèmes physiques d'émulation sont complexes et onéreux. En général les processeurs / SoC fournissent des interfaces permettant de les piloter comme s'ils étaient des émulateurs. On parle d'émulation in-circuit (ICE, In-Circuit Emulation, ou OCD, On-Chip Debug).

On ne déploie pas exactement les mêmes outils si on veut développer un logiciel applicatif ou un logiciel de support (noyau, intergiciel, chargeur de démarrage...).

Même si en général les outils « bas niveau » sont adaptés aux deux, ce n'est pas nécessairement le cas des outils plus complexes (iOS n'est pas développé avec Xcode ou Android avec Android Studio).

Finalement, les outils « haut niveau » reposent systématiquement sur l'existence d'outils de plus bas niveau pour les basses besognes (génération d'images adaptées à la cible ou à son système d'exploitation).

Si on veut avoir une bonne maîtrise du développement de logiciels embarqués, il est utile d'avoir une vision globale partant de la base.

- 1 Introduction
- 2 Développement embarqué
- 3 Une chaîne de bas niveau
- 4 Communication hôte / cible
- 5 Débogage HW
- 6 Conclusion

Le choix d'une chaîne de développement dépend de la cible et de l'application.

Chaque cible est différente

Les outils présentés avant reposent sur l'existence de moyens :

- de création d'images mémoire exécutables sur la cible
- de création d'images exécutables par l'OS de la cible
- de communication avec la cible

Chaque application est différente

Certaines applications ne justifient pas la mise en œuvre d'un OS :

- Applications monolithiques peu complexes (firmwares),
- Trop de contraintes matérielles (mémoire disponible...),
- Besoin de performances élevées.

Outils adaptés en général fournis par le fabricant du système. Mais que faire s'il n'existent pas?

Cible nue, ou « Bare metal »

Une cible sans logiciel ou firmware est dite « nue » (ou bare metal). La première étape avant de pouvoir l'utiliser est de disposer d'une chaîne de compilation qui permet de créer un firmware et d'outils permettant de transférer les images mémoire obtenues sur la cible.

Firmware

Il s'agit d'une image mémoire *persistante* directement exécutable, qui contient à la fois les données et le code de l'application nécessaire au fonctionnement de la cible et qui est exécutée à sa mise sous tension.

De quoi a-t-on besoin?

Pour créer une image mémoire exécutable à partir d'un code source il faut au minimum :

- un assembleur croisé ou cross-assembler,
- un compilateur croisé ou cross-compiler (en général C),
- un éditeur de liens,
- un outil de création d'images mémoire.

Outils développement croisé

Il s'agit d'outils de développement exécutables sur l'hôte mais qui génèrent des images pour une architecture différente de l'hôte. Par exemple, un compilateur C qui cible l'architecture ARM mais qui s'exécute sur PC est un compilateur croisé.

Les outils GNU

Les outils GNU permettent de remplir tous ces besoins.

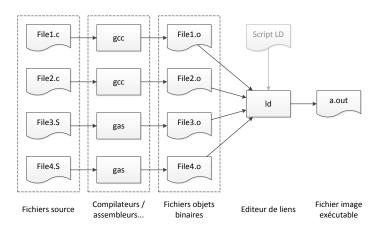
- gcc : compilateur C
- gas : assembleur
- 1d : éditeur de liens
- objcopy : outil de manipulation d'images

Les outils de développement GNU comprennent aussi (entre autres...)

- objdump : introspection des fichiers objet
- readelf: introspection des images exécutables au format ELF (Executable and Linkable Format)

Structure d'une chaîne de compilation

La chaîne de compilation produit une image exécutable à partir de fichiers source au travers de fichiers binaires objet.



La chaîne de compilation doit être créée et configurée afin qu'elle produise des exécutables

- conformes à *l'architecture* de la cible
- conformes à l'organisation mémoire de la cible

Il faut disposer d'outils de compilation croisés correctement configurés. Leur construction pour une *architecture* particulière sort du cadre de ce cours mais n'est pas très difficile — voir http://wiki.osdev.org par exemple à ce sujet.

En revanche, on va s'intéresser à leur configuration pour une cible spécifique en étudiant comment ces outils produisent une image exécutable et en adaptant ce processus.

