Bare-Metal Development
M2M Lectures
Grenoble University
Your Names Here
January 12, 2021

1) Preface by Pr. Olivier Gruber

This document is your work log for the first step in the M2M course, master-level, at the University of Grenoble, France. You will have such a document for each step of our course together.
This document has two parts. One part is about diverse sections, each with a bunch of questions that you have to answers. The other part is really a laboratory log, keeping track of what you do, as you do it.
The questions provide a guideline for your learning. They are not about getting a good grade if you answer them correctly, they are about giving your pointers on what to learn about.
The goal of the questions is therefore not to be answered in three lines of text and be forgotten about. The questions must be researched and thoroughly understood. Ask questions around you if things are unclear, to your fellow students and to me, your professor.
Writing down the answers to the questions is a tool for helping your learn and remember. Also, it keeps track of what you know, the URLs you visited, the open questions that you are trouble with, etc. The tools you used. It is intended to be a living document, written as you go.
Ultimately, the goal of the document is to be kept for your personal records. If ever you will work on embedded systems, trust me, you will be glad to have a written trace about all this.
REMEMBER: plaggia is a crime that can get you evicted forever from french universities... The solution is simple, write using your own words or quote, giving the source of the quoted text. Also, remember that you do not learn through cut&paste. You also do not learn much by watching somebody else doing.

2) QEMU

What is QEMU? Why is it necessary here?

→ QEMU est utilisé pour émuler du matériel : ici on veut émuler une “ARM-based board, the VersatilePB board” (README - ligne 2), on peut parametrer la machine émuler : la quantité de RAM, le type de processeur etc …
C’est nécessaire ici car nous n’avons pas d’exemplaire de carte chez nous, on se contente donc de simuler son comportement de façon logicielle.

Read the README-QEMU-ARM.

Try it... with ”make run”
→ les installations sur mon environnement de travail on été faites : ça tourne

3) GNU Debugger

Here, try single-stepping the code via the GNU debugger (GDB).
Read the README-GDB.
Try it, in one shell, launch QEMU in debug mode and in another shell launch gdb.

→ dans un terminal :

* make debug

ça lance le programme sur qemu avec l'option -S etc pour permettre à gdb de se connecter sur le port 1234.

Dans un autre terminal :

* gdb-multiarch kernel.elf

quand le prompt gdb apparait :

* target remote:1234

A partir de la on peut utiliser gdb de façon normale en ligne de commande.

4) Makefile

You need to read and fully understand the provided makefile. Please find a few questions below highlighting important points of that makefile. These questions are there only to guide your reading of the makefile. Make sure they are addressed in your overall writing about the makefile and the corresponding challenge of building bare-metal software.

1. What is the TOOLCHAIN?

La "Tool chain" définit l'ensemble des paquets utilisé pour la compilation d'un programme,
ici on veut cibler une architecture ARM donc on utilise "arm-none-eabi".

2. What are VersatilePB and VersatileAB?

On a vu que l'on pouvait définir la machine émulé par qemu, on voit que le makefile définit la variable "MACHINE" comme étant égal à " VersatilePB" et " VersatileAB" : Ce sont les machines que l'on va émuler avec la commande "make run"

3. What is a linker script? Look at the linker option "-T"

Le "linker script" est la suite de commande qui va faire le lien entre les différents .o obtenu par la compilation des fichiers .s, pour obtenir un .elf → l'executable que l'on obtient à la fin de la chaine de compilation.

L'option "-T" sert à lire le fichier .ld qui définit comment doivent être lié les différents .o (voir "man arm-none-eabi-ld") pour contrôler la façon dont les fichiers objets seront liés.

4. Read and understand the linker script that we use

Le fichier "kernel.ld" définit le format du fichier de sorti : kernel.elf (interprétation grâce au lien suivant : https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_chapter/ld_3.html#SEC17) :

- Définition du point d'entrée du programme :
l'adresse de la première instruction qui sera executée au lancement du programme
- Définiton des différentes sections :
.startup : zone text de startup.o
.text : zone text de tous les autres .o : le code de tous les autres fichiers
.data : zone data de tous les .o
.bss : pas compris

5. Why do we translate the ”kernel.elf” into a ”kernel.bin” via ”objcopy”

C’est grâce à cela que le code peut être chargé en mémoire sur la board, la board ne peut pas comprendre les fichiers .elf.

6. What ensures that we can debug?

C’est le fait que les .o sont obtenu par une compilation qui utilise l’option **“-g”** : le code compilé est un peu plus lourd car il contient des informations qui permettent le debogage par gdb.

