

# Environnement Linux embarqué et programmation noyau Linux

L'ensemble des modules et programmes utilisateurs du deuxième et troisième laboratoire sont mis à disposition en annexe.

## Laboratoire 1 : Environnement Linux embarqué

### Déploiement

L'objectif de ce premier laboratoire est d'avoir un environnement de développement fonctionnel sous Docker. Afin de faciliter les tests, le protocole CIFS / SMB est utilisé. Il permet d'accéder à des dossiers du conteneur depuis la cible. Cela permet de ne pas avoir à devoir flasher à nouveau la carte SD à chaque modification d'un programme.

Le conteneur ne fonctionnant pas directement, il est nécessaire de choisir une version précise de Ubuntu. La modification est effectuée dans le fichier ".devcontainer/toolchain/Dockerfile".

```
ARG VARIANT=ubuntu-22.04
```

*Figure 1. Version fonctionnelle de Ubuntu*

Le conteneur, étant démarré, je récupère buildroot avec "sudo get-buildroot.sh". Je le compile en me rendant dans /buildroot et en effectuant "make -j8".

Suite à la compilation, je copie l'image de la carte SD dans le workspace avec "cp /output/images/sdcard.img /workspace/". Ce dossier étant partagé, cela permet d'échanger des informations entre l'hôte et le conteneur.

Depuis la machine hôte, je flashe la carte SD avec l'application Balena Etcher en utilisant l'image générée. La carte démarre correctement.

```
Starting sshd: [ 12.179500] NET: Registered PF_INET6 protocol family
[ 12.186035] Segment Routing with IPv6
[ 12.189736] In-situ OAM (IOAM) with IPv6
OK
Welcome to FriendlyARM Nanopi NEO Plus2
csel login: root
#
```

*Figure 2. Linux fonctionnel sur le Nano-Pi*

Afin de faciliter le développement, une connexion SSH est mise en place. J'indique dans les paramètres de mon PC sous Ubuntu l'adresse et le masque de la cible.

Details Identity **IPv4** IPv6 Security

**IPv4 Method**

☐ Automatic (DHCP) ☐ Link-Local Only

☒ **Manual** ☐ Disable

☐ Shared to other computers

**Addresses**

Address	Netmask	Gateway
192.168.0.4	255.255.255.0	

Figure 3. Configuration IP du port Ethernet pour le Nano-Pi

L'adaptateur USB / Ethernet ne me permet pas de me connecter via SSH. Uniquement une connexion directe via Ethernet fonctionne. Heureusement, mon PC dispose d'une interface. Pour me connecter, j'effectue "ssh root@192.168.0.14".

J'effectue la commande "uname -a" afin de s'assurer que le système d'exploitation est bien celui escompté.

```
# uname -a
Linux csel 5.15.21 #1 SMP PREEMPT Fri Feb 24 16:27:34 UTC 2023 aarch64 GNU/Linux
#
```

Figure 4. Version du kernel linux

Je monte manuellement le dossier distant "workspace" du conteneur avec la commande "mount -t cifs -o vers=1.0,username=root,password=toor,port=1445,noserverino //192.168.0.4/workspace /workspace". Le dossier doit être préalablement créé.

```
[ 1317.340190] CIFS: VFS: Use of the less secure dialect vers=1.0 is not recommended unless required for access to very old servers
[ 1317.356208] CIFS: Attempting to mount \\192.168.0.4\workspace
# cd /workspace/
# ls
README.md config sdcard.img src
```

Figure 5. Montage avec succès du dossier distant "workspace"

Afin d'éviter de saisir cette commande manuellement à chaque démarrage du Nano Pi, j'ajoute la commande proposée dans "/etc/fstab".

```
# <file system> <mount pt> <type> <options> <dump> <pass>
/dev/root / ext2 rw,noauto 0 1
proc /proc proc defaults 0 0
devpts /dev/pts devpts defaults,gid=5,mode=620,ptmxmode=0666 0 0
tmpfs /dev/shm tmpfs mode=0777 0 0
tmpfs /tmp tmpfs mode=1777 0 0
tmpfs /run tmpfs mode=0755,nosuid,nodev 0 0
sysfs /sys sysfs defaults 0 0
//192.168.0.4/workspace /workspace cifs vers=1.0,username=root,password=toor,port=1445,noserverino
```

Figure 6. Contenu du fichier /etc/fstab

Il suffit alors d'effectuer la commande "mount -a" afin de monter le répertoire distant.

```
# mount -a
[ 13.683771] Use of the less secure dialect vers=1.0 is not recommended unless required for access to very old servers
[ 13.683771]
[ 13.695906] CIFS: VFS: Use of the less secure dialect vers=1.0 is not recommended unless required for access to very old servers
[ 13.707532] CIFS: Attempting to mount \\192.168.0.4\workspace
```

Figure 7. Montage avec succès du dossier distant "workspace" avec "mount -a"

Le workspace contient uniquement les informations ajoutées par le développeur. Cependant, si une modification est effectuée dans le système de fichier racine, il faut flasher la carte. Il est plus pratique qu'il soit chargé via CIFS / SMB au démarrage.