Un fichier source

Éléments d'un programme C

- Des données non initialisées
- Des données initialisées
- 3 Des constantes
- 4 Des instructions

Éléments d'un programme C

Un fichier source

```
int a;

int a;

static int b;

int c = 1;

char *s = "Salut";

int main(void) {
    a = a + b + c + s[i] + f();
    return(a);
}

int a;

Des données non initialisées

Des données initialisées

Des constantes

Des instructions

Des instructi
```

Un fichier source

Éléments d'un programme C

```
int a;
int a;
static int b;
int c = 1;

Char *s = "Salut";
int main(void) {
    a = a + b + c + s[i] + f();
    return(a);
}

Des données non initialisées

Des constantes

Des constantes

Des instructions

Des instructions
```

Éléments d'un programme C

Un fichier source

```
int a;
int a;
static int b;

int c = 1;

char *s = "Salut";

int main(void) {
    a = a + b + c + s[i] + f();
    return(a);
}

Des données non initialisées

Des constantes

Des constantes

Des instructions

Interpretation of the properties of the properties
```

Un fichier source

```
int a;
int a;
static int b;
int c = 1;

Char *s = "Salut";
int main(void) {
    a = a + b + c + s[1] + f();
    return(a);
}

Des données non initialisées

Des constantes

Des constantes

Des instructions

4 Des instructions
```

Les sections

Qu'est-ce qu'une section?

- La chaîne de compilation produit des exécutables à partir de plusieurs fichiers source.
- Les langages utilisés peuvent être différents (par ex. C et assembleur)
- Chaque fichier source peut définir des éléments divers qui devront être regroupés selon leur nature dans l'image finale

Dans les fichiers objet, chaque type d'élément est émis dans une **section** particulière. L'éditeur de liens se sert des sections pour regrouper les objets de même catégorie dans l'image exécutable finale.

Les sections

Sections standard

Les différents outils de compilation ou d'assemblage permettent de définir des objets de même nature. Pour s'y retrouver, il faut que tous les outils les rangent dans les mêmes sections. Certaines sections ont des noms conventionnels, les plus importantes sont :

- .text pour les instructions
- .data pour les données initialisées
- **.bss** pour les données non initialisées
- .rodata pour les données constantes

Il est possible de définir d'autres sections dans les fichiers source, par exemple pour séparer des codes qui ne doivent pas être au même emplacement dans l'image exécutable.

Attribution des objets aux sections - Langage C

Les sections

En langage C, le compilateur décide automatiquement d'affecter chaque élément à la bonne section. On peut toutefois l'imposer dans gcc en utilisant la construction suivante.

```
int bar __attribute__((section ("machin"))) = 1;  /* bar est dans la section machin */
int foo(void) __attribute__((section ("truc")))  /* foo est dans truc */
{
    return(0);
}
```

En assembleur GNU, il faut rendre les sections explicites dans le code source en nommant la section dans laquelle tout ce qui suit dans le fichier doit être émis.

```
.file
             "foo.S"
2
                                   # Tout ce qui suit va dans .text
     .section .text
     .thumb
             ldr r3,=0xe0000000
6
             ldr
                    r2,=GPIO_D
             1dr
                    r0,[%r2,#GIUS]
9
10
11
     .section .data
                                   # Tout ce qui suit va dans .data
12
13
                     Oxcafebabe
             .word
14
15
     # Tout ce qui suit va dans la section "truc".
     # "truc" doit être allouée (a), est exécutable (x) et contient
16
     # des données à émettre dans le fichier objet (%progbits)
18
     .section "truc", "ax", %progbits
19
     . thumb
20
                     r0.10
             mov
21
     100p:
             subs
                     r0.r0.#1
22
                     loop
             bne
23
     .end
```

Les sections

Le format *ELF*

Le format *ELF* (*Executable and Linkable Format*) est un format de fichier objet et d'image exécutable très répandu dans le monde Unix. Il est utilisé par les outils GNU et Linux par exemple. Chaque fichier est organisé comme suit :

- Un en-tête ELF décrivant le type de fichier et ses attributs, dont l'architecture de la machine auquel il est destiné
- Une table d'en-têtes de segments, décrivant zéro ou plus segments de mémoire (pour les exécutables)
- Une table d'en-têtes de sections, décrivant zéro ou plus sections
- Les données proprement dites, auxquelles font référence les en-têtes de segments et de sections

Chaque fichier objet produit par gcc ou gas respecte ce format. Il s'agit de la représentation binaire du code machine et des données organisées en sections.

On peut utiliser « objdump » pour visualiser le contenu d'un fichier objet.

```
$ objdump -fh file1.o
file1.o:
             file format elf32-littlearm
architecture: armv7e-m, flags 0x00000011:
Sections:
Idx Name
                  Size
                            VMA
                                      LMA
                                                File off
                                                          Algn
  0 text
                  00000044 00000000 00000000 00000034
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 data
                  00000008 00000000 00000000 00000078
                                                          2**2
                  CONTENTS, ALLOC, LOAD, RELOC, DATA
  2 .bss
                  00000004 00000000 00000000
                                                00000080
                                                          2**2
                  ALLOC
  3 .rodata
                  00000006 00000000 00000000
                                                00000080
                                                          2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 comment
                  0000005a 00000000 00000000 00000086
                  CONTENTS . READONLY
```

Les sections sont marquées à allouer (ALLOC) et à charger (LOAD) à partir du contenu du fichier (CONTENTS).

