

---

# **Environnement Linux embarqué et programmation noyau Linux**

---

MA\_CSEL1

15.4.2023

HES-SO Master - Master of Science in Engineering (MSE)  
**Computer Sciences**

Luca Srdjenovic & Louka Yerly

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>SP1 - Environnement Linux embarqué</b>	<b>2</b>
2.1	Travail à effectuer . . . . .	2
2.2	Réponse aux questions . . . . .	5
2.3	Synthèse sur ce qui a été appris/exercé . . . . .	7
2.4	Remarques et choses à retenir . . . . .	7
2.5	Feedback personnel sur le laboratoire . . . . .	7
<b>3</b>	<b>SP3 -Programmation Noyaux (module noyaux)</b>	<b>8</b>
3.1	Travail à effectuer . . . . .	8
3.2	Synthèse sur ce qui a été appris/exercé . . . . .	14
3.3	Remarques et choses à retenir . . . . .	14
3.4	Feedback personnel sur le laboratoire . . . . .	14
<b>4</b>	<b>SP4 - Programmation Noyau (module noyaux)</b>	<b>15</b>
4.1	Travail à effectuer . . . . .	15
4.2	Synthèse sur ce qui a été appris/exercé . . . . .	19
4.3	Remarques et choses à retenir . . . . .	19
4.4	Feedback personnel sur le laboratoire . . . . .	19
<b>5</b>	<b>SP5 - Programmation Noyau (Pilotes de périphériques)</b>	<b>20</b>
5.1	Travail à effectuer . . . . .	20
5.2	Synthèse sur ce qui a été appris/exercé . . . . .	29
5.3	Remarques et choses à retenir . . . . .	29
5.4	Feedback personnel sur le laboratoire . . . . .	29
<b>6</b>	<b>SP6 - Programmation Noyau (Pilotes de périphériques)</b>	<b>30</b>
6.1	Travail à effectuer . . . . .	30
6.2	Synthèse sur ce qui a été appris/exercé . . . . .	35
6.3	Remarques et choses à retenir . . . . .	35
6.4	Feedback personnel sur le laboratoire . . . . .	35
<b>7</b>	<b>Conclusion</b>	<b>36</b>

## 1 Introduction

Ce rapport technique a pour but d'offrir une documentation sur les laboratoires réalisés dans le cadre du cours au sujet de la programmation sous Linux : la prise en main d'un environnement Linux embarqué (buildroot) et la programmation dans le noyau Linux, plus particulièrement l'intégration de modules noyaux et de pilotes de périphérique.

Dans la première partie, nous aborderons l'installation de l'environnement de développement à l'aide de Docker, la mise en place de l'infrastructure pour la cible et la machine hôte, le debugging d'une application simple depuis la machine hôte et la mise en production de l'application. Les étapes détaillées pour chaque activité seront présentées dans les sections suivantes.

Dans la deuxième partie, nous nous pencherons sur la programmation noyaux Linux, en présentant les étapes à suivre pour programmer des modules noyaux et des pilotes de périphériques. Les sujets abordés dans cette section comprennent la gestion de la mémoire, les bibliothèques et les fonctions utiles, l'accès aux entrées/sorties, les threads du noyau, la mise en sommeil et les interruptions. En plus des sujets mentionnés précédemment, le rapport traitera également de la programmation de pilotes de périphériques orientés mémoire et orientés caractère, de l'utilisation du sysfs, les opérations bloquantes ainsi que l'écriture de pilotes orientés mémoire.

## 2 SP1 - Environnement Linux embarqué

Le travail pratique réalisé dans ce laboratoire avait pour objectif de mettre en œuvre un système embarqué sous Linux en utilisant Docker pour créer l'environnement de développement. Les activités entreprises comprenaient l'installation de la machine hôte, la création de l'espace de travail, la génération de l'environnement de développement, la mise en place de l'infrastructure et la gravure de la carte SD pour la cible. Des tests ont été effectués pour valider l'environnement développement, ainsi que le debugging d'une application simple depuis la machine hôte.

### 2.1 Travail à effectuer

1. **installez l'environnement de développement sur la machine hôte, selon les instructions ci-dessus, et configurez la cible en mode de développement avec CIFS/SMB.**

Pas de remarques.

2. **Installez/configurez SSH pour un accès à distance**

Pas de remarques.

3. **Créez un script permettant de générer la carte SD.**

Pas de remarques.

4. **Testez les différentes méthodes et techniques de débogage proposées par l'environnement Linux.**

Pas de remarques.

5. **Créez une partition ext4 avec l'espace restant sur la carte SD. Démarrez ensuite automatiquement (mode production) un petit programme que vous aurez préalablement placé dans /opt**

Pour simplifier la démarche, nous avons créé une partition de 1GiB à la fin de la carte SD.

En effet, nous savons que la mémoire à la fin de la carte SD n'est pas utilisée.

Pour trouver le secteur de début nous avons simplement utilisé le secteur de fin et la taille en bytes d'un secteur. Le calcul est présenté ci-dessous :

$$last\_sector - \frac{\#bytes}{bytes\_per\_sector} = 31176703 - \frac{1 * 1024^3}{512} = 29'079'551$$

L'extrait de commandes shell ci-dessous présente comment créer la partition :

```
1 # fdisk /dev/mmcblk2
2 Command (m for help): n
3 Partition type
4     p   primary partition (1-4)
5     e   extended
6
7 p
8 Partition number (1-4): 3
9 First sector (16-31176703, default 16): 29079551
10 Last sector or +size{K,M,G,T} (29079551-31176703, default 31176703):
11 Using default value 31176703
12
13 Command (m for help): w
14 The partition table has been altered.
15 Calling ioctl() to re-read partition table
16 # ls /dev/mmcblk2*
17 /dev/mmcblk2  /dev/mmcblk2p1  /dev/mmcblk2p2  /dev/mmcblk2p3
```

Pour formater la partition en ext4 et la monter sur `/opt`, il suffit d'utiliser la commande ci-dessous :

```
1 # mkfs.ext4 -L opt /dev/mmcblk2p3
2 # mount /dev/mmcblk2p3 /opt
3
```

Le petit programme placé dans `/opt` est alors le programme disponible dans le workspace `/workspace/src/01_environment/daemon/daemon.c`

```
1 # cd /workspace/src/01_environment/daemon/
2 # make
3 # scp app csel:/rootfs/opt/mini-program
4
```

Ensuite, pour que ce programme soit automatiquement lancé au démarrage, nous avons créé un script de démarrage. Ce dernier est montré ci-dessous :

```
S61mini-program
1  #!/bin/sh
2  #
3  # Daemon application
4  #
5  case "$1" in
6      start)
7          mount /dev/mmcblk2p3 /opt/
8          /opt/mini-program
9          ;;
10     stop)
11         killall mini-program
12         ;;
13     restart|reload)
14         killall mini-program
15         /opt/mini-program
16         ;;
17     *)
18         echo $"Usage: $0 {start|stop|restart}"
19         exit 1
20     esac
21
22     echo "Daemon application launched"
23
24     exit $?
25
```

Ce script a été placé dans `/etc/init.d/`.

