

## TP5

Objectif : Installer le compilateur ARM aarch64, compiler un premier exemple de code « bare metal » multi-cœurs, charger le code sur la carte SBC Raspberry Pi 3 et avec traçage par console série.

Matériel et logiciel nécessaire :

- Carte Raspberry Pi 3 B+
- Carte microSD pour la B
- Adaptateur PC pour carte microSD
- Câble console USB<->UART compatible Raspberry Pi
- Alimentation micro-USB pour Raspberry Pi 3 B+
- Lecteur de cartes SD de votre portable
- Les sources C (demo-self-contained.tar.gz)
- Connexion internet
- Temporairement, pour tester la carte SD, un câble HDMI, un moniteur, un clavier et une souris (un ou deux ensembles devraient suffire à tout le monde).

### 1. Mise en œuvre matérielle de la carte

#### a. Création de la carte SD de démarrage compatible Raspberry Pi

Suivre les instructions d'installation de Raspberry Pi OS, disponibles ici :

- <https://www.raspberrypi.org/software/>

La procédure vise à installer Raspberry Pi OS sur une carte SD. Pour vérifier que l'installation s'est bien passée, il faut :

- Placer la carte SD dans la Raspberry Pi
- Connecter la carte Raspberry Pi à un moniteur (par HDMI), à un clavier et à une souris (par USB).
- Lancer l'exécution en connectant le câble d'alimentation.
- Une fois l'OS lancé, l'arrêter proprement (par l'interface graphique). Arrêter l'alimentation brutalement peut endommager les données sur la carte SD.

Pour ceux qui ont déjà une carte servant à l'autre cours :

- Vous pouvez en principe vous servir de cette carte, sans avoir à la changer.
- ATTENTION : il vous faut faire très attention, car corrompre la carte SD implique une corruption potentielle du disque pour l'autre cours. Attention donc au point (b).

Ouvrir dans la partition "bootfs" de la carte SD le fichier config.txt et placer à sa fin les lignes suivantes

```
[all]
enable_uart=1
core_freq=250
```

Ces lignes forcent l'activation du lien série UART et fixent la fréquence du GPU.

### b. Connexion électrique du câble UART à la Raspberry Pi

Brancher le câble USB/UART à la carte Raspberry Pi. Pour ce faire, servez-vous du plan (fig. 1) des GPIO de la carte Raspberry Pi et du pin-out du câble console.

**GPIO Numbers**

Raspberry Pi B Rev 1 P1 GPIO Header			Raspberry Pi A/B Rev 2 P1 GPIO Header			Raspberry Pi B+ B+ J8 GPIO Header		
Pin No.			Pin No.			Pin No.		
3.3V	1	2	3.3V	1	2	3.3V	1	2
GPIO0	3	4	GPIO2	3	4	GPIO2	3	4
GPIO1	5	6	GPIO3	5	6	GPIO3	5	6
GPIO4	7	8	GPIO4	7	8	GPIO4	7	8
GND	9	10	GND	9	10	GND	9	10
GPIO17	11	12	GPIO17	11	12	GPIO17	11	12
GPIO21	13	14	GPIO27	13	14	GPIO27	13	14
GPIO22	15	16	GPIO22	15	16	GPIO22	15	16
3.3V	17	18	3.3V	17	18	3.3V	17	18
GPIO10	19	20	GPIO10	19	20	GPIO10	19	20
GPIO9	21	22	GPIO9	21	22	GPIO9	21	22
GPIO11	23	24	GPIO11	23	24	GPIO11	23	24
GND	25	26	GND	25	26	GND	25	26
						DNC	27	28
						GPIO6	29	30
						GPIO6	31	32
						GPIO13	33	34
						GPIO19	35	36
						GPIO26	37	38
						GND	39	40

Key

Power+	UART
GND	SPI
IC	GPIO

Figure 1. Plan des GPIO de la carte Raspberry Pi (pris sur raspberrypi.org). La RaspberryPi 3 B+ correspond au schéma de droite.