- VMA : adresse virtuelle de la section
- LMA : adresse de chargement de la section



Structure d'un fichier objet binaire ELF

Le cas .bss

Dans les sections, .bss est marquée simplement ALLOC. Elle ne doit pas être chargée à partir de l'image. Pourquoi?

Les données contenues dans .bss ne sont pas initialisées (pas de valeur) : il n'y a rien a stocker dans le fichier objet. Il faut juste leur allouer assez de place dans l'image mémoire finale.

objdump permet aussi de voir à quelle section a été affecté chacun des éléments du code source.

```
$ objdump -t file1.o
file1 o
             file format elf32-littlearm
SYMBOL TABLE:
00000000 1
              df *ABS* 00000000 file1.c
00000000 1
                .text 00000000 .text
00000000 1
              d .data 00000000 .data
00000000 1
              d .bss
                        00000000 .bss
00000000 1
                 .bss
                        00000004 b
00000000 1
              d .rodata
                           00000000 .rodata
                .comment 00000000 .comment
00000000 1
00000000 1
                .ARM.attributes 00000000 .ARM.
 attributes
00000004
               0 *COM* 00000004 a
00000000 g
               0 .data 00000004 c
00000004 g
               0 .data 00000004 s
00000000 g
               F .text 00000044 main
00000000
                 *UND* 00000000 f
```

- a n'est dans aucune section?
- f est non défini?

objdump permet aussi de voir à quelle section a été affecté chacun des éléments du code source.

```
$ objdump -t file1.o
file1 o
             file format elf32-littlearm
SYMBOL TABLE:
00000000 1
              df *ABS* 00000000 file1.c
00000000 1
                 .text 00000000 .text
00000000 1
              d .data 00000000 .data
00000000 1
              d .bss
                        00000000 .bss
00000000 1
                 .bss
                        00000004 ъ
00000000 1
              d .rodata
                           00000000 .rodata
                 .comment 00000000 .comment
00000000 1
00000000 1
                 .ARM.attributes 00000000 .ARM.
 attributes
00000004
               0 *COM* 00000004 a
00000000 g
               0 .data 00000004 c
00000004 g
               0 .data 00000004 s
00000000 g
               F .text 00000044 main
00000000
                 *UND* 00000000 f
```

- a n'est dans aucune section?
- f est non défini?

objdump permet aussi de voir à quelle section a été affecté chacun des éléments du code source.

```
$ objdump -t file1.o
file1 o
             file format elf32-littlearm
SYMBOL TABLE:
00000000 1
              df *ABS* 00000000 file1.c
00000000 1
                 .text 00000000 .text
00000000 1
              d .data 00000000 .data
00000000 1
              d .bss
                        00000000 .bss
00000000 1
                 .bss
                        00000004 ъ
00000000 1
              d .rodata
                           00000000 .rodata
                 .comment 00000000 .comment
00000000 1
00000000 1
                 .ARM.attributes 00000000 .ARM.
 attributes
00000004
               0 *COM* 00000004 a
00000000 g
               0 .data 00000004 c
00000004 g
               0 .data 00000004 s
00000000 g
               F .text 00000044 main
00000000
                 *UND* 00000000 f
```

- a n'est dans aucune section?
- f est non défini?

Symboles communs et non définis

- Symboles communs: *COM* Variables globales potentiellement communes à plusieurs fichiers objet dont toutes les définitions doivent aboutir au même emplacement mémoire. La section *COM* est destinée à recevoir toutes les variables qui pourront être potentiellement fusionnées.
- Symboles inconnus: *UND* Le C permet de faire référence à des symboles définis dans une autre unité de compilation (fichier source). Un tel symbole est inconnu du fichier objet, son adresse devra être mise à jour par l'éditeur de liens quand elle sera connue, lors de la phase de **relogeage** (relocation).

```
$ obidump -s file1.o
Contents of section .text:
 0000 98b500af 0b4b1a68 0b4b1b68 1a440b4b .....K.h.K.h.D.K
 0010 1b681344 0a4a1268 01321278 9c18fff7
                                           .h.D.J.h.2.x...
 0020 feff0346 2344034a 1360024b 1b681846
                                           ...F#D.J.'.K.h.F
 0030 98bd00bf 00000000 00000000 00000000
 0040 00000000
Contents of section .data:
 0000 01000000 00000000
Contents of section .rodata:
 0000 53616c75 7400
                                           Salut.
Contents of section .comment:
 0000 00474343 3a202831 353a392d 32303139 .GCC: (15:9-2019
 0010 2d71342d 30756275 6e747531 2920392e -q4-0ubuntu1) 9.
 0020 322e3120 32303139 31303235 20287265 2.1 20191025 (re
 0030 6c656173 6529205b 41524d2f 61726d2d lease) [ARM/arm-
 0040 392d6272 616e6368 20726576 6973696f 9-branch revisio
 0050 6e203237 37353939 5d00
                                           n 277599].
Contents of section .ARM.attributes:
 0000 41310000 00616561 62690001 27000000
                                           A1...aeabi..'...
                                           .7E-M....M.....
 0010 0537452d 4d00060d 074d0902 0a081204
 0020 14011501 17031801 19011a01 1c011e06
 0030 2201
```