Je crée un dossier “boot-scripts” dans “/workspace”. J’y crée le fichier “boot\_cifs.cmd”. Il contient entre autres, les paramètres fournis par U-boot au kernel. Ils permettent ici de charger le rootfs via le réseau au lieu de la carte SD.

```
setenv myip 192.168.0.14
setenv serverip 192.168.0.4
setenv netmask 255.255.255.0
setenv gatewayip 192.168.0.4
setenv hostname myhost
setenv mountpath rootfs
setenv bootargs console=ttyS0,115200 earlyprintk rootdelay=1 root=/dev/cifs rw cifsroot=//$serverip/$mountpath

fatload mmc 0 $kernel_addr_r Image
fatload mmc 0 $fdt_addr_r nanopi-neo-plus2.dtb

booti $kernel_addr_r - $fdt_addr_r
```

Figure 8. Contenu du fichier “boot\_cifs.cmd”

J’y ajoute également un Makefile afin de le compiler.

```
boot.cifs: boot_cifs.cmd
mkimage -T script -A arm -C none -d boot_cifs.cmd boot.cifs
```

Figure 9. Makefile pour compiler “boot\_cifs.cmd”

Suite à la commande “make”, j’ajoute le fichier généré “boot.cifs” dans la partition “boot” de la carte SD. J’indique à U-boot d’utiliser ces nouvelles commandes.

```
=> setenv boot_scripts boot.cifs
=> saveenv
Saving Environment to FAT... OK
=> boot
switch to partitions #0, OK
mmc0 is current device
Scanning mmc 0:1...
Found U-Boot script /boot.cifs
```

Figure 10. Utilisation du nouveau script de boot par U-boot

Le dossier chargé par la cible n’existant pas, je copie le contenu du rootfs généré. Le dossier source utilisé est “target”. Il s’agit de celui utilisé pour construire “sdcard.img”. Cela fonctionne correctement.

```
Welcome to FriendlyARM Nanopi NEO Plus2
csl login: root
```

Figure 11. Utilisation du rootfs distant par la Nano-Pi

## Automatisation du déploiement

Cette partie n’est pas demandée et permet de faciliter un nouveau déploiement. L’objectif est d’effectuer automatiquement un maximum d’opérations.

Actuellement, le rootfs distant utilisé par la cible doit être mis à jour manuellement à chaque modification de l’original. De même qu’à chaque déploiement de l’image de la carte SD, le fichier “boot.cifs” disparaît car il n’est pas inclus par Buildroot. Pour effectuer ces opérations automatiquement, j’ajoute des commandes au script appelé avant la création de “sdcard.img”. Il se trouve au chemin “board/friendlyarm/nanopi-neo-plus2/post-build.sh”.

```

#!/bin/sh
BOARD_DIR="$(dirname $0)"

install -m 0644 -D $BOARD_DIR/extlinux.conf $BINARIES_DIR/extlinux/extlinux.conf

#Custom code
#Populate distant rootfs
mkdir -p /rootfs
cp -rf /buildroot/output/target/* /rootfs
mv /rootfs/THIS_IS_NOT_YOUR_ROOT_FILESYSTEM /rootfs/THIS_IS_YOUR_REMOTE_ROOT_FS

#Compile new uboot commands to use distant rootfs
curr_dir=$(pwd)
cd /workspace/boot-scripts
make
cd $curr_dir
cp /workspace/boot-scripts/boot.cifs /buildroot/output/images/

```

Figure 12. Modification du script post-build.sh

Le script “genimage” est appelé par buildroot afin de créer les différentes partitions et l’image finale “sdcard.img”. Il utilise pour cela la configuration contenue dans “board/friendlyarm/nanopi-neo-plus2/genimage.cfg”. J’ajoute dans la partition “boot”, le fichier “boot.cifs” afin qu’il soit contenu dans cette partition.

```

image boot.vfat {
    vfat {
        label = "boot"
        files = {
            "Image",
            "nanopi-neo-plus2.dtb",
            "boot.scr",
            "boot.cifs"
        }
    }

    size = 64M
}

```

Figure 13. Modification de genimage.cfg

En ce qui concerne U-boot, suite à un déploiement d’une nouvelle image, le fichier de commande “boot.cifs” n’est plus utilisé comme script de boot. J’ai créé un fichier et ajouté un environnement par défaut dans le menu de configuration de U-boot (make uboot-menuconfig).

```

[*] Create default environment from file
(/buildroot/board/friendlyarm/nanopi-neo-plus2/init_uboot_venv) Path to default environment file

```

Figure 14. Configuration d’un environnement par défaut

```

boot_scripts=boot.cifs

```

Figure 15. Contenu de l’environnement par défaut

Malheureusement, cette solution ne fonctionne pas car l’intégralité des variables d’environnements sont remplacées. Cette solution est donc abandonnée.

La dernière action consisterait à modifier “/etc/fstab” afin d’y ajouter les paramètres de chargement distant du workspace.

## Débogage avec GDB

Le projet dispose d’un dossier “.devcontainer” contenant les différents workspaces permettant d’effectuer du débogage. J’ouvre celui de fibonacci depuis VS Code.

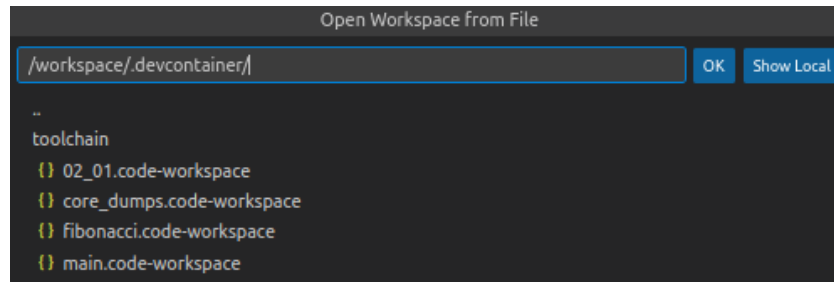


Figure 16. Menu de sélection des différents workspaces

Je compile en effectuant “make” dans le dossier “/workspace/src/01\_environment/fibonacci”. Je démarre le serveur gdb sur la cible depuis VS Code en faisant “Run Task...” en indiquant “gdbserver” puis en sélectionnant “continue without scanning the task output”.

```
* Executing task in folder fibonacci: ssh -t root@192.168.0.14 '/usr/bin/gdbserver :1234 /workspace/src/01_environment/fibonacci/app 2'
The authenticity of host '192.168.0.14 (192.168.0.14)' can't be established.
ED25519 key fingerprint is SHA256:Bilyqo48Xo0Fbeu0d/MKtcSCVjC/LUrZHNTDXPFic04.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '192.168.0.14' (ED25519) to the list of known hosts.
Process /workspace/src/01_environment/fibonacci/app created; pid = 269
Listening on port 1234
```