Après redémarrage du nanopi avec la commande `reboot`, on peut voir sur l'extrait ci-dessous que le programme s'exécute bien :

```
nanopi
1  # ps -aux | grep "daemon"
2  daemon      250  0.0  0.0   2120    76 ?        S    00:08   0:00
   ↪ /opt/mini-program
3  root        257  0.0  0.0   3008   324 pts/0    S+   00:09   0:00 grep daemon
4
```

## 2.2 Réponse aux questions

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

U-Boot est généré automatiquement au moment de la compilation dans buildroot. Il suffit de lancer la compilation avec la commande `make` dans le répertoire `/buildroot`.

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

Il suffit d'utiliser la commande `make menuconfig` et de sélectionner le paquet que l'on veut.

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

Pour modifier la configuration du noyau Linux il suffit d'utiliser la commande `make linux-menuconfig` et de sélectionner/supprimer les options que l'on désire.

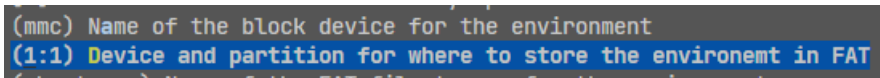
### 4. Comment faut-il faire pour générer son propre rootfs ?

Le dossier `/buildroot/board/friendlyarm/nanopi-neo-plus2/rootfs_overlay/` permet de placer les fichiers/dossiers que l'on veut introduire dans le rootfs final. Ce répertoire est alors "mergé" avec le dossier `/buildroot/output/target/`.

Une fois le dossier "rootfs\_overlay" modifié, il est nécessaire de recréer l'image. Pour cela, il suffit d'utiliser la commande `make` dans le dossier `/buildroot`

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

La première chose à faire est de modifier la configuration de uboot. L'image 1 montre cela :



(mmc) Name of the block device for the environment  
(1:1) Device and partition for where to store the environment in FAT  
(ubootenv) Name of the FAT file system for the environment

FIGURE 1 – Modification de uboot ("make uboot-menuconfig")

La seconde chose à faire est d'indiquer à uboot quel stockage utilisé. Ceci est décrit dans le fichier "boot.cmd". Ce dernier se trouve au chemin suivant : `/buildroot/board/friendlyarm/nanopi-neo-plus2/boot.cmd`. Il faut donc modifier le fichier "boot.cmd" en précisant que l'on ne veut plus démarrer le rootfs depuis `/dev/mmcblk2p2` mais depuis `/dev/mmcblk1p2`. Il faut aussi préciser la bonne mmc. (La bonne mmc peut être trouvée avec la commande `mmc list` dans uboot)

```
/buildroot/board/friendlyarm/nanopi-neo-plus2/boot.cmd
1 | setenv bootargs console=ttyS0,115200 earlyprintk root=/dev/mmcblk1p2 rootwait
2 |
3 | fatload mmc 1 $kernel_addr_r Image
4 | fatload mmc 1 $fdt_addr_r nanopi-neo-plus2.dtb
5 |
6 | booti $kernel_addr_r - $fdt_addr_r
7 |
```

Pour que cette modification soit prise en compte, il faut régénérer le fichier "boot.scr". L'extrait ci-dessous montre cela :

```
1 | # cd /buildroot
2 | # make host-uboot-tools-rebuild
3 |
```

Ensuite, on peut recompiler le tout. Cela permet de régénérer le fichier "sdcard.img". Pour faire cela il suffit d'utiliser la commande `make` dans `/buildroot`

Finalement, il suffit de copier la "sdcard.img" sur l'interface "mmcblk1". Ceci est montré dans le code ci-dessous : (Avant de réaliser cette étape, il ne faut pas oublier d'utiliser la commande "sync-images.sh" afin de recevoir la bonne version de la "sdcard.img")

on the nanopi

```
1 | # dd if=/workspace/buildroot-images/sdcard.img of=/dev/mmcblk1
2 |
```

#### 6. Qu'elle serait la configuration optimale pour le développement uniquement d'applications en espace utilisateur ?

La configuration de développement la plus pratique consiste à effectuer une cross-compilation sur la machine hôte, ensuite de monter l'espace de travail sur le périphérique cible et à utiliser l'éditeur VS Code avec un serveur gdb pour exécuter le code et le déboguer. Cet environnement est déjà proposé dans ce cours. Il nous permet déjà de développer des applications dans l'espace utilisateur de manière confortable mais cela uniquement pour le langage C et éventuellement C++. Si le but est réellement de développer uniquement en userspace, alors il ne serait pas nécessaire d'inclure les headers files du code source de linux (kernel). Dans le cas où l'on voudrait développer avec d'autre langage alors il serait nécessaire de modifier ce environnement en ajoutant les outils de compilation (sur le host) ou d'interprétation (sur le système embarqué).



## 2.3 Synthèse sur ce qui a été appris/exercé

— **Non acquis :**

1. Le passage de la carte SD vers la flash du système embarqué

— **Acquis, mais à exercer :**

1. Debugging avec VS Code

— **Parfaitement acquis :**

1. Modification du `rootfs_overlay` pour inclure des fichiers lors de la compilation avec `buildroot`
2. L'environnement de travail VSCode/Docker

## 2.4 Remarques et choses à retenir

Toutes les remarques où choses importantes à retenir se trouvent dans le chapitre 2.1

## 2.5 Feedback personnel sur le laboratoire

### Louka Yerly

L'environnement est très confortable pour programmer. Il est donc plaisant de faire les exercices. Pour la question concernant l'utilisation de la mmc au lieu de la carte SD nous ne sommes pas entièrement sûr de la réponse, car cela n'a pas été testé jusqu'au bout. Laboratoire très intéressant.

### Luca Srdjenovic

Dans cet environnement, il est facile de coder, de construire et d'exécuter quelque chose sur la cible, grâce aux devcontainers. De plus, la machine embarquée (Nanopi) est facile à flasher et à configurer. Merci au fait que j'ai déjà suivi le cours de MA\_SeS. Donc j'ai déjà un peu d'expérience avec la matière.