3c: R_ARM_ABS32 40: R_ARM_ABS32

Relocations

```
$ objdump -rd file1.o
Disassembly of section .text:
00000000 <main>:
                  push {r3, r4, r7, lr}
   0: b598
   2: af00
                  add
                        r7, sp, #0
                        r3, [pc, #44]
                                       ; (34 <main+0x34>)
   4: 4b0b
                  ldr
                        r2, [r3, #0]
   6: 681a
                  ldr
   8: 4b0b
                  ldr
                        r3, [pc, #44]
                                       ; (38 <main+0x38>)
                  ldr
                        r3, [r3, #0]
   a: 681b
   c: 441a
                  add
                        r2. r3
                        r3, [pc, #44] ; (3c <main+0x3c>)
   e: 4b0b
                  ldr
  10: 681b
                  ldr
                        r3, [r3, #0]
  12: 4413
                  add
                        r3, r2
                                       : (40 <main+0x40>)
  14: 4a0a
                  ldr
                        r2. [pc. #40]
  16: 6812
                  ldr
                        r2, [r2, #0]
                  adds r2, #1
  18: 3201
  1a: 7812
                  ldrb
                        r2, [r2, #0]
  1c: 189c
                  adds r4, r3, r2
  1e: f7ff fffe
                  bl 0 <f>
         1e: R_ARM_THM_CALL
  22: 4603
                  mov
                         r3, r0
  24: 4423
                  add
                         r3. r4
  26: 4a03
                  ldr
                         r2, [pc, #12]
                                        : (34 <main+0x34>)
  28: 6013
                  str
                        r3, [r2, #0]
  2a: 4b02
                  ldr
                        r3, [pc, #8]
                                        : (34 <main+0x34>)
                         r3, [r3, #0]
  2c: 681b
                  ldr
  2e: 4618
                  mov
                        r0, r3
  30: bd98
                   pop
                        {r3, r4, r7, pc}
  32: bf00
                   nop
```

.bss

34: R_ARM_ABS32 38: R_ARM_ABS32

3c: R_ARM_ABS32 40: R_ARM_ABS32

Relocations

L'emplacement futur des sections dans l'image mémoire finale est inconnu. Les adresses devant être ajustées après que l'adresse mémoire de chaque section est déterminée sont marquées. L'éditeur de liens y placera les valeurs correctes lors de la relocation.

Que fait l'éditeur de liens

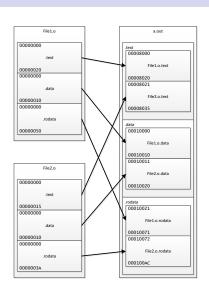
Rôle de l'éditeur de liens

L'éditeur de liens combine les fichiers objet produits par le compilateur / assembleur ou provenant de bibliothèques et produit un fichier image exécutable sur l'architecture pour laquelle il a été configuré. Il le fait en deux étapes :

- Assemblage des sections
- Traitement des relocations

Première étape : assemblage des sections

- Toutes les sections de même nom sont concaténées dans l'ordre d'apparition des fichiers sur la ligne de commande,
- L'adresse de début de chaque section résultante est calculée en fonction de sa position définitive dans l'image finale,
- Au fur et à mesure que les adresses des symboles sont fixées, les couples (symbole, adresse) sont ajoutés à une table interne.