Sur ce plan, identifier les GPIO 14 et 15 qui correspondent respectivement au TX et RX du périphérique UART0. En faisant l'hypothèse que vous utilisez un câble Adafruit à 4 pins, il faut connecter :

- le fil noir sur le pin 6 (GND)
- le fil blanc sur le pin 8 (GPIO14)
- le fil vert sur le pin 10 (GPIO15)
- le fil rouge reste non-connecté (attention qu'il ne touche pas à d'autres pins)

Pour les câbles à 3 pins, il faut connecter :

- le fil noir sur le pin 6 (GND)
- le fil jaune sur le pin 8 (GPIO14)
- le fil orange sur le pin 10 (GPIO15)

**ATTENTION – une mauvaise connexion des pins peut facilement endommager la carte Raspberry Pi.**

## 2. Installation d'un compilateur ARM 64 bits (aarch64)

L'objectif de cette étape est l'installation du « cross-compileur » capable de produire sur votre machine x86 des binaires aarch64 s'exécutant sans l'aide d'un OS et avec l'ABI (application binary interface) EABI (embedded ARM ABI).

Je déconseille la compilation de ce compilateur à partir de sources. Elle peut être réalisée sous tout OS moderne, mais demande une bonne pratique de compilation de projets de très grande taille (et souvent une certaine perte de temps). Dans le doute, voici un tutoriel sur comment on compile un cross-compileur (et les autres logiciels nécessaires à la compilation, groupés dans le package « binutils ») :

[https://github.com/bztsrc/raspi3-tutorial/tree/master/00\\_crosscompiler](https://github.com/bztsrc/raspi3-tutorial/tree/master/00_crosscompiler)

La procédure d'installation que je recommande implique l'utilisation de paquets standards précompilés. Le nom de ces paquets est :

- Linux Debian/Ubuntu – gcc-aarch64-linux-gnu
- ArchLinux – aarch64-linux-gnu-gcc
- NixOS – aarch64-none-elf
- MacOS – installation avec HomeBrew, le nom du « cask » est gcc-aarch64-embedded

Pour une machine Windows, ou si vous avez des problèmes à installer le compilateur, le plus simple est de créer une machine virtuelle Ubuntu sous VirtualBox et y installer le package gcc-aarch64-linux-gnu.

### **3. Compilation et installation des sources**

La décompression du fichier demo.tar.gz produit la structure de répertoires suivante :

```
demo - aarch64-cortexA53-rpi3-runtime
|
|--- librpil3
|    Low-level aarch64/cortexA53/rpi3 libraries
|--- bootloader-rpi-fixed (*)
|    Minimal bootloader
|    Résultat de la compilation : kernel8.img, à placer
|    directement sur la carte SD
|--- bootloader-rpi-loaded (*)
|    Higher-level bootloader, itself loaded
|    Résultat de la compilation : kernel8.img, à placer
|    dans le répertoire « bootloader-host » sous le nom boot.img
|--- application (*)
|    Application loaded by the bootloader
|    Résultat de la compilation : kernel8.img, à placer
|    dans le répertoire « bootloader-host » sous le nom el0.img
|--- bootloader-host (*)
|    Hostserver-part of the bootloader
|    Résultat de la compilation : boot
|--- simple-example
|    |--- gen-t1042-threads
```

Dans les TP suivants, vous devrez modifier ce code. Toujours faire une copie de cette hiérarchie (ne jamais modifier l'original, pour avoir une référence). Pour commencer, créer une copie demo-test de cette hiérarchie.

Le processus de compilation doit être appliqué séparément dans chacun des répertoires marqués avec (\*) dans la hiérarchie.

- La compilation des premiers 3 répertoires marqués de (\*) doit être réalisée avec le cross-compilateur aarch64. Avant de réaliser la compilation, ouvrir le fichier « Makefile », y rechercher la définition de la variable « ARMGNU » et la modifier avec le bon préfix de votre chaîne d'outils de compilation. La version existante « aarch64-none-elf » est correcte si le cross-compilateur à appeler est « aarch64-none-elf-gcc ».

Appelez « make » dans chacun des 4 répertoires. Pour les premiers 3 répertoires, le résultat de compilation est appelé « kernel8.img » (c'est une image binaire, directement exécutable sans chargement compliqué comme pour un fichier en format ELF).

- Le premier doit être placé sur la carte SD, dans la partition « bootfs », à la place du kernel8.img existant. Si la carte SD doit servir par la suite, sauvegarder les fichiers kernel\*.img sur le PC. En tout cas, effacer les autres fichiers kernel\*.img.
- Le second doit être copié sous bootloader-host/boot.img. Il sera automatiquement chargé sur la carte par le lien série. Cette opération de copie doit être réalisée une seule fois.
- Le troisième doit être copié sous bootloader-host/el0.img. Il sera automatiquement chargé sur la carte par le lien série. Cette opération de copie doit être réalisée à chaque fois que l'application change, car ce fichier contient le code multi-threads à exécuter.