7. What is the meaning of the ”-nostdlib” option? Why is it necessary?

C’est pour ne pas inclure la librairie standard lors de la compilation des fichiers .c, nécessaire ici car cela voudrait dire que la librairie est définie quelque part sur la carte émulé, or ici ce n’est pas le cas (besoin de précisions).

8. Try MEMORY=32K, it fails, why? Look at the linker script.

→ (3) Your guest kernel has a bug and crashed by jumping off into nowhere
Dans kernel.ld on définit le point d’entrée à 0x100000, si on que a 32kB de mémoire, un tel point d’entrée est en dehors de la zone mémoire aloué par qemu pour émuler la machine.

9. Could you use printf in the code? Why?

Non on ne peut pas utiliser de printf dans le code, c’est une fonction de la librairie standard donc comme on ne l’inclue pas dans la compilation notre programme n’aurait pas connaissance du code de la fonction printf et ne pourrait donc pas l’utiliser.

4.1) Linker Script
Detail here your understanding of the linker script that we use.
Why do we translate the ”kernel.elf” into a ”kernel.bin” via ”objcopy”
C’est grâce à cela que le code peut être chargé en mémoire sur la board, la board ne peut pas comprendre les fichiers .elf.

Why do we link our code to run at the 0x10000?
Car il semble que c’est à cette adresse que le processeur va commencer à lire le code quand il va “se réveiller”

Why do we make sure the code for the object file ”startup.o” is first?
Car c’est le bout de code qui permet de bouger le code à l’adresse 0x10000, si ce n’est pas déjà le cas et cela doit être fait avant de commencer à exécuter le reste de notre code.

4.2) ELF Format
What is the ELF format?
→ D’après Wikipedia (https://fr.wikipedia.org/wiki/Executable_and_Linkable_Format) :

Chaque fichier ELF est constitué d'un en-tête fixe, puis de segments et de sections. Les segments contiennent les informations nécessaires à l'exécution du programme contenu dans le fichier, alors que les sections contiennent les informations pour la résolution des liens entre fonctions et le replacement des données.

Why is it used as an object file format and an executable file format.
Il est utilisé comme executable par qemu par exemple qui est capable de comprendre le code qui est représenté à l'intérieur, mais il est est aussi utilisé comme fichier objet par objcopy pour être traduit en binaire afin de pouvoir être éxécuté sur la board.

How does the ELF executable contain debug information? Which option must be given to the compiler and linker? Why to both?
En faisant arm-none-eabi-objdump -D kernel.elf j'ai vu les sections ".debug_line" ".debug_info" ".debug_abbrev" ".debug_aranges" ".debug_str" ".debug_frame", j'imagine que ces sections sont celles qui ont été généré à la compilation grâce à l'option "-g" donné à gcc.
Il y a une option donné au linker pour avoir des infos de debug ? Il me semble que les sections que j'ai décris plus haut sont déjà présentes dans les .o, le linker script ne fait que prendre tout le contenu des .o.

Confirm what ELF object files and the final ELF executable are with the shell command "file".
Il y a deux fichiers ELF ?
$ file kernel.elf
kernel.elf: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), statically linked, with debug_info, not stripped
Le fichier est bien reconnu comme un executable.

Look at the ELF object files and the final ELF executable with the tool: arm-none-eabi-objdump.
→ fait, utilisé pour répondre et comprendre les questions d'avant

5) Startup Code
Read and understand the startup code in the file "startup.s".
Explain here what it does.

Ce morceau de code regarde l'endroit en mémoire ou est chargé ce programme (_load), et si cette adresse est différente de _start le code est déplacé à _start

6) Main Code
Read and understand the main code in the file "main.c".

Explain here what it does. In particular, explains how the characters you type in the terminal window actually appear on the terminal window.
With a regular shell, the shell echoes the character as you type them. It only sends the characters once you hit the return key, as a complete line.
It is the behavior you notice here?

Dans main.c il y a :

une fonction qui permet d'afficher un caractère sur la sortie standard
une fonction qui utilise la précédente pour affciher une chaine de caractère
Une fonction qui récupère des caractères sur l'entrée standard

Le point d'entrée du programme est définit comme étant la fonction c_entry() dans le linker script, cette fonction boucle indéfiniment sur une portion de code qui :

tous les 50 millions de tours de boucle affichage du message "Zzzz…." pour montrer à l'utilisateur que le programme tourne toujours.