Seconde étape : la relocation

```
/* file1.c */
                                                          /* file2.c */
   int a:
    static int b:
                                                          extern int c:
4 int c = 1;
  char *s1 = "Salut":
                                                         int f(void) {
  int main(void) {
                                                              return (c * 2):
        return a + b + c + s1[1] + f():
    file1.o: file format elf32-littlearm
                                                          file2.o: file format elf32-littlearm
                                                          RELOCATION RECORDS FOR [.text]:
    RELOCATION RECORDS FOR [.text]:
    OFFSET TYPE
                               VALUE
                                                          OFFSET
                                                                   TYPE
                                                                                     VALUE
    0000001e R_ARM_THM_CALL
                                                          00000014 R_ARM_ABS32
    00000034 R ARM ABS32
    00000038 R ARM ABS32
                               .bss
    0000003c R ARM ABS32
    00000040 R ARM ABS32
    RELOCATION RECORDS FOR [.data]:
    OFFSET
             TYPE
    00000004 R ARM ABS32
                               .rodata
```

- recherche dans la table interne de l'adresse de chaque symbole devant subir une relocation
- remplacement dans la section des octets correspondants par l'adresse finale



L'éditeur de liens produit une image exécutable *ELF*. Ce n'est pas encore une image de la mémoire. Ce fichier commence par un en-tête qui décrit son contenu :

```
$ readelf -a a.out
ELF Header:
 Magic:
           7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
 Class:
                                      ELF32
 Data:
                                      2's complement, little endian
 Version:
                                      1 (current)
 OS/ABT:
                                      UNIX - System V
 ABT Version:
 Type:
                                      EXEC (Executable file)
 Machine:
                                      ARM
 Version:
                                      0 \times 1
 Entry point address:
                                      0x8000
 Start of program headers:
                                      52 (bytes into file)
 Start of section headers:
                                      33740 (bytes into file)
 Flags:
                                      0x5000200, Version5 EABI, soft-float ABI
 Size of this header:
                                      52 (bytes)
 Size of program headers:
                                      32 (bytes)
 Number of program headers:
 Size of section headers:
                                      40 (bytes)
 Number of section headers:
                                      11
 Section header string table index: 10
```

Suit la liste des sections et des segments, ainsi que l'association entre les deux.

```
Section Headers:
  [Nr] Name
                         Type
                                         Addr
                                                  Off
                                                          Size
                                                                 ES Flg Lk Inf Al
  Γ 01
                         NULL
                                         00000000 000000 000000 00
  [ 1] .text
                         PROGBITS
                                         00008000 008000 00005c 00
  [ 2] .rodata
                         PROGBITS
                                         0000805c 00805c 000006 00
  [3] .data
                                         00018064 008064 000008 00 WA 0
                         PROGBITS
  [ 4] .bss
                         NOBITS
                                         0001806c 00806c 000008 00
  [ 5] .comment
                        PROGBITS
                                         00000000 00806c 000059 01
  [ 6] .ARM.attributes ARM_ATTRIBUTES 00000000 0080c5 00002e 00
  [ 7] .noinit
                         PROGRITS
                                         00018074 0080f3 000000 00
  [8] .symtab
                                                                         9 18 4
                         SYMTAB
                                         00000000 0080f4 000210 10
  [ 9] .strtab
                                         00000000 008304 000070 00
                                                                             0 1
                         STRTAB
  [10] .shstrtab
                         STRTAB
                                         00000000 008374 000055 00
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  y (purecode), p (processor specific)
There are no section groups in this file.
Program Headers:
                                                FileSiz MemSiz Flg Align
  Type
                 Offset
                          VirtAddr
                                     PhysAddr
  LOAD
                 0x000000 0x00000000 0x00000000 0x08062 0x08062 R E 0x10000
  T. N A D
                 0x008064 0x00018064 0x00018064 0x00008 0x00010 RW 0x10000
 Section to Segment mapping:
  Segment Sections ...
   00
          .text .rodata
   0.1
          .data .bss
```

Transformation d'une image exécutable en image mémoire

Image sur une cible avec OS

Sur une cible qui dispose d'un OS, le travail est terminé. Il suffit de :

- Copier l'image exécutable sur le système de fichier de la cible.
- Laisser l'OS créer un processus à partir de l'image.
- Le chargeur s'occupe d'allouer les segments en mémoire et de copier les sections à leurs places dans les segments, définies dans l'image.

Image sur une cible nue

Sur une cible sans OS, il faut extraire de l'image exécutable l'image mémoire qui serait créée par le chargeur. C'est le rôle du programme objcopy :

- \$ objcopy -o binary -s a.out memory_image
- Pour chaque segment défini dans l'image exécutable ELF, objcopy copie dans le fichier de destination memory_image les octets de l'image ELF
- S'il existe un trou entre deux segments, objcopy le remplira de 0 dans le fichier de sortie.
- Exemple: Premier segment à l'adresse 0x00000000, de longueur 42. Second segment à l'adresse 0x1000000, de longueur 8. L'image mémoire aura une taille de plus de 256 Mo et non 50 octets comme attendu.