Figure 17. Serveur gdb fonctionnel sur la cible

Je démarre gdb depuis VSCode en faisant “Run” puis “Start Debugging”.

```
Process /workspace/src/01_environment/fibonacci/app created; pid = 275
Listening on port 1234
Remote debugging from host 192.168.0.4, port 36776
fibonacci.c, 78, Mar  3 2023, 16:06:34
The first 2 Fibonacci numbers are:
0, 1
```

Figure 18. gdb démarré et connecté à la cible

## Mode production sur une nouvelle partition

Le système d’exploitation de ma machine hôte étant Ubuntu, la création de la partition n’est pas effectuée depuis le conteneur. Suite à l’insertion de la carte SD sur la machine hôte, je démarre l’utilitaire de gestion de partitions avec “fdisk /dev/sda”. J’indique lors de la création de la partition, l’intervalle le plus grand, d’environ 10GB.

```

Command (m for help): F
Unpartitioned space /dev/sda: 12.8 GiB, 13745823744 bytes, 26847312 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes

   Start      End  Sectors  Size
   ----      -
2048      2127        80   40K
4329472 31176703 26847232 12.8G

Command (m for help): n
Partition type
   p   primary (2 primary, 0 extended, 2 free)
   e   extended (container for logical partitions)
Select (default p): p
Partition number (3,4, default 3): 3
First sector (2048-31176703, default 2048): 4329472
Last sector, +/-sectors or +/-size{K,M,G,T,P} (4329472-31176703, default 31176703): 26847232

Created a new partition 3 of type 'Linux' and of size 10.7 GiB.

Command (m for help): p
Disk /dev/sda: 14.87 GiB, 15962472448 bytes, 31176704 sectors
Disk model: STORAGE DEVICE
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x00000000

Device      Boot  Start      End  Sectors  Size Id Type
/dev/sda1   *      2128    133199    131072   64M c W95 FAT32 (LBA)
/dev/sda2           133200   4327503   4194304    2G 83 Linux
/dev/sda3      4329472 26847232 22517761 10.7G 83 Linux

```

Figure 19. Création de la partition avec fdisk

La partition étant créée, un système de fichier ext4 est installé avec “sudo mkfs.ext4 /dev/sda3”. J’y insère le script que je souhaite exécuter au démarrage.

```

#!/bin/sh -x
echo "My init script executed :)"

```

Figure 20. Installation d’un système de fichiers ext4

init.d étant utilisé pour effectuer des tâches au démarrage, je crée une copie du script pour démarrer SSH nommé “S50sshd”. Il est nommé S60myscript. Le numéro après le “S” est volontairement le plus grand de l’ensemble afin qu’il soit exécuté en dernier.

```

#!/bin/sh
#
# myscrip      myscrip
#
umask 077

start() {
    mount /dev/mmcb1k2p3 /opt
    ./opt/init_script.sh
}
stop() {
    umount /opt
}
restart() {
    stop
    start
}

case "$1" in
    start)
        start
        ;;

```

Figure 21. Contenu de S60myscript

```
pts: (null). Quota mode: none.  
+ echo 'My init script executed :)'  
My init script executed :)  
  
Welcome to FriendlyARM Nanopi NEO Plus2  
csl login: root  
#
```

Figure 22. Script correctement exécuté au démarrage

## Questions

1. Comment faut-il procéder pour générer l'U-Boot ?

Uboot est généré automatiquement lors de la compilation de Buildroot. Cette dernière s'effectue au travers de la commande "make" appelée dans le dossier de Buildroot. Si l'on souhaite générer uniquement un package, il est possible de faire "make PACKAGE\_NAME-build". Dans le cas de U-Boot, "make uboot-build".

2. Comment peut-on ajouter et générer un package supplémentaire dans le Buildroot ?

L'ensemble des packages utilisables sont présents dans le dossier "package" à la racine de Buildroot. Si l'on souhaite en ajouter un, il suffit d'ajouter un dossier contenant une structure de fichiers à respecter.

3. Comment doit-on procéder pour modifier la configuration du noyau Linux ?

Le moyen le plus simple de modifier la configuration du noyau Linux est d'utiliser la commande "make linux-menuconfig".

4. Comment faut-il faire pour générer son propre rootfs ?

Si l'on souhaite modifier légèrement le rootfs, le moyen le plus simple est d'utiliser un overlay. Dans le cas où l'on en souhaite un complètement personnalisé, il faudrait créer son rootfs dans "postbuild.sh". Ce script étant appelé avant la génération de l'image par "genimage.sh". Il faut ensuite indiquer dans "genimages.cfg", d'utiliser notre rootfs.

5. Comment faudrait-il procéder pour utiliser la carte eMMC en lieu et place de la carte SD ?

Il faut dans un premier temps flasher la carte eMMC. Ensuite, modifier le fichier "boot.cmd" utilisé par U-Boot afin de charger le noyau et le device tree depuis la carte eMMC. Les paramètres donnés au kernel doivent également être modifiés pour indiquer que le rootfs se trouve sur la eMMC.

6. Dans le support de cours, on trouve différentes configurations de l'environnement de développement. Qu'elle serait la configuration optimale pour le développement uniquement d'applications en espace utilisateur ?

La configuration optimale serait d'avoir uniquement l'espace utilisateur distant au travers CIFS/SMB. Le rootfs, kernel et uboot n'étant pas modifiés, ils n'ont pas besoin d'être distant.

Cela permet d'éviter de devoir reflasher la carte SD à chaque nouvelle modification d'un programme.