### 3 SP3 -Programmation Noyaux (module noyaux)

Le but de cette session est de se familiariser avec le développement de modules noyaux pour Linux. Cela va permettre d'utiliser différents outils afin d'insérer ces modules et également passer des paramètres à ces modules.

#### 3.1 Travail à effectuer

##### 1. Module noyau out of tree

La première chose que l'on demande de réaliser est de pouvoir afficher des messages. Il est possible d'utiliser la fonction `printk()` pour faire cela, cependant il est recommandé d'utiliser les "alias function". Ces fonctions sont présentées dans la figure 2. Du côté utilisateur, on peut voir les logs du kernel avec la commande `dmesg`.

```
1 # insmod mymodule.ko
2 # rmmod mymodule.ko
3 # dmesg
4 [ 784.820689] Linux module 01 skeleton loaded
5 [ 784.824946] text: dummy text
6 [ 784.824946] elements: 1
7 [ 790.353411] Linux module skeleton unloaded
8
```

La seconde chose demandée est d'utiliser la commande `modinfo`. Cette commande doit tout d'abord être installée avec `apt install kmod`. Ceci installe "modinfo" et "modprobe". La commande retourne alors les informations ci-dessous :

```
1 # modinfo mymodule.ko
2 filename:      /workspace/src/02_modules/exercice01/mymodule.ko
3 license:      GPL
4 description:   Module skeleton
5 author:       Daniel Gachet <daniel.gachet@hefr.ch>
6 depends:
7 name:         mymodule
8 vermagic:     5.15.21 SMP preempt mod_unload aarch64
9 parm:        text:charp
10 parm:        elements:int
11
```

On observe les choses suivantes :

- (a) On retrouve le nom de l'auteur du fichier. Comme nous ne l'avons pas modifié pour le moment nous avons laissé "Daniel Gachet".
- (b) On retrouve les paramètres qui peuvent être passés au module. Dans ce cas il s'agit d'une chaîne de caractère (charp) pour "text" et d'un entier (int) pour "elements".
- (c) On peut voir que ce module n'a actuellement pas de dépendance.

La troisième chose demandée est de comparer les informations entre `/proc/modules` et `lsmod`. L'extrait ci-dessous illustre cela :

```
1 # cat /proc/modules | grep "mymodule"
2 mymodule 16384 0 - Live 0xffff80000116d000 (0)
3 # lsmod | grep "odule"
4 Module                Size  Used by    Tainted: G
5 mymodule              16384  0
6 #
7
```

On peut voir que la commande permet de présenter les informations du module de façon plus jolie. Il manque cependant les informations concernant le status (ici "Live") ainsi que le kernel offset.

La dernière tâche demandée pour cette exercice est d'adapter le "Makefile" pour permettre l'insertion d'un module avec `modprobe`. Pour arriver à faire cela, il faut ajouter dans le "Makefile" le chemin du "rootfs".

Dans ce cas, le "Makefile" a été modifié ainsi :

```
/workspace/src/02_modules/exercice01/Makefile

1 ...
2 # Part executed when called from kernel build system:
3 ifneq ($(KERNELRELEASE),)
4 ...
5 # Part executed when called from standard make in module source directory:
6 else
7 ...
8 MODPATH := /rootfs
9 ...
10 install:
11     $(MAKE) -C $(KDIR) M=$(PWD) INSTALL_MOD_PATH=$(MODPATH) modules_install
12 endif
```

Une fois le fichier modifié, il est possible d'utiliser la commande `make install` dans le répertoire. On peut alors installer le module simplement avec `modprobe` :

```
1 # modprobe mymodule
2 # modprobe -r mymodule
3 # dmesg
4 [ 7625.778502] Linux module 01 skeleton loaded
5 [ 7625.782740]   text: dummy text
6 [ 7625.782740]   elements: 1
7 [ 7632.728783] Linux module skeleton unloaded
8
```

## 2. Réception de paramètres

Pour cet exercice, les paramètres utilisés sont "text" et "elements".

Comme présenté dans l'exercice précédant, il suffit de déclarer les paramètres que l'on veut utiliser de la manière suivante :

```
1 | ...
2 | module_param(text, charp, 0664);
3 | static int elements = 1;
4 | module_param(elements, int, 0);
5 | ...
```

Avec la commande `insmod`, on peut voir ci-dessous que le passage de paramètre fonctionne bien :

```
1 | # insmod mymodule.ko elements=150 'text="Test exercice 2"'
2 | # dmesg
3 | [ 8227.704385] Linux module 01 skeleton loaded
4 | [ 8227.708641]   text: Test exercice 2
5 | [ 8227.708641]   elements: 150
```

Afin de pouvoir utiliser `modprobe`, il est nécessaire de modifier le fichier `/etc/modprobe.conf` afin de spécifier ces paramètres.

`/rootfs/etc/modprobe.conf`

```
1 | options mymodule elements=150 text="Test exercice 2"
```

Une fois ce fichier complété, le module inséré avec `modprobe` prend les bons paramètres :

```
1 | # modprobe mymodule
2 | # dmesg
3 | [ 8620.704490] Linux module 01 skeleton loaded
4 | [ 8620.708745]   text: Test exercice 2
5 | [ 8620.708745]   elements: 150
```

### 3. Les 4 valeurs de /proc/sys/kernel/printk

Comme le montre l'exemple ci-dessous, lorsque l'on affiche le contenu du fichier `/proc/sys/kernel/printk`, quatre valeurs distinctes apparaissent :

```
1 # cat /proc/sys/kernel/printk
2 7      4      1      7
3
```

En prenant la documentation du kernel, ces valeurs font directement référence au log level. Ces différents log level sont décrits avec la figure 2

Name	String	Alias function
KERN_EMERG	"0"	<code>pr_emerg()</code>
KERN_ALERT	"1"	<code>pr_alert()</code>
KERN_CRIT	"2"	<code>pr_crit()</code>
KERN_ERR	"3"	<code>pr_err()</code>
KERN_WARNING	"4"	<code>pr_warn()</code>
KERN_NOTICE	"5"	<code>pr_notice()</code>
KERN_INFO	"6"	<code>pr_info()</code>
KERN_DEBUG	"7"	<code>pr_debug()</code> and <code>pr_devel()</code> if <code>DEBUG</code> is defined
KERN_DEFAULT	""	
KERN_CONT	"c"	<code>pr_cont()</code>

FIGURE 2 – Log Level du kernel [3]

Ces valeurs du fichier "printk" ont les significations suivantes :

- Le premier nombre définit le "current console\_loglevel". Cela permet de limiter les messages qui sont affichés à la console.
- Le second nombre définit le "default log level". Ce mode est utilisé lorsque dans l'appel de "printk" aucun mode n'est spécifié.
- Le troisième nombre définit le "minimum log level" (donc la valeur des messages les plus critiques).
- Le quatrième nombre définit "boot-time-default log level".