Et ensuite?

On sait comment une image exécutable est organisée. Il faut adapter cette organisation en fonction des contraintes de la cible.

- Obtenir une chaîne de compilation adaptée à l'architecture de la cible
- Configurer l'organisation des sections en mémoire pour qu'elle se conforme à la cible

C'est ce deuxième point que nous allons traiter maintenant, en voyant comment on peut forcer l'éditeur de liens à organiser la mémoire de manière correcte en fonction de la carte mémoire de la cible.

Une carte mémoire décrit l'occupation physique de l'espace d'adressage de la cible. Elle comprend :

- Les mémoires RAM (SRAM, SDRAM), qui peuvent être non contiguës
- Les mémoires Flash, qui peuvent être non contiguës et de types différents (NAND, NOR)
- L'emplacement des registres d'état, de contrôle et de données des périphériques
- **...**

L'éditeur de liens doit produire des images dont les sections prennent place au bon endroit dans cette carte mémoire.

Carte mémoire d'un processus

Il s'agit de l'emplacement de chaque section dans l'espace d'adressage du processus.

Rôle des scripts d'édition de liens

L'éditeur de liens décide des adresses et de l'organisation des images exécutables en suivant les instructions données dans des scripts (*linker scripts*). Ces scripts permettent de configurer son fonctionnement. Il faut donc en fournir un adapté à la cible.

Sous un système Linux, on les trouve dans /usr/lib/ldscripts/. L'un d'entre eux est utilisé par défaut, mais on peut le changer en ligne de commande en exécutant

\$ ld -T script.x file1.o file2.o

- Spécification du format du fichier image et de l'architecture
- Symbole prenant l'adresse du point d'entrée (première instruction) du programme
- Nom de la section de sortie dans le fichier image
- Nom du fichier objet contenant la section d'entrée
- Nom de la section d'entrée à copier depuis le fichier objet dans la section de sortie

- Spécification du format du fichier image et de l'architecture
- Symbole prenant l'adresse du point d'entrée (première instruction) du programme
- Nom de la section de sortie dans le fichier image
- Nom du fichier objet contenant la section d'entrée
- Nom de la section d'entrée à copier depuis le fichier objet dans la section de sortie

- Spécification du format du fichier image et de l'architecture
- Symbole prenant l'adresse du point d'entrée (première instruction) du programme
- Nom de la section de sortie dans le fichier image
- Nom du fichier objet contenant la section d'entrée
- Nom de la section d'entrée à copier depuis le fichier objet dans la section de sortie

- Spécification du format du fichier image et de l'architecture
- Symbole prenant l'adresse du point d'entrée (première instruction) du programme
- Nom de la section de sortie dans le fichier image
- Nom du fichier objet contenant la section d'entrée
- Nom de la section d'entrée à copier depuis le fichier objet dans la section de sortie

- Spécification du format du fichier image et de l'architecture
- Symbole prenant l'adresse du point d'entrée (première instruction) du programme
- Nom de la section de sortie dans le fichier image
- Nom du fichier objet contenant la section d'entrée
- Nom de la section d'entrée à copier depuis le fichier objet dans la section de sortie

- Spécification du format du fichier image et de l'architecture
- Symbole prenant l'adresse du point d'entrée (première instruction) du programme
- Nom de la section de sortie dans le fichier image
- Nom du fichier objet contenant la section d'entrée
- Nom de la section d'entrée à copier depuis le fichier objet dans la section de sortie

Noms des sections et des fichiers objets

Peuvent être spécifiés explicitement ou au moyen de jokers.

```
OUTPUT_FORMAT("elf32_littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)

SECTIONS {
          ctt.o(.init)
          *(.text)
          *(.text.*)
       }
          ...
}
```

La section de sortie .text contiendra dans l'ordre :

- la section .init du fichier objet crt.o
 s'il existe
- les sections .text de tous les fichiers objets (y compris crt.o)
- les sections commençant par .text. de tous les fichiers objets (y compris crt.o)

Ordre de traitement des fichiers objets

Les fichiers objets sont traités dans l'ordre de leur apparition sur la ligne de commande d'invocation de l'éditeur de liens.

Le location counter « . »

L'éditeur de lien maintient une variable spéciale qui contient l'adresse du prochain octet qui sera émis dans le fichier de sortie. Elle se note « . » (un point seul) et peut être utilisée pour forcer les adresses auxquelles sont émises les sections.

- La section foo est émise à l'adresse 1000
- La section bar est émise 1000 octets après la section foo.
- Remarque : le compteur ne peut jamais décroître!

Le location counter « . » – Adresse absolue ou relative?