Ce laboratoire m'a permis de découvrir CIFS / SMB afin d'accéder à un répertoire distant. Cela est très pratique dans le cadre de développement embarqué. Également, je n'avais jamais effectué de débogage distant. Enfin, l'utilisation d'un environnement Docker est appréciable. Il permet d'avoir un déploiement facile sans les problèmes de virtualisations, lenteurs, stockage importants demandés par les machines virtuelles.

## Laboratoire 2 : Programmation Noyau / Modules noyaux

Ce laboratoire aborde la programmation dans le noyau Linux au travers de modules développés par nos soins. Il permet de voir quelles sont les restrictions, les actions et bibliothèques disponibles.

### Exercice 1:

L'objectif est de générer un module à l'extérieur des sources du kernel. Je crée un simple module qui affiche un message au chargement et au déchargement. Je le compile avec le Makefile mis à disposition dans le chapitre du cours "Génération d'un module".

Depuis la machine hôte, j'effectue la commande "modinfo my\_module.o".

```
filename:      /home/quentin/Documents/Master/MA_CSEL/csel-workspace/my_work/tp02/mymodule.ko
license:      GPL
description:   Personalized module for TP02 CSEL
author:       Quentin Rod <quentin.rod@hes-so.ch>
depends:
name:         mymodule
vermagic:     5.15.21 SMP preempt mod_unload aarch64
```

Figure 23. Résultat de modinfo my\_module.o

On retrouve les informations fournies au module au travers des différentes macros. Il est affiché en plus la version du kernel ainsi que l'architecture du module.

Suite au montage distant du dossier "workspace" sur la cible, j'insère le module avec la commande "insmod mymodule.ko". La commande "dmesg" affiche bien notre message de chargement du module.

```
[ 161.247763] mymodule: loading out-of-tree module taints kernel.
[ 161.254343] My module loaded
```

Figure 24. Résultat de la commande dmesg

J'affiche la liste des modules chargés par le système avec "lsmod". Le notre apparaît bien dedans.

Module	Size	Used by	Tainted: G
mymodule	16384	0	

Figure 25. Module présent dans la liste des modules chargés

Il apparaît également lors de la commande "cat /proc/modules".

```
mymodule 16384 0 - Live 0xffff80000121e000 (0)
```

Figure 26. Informations du module présents dans "cat /proc/modules"



Cette dernière commande apporte des indications supplémentaires. La 4ème colonne étant “-”, indique qu’il n’y a pas de dépendance avec d’autres modules. “Live” indique qu’il est chargé. Enfin, la valeur hexadécimal est l’adresse où est chargé le module dans le kernel.

Je retire mon module avec “rmmod mymodule.ko”. Avec “dmesg”, le message de déchargement est bien affiché, le module est par conséquent bien déchargé.

```
[ 6700.358993] My module unloaded
```

*Figure 27. Module correctement déchargé*

Afin de pouvoir utiliser modprobe pour charger le module, il est nécessaire que ce dernier soit dans le répertoire des modules dans le rootfs. Il doit également figurer dans le fichier “modules.dep” qui indique s’il possède ou non des dépendances avec d’autres modules.

Une seule ligne suffit à être ajoutée. Je me suis pour cela basé sur la documentation du kernel à l’adresse <https://docs.kernel.org/kbuild/modules.html>.

Je compile le module en effectuant la commande “make” dans le dossier du module puis “make” dans le dossier de buildroot. Cette dernière action va mettre à jour le nouveau système de fichier racine utilisé par la cible.

Sur le Nano-Pi, je charge le module avec “modprobe mymodule”.

```
[ 23.992199] mymodule: loading out-of-tree module taints kernel.  
[ 23.998722] My module loaded
```

*Figure 28. Chargement du module avec modprobe avec succès*

## Exercice 2

L’objectif de cet exercice est d’apprendre à pouvoir configurer un module en lui renseignant des paramètres à l’insertion.

Pour créer un paramètre, il suffit de créer une variable et de la fournir à la macro module\_param(). Le module réalisé reçoit le nom, prénom et l’âge d’une personne. Il écrit un message personnalisé au chargement.

Suite à la compilation et au déploiement du module, je le charge avec la commande modprobe mymodule 'firstname="Quentin"' 'lastname="Rod"' age=22.

```
[ 125.215920] My module loaded  
[ 125.218927] Hello Quentin Rod, 22 years old
```

*Figure 29. Message personnalisé affiché par le module*

## Exercice 3

La commande “cat /proc/sys/kernel/printk” retourne le contenu suivant.

```
7 4 1 7
```

*Figure 30. Contenu de “cat /proc/sys/kernel/printk”*

La signification de ces chiffres a été trouvée sur la documentation du kernel à l’adresse <https://www.kernel.org/doc/html/latest/core-api/printk-basics.html>. De gauche à droite, ils indiquent respectivement le niveau de log de la console courant, par défaut, minimum et au

démarrage. Uniquement les messages avec un niveau plus faible que le niveau de log courant de la console sont affichés sur la console.

## Exercice 4

L'objectif de cet exercice est de manipuler des structures de données mises à disposition dans le noyau ainsi que l'allocation dynamique de ressources.

Souhaitant utiliser une liste globale, il suffit d'utiliser la macro LIST\_HEAD() en lui donnant en paramètre le nom de la liste. Il n'y a pas à créer de variables au préalable contrairement à la macro INIT\_LIST\_HEAD().

Le nombre d'éléments de la liste étant donnés en paramètre au module, il est nécessaire d'effectuer une allocation dynamique. Dans le noyau, la méthode disponible est kalloc(). Ces espaces mémoires sont ensuite libérés avec kfree() lors de la désinstallation du module.

La macro list\_for\_each\_entry() permet de parcourir la liste. Elle est utilisée pour afficher chaque élément de la liste ainsi que pour la suppression.

```
# modprobe mymodule nbElements=5 'initString="hello"'
[ 53.280072] My module loaded
[ 53.283053] [ID : 0, string : hello]
[ 53.286680] [ID : 1, string : hello]
[ 53.290279] [ID : 2, string : hello]
[ 53.293886] [ID : 3, string : hello]
[ 53.297477] [ID : 4, string : hello]
#
# rmmod mymodule
[ 58.417790] My module unloaded
```

Figure 31. Liste chaînée composée de 5 éléments.