**Remarque :** Cela ne fonctionne qu'avec un port console, avec SSH cela n'est pas fonctionnel.

Si l'on désire modifier le "current console\_loglevel", il suffit d'utiliser la commande ci-dessous :

```
1 # echo "8" > /proc/sys/kernel/printk
2 # cat /proc/sys/kernel/printk
3 8      4      1      7
4 # modprobe mymodule
5 [11082.867633] Linux module 01 skeleton loaded
6 [11082.871903]   text: Test exercice 2
7 [11082.871903]   elements: 150
8
```

#### 4. Gestion de la mémoire, bibliothèque et fonction utile

Pour cet exercice, nous avons créé notre propre code. L'output de la console peut être visualisé ci-dessous :

```
1 # modprobe mymodule
2 # dmesg -c
3 [17787.156456] Linux module 01 skeleton loaded
4 [17787.160741]   ->Element 0, text: Test exercice 4
5 [17787.165384]   ->Element 1, text: Test exercice 4
6 [17787.170006]   ->Element 2, text: Test exercice 4
7 [17787.174639]   ->Element 3, text: Test exercice 4
8 [17787.179267]   ->Element 4, text: Test exercice 4
9 [17787.183894]   ->Element 5, text: Test exercice 4
10 [17787.188518]   ->Element 6, text: Test exercice 4
11 [17787.193144]   ->Element 7, text: Test exercice 4
12 [17787.197767]   ->Element 8, text: Test exercice 4
13 [17787.202395]   ->Element 9, text: Test exercice 4
14 # modprobe -r mymodule
15 # dmesg
16 [17809.092154]   ->Element 0 freed
17 [17809.092193]   ->Element 1 freed
18 [17809.095416]   ->Element 2 freed
19 [17809.098587]   ->Element 3 freed
20 [17809.101728]   ->Element 4 freed
21 [17809.104883]   ->Element 5 freed
22 [17809.108034]   ->Element 6 freed
23 [17809.111190]   ->Element 7 freed
24 [17809.114340]   ->Element 8 freed
25 [17809.117478]   ->Element 9 freed
26 [17809.120631] Linux module skeleton unloaded
27
```

Pour la création des éléments, le code utilisé est sensiblement le même que celui proposé par le cours. Cependant, pour la libération des ressources, nous avons utilisé la macro `list_for_each_safe()`. Cette macro semble même recommandée pour réaliser ce type d'opération. Le code ci-dessous illustre la libération de la mémoire :

```
1  static void free_list(void) {
2      struct element* ele;
3      struct list_head *p, *n;
4
5      // reference: https://stackoverflow.com/questions/63051548/linux-kernel-
   ↪ list-freeing-memory
6      list_for_each_safe(p, n, &my_list) {
7          ele = list_entry(p, struct element, list);
8          pr_info("  ->Element %i freed", ele->unique_id);
9          kfree(ele);
10     }
11 }
12
13 static void __exit skeleton_exit(void)
14 {
15     free_list();
16     pr_info ("Linux module skeleton unloaded\n");
17 }
```

### 3.2 Synthèse sur ce qui a été appris/exercé

— Non acquis :

-

— Acquis, mais à exercer :

1. L'utilisation de la macro `list_for_each_safe()` dans notre code personnel pour les listes chaînées.
2. L'utilisation des listes chaînées dans le kernel.

— Parfaitement acquis :

1. L'utilisation de `modprobe`.
2. Les différentes valeurs possibles de `printk()`.
3. L'utilisation de `insmod`, `rmmmod`, `lsmod`.

### 3.3 Remarques et choses à retenir

Nous avons consolidé nos connaissances fondamentales sur l'utilisation des modules noyaux, y compris leur chargement, leur déchargement et leur installation, avec la différence entre les commandes `insmod` et `modprobe` pour ajouter des modules au noyau. De plus, en explorant des concepts tels que `printk()` permettant la transmission d'informations du noyau vers l'espace utilisateur, nous avons acquis une meilleure compréhension du fonctionnement interne de la commande `dmesg`. Toutes les autres remarques ou choses importantes à retenir se trouvent dans le chapitre 3.1.

### 3.4 Feedback personnel sur le laboratoire

Les feedbacks ont été faits lors de la fin de ce thème. Ceux-ci se trouvent au chapitre *Programmation Noyau 4.4*



## 4 SP4 - Programmation Noyau (module noyaux)

Le but de cette session est de réaliser des modules noyaux plus complexes en utilisant des threads et des GPIOs. Cette session va également permettre de lire des données dans la RAM.

### 4.1 Travail à effectuer

#### 5. Module noyau permettant d'afficher le Chip-ID du processeur, la température du CPU et la MAC adresse du contrôleur Ethernet

Pour éviter les problèmes de concurrence, le kernel Linux met en place un mécanisme de réservation d'adresse mémoire. Ce mécanisme ne fait aucune sorte de mapping, c'est un mécanisme de réservation pur, qui repose sur le fait que tous les pilotes de périphériques du noyau doivent être sympathiques, et qu'ils doivent appeler `request_mem_region()`, vérifier la valeur de retour, et se comporter correctement en cas d'erreur.

Il est cependant obligatoire de mapper cette adresse mémoire. L'adresse souhaitée peut être mappée avec le MMU configuré en utilisant la fonction `ioremap()` pour créer un mappage de mémoire utilisable pour la zone souhaitée. Cela permet d'accéder au matériel via la mémoire de manière sécurisée et sans conflit.

Il est recommandé de demander et d'utiliser l'intégralité de la page mémoire de 4kiB pour une bonne pratique. Cependant, il est important de noter que le noyau est capable de gérer la demande et l'utilisation de l'adresse exacte, comme nous avons pu le voir en classe. Par conséquent, nous avons suivi cette méthode pour demander et utiliser l'adresse précise.