La valeur du *location counter* est relative au bloc où il est utilisé. S'il est utilisé directement dans le bloc SECTIONS, alors il est relatif au début de l'image et contient une adresse *absolue*. Sinon, il contient une adresse relative au début de la section en cours.

- foo est émise à l'adresse 1000
- foo se termine 1000 octets après son début, quelque soit sont contenu. Si son contenu dépasse 1000 octets, une erreur sera générée.
- bar est émise après foo, à l'adresse de chargement 2000.
- bar contient 1000 octets vides supplémentaires à la fin.

Symboles

On peut définir des symboles qui seront connus à l'édition de liens et peuvent être utilisés dans les programmes.

- Les symboles foo_start et foo_end sont relogeables – adresse relative à la section de sortie
- Les symboles foo_len et bar_end sont absolus – adresse fixe par rapport au début de l'image
- Le macro SIZEOF donne la taille d'une section et ADDR son adresse de départ.

Scripts d'édition de liens

Zone de mémoire

On peut aussi définir des zones de mémoire.

```
MEMORY {
    MEM1 (rwx): org = 8000, len = 1000
    MEM2 (rx): org = 10000, len = 500
SECTIONS {
    foo: {
      *(foo)
    } > MEM1
    bar: {
      *(bar)
    } > MEM2
}
```

- La section foo prend place dans la mémoire MEM1, à l'adresse 8000.
- La section bar prend place dans la mémoire MEM2, à l'adresse 10000.
- Si une section déborde de la zone prévue, l'édition de liens s'arrête sur une erreur

Document utile

Ce cours n'est qu'une brève introduction. Les scripts d'édition de liens sont un mécanisme très puissant et peuvent être complexes. Rien ne vaut la documentation!

■ Le manuel de LD : http://sourceware.org/binutils/docs/ld/ On peut aussi s'inspirer des nombreux scripts disponibles dans toute distribution Linux qui se respecte...

Choix du langage de programmation

- Jusqu'à présent, on s'est concentré sur le C.
- Ce n'est pas si simple : un programme C commence par la fonction main... mais qui appelle main?
- Il faut écrire un programme de démarrage en assembleur

Il doit:

- S'assurer que le matériel de la cible est dans un état correct
- S'assurer que l'état du processeur et de la mémoire est adapté à l'exécution correcte d'un programme C.
- Faire un appel correct à main pour donner la main au code C
- Faire quelque chose d'intelligent au retour de main.

Assembleur ou C?

Hypothèses vérifiées lors de l'appel à main

Un programme écrit en C s'attend à ce que deux hypothèses au minimum soient vérifiées avant de s'exécuter. Ces hypothèses sont :

- les variables globales non initialisées doivent valoir 0. Les pointeurs globaux doivent valoir NULL.
- il doit exister une zone de pile valide (pour les passages de paramètres, cadres de pile et variables locales)

Une des fonction du code d'initialisation est de garantir que ces hypothèses sont vérifiées.

Liste de courses

Afin d'adapter une chaîne de compilation GNU à une cible, l'ingénieur responsable doit au moins :

- Construire les outils de compilation croisée adaptés à la cible (binutils et gcc au moins, puis libgcc)
- Construire un script d'édition de liens adapté à la cible
- Fournir les fichiers objet d'initialisation corrects

Étant donnés ces éléments, il est possible de développer en C des logiciels de type firmware ou noyau. Pour développer des logiciels d'application plus complexes avec un OS, il faut aussi disposer d'une bibliothèque d'exécution C adaptée à la cible.

On peut ensuite proposer le support d'autres langages, comme le C++, qui suppose un code d'initialisation et un script d'édition de liens bien plus complexes.

Plan

- 1 Introduction
- 2 Développement embarqué
- 3 Une chaîne de bas niveau
- 4 Communication hôte / cible
- 5 Débogage HW
- 6 Conclusion

En plus d'une chaîne de compilation, on a besoin d'un moyen pour communiquer avec la cible, simplement pour pouvoir charger l'image de l'application dans sa mémoire. On peut vouloir en plus, par exemple :

- Contrôler plus précisément la cible (démarrage / arrêt du programme, réinitialisation...)
- Examiner l'exécution du programme (placement de points d'arrêt, examen de l'état des processeurs...)

À cette fin, toutes les cibles disposent au moins d'une interface de programmation qui permet le transfert d'une image mémoire. Cette interface peut être

- Propriétaire, nécessitant l'utilisation du matériel et logiciel de programmation fourni (et vendu) par le concepteur du système (Interfaces ICSP sur les microcontrôleurs PIC, ST-LINK sur les SoC STMicro par exemple)
- Standard ou normalisée, ce qui offre plus de souplesse dans le choix du matériel et du logiciel (existence de solutions libres par exemple) : interfaces RS-232, JTAG.