Je n'ai pas compris le mécanisme utilisé par la macro list\_entry() pour déterminer l'adresse de l'élément à partir du membre "struct list\_head" qu'il contient. De plus, les macros permettant d'effectuer un ajout prennent en paramètres uniquement le membre "struct list\_head" de notre structure.

## Exercice 5

L'objectif de cet exercice est d'accéder à des registres physiques depuis notre module.

Le module ne peut pas accéder directement à une adresse mémoire physique. Le processus doit impérativement mapper une adresse virtuelle à une adresse physique. La fonction utilisée est ioremap(). Également, il est préférable de réserver la plage d'adresse physique utilisée afin d'éviter des conflits avec d'autres pilotes. Toutefois, il est quand même possible d'y accéder sans réservation. Elle s'effectue avec la méthode request\_mem\_region().

```
# modprobe up_thermo
[ 868.015216] Impossible to reserve temperature space
[ 868.020167] Impossible to reserve id space
[ 868.024318] Impossible to reserve mac space
[ 868.028530] temp : 37204
[ 868.031391] id part 1 : 9400470482800001
[ 868.035782] id part 2 : 0
[ 868.038607] mac : 31db01020000edd2
```

Figure 32. Données affichées par le module

L'intégralité des espaces sont déjà réservés mais sont quand même lus. Par conséquent, nos réservations ne sont pas présentes dans "/proc/iomem".

L'adresse MAC s'obtient avec la commande ifconfig.

```
Link encap:Ethernet HWaddr 02:01:DB:31:D2:ED
```

Figure 33. Valeur de l'adresse MAC

En ce qui concerne la valeur de la température, elle s'obtient avec "cat /sys/class/thermal/thermal\_zone0/temp".

```
35180
```

Figure 34. Valeur de la température

On peut observer que les valeurs des températures sont proches. L'adresse Ethernet est également correcte. Uniquement l'ordonnancement des octets est différent.

## Exercice 6

L'objectif de cet exercice est de créer un thread fonctionnant en mode noyau au travers d'un module.

Pour créer un thread, il suffit d'appeler kthread\_run() en lui donnant en paramètre le nom de la fonction à exécuter. L'appel est effectué à l'initialisation de mon module. Le thread doit vérifier en continu s'il doit s'arrêter ou non en appelant la fonction kthread\_should\_stop(). L'arrêt est demandé au déchargement de mon module avec la commande kthread\_stop().

Suite à l'insertion du module, le message est bien émis par le thread à intervalle de 5 secondes.

```
[ 2352.094110] Hello from kernel ! :)  
[ 2357.214108] Hello from kernel ! :)  
[ 2362.334107] Hello from kernel ! :)  
[ 2367.454112] Hello from kernel ! :)  
..
```

Figure 35. Messages affichés par le thread

Suite au déchargement du module, il n'y a plus de message affiché, signifiant que le thread est bien arrêté.

## Exercice 7

L'objectif de cet exercice est d'effectuer une synchronisation d'un thread à l'aide d'une "waitqueue".

La fonction wait\_event() reçoit deux paramètres qui sont la "waitqueue" et la condition. Lorsqu'elle est appelée, le thread est en sommeil tant qu'il n'y a pas d'évènement et que la condition n'est pas vraie. Dans mon module, le thread receveur appelle cette méthode en continu. Lorsqu'il en sort, il rend la condition fausse pour retourner en sommeil.

La fonction wake\_up() reçoit en paramètre la "waitqueue" et permet de créer un évènement. Dans mon module, le thread notifieur effectue un appel à cette fonction toutes les 5 secondes.

Le thread notifieur est bien réveillé toutes les 5 secondes et entraîne le réveil du receveur.

```
# insmod thread_mod.ko
[ 215.789878] Notifier awake
[ 215.790015] Run kernel threads
[ 215.792778] Receiver awake
# [ 220.894353] Notifier awake
[ 220.897105] Receiver awake
[ 226.014351] Notifier awake
[ 226.017084] Receiver awake
[ 231.134344] Notifier awake
[ 231.137072] Receiver awake
[ 236.254341] Notifier awake
[ 236.257071] Receiver awake
# rmmod thread_mod.ko
[ 240.319847] Stop kernel threads
[ 241.278417] Receiver awake
```

Figure 36. Messages transmis par le thread notifieur et receveur

## Exercice 8

L'objectif de cet exercice est de réaliser une action lors d'une interruption matérielle. Celle souhaitée provient du GPIO.

La première étape consiste à réserver un port GPIO. Cela s'effectue avec la fonction `gpio_request()` en lui donnant le numéro de la pin. Il est ensuite possible de récupérer le numéro de l'IRQ correspondant en appelant `gpio_to_irq()`. Enfin, l'interruption est configurable avec `request_irq()`.

Dans mon module, la callback est identique pour chaque bouton. Cependant, le `dev_id` donné en paramètre dépend de chaque bouton. Lors du déchargement du module, la libération des ressources est effectuée avec `gpio_free()` ainsi que `free_irq()`.

```
# [ 26.712116] Button pushed k2!
[ 27.083254] Button pushed k3!
[ 27.468709] Button pushed k1!
```

Figure 37. Messages transmis suite à l'appui des boutons

J'ai particulièrement apprécié ce dernier exercice. Ayant fait beaucoup de micro-contrôleurs durant mon Bachelor, je ne pensais pas qu'il était possible de définir des callbacks à des interruptions sur Linux. Jusqu'à présent, dans mes projets personnels, je ne faisais que du polling. La latence était très élevée.