Le module chargé affiche toutes les informations au chargement :

```
1 # insmod mymodule.ko
2 [ 2801.737018] Linux module 05 skeleton loaded
3 [ 2801.741287] Error while reserving memory region... [0]=1, [1]=1, [2]=1
4 [ 2801.747899] chipid=82800001'94004704'5036c304'0c2c084e
5 [ 2801.753134] temperature=38634 (1548)
6 [ 2801.756772] mac-addr=02:01:c6:09:1b:ea
7 # cat /sys/class/thermal/thermal_zone0/temp
8 38276
9 # ifconfig -a | grep -ioE '([a-z0-9]{2:}){5}..'
10 02:01:C6:09:1B:EA
11
```

Nous avons constaté que l'appel à la fonction `request_mem_region()` a échoué pour les trois zones de mémoire que nous avons essayé de réserver. Cela indique que ces zones sont déjà utilisées par un autre module pour lire ou écrire des données. Il est possible que ces zones aient été réservées par des modules natifs qui fournissent déjà des informations telles que la température du CPU, l'adresse MAC et l'ID du chip. Dans le cadre de cet exercice, nous considérons que l'utilisation de ces zones de mémoire ne pose pas de problème car nous nous contentons de lire des données.

Nous avons pu observer que la température du CPU correspond à celle affichée par le module de capteur de température du Nanopi, et que l'adresse MAC est identique à celle affichée par la commande `ifconfig`.

## 6. Module permettant d'instancier un thread dans le noyau

Cet exercice fait usage des fonctions expliquées durant le cours concernant les threads kernel [7].

Afin de pouvoir démarrer un thread, il faut utiliser la macro `kthread_run()`. Cette macro permet de démarrer facilement un thread kernel. En effet, elle englobe la fonction `kthread_create()` et la fonction `wake_up_process()`.

Finalement, il faut gérer la fin d'exécution d'un thread kernel avec les fonctions `kthread_stop()` et `kthread_should_stop()`.

L'extrait ci-dessous illustre cela :

```
1 static struct task_struct* my_thread;
2 static int skeleton_thread (void* data) {
3     while(!kthread_should_stop()) {
4         // DO STUFF !
5     }
6     return 0;
7 }
8 static int __init skeleton_init(void) {
9     my_thread = kthread_run (skeleton_thread, 0, "s/thread");
10    return 0;
11 }
12 static void __exit skeleton_exit(void) {
13     kthread_stop (my_thread);
14 }
```

Un fois le module inséré, on peut voir que le thread réalise bien son travail :

```
1 # modprobe mymodule
2 # sleep sleep 20
3 # modprobe -r mymodule
4 # dmesg
5 [ 2304.348059] Linux module 06 skeleton loaded
6 [ 2304.352544] skeleton thread is now active...
7 [ 2309.598271] skeleton thread is kick every 5 seconds...
8 [ 2314.718285] skeleton thread is kick every 5 seconds...
9 [ 2319.838278] skeleton thread is kick every 5 seconds...
10 [ 2324.958278] skeleton thread is kick every 5 seconds...
11 [ 2324.963510] Linux module skeleton unloaded
```

## 7. Module permettant d'instancier deux threads avec mise en sommeil

Pour réaliser cet exercice, deux types de variables partagées sont utilisées :

- `atomic_t` : Ce type permet de partager un entier. Les opérations de lecture avec `atomic_read()` et d'écriture avec `atomic_inc()/dec()` et `atomic_set()` sont réalisées de manière atomique.
- `wait_queue_head` : Ce type est déclaré grâce à la macro `DECLARE_WAIT_QUEUE_HEAD()`. Un thread peut alors attendre dans cette queue, sur une certaine condition. Dans le cadre de cet exercice, l'attente est réalisée de la façon présentée dans l'extrait de code ci-dessous. La variable `queue_1` est de type `wait_queue_head` et la variable `is_kicked` est de type `atomic_t`.

```
1 | wait_event_interruptible(queue_1, (atomic_read(&is_kicked) != 0)) ||  
   | ↪ kthread_should_stop();
```

## 8. Module permettant de capturer les pressions exercées sur les switches de la carte d'extension par interruption

Ces boutons sont connectés à travers l'interface GPIO, et chacun possède un identifiant distinctif qui peut être utilisé pour l'identifier. Selon les schémas sur la figure 3 de la carte, leurs identifiants sont 0, 2 et 3.

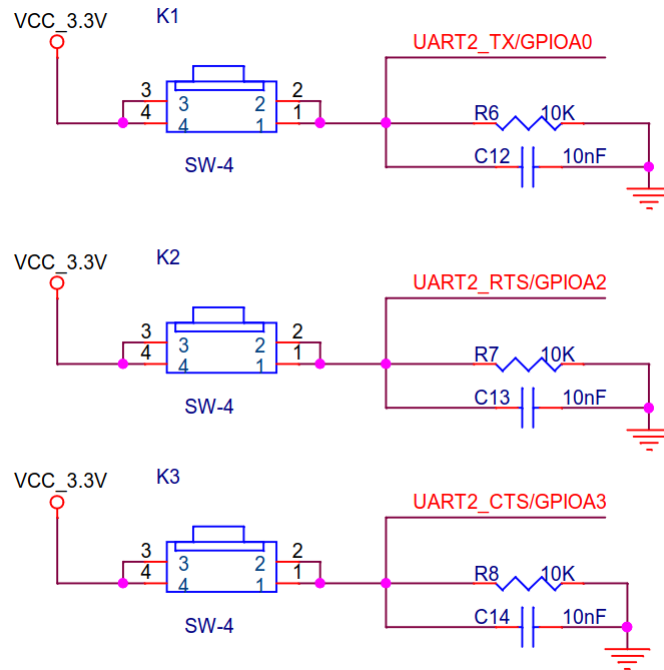


FIGURE 3 – Schéma des interfaces des switches [2]

Afin de pouvoir utiliser les GPIOs, la première chose à faire est d'utiliser la fonction `gpio_request()`. Cette fonction (selon LWN[1]) permet d'allouer et d'obtenir l'ownership d'une gpio. Ensuite, il suffit d'utiliser la fonction `request_irq()` pour pouvoir ajouter une fonction de callback lors d'une IRQ. L'utilisation de la fonction s'est faite ainsi :

```
1 request_irq(gpio_to_irq(K1), gpio_isr, IRQF_TRIGGER_FALLING | IRQF_SHARED,
   ↪ k3, k3);
```

Afin de désenregistrer les fonctions de callback et de libérer la GPIO, les fonctions pendantes sont `free_irq()` et `gpio_free()`.

## 4.2 Synthèse sur ce qui a été appris/exercé

— Non acquis :

-

— Acquis, mais à exercer :

1. La synchronisation des threads kernel.
2. L'utilisation des GPIO. Par exemple comment trouver les valeur numérique des GPIO.