Les interfaces JTAG

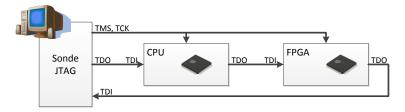
Le standard JTAG

Sur les cibles complexes, on trouve en général une interface au standard JTAG (*Joint Test Action Group*, norme IEEE 1149.1).

- Interface série synchrone
- Créée pour permettre le test fonctionnel et physique de circuits intégrés après encapsulation
- Le protocole d'échange admet des extensions
- Ces extensions permettent de nombreuses fonctionnalités supplémentaires, comme le débogage matériel ou la prise de contrôle totale d'une cible par un hôte

Connexion à la cible

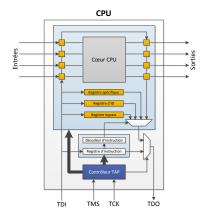
La connexion à la cible se fait au travers d'une Sonde JTAG



- Les composants sont identifiés par un numéro unique pour chaque type
- Les composants sont reliés en série et forment une chaîne JTAG
- L'hôte communique avec la sonde au moyen d'une liaison série, USB ou parallèle

Le TAP (*Test Access Port*)

Chaque composant qui supporte le standard JTAG doit être muni de logique supplémentaire qui permet à la sonde d'accéder aux ressources internes de ce composant : le *Test Access Port* ou *Debug Access Port*



- Les signaux internes du processeur sont activés en fonction de l'instruction JTAG reçue dans le registre d'instructions et du contenu de registres spécifiques
- Un TAP/DAP peut être configuré pour rendre le composant transparent sur la chaîne (bypass)

Le logiciel

Une sonde JTAG est une interface matérielle. Il faut un logiciel pour la piloter.

- Logiciels propriétaires, adaptés à un outil de développement ou une API spécifique
- Logiciel libre OpenOCD, capable de gérer la plupart des sondes du marché (en particulier pour communiquer avec des cibles ARM).

OpenOCD est un logiciel client / serveur.

- Côté serveur, il émet des instructions aux cibles au travers d'une sonde JTAG
- Côté client, il accepte des ordres venant de débogueurs comme GDB ou fournis par l'utilisateur en ligne de commande par une interface Telnet

Pour fonctionner, il a besoin d'un fichier de configuration qui décrit la chaîne JTAG connectée à la sonde.

- Lancement par openocd -f configfile
- Connection à l'interface de commande par telnet localhost 4444
- reset init réinitialise la cible
- halt stoppe l'exécution en cours et rend le processeur réceptif à d'autres commandes.
- resume reprend l'exécution à la valeur du compteur de programme ou à l'adresse spécifiée.
- load_image nom_image addr charge le fichier image mémoire « nom_image » à l'adresse spécifiée.
- flash write_image erase unlock nom_image addr copie le fichier image « nom_image » dans la mémoire flash de la cible.

- 1 Introduction
- 2 Développement embarqué
- 3 Une chaîne de bas niveau
- 4 Communication hôte / cible
- 5 Débogage HW
- 6 Conclusion

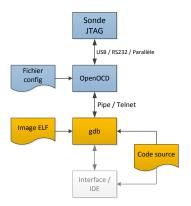
Il s'agit de la prise de contrôle de la cible au travers d'un débogueur pour suivre l'exécution du programme en détails, examiner l'état du processeur...

Deux types :

- Le débogage in-circuit (ICD, In-Circuit Debugging). Dans ce cas, le processeur réel est utilisé et le programme s'exécute dessus.
- L'émulation in-circuit (ICE, In-Circuit Emulation). Dans ce cas, un émulateur matériel du processeur cible est utilisé à sa place.

En pratique, sur les cibles modernes, quelques fonctionnalités réservées à l'ICE sont disponibles en ICD (positionnement de points d'arrêt matériels ou de données par exemple).

- Il offre une plus grande visibilité sur « ce qui se passe » dans la cible.
- Il permet d'étudier le fonctionnement de code habituellement difficile à déboguer (noyau temps-réel ou gestionnaires d'interruptions).



Un débogueur hardware complet peut être réalisé en combinant GDB et openOCD. Il permet alors de faire du débogage au niveau source.

Conclusion

La combinaison de tous ces éléments permet d'obtenir une chaîne complète de développement / débogage efficace pour de nombreux systèmes embarqués, non limitée à une architecture particulière.

Certains produits commerciaux comme CodeSourcery de Mentor Graphics ou Code Composer Studio de Texas Instruments sont d'ailleurs basés sur ces même outils.