S'il y a eut une entrée saisie par l'utilisateur sur l'entrée standard : affichage sur la sortie standard, sinon rien.

1. What is an UART and a serial line?

UART = un composant physique qui transforme un signal electrique en bit.
ligne serie = la ou circule les bits de données

2. What is the purpose of a serial line here?

Pour permettre la communication avec la board pour obtenir des entrées / fournir des sorties (saisies clavier/ affichage à l'écran).

3. What is the relationship between this serial line and the Terminal window running a shell on your laptop?

Le terminale remplace la ligne série : c'est par le terminal que l'on passe pour donner les entrées clavier à qemu, et c'est ici que s'affiche ce que qemu renvoie

4. What is the special testing of the value 13 as a special character and why do we send back '\r' and '\n' ?

Caractère 13 sur la table ASCII = carriage return : c'est pour afficher un retour à la ligne sur le terminal quand on appuie sur la touche "entrée".
Car quand la simulation de la board renvoie /n c'est ensuite interprété par notre terminal qui comprend que cela correspond a descendre d'une ligne, et /r pour revenir à la colonne 0

5. Why can we say this program polls the serial line? Although it works, why is it not a good idea?

Car ce programme lit en boucle sur la ligne série dans une boucle infinie, ce n'est pas une bonne idée car les capacités de calcul sont utilisées pour une attente active de contenue sur la ligne série.

6. How could using hardware interrupts be a better solution?

Ceci permet au système de se mettre en pause jusqu'a ce qu'il recoive un signal : moindre consommation énergietique.

7. Could we say that the function uart send may block? Why?

Cela peut bloquer si le buffer de transmission est plein

Ligne de code suivante :

```
while (*uart_fr & UART_TXFF);
```

8. Could we say that the function uart receive is non-blocking? Why?

C'est non bloquant car s'il n'y a rien a récupérer : on renvoie 0, sinon on renvoie 1 et ce qui a été saisie est stoqué dans la variable pointé par le pointeur donné en paramètre.

9. Explain why uart send is blocking and uart receive is non-blocking.

La fonction d'envoie est bloquante car le programme fait envoyer au processeur des bits d'informations à l'uart, on est donc limité par le buffer de transmission

La fonction uart receive est non bloquante car elle lit sur le buffer de réception de l'uart, s'il est vide on n'a pas récupéré de donnée, sinon on en récupère, ceci sans blocage possible

7) Test Code

7.1) Blocking Uart-Receive

Change the code so that the function uart receive is blocking.
Why does it work in this particular test code?
Why would it be an interesting change in this particular setting?
J'ai écris une fonction "uart_receive_blocking" similaire à la fonction "receive" de base en mettant une boucle while similaire à la fonction "send".
Je ne vois pas particulièrement pourquoi c'est plus intéressant, ni ce qui fait que que cela la marche en particulier dans notre code de test

7.2) Adding Printing

We provided you with the code of a kernel-version of printf, the function called "kprintf" in the file "kprintf.c".
Add it to the makefile so that it is compiled and linked in.

J'ai modifié le makefile, et crée in fichier kprintf.h que j'ai inclue dans main.c

Look at the function "kprintf" and "putchar" in the file "kprintf.c". Why is the function "putchar" calling the function "uart send"?

Car c'est la fonction qui permet d'envoyer un caractère vers la sortie standard

Use the function kprintf to actually print the code of the characters you type
and not the characters themselves.
Hit the following special keys:
• left and right arrow.
• backspace and delete key.
Explain what you see.
Fleche gauche : 27 + 91 + 68 (ESC + [+ D)
fleche droite : 27 + 91 + 67 (ESC + [+ C)
backspace : 127 (DEL)
delete : 51 + 126 (3 + ~)
On voit que les flèches de gauche et de droite ainsi que la touche delete sont interprêtés : la board renvoie plusieurs caractères pour les représenter

7.3) Line editing

The idea is now to allow the editing of the current line:
• Using the left and right arrows
• Using the ”backspace” and ”delete” keys
First, experiment using the left/right arrows... and the backspace/delete keys...
• Explain what you see
• Explain what is happening?
Now that you understand, write the code to allow for line editing.