— Parfaitement acquis :

1. La création de threads kernel.

## 4.3 Remarques et choses à retenir

Lors de la programmation dans l'espace noyau, plusieurs concepts clés doivent être pris en compte, comme : (1) la maîtrise de l'allocation et la libération dynamiques de la mémoire, (2) la considération de toutes les éventuelles défaillances, (3) l'évitement des fuites de mémoire dans le noyau, et (4) la cartographie de la mémoire matérielle et des registres. Pour trouver les adresses et les GPIOs nécessaires, il est recommandé de se référer aux fiches techniques et aux schémas correspondants.

Nous avons appris (1) à gérer la concurrence et le code asynchrone en dehors de l'init, en utilisant notamment (2) la gestion des threads du noyau, (3) la communication inter-threads, (4) les opérations de mémoire atomique, (5) les temporisateurs et le sommeil, (6) l'attente d'un certain temps, (7) l'attente d'un événement, (8) les interruptions, (9) l'enregistrement des gestionnaires, ainsi que (10) l'externalisation du travail des gestionnaires pour minimiser leur taille et aussi éviter un blocage au niveau du système d'exploitation.

Toutes les autres remarques où choses importantes à retenir se trouvent dans le chapitre 4.1.

## 4.4 Feedback personnel sur le laboratoire

### Louka Yerly

J'avais déjà programmer au niveau du kernel plusieurs fois dans différent cours, mais c'est la première fois que la notion de thread kernel a été abordée. C'est également la première fois que j'utilise le passage de paramètres à un module.

Le TP est très intéressant.

### Luca Srdjenovic

Au début, je me suis senti un peu intimidé lorsque j'ai commencé à programmer dans l'espace noyau car étant ingénieur software, tout ce qui touche à l'embarqué est encore obscure pour moi. Toutefois, cela a été une surprise pour moi de constater que les bibliothèques proposées pour l'allocation dynamique de la mémoire, la gestion des threads et autres, étaient très similaires à celles que j'utilisais dans l'espace utilisateur. Les exercices proposés étaient bien structurés et m'ont aidé à acquérir de nouvelles compétences en programmation. J'ai particulièrement apprécié l'apprentissage à travers les méthodologies de gestion des ressources (e.g. synchronisation des threads). Dans l'ensemble, cette expérience m'a permis de mieux comprendre le fonctionnement de l'espace noyau et de renforcer mes compétences en langage C.

## 5 SP5 - Programmation Noyau (Pilotes de périphériques)

Cette session a pour but se familiariser avec les pilotes de périphériques. Un premier exercice va faire exception à cela en montrant qu'il est possible d'accéder à la mémoire depuis l'espace utilisateur. Les exercices suivants permettront quant à eux de mieux comprendre les pilotes de périphériques et le sysfs.

### 5.1 Travail à effectuer

#### 1. Pilote orienté mémoire permettant de lire l'identification du $\mu P$

Le but de cet exercice est de réaliser une application dans le **userspace** qui permet de lire le chip id du Nanopi.

Pour faire cela, il est nécessaire d'utiliser le fichier `/dev/mem`. Selon la man page [4], `/dev/mem` est un fichier de périphérique de caractères qui est une image de la mémoire principale de l'ordinateur. Il peut être utilisé, par exemple, pour examiner (et même corriger) le système. Les adresses sont interprétées comme des adresses de mémoire physique. Les références à des emplacements inexistants entraînent le retour d'erreurs.

Pour mapper les adresses physiques aux adresses virtuelles, la fonction `mmap` doit être utilisée. Dans le cadre de ce laboratoire cela est fait de la façon suivante :

```
1  size_t psz      = getpagesize();
2  off_t dev_addr = 0x01c14200;
3  off_t ofs      = dev_addr % psz;
4  off_t offset   = dev_addr - ofs;
5  volatile uint32_t* regs = mmap(0, psz, PROT_READ | PROT_WRITE, MAP_SHARED,
   ↪  fd, offset);
6
```

la man page de la fonction `mmap` indique qu'il faut que l'offset soit un multiple de `pagesize()`. C'est pour cela qu'il est nécessaire d'utiliser le modulo.

La valeur de "offset" vaut donc :

$$offset = 0x01c14200 - (0x01c14200 \% 0x1000) = 0x1C14000$$

Le code minimal pour lire une adresse est présenté ci-dessous :

```
1  int main()
2  {
3      int fd = open("/dev/mem", O_RDWR);
4      if (fd < 0) {
5          return -1;
6      }
7
8      size_t psz      = getpagesize();
9      off_t dev_addr = 0x01c14200;
10     off_t ofs      = dev_addr % psz;
11     off_t offset   = dev_addr - ofs;
12
```

```
13     volatile uint32_t* regs = mmap(0, psz, PROT_READ | PROT_WRITE,  
    ↪ MAP_SHARED, fd, offset);  
14  
15     if (regs == MAP_FAILED) {  
16         printf("mmap failed, error: %i:%s \n", errno, strerror(errno));  
17         return -1;  
18     }  
19  
20     // USE THE REGS !  
21  
22     munmap((void*)regs, psz);  
23     close(fd);  
24  
25     return 0;  
26 }
```

## 2. Implémenter un pilote de périphérique orienté caractère

Afin que le userspace ait un accès de communication avec le module situé dans le kernel, la notion de MAJOR et de MINOR est utilisée.

- **MINOR\_NUMBER** : identifie le pilote associé au dispositif. Un numéro majeur peut également être partagé par plusieurs pilotes de périphériques.
- **MAJOR\_NUMBER** : identifie le pilote correspondant. Plusieurs appareils peuvent utiliser le même MAJOR\_NUMBER, il faut donc attribuer un numéro à chaque appareil qui utilise le même MAJOR\_NUMBER. En d'autres termes, le pilote de périphérique utilise le MINOR\_NUMBER pour distinguer les périphériques.

Dans cet exercice, le MAJOR\_NUMBER est créé lors de l'insertion du module, il n'est donc pas connu à l'avance. Ceci a pour but de rendre le périphérique plus dynamique. Cette attribution de major se fait par l'utilisation de la fonction `alloc_chrdev_region()` qui permet de spécifier également le nombre de MINOR\_NUMBER voulu.