Au cours de mon Bachelor, j'ai eu l'occasion d'aborder la programmation noyau au travers de la programmation d'un driver. J'ai été agréablement surpris de l'ensemble des outils mis à disposition au programmeur. Les difficultés rencontrées sont principalement dûes au faible nombre d'articles disponibles sur le net ainsi qu'une documentation parfois inexistante.

## Laboratoire 3 : Programmation Noyau / Pilotes de périphériques

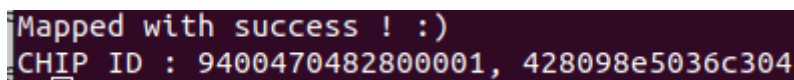
### Exercice 1

L'objectif de cet exercice est de réaliser un pilote orienté mémoire. L'avantage est qu'il est directement possible d'accéder aux registres depuis l'espace utilisateur. Il n'y a pas besoin d'utiliser un driver comme effectué dans le laboratoire précédent. La commande mmap est utilisée pour mapper les adresses physiques à des adresses virtuelles du processus. Le périphérique utilisé est "/dev/mem" car il gère la callback correspondante pour l'ensemble des adresses physiques.

L'adresse physique mappée avec mmap doit impérativement être alignée sur la taille d'une page (4096 octets).

L'adresse physique souhaitée est 0x1c14200 et correspond en décimal à 29442560.

Cela correspond au nombre de pages suivant :  $29442560 / 4096 = 7188.125$ . Comme le nombre n'est pas entier, on mappe l'adresse physique inférieure la plus proche étant alignée. Cette différence est un offset. On déplace ensuite le pointeur sur la mémoire virtuelle de cet offset afin d'avoir le registre.



```
Mapped with success ! :)  
CHIP ID : 9400470482800001, 428098e5036c304
```

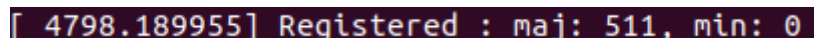
Figure 38. Chip ID récupéré par le pilote mémoire

### Exercice 2

Cet exercice a pour objectif de réaliser un périphérique virtuel de type caractère. Il doit être possible d'écrire une donnée et de pouvoir la lire.

Je reprends le modèle fourni dans le chapitre Aspects pratiques : Pilotes orientés caractère. Il répond à l'exercice sans besoin d'être modifié. Dans l'initialisation du module, la première étape consiste à allouer le numéro de pilote ainsi que ceux des périphériques. Cela se fait à l'aide de la fonction alloc\_chrdev\_region(). Il faut ensuite associer au pilote les différentes callbacks avec la fonction cdev\_init(). Il suffit enfin d'ajouter au noyau les périphériques avec la fonction cdev\_add(). Les callbacks write() et read() échangent des données avec l'espace utilisateur. Elles ne peuvent cependant pas y accéder directement. Elles utilisent pour cela copy\_from\_user() et copy\_to\_user(). Les ressources sont libérées au déchargement du module avec cdev\_del() et unregister\_chrdev\_region().

Lors de l'initialisation du module, j'affiche le numéro majeur et mineur du device créé.



```
[ 4798.189955] Registered : maj: 511, min: 0
```

Figure 39. Numéro majeur et mineur du périphérique

Le fichier dans "/dev" doit être créé manuellement. La commande utilisée est "mknod /dev/my\_char\_dev c 511 0". Elle nécessite le numéro mineur et majeur ainsi que le type de périphérique.

```

cpu_dma_latency  mmcb1k1p1  ptypc
dri              mmcb1k1p2  ptypd
fd              mmcb1k2   ptype
full            mmcb1k2p1  ptypf
fuse            mmcb1k2p2  random
gpiochip0       my_char_dev rfkill

```

Figure 40. Périphérique présent dans “/dev”

Suite à une écriture avec la commande “echo”, le texte lu avec “cat” est bien celui précédemment écrit.

```

# echo coucou > /dev/my_char_dev
[ 6384.395215] Device opened : major:511, minor:0
[ 6384.399800] Write to device: 7 bytes
[ 6384.403417] Release device
# cat /dev/my_char_dev
[ 6387.808010] Device opened : major:511, minor:0
[ 6387.812716] Read from user: 10000 bytes
coucou
[ 6388.477391] Read from user: 0 bytes
[ 6388.486536] Release device

```

Figure 41. Callbacks read() et write() fonctionnelles

### Exercice 3

L'objectif de cet exercice est d'avoir un seul pilote mais plusieurs périphériques indépendants. Le comportement doit être identique à l'exercice précédent mais les espaces mémoires entre les périphériques indépendants.

Comme le nombre d'instances est donné en paramètres, un “char \*\*” est utilisé comme tableau de chaînes de caractères et alloué dynamiquement à l'initialisation du module.

Le pilote étant le même pour chacun des périphériques, la même callback est appelée pour tous. Par conséquent, un moyen doit être mis en place pour permettre au pilote de les différencier. Les callbacks open(), write() et read() reçoivent en paramètre un “struct file\*”. Elle contient un membre libre nommé “private\_data”. La technique utilisée consiste à renseigner dedans le numéro mineur. Comme l'allocation des numéros mineurs commence à 0 et est contiguë, la valeur peut directement être utilisée comme indice du tableau des chaînes de caractères.

J'insère le module en spécifiant en paramètre le nombre de périphériques à créer avec la commande “insmod my\_char\_dev.ko nbDevices=3”. Je crée ensuite trois fichiers dans “/dev” pour chacun des devices.