Pour le répertoire bootloader-host, la compilation produit l'exécutable « boot » qui réalise le chargement et ensuite trace l'exécution.

#### **4. Exécution**

Celle-ci est possible seulement sous Linux ou MacOS (l'unique utilisateur de Windows devra se mettre en binôme pour l'exécution, exécuter sur une autre machine, ou installer une machine virtuelle Ubuntu en donnant à la VM accès au port COM du connecteur série).

Les instructions supposent que :

- La carte Raspberry Pi est connectée à l'ordinateur à l'aide du câble série. Suite à cette opération, un nouveau device apparaît sous /dev (vous pouvez l'identifier en comparant le contenu du répertoire avant et après connexion du câble UART au PC). Sous MacOS, ce device est typiquement /dev/cu.<que\_chose>.
- La compilation et la copie de fichiers a été réalisée.
- Le répertoire courant est bootloader-host, qui contient les fichiers « boot », « boot.img » et « el0.img ».
- L'alimentation n'est pas encore branchée à la carte Raspberry Pi.

Etape 1 : lancer le gestionnaire avec la ligne de commande :

`./boot <nom_du_device>`

Exemple :

```
bootloader-host dpotop$ ./boot /dev/cu.usbserial-1110
TTY file to open: /dev/cu.usbserial-1110
Opening tty /dev/cu.usbserial-1110 succeeded.
COMMAND:
```

Etape 2 : brancher l'alimentation sur la carte Raspberry Pi. Attendre que le message suivant est affiché :

```
RPI:-----
RPI:Loader code started and UART initialized
RPI:Base addr:0x80000 EL2
RPI:-----
```

Etape 3 : donner la commande de chargement « script ». On vous demandera le nom d'un script de chargement, il faut faire directement « enter » car le nom est celui donné par défaut (« load.cfg »). Cette commande (en noir plus bas)

déclenche le chargement (en bleu), la configuration (en vert) et l'exécution (en rouge) :

```
script
  Script to execute(load.cfg):
EXECUTING SCRIPT load.cfg
=====
SYNC: start flush writing
SYNC: finished writing
SYNC: flush reading: start
SYNC: FLUSH COMPLETE
SYNC: sync with RPi write: start.
SYNC: sync with RPi write: finished.
SYNC: sync reading: start
SYNC: SYNC COMPLETE
=====
UPLOAD boot.img to ADDR:0x90000 SIZE:0x4bd8 CRC: 0x62c0
1ko blocks:XXXXXXXXXXXXXXXXXX
RPI:RECEIVED FILE: ADDR:0x90000 SIZE:0x4BD8 CRC:0x62C0
=====
UPLOAD el0.img to ADDR:0x80000 SIZE:0x1cf8 CRC: 0x541d
1ko blocks:XXXXXXX
RPI:RECEIVED FILE: ADDR:0x80000 SIZE:0x1CF8 CRC:0x541D
=====
RUN command with address 0x90000
RPI:RUN command. Branching to address: 0x90000
RPI:-----
RPI:Core 0 entered in EL2.
RPI:Core 0: MMU initialized at EL2.
RPI:Core 0: EL1 stack and registers set from EL2.
RPI:Core 0: Prepare branch to EL1.
RPI:C0:Page table created.
RPI:C0:MMU initialized.
RPI:C0: execute mmu_init on CPU1...Done
RPI:C0: execute mmu_init on CPU2...Done
RPI:C0: execute mmu_init on CPU3...Done
RPI:C0:=====start=concurrent=execution=====
RPI:Core1: f()->(x=173)
RPI:Core0: f(z_0=123)->(y=143)
RPI:Core0: h(x=173,y=143)->(z_1=316)
RPI://=====cycle start=====
RPI:Core0: f(z_0=316)->(y=336)
RPI:Core1: f()->(x=223)
RPI:Core0: h(x=223,y=336)->(z_1=559)
RPI://=====cycle start=====
RPI:Core0: f(z_0=559)->(y=579)
RPI:Core1: f()->(x=273)
RPI:Core0: h(x=273,y=579)->(z_1=852)
...

```

Pour arrêter l'exécution, faire CTRL-C, et ensuite débrancher l'alimentation de la Raspberry Pi. La connexion par câble série peut rester en place pour la longueur du TP.

L'exécution est celle de l'application simple décrite dans le cours.