Pour trouver le numéro de major number (créé dynamiquement), il est possible d'utiliser la commande présentée ci-dessous. En effet le fichier `/proc/devices` permet de lister les pilotes de périphériques configurés dans le noyau en cours d'exécution (bloc et caractère).

```
1 # cat /proc/devices | grep "mymodule"
2 511 mymodule
```

On peut alors créer le fichier d'accès avec la commande ci-dessous. On peut voir que le fichier est bien créé avec les bons numéros de majeur et de mineur :

```
1 # mknod /dev/mymodule c 511 0
2 # ls -al /dev/mymodule
3 crw-r--r-- 1 root root 511, 0 Jan  1 00:58 /dev/mymodule
```

Finalement le comportement demandé (d'écriture et de lecture) est fonctionnel :

```
1 # echo "test" > /dev/mymodule
2 # cat /dev/mymodule
3 test
```

Un point important à noter est que l'échange de données entre l'application en espace utilisateur et le pilote de périphérique en espace noyau n'est généralement pas autorisé avec un accès direct basé sur la déréréférenciation du pointeur buf. Il est donc nécessaire d'utiliser les méthodes `copy_to_user()` et `copy_from_user()`.



Le code ci-après présente le code minimal pour la création du pilote de périphérique orienté caractère :

```
1 static dev_t skeleton_dev;
2 static struct cdev skeleton_cdev;
3 static int skeleton_open(struct inode* i, struct file* f) {
4     // DO STUFF
5     return 0;
6 }
7 static int skeleton_release(struct inode* i, struct file* f) {
8
9     return 0;
10 }
11 static ssize_t skeleton_read(struct file* f, char __user* buf, size_t count,
12     ↪ loff_t* off) {
13     // DO STUFF
14     return count;
15 }
16 static ssize_t skeleton_write(struct file* f, const char __user* buf, size_t
17     ↪ count, loff_t* off) {
18     //DO STUFF
19     return count;
20 }
21 static struct file_operations skeleton_fops = {
22     .owner    = THIS_MODULE,
23     .open     = skeleton_open,
24     .read     = skeleton_read,
25     .write    = skeleton_write,
26     .release  = skeleton_release,
27 };
28 static int __init skeleton_init(void) {
29     int status = alloc_chrdev_region(&skeleton_dev, 0, 1, "mymodule");
30     if (status == 0) {
31         cdev_init(&skeleton_cdev, &skeleton_fops);
32         skeleton_cdev.owner = THIS_MODULE;
33         status              = cdev_add(&skeleton_cdev, skeleton_dev, 1);
34     }
35     return 0;
36 }
37 static void __exit skeleton_exit(void){
38     cdev_del(&skeleton_cdev);
39     unregister_chrdev_region(skeleton_dev, 1);
40 }
41 module_init(skeleton_init);
42 module_exit(skeleton_exit)
```

### 3. Étendre la fonctionnalité du pilote de l'exercice précédent afin que l'on puisse à l'aide d'un paramètre module spécifier le nombre d'instances

Comme on peut le voir dans l'extrait de code ci-dessous, le nombre d'instances est spécifié avec le paramètre `instances`. Dans la fonction d'initialisation, il est nécessaire d'allouer les ressources pour la variable `buffers`.

```
1 static int instances = 3;
2 module_param(instances, int, 0);
3 static char** buffers = 0;
4 static int __init skeleton_init(void) {
5     ...
6     buffers = kzalloc(sizeof(char*) * instances, GFP_KERNEL);
7     for (i = 0; i < instances; i++)
8         buffers[i] = kzalloc(BUFFER_SZ, GFP_KERNEL);
9 }
10
11
```

Lorsqu'une application ouvre le fichier d'accès, la méthode `skeleton_open()` assigne à la variable `private_data` le buffer correspondant au numéro de mineur :

```
1 static int skeleton_open(struct inode* i, struct file* f) {
2     ...
3     f->private_data = buffers[iminor(i)];
4 }
5
```

Afin de pouvoir utiliser ce module, il est nécessaire de créer les fichiers d'accès. Ceci peut être fait avec les commandes ci-dessous :

```
1 # mknod /dev/mymodule0 c 511 0
2 # mknod /dev/mymodule1 c 511 1
3 # mknod /dev/mymodule2 c 511 2
4 # ls -al /dev/mymodule*
5 crw-r--r-- 1 root root 511, 0 Jan  1 00:58 /dev/mymodul
6
```

Finalement, on peut voir le bon fonctionnement :

```
1 # echo "test0" > /dev/mymodule0
2 # echo "test1" > /dev/mymodule1
3 # echo "test2" > /dev/mymodule2
4 # cat /dev/mymodule0
5 test0
6 # cat /dev/mymodule1
7 test1
8 # cat /dev/mymodule2
9 test2
10
```

#### 4. Application en espace utilisateur permettant d'accéder à ces pilotes

L'accès à un pilote se fait de la même façon qu'à un fichier traditionnel. On utilise donc les méthode `open()`, `read()`, `write()` et `close()`.

Pour l'exercice 4, il suffit d'ouvrir les fichiers d'accès préalablement créés avec `mknod`. Il est alors nécessaire de préciser à l'exécutable quel fichier d'accès ouvrir en donnant en paramètre de l'application le nom du fichier.

```
1 # ./app /dev/mymodule0
2 /dev/mymodule0
3 bonjour le monde
4 ce mois d'octobre est plutot humide...
5 ce n'est qu'un petit texte de test...
6
7 et voici un complement au premier text..
8 ce n'est qu'un deuxieme petit texte de test...
9 blabla blabla blabla blabla blabla blabla blabla
10 ...
11
```

Pour que l'affichage se fasse correctement, il ne faut pas oublier de réserver un caractère pour le `'\0'` : `ssize_t sz = read(fdr, buff, sizeof(buff) - 1);`.

## 5. Développement d'un pilote de périphérique orienté caractère permettant de valider la fonctionnalité du sysfs

Afin de créer une classe dans le sysfs, il est nécessaire d'utiliser la fonction `class_create()`. Cette fonction a pour effet de créer un nouveau répertoire dans `/sys/class`. On peut alors créer à l'intérieur de cette classe un device avec la méthode `device_create()`. Finalement, les attributs sont créés avec la méthode `device_create_file()`. Cela a pour effet de créer des fichiers. De cette manière on respecte la figure 4 présentée en cours.