- mknod /dev/my\_char\_dev\_0 c 511 0
- mknod /dev/my\_char\_dev\_1 c 511 1
- mknod /dev/my\_char\_dev\_2 c 511 2

Le comportement est bien identique à l'exercice précédent et les mémoires distinctes entre les périphériques.

```

# echo "coucou" > /dev/my_char_dev_0
[19959.883786] Device opened : major:511, minor:0
[19959.888323] (0) Write to device: 7 bytes
[19959.892284] Release device
# echo "les" > /dev/my_char_dev_1
[19966.603739] Device opened : major:511, minor:1
[19966.608269] (1) Write to device: 4 bytes
[19966.612229] Release device
# echo "amis" > /dev/my_char_dev_2
[19975.667190] Device opened : major:511, minor:2
[19975.671725] (2) Write to device: 5 bytes
[19975.675723] Release device
# cat /dev/my_char_dev_0
[19985.120312] Device opened : major:511, minor:0
[19985.124997] (0) Read from user: 100 bytes
coucou
[19985.129111] (0) Read from user: 0 bytes
[19985.138538] Release device
# cat /dev/my_char_dev_1
[19987.622336] Device opened : major:511, minor:1
[19987.627036] (1) Read from user: 100 bytes
les
[19987.631146] (1) Read from user: 0 bytes
[19987.640590] Release device
# cat /dev/my_char_dev_2
[19989.805433] Device opened : major:511, minor:2
[19989.810215] (2) Read from user: 100 bytes
amis
[19989.814298] (2) Read from user: 0 bytes
[19989.823783] Release device

```

Figure 42. Comportement identique au précédent mais mémoires indépendantes

## Exercice 4

L'objectif de cet exercice est de pouvoir utiliser notre périphérique depuis notre propre programme utilisateur et non depuis les commandes existantes comme "cat" et "echo".

Il suffit d'ouvrir le fichier correspondant au périphériques dans "/dev" avec l'appel système open() puis d'effectuer read() ou write() avec le descripteur de fichier obtenu. Dans mon module, uniquement un seul périphérique est utilisé.

De la façon dont est faite le driver, il est obligatoire d'avoir un descripteur de fichier pour la lecture et l'écriture. Chacun contient son propre offset dans le fichier. À chaque read ou write, l'offset est augmenté. Si par exemple on effectue un write puis un read, les données lues seront celles qui suivent celles écrites. Pour éviter cela, il faudrait implémenter la callback seek.

La chaîne de caractère écrite est bien celle lue ensuite.

```

[ 1938.650096] Device opened : major:511, minor:0
[ 1938.654662] (0) Write to device: 9 bytes
[ 1938.658653] Release device
[ 1938.661384] Device opened : major:511, minor:0
[ 1938.665891] (0) Read from user: 9 bytes
String read : "Coucou !"
[ 1938.670020] Release device

```

Figure 43. Lecture de données écrites depuis un programme utilisateur

## Exercice 5

L'objectif de cet exercice est d'utiliser le système de fichier "/sys/" permettant généralement de configurer un périphérique. Il doit être possible de modifier la valeur d'un attribut. Dans mon module, ce dernier est unique et partagé entre l'ensemble des périphériques. C'est au développeur de choisir où le périphérique sera accessible dans la hiérarchie. Dans mon cas,



il est possible de sélectionner si il se trouve dans le bus “platform” ou dans une “class”. Ce premier regroupe les appareils qui n’ont pas de bus permettant d’être détectés automatiquement comme SPI ou I2C. Le second représente un ensemble de périphériques définis par le développeur.

Un attribut est créé à l’aide de la macro `DEVICE_ATTR()` qui prend en paramètre le nom, le mode, la callback `show()` et `store()`. Un attribut est présenté sous la forme d’un fichier et doit donc pouvoir être lu ou écrit. La macro crée automatiquement un “struct device\_attribute dev\_attr\_X” avec X étant le nom de l’attribut. Dans mon module, l’attribut configure une chaîne de caractère.

Dans le cas du bus “platform”, la première étape consiste à créer le dossier du périphérique avec `platform_device_register()`. Il est ensuite possible d’y ajouter l’attribut avec `device_create_file()`. Ces opérations sont effectuées dans l’initialisation du module. Lors du déchargement, la suppression de l’attribut se fait avec `device_remove_file()` suivi de `platform_device_unregister()`.

Dans le cas de la “class”, il faut premièrement créer le dossier de la classe avec `class_create()`. Ensuite, créer le dossier du périphérique avec `device_create()` et enfin ajouter l’attribut avec `device_create_file()`. Lors du déchargement, la suppression de l’attribut se fait avec `device_remove_file()` suivi de `device_destroy()` et `class_destroy()`.

Lors de l’utilisation en mode “platform”, suite à l’insertion du module, deux dossiers sont apparus. Il s’agit de “./devices/platform/my\_dev\_sysfs” ainsi que “/sys/bus/platform/devices/my\_dev\_sysfs”. Leurs contenus sont identiques.

```
driver_override modalias power subsystem uevent val
```

Figure 44. Contenu des dossiers du périphérique dans “/sys” en mode “platform”

Suite à l’écriture dans le fichier de l’attribut d’une chaîne de caractère, elle est bien affichée lors de la lecture.

```
# echo "coucou" > val
# cat val
coucou
```

Figure 45. Attribut pouvant être écrit et lu en mode “platform”

Lors de l’utilisation en mode “class”, suite à l’insertion du module, un dossier est créé dans “/sys/class” nommé “my\_sysfs\_class”. À l’intérieur se trouve le périphérique sous la forme d’un répertoire

Il contient les éléments suivants.

```
power subsystem uevent val
```

Figure 46. Contenu du dossier du périphérique dans “/sys” en mode “class”

Le comportement est bien identique au mode “platform” pour l’écriture et la lecture de l’attribut.

```
# echo "coucou" > val
# cat val
coucou
```

Figure 47. Attribut pouvant être écrit et lu en mode “class”



## Exercice 5.1

Dans cet exercice, l'objectif est de fournir à l'utilisateur à la fois un périphérique dans `"/dev"` et un moyen de le configurer dans `"/sys"`. Contrairement à l'exercice précédent, il y a cette fois-ci plusieurs périphériques, le nombre étant défini par l'utilisateur. Il doit être possible de configurer indépendamment chacun des périphériques.