(je passe cette partie pour le moment afin de pouvoir passer à la suite j’y reviendrai plus tard)

8) Laboratory Log

Interruptions handling :

J’ai crée les fichiers uart.c et uart.h, en y mettant le contenu du main.c et main.h actuel qui sont en rapport avec l’uart (receive / send), j’ai ajouté les règles de compilation nécessaire au makefile puis changé les includes dans les fichiers concernés.
J’ai aussi ajouté la déclaration

Ce que j’ai compris du chapitre “Processor modes” et “Exceptions” de la doc de l’architecture ARMv5 :

De base le processeur execute le programme en mode “user” (ou “Supervisor” étant donné que notre programme “remplace l’os”), c’est à nous de la faire passer dans un autre mode dit “privilégié” ou “mode d’exception” pour avoir accès à d’autres ressources sur le matériel physique depuis le programme.
Nous voulons donc passer en mode d’éxécution IRQ pour pouvoir gérer l’évenement externe au système “un caractère est tapé au clavier et arrive par la ligne série”.
Quand une exception est levée, l’execution du processeur est forcé sur une adresse donnée (appellé “exception vector”) en fonction de quel type d’exception il s’agit : pour une exception IRQ c’est l’adresse : (voir doc)
Les interruptions sont désactivées quand le bit “I” dans le registre CPSR est a 1, s’il est à 0, ARM va regarder s’il y a des “regular interruption request”.

Le CPSR (Current Program Status Register)est un registre particulier qui contient : l’état de certains flags, le mode de processeur courrant, et d’autres informations de status et de contrôle
Le SPSR (Saved Program Status Register) est un registre qui sert a sauvegarder l’état du CPSR en cas de gestion d’une exception.

Certains registre ne sont pas “partagés” entre les différents mode de processeur, par exemple en mode IRQ les registre 13 et 14 (et) ne sont pas les mêmes qu’en mode Supervisor par exemple.

Le processeur ne peut gérer qu’une seule interruption à la fois (on se limite à cela dans notre cas du moins)

Il y a un registre spécial (UART Masked Interrupt Status Register) qui indique selon la valeur qu’il contient, s’il y a une exception autorisées de l’UART qui est levée. Registre en read-only
Il y a un registre le (UART Interrupt Mask Set/Clear Register) qui indique si l’UART est autorisé à lever des exceptions. Registre read/write : on peut modifier ici si on autorise les exceptions de l’UART ?

Il y a également un registre pour indiquer que l'on a finit de traiter l'exception de l'UART (UART Interrupt Clear Register), il faut y écrire 1 pour indiquer que l'on a terminé de traiter l'interruption. Il est nécessaire de dire à l'UART que l'on a finit de traiter son exception pour qu'il puisse de nouveau en lever une quand il en aura besoin.

Donc pour pouvoir gérer les exceptions sur notre board et pouvoir les utiliser dans notre contexte il faut dans un premier temps changer des paramètres contenus dans des registres spéciaux afin que le système détecte les exceptions levé par l'UART quand un caractère est tapé au clavier.
Il faut faire le nécéssaire pour que le processeur passe en mode "IRQ" quand une telle exception est levée, afin de traiter l'exception.
Il faut appeler la fonction _wfi depuis notre programme C pour endormir le processeur quand cela est nécéssaire.
Il faut définir une structure de donnée qui sera remplie par les caractères tapé au clavier : ce sera stoquée en mémoire lorsque le processeur sera en mode "IRQ" : la pile n'étant pas la même (registre r13) quand le processeur est en mode "IRQ" que quand il est en mode "Supervisor", il faut définir dans le linker script un espace mémoire pour cette nouvelle pile, et faire le paramétrage du mode IRQ pour que r13_IRQ pointe sur cet espace mémoire.

La partie "ISR" (Interrupt Service Routine) est lancé par "_irq_handler" en fonction de qui a levé une exception IRQ, dans notre cas on el lancera quand l'UART aura levé son exception.
ISR gerera donc la manière dont l'exception est traitée : lecture des caractères au niveau de l'UART et remplissage du buffer circulaire tant qu'il y a des caractères / tant qu'il y a de la place dans le buffer
Il faut ensuite faire des manipulations sur des registres particuliers pour dire à l'UART qu'on a finit de traiter son exception, l'exécution du processeur repasse en mode "Supervisor" et notre programme principal va reprendre son execution normale : lire dans le buffer les caractères qui ont été mit pour les affiché à l'écran.

TODO:
Explication des fonctions dans exceptions.s :

_wfi :
_irqs_setup :
_irqs_enable :
_irqs_disable :
_copy_vector :
_irq_handler :

autres handlers: on ne veut rien mettre en place dans notre cas pour les autres type d'exceptions