Les callbacks `"show"` étant `"store"` étant les mêmes pour chaque instance, il faut un moyen de les différencier. Les callbacks pour le périphérique dans `"/dev"`, utilisent le numéro mineur obtenu lors de la callback `"open"` et fourni au travers du membre `"private"` de la structure `"file"`. Les callbacks `"show"` et `"store"` reçoivent en paramètre une structure `"device"` contenant le numéro majeur et mineur du périphérique associé dans `"/dev"`. Le numéro mineur est donc également utilisé pour les différencier.

Dans mon module, le mode `"platform"` est utilisé. L'attribut `".dev.dev"` permet de créer directement les périphériques dans `"/dev"`. Les opérations de créations des fichiers dans `"sys"` et de suppression sont identiques à celles de l'exercice précédent. Elles sont simplement répétées autant de fois qu'il y a de périphériques. Egalement, un seul attribut est utilisé et le même est ajouté à chacun. Cela ne pose pas de problème car les callbacks différencient les périphériques grâce au numéro mineur.

Le module est inséré en demandant 10 périphériques avec `"insmod char_dev_sysfs.ko nbDevices=10"`. Dans `"/dev"`, il est créé autant de fichiers `"my_dev_sysfs"`. De même pour les dossiers dans `"/sys/bus/platform/devices/"`

autofs	mmcblk1boot0	1c20ca0.watchdog	my_dev_sysfs.4
bus	mmcblk1boot1	1c22c00.codec	my_dev_sysfs.5
console	mmcblk1p1	1c25000.thermal-sensor	my_dev_sysfs.6
cpu_dma_latency	mmcblk1p2	1c28000.serial	my_dev_sysfs.7
dri	mmcblk2	1c30000.ethernet	my_dev_sysfs.8
fd	mmcblk2p1	1c62000.dram-controller	my_dev_sysfs.9
full	mmcblk2p2	1e00000.deinterlace	platform-framebuffer.0
fuse	my_dev_sysfs.0	1e80000.gpu	pmu
gpiochip0	my_dev_sysfs.1	1ef0000.hdmi-phy	psci
gpiochip1	my_dev_sysfs.2	1f00000.rtc	psci-cpuidle
kmsg	my_dev_sysfs.3	1f015c0.codec-analog	reg-dummy
kvm	my_dev_sysfs.4	1f02c00.pinctrl	regulatory.0
log	my_dev_sysfs.5	Fixed MDIO bus.0'	serial8250
loop-control	my_dev_sysfs.6	alarmtimer.2.auto	snd-soc-dummy
loop0	my_dev_sysfs.7	gmac-3v3	soc
loop1	my_dev_sysfs.8	gpio-regulator	timer
loop2	my_dev_sysfs.9	usb-hdrc.1.auto	usb_phy_generic.0.auto
		my_dev_sysfs.0	vcc3v3
		my_dev_sysfs.1	wifi_pwrseq
		my_dev_sysfs.2	
		my_dev_sysfs.3	

Figure 49. Fichiers des périphériques respectivement créés dans `"/dev"` et `"/sys"`

La configuration d'un attribut est bien indépendante à chaque périphérique.

```
# echo "hey" > /sys/bus/platform/devices/my_dev_sysfs.0/val
# echo "you" > /sys/bus/platform/devices/my_dev_sysfs.2/val
# cat /sys/bus/platform/devices/my_dev_sysfs.0/val
hey
# cat /sys/bus/platform/devices/my_dev_sysfs.2/val
you
```

Figure 50. Configuration de l'attribut indépendante

## Exercice 7

L'objectif de cet exercice est de réaliser une application utilisateur qui attend passivement qu'une interruption matérielle ait eu lieu. Elle provient d'un bouton poussoir de la carte d'extension.

Afin de réaliser cet exercice, je me suis aidé de l'article [poll and select - Linux Device Drivers, Second Edition \[Book\]](#) ainsi que du forum [How do poll wait\(\) and wake\\_up\\_interruptible\(\) work in sync? - Stack Overflow](#).

L'appel système "select" permet d'attendre passivement qu'il soit possible d'effectuer une action sur un fichier. Les fichiers sont donnés en paramètres sous la forme de groupe de descripteurs de fichiers. Dans le cadre de l'exercice, uniquement la lecture doit être possible. Le descripteur de fichier est donc ajouté au groupe de lecture.

L'appel système utilisé étant "select", le driver doit implémenter la callback "poll". Elle effectue directement un appel à poll\_wait() en indiquant une "waiting queue". Si aucun événement n'a eu lieu, cette fonction modifie l'état du processus afin qu'il ne soit pas ordonnancé et qu'il soit mis en sommeil à la fin de la callback "poll". Si il y a eu un événement, la callback retourne des flags indiquant la ou les actions pouvant être effectuées. Lorsque l'interruption matérielle survient, sa callback indique l'événement à la "waiting queue" avec wake\_up() qui entraînera le réveil du programme utilisateur.

Dans mon module, la variable permettant d'indiquer qu'une action est possible est atomique. En effet, comme il n'y a pas d'anti rebond, il est possible de lire la valeur et en même temps qu'elle soit écrite par l'interruption.

Dès que le bouton est appuyé, le programme utilisateur reçoit l'indication que la lecture est possible.

```
[13297.949495] Button pushed !
[13297.949506] Read is possible by user !
```

Figure 51. Lecture possible suite à l'appui du bouton

Durant cet exercice, j'ai eu des difficultés à comprendre les mécanismes utilisés par la callback poll() comme poll\_wait().

Durant mon cursus de Bachelor, je n'avais réalisé qu'un simple driver. Le code était fourni et n'était pas expliqué par le professeur. Grâce à ce laboratoire incrémental, j'ai compris les lignes de codes utilisées ainsi que l'utilité des systèmes de fichiers comme "/sys".