Interne au noyau	Espace utilisateur
Objets du noyau	Répertoires
Attributs des objets	Fichiers
Relations entre objets	Liens symboliques

FIGURE 4 – Principe du sysfs[6]

Le code ci-dessous présente l'utilisation des méthode décrite dans le paragraphe ci-dessus.

```

1 static struct class* sysfs_class;
2 static struct device* sysfs_device;
3
4 static int __init skeleton_init(void) {
5     sysfs_class = class_create(THIS_MODULE, "my_sysfs_class");
6     sysfs_device = device_create(sysfs_class, NULL, 0, NULL, "my_sysfs_device");
7     if (status == 0) status = device_create_file(sysfs_device, &dev_attr_val);
8     if (status == 0) status = device_create_file(sysfs_device, &dev_attr_cfg);
9 }

```

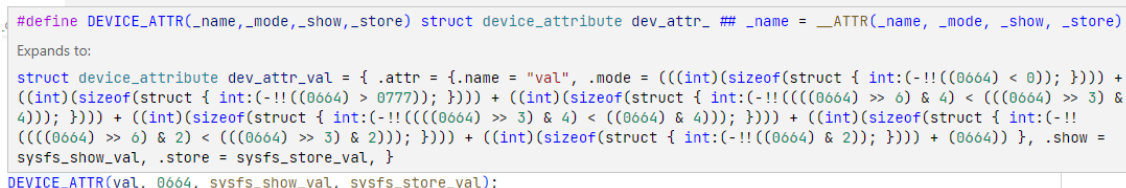
Les structure `dev_attr_val` et `dev_attr_cfg` sont créés grâce à l'utilisation de la macro `DEVICE_ATTR()`. Ceci est montré ci-dessous :

```

1 DEVICE_ATTR(val, 0664, sysfs_show_val, sysfs_store_val);
2 DEVICE_ATTR(cfg, 0664, sysfs_show_cfg, sysfs_store_cfg);

```

L'éditeur VSCode permet de montrer que la macro génère bien les structures `dev_attr_val` et `dev_attr_cfg`. La figure 5 présente cela.



```
#define DEVICE_ATTR(_name, _mode, _show, _store) struct device_attribute dev_attr_ ## _name = __ATTR(_name, _mode, _show, _store)

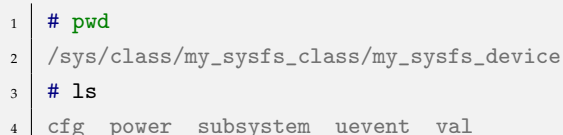
Expands to:

struct device_attribute dev_attr_val = { .attr = { .name = "val", .mode = (((int)(sizeof(struct { int:(-!!((0664) < 0)); }))) +
((int)(sizeof(struct { int:(-!!((0664) > 0777)); }))) + ((int)(sizeof(struct { int:(-!!(((0664) >> 6) & 4) < (((0664) >> 3) &
4)); }))) + ((int)(sizeof(struct { int:(-!!(((0664) >> 3) & 4) < ((0664) & 4)); }))) + ((int)(sizeof(struct { int:(-!!
(((0664) >> 6) & 2) < (((0664) >> 3) & 2)); }))) + ((int)(sizeof(struct { int:(-!!((0664) & 2)); }))) + (0664)) }, .show =
sysfs_show_val, .store = sysfs_store_val, }

DEVICE_ATTR(val, 0664, sysfs_show_val, sysfs_store_val);
```

FIGURE 5 – Expansion de la macro `DEVICE_ATTR()`

Une fois le module installé, on retrouve l'arborescence souhaitée. L'extrait de session shell ci-dessous montre cela :

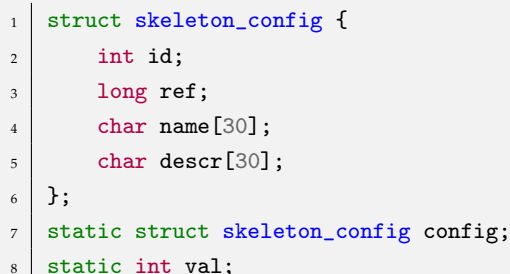


```
1 # pwd
2 /sys/class/my_sysfs_class/my_sysfs_device
3 # ls
4 cfg power subsystem uevent val
```

On peut voir qu'en plus des fichiers `cfg` et `val`, trois autres fichiers sont présents. Selon [kernel.org\[5\]](http://kernel.org[5]), ces dossiers ont les fonctions suivantes :

- **power** : contient des attributs permettant à l'utilisateur de vérifier et de modifier certaines propriétés liées à la gestion de l'énergie.
- **subsystem** : lien symbolique qui pointe vers le sous-système propriétaire. Dans ce cas il s'agit de "my\_sysfs\_class".
- **uevent** : fichier permettant de "ADD" ou de "REMOVE" des événements qui peuvent être traités par "udev".

Finalement, les accès aux fichiers `val` et `cfg` permettent de lire et de définir les valeurs contenues dans les variables déclarées dans le code. Ces valeurs sont les suivantes :



```
1 struct skeleton_config {
2     int id;
3     long ref;
4     char name[30];
5     char descr[30];
6 };
7 static struct skeleton_config config;
8 static int val;
```

On peut alors simplement passer des valeurs. L'exemple ci-dessous permet de donner des valeurs pour la structure `dev_attr_cfg`.

```
1 | # echo "20 30 my_name my_descr" > cfg
2 | # cat cfg
3 | 20 30 my_name my_descr
```

## 5.2 Synthèse sur ce qui a été appris/exercé

- Non acquis :

- 

- Acquis, mais à exercer :

1. création de pilotes orientés mémoire
2. création de pilotes orientés caractère

- Parfaitement acquis :

1. création de classes dans le sysfs
2. création de devices dans une classes
3. création d'attributs dans un device

## 5.3 Remarques et choses à retenir

Ce travail pratique a tout d'abord permis de comprendre comment fonctionne le fichier `/dev/mem` afin d'accéder à la mémoire depuis l'espace utilisateur avec l'utilisation de `mmap()`. Puis, les autres exercices se sont concentrés sur la création de pilotes de périphériques. Les notions de MINOR et MAJOR ont été traitées.

Finalement une introduction au sysfs a été faite. Ce TP a permis de créer une nouvelle classe avec la méthode `class_create()`. Dans cette classe un nouveau device a été créé avec la méthode `device_create()`. Pour terminé, les attributs ont été exposés avec la méthode `device_create_file()`.

Toutes les autres remarques où choses importantes à retenir se trouvent dans le chapitre 5.1

## 5.4 Feedback personnel sur le laboratoire

Les feedbacks ont été faits lors de la fin de ce thème. Ceux-ci se trouve au chapitre *Programmation Noyau* 6.4.

## 6 SP6 - Programmation Noyau (Pilotes de périphériques)

Dans cette séance, nous allons implémenter un pilote de périphérique orienté caractère pour un *miscdevice* qui permettra de vérifier la fonctionnalité de sysfs. Le pilote devra proposer plusieurs attributs accessibles en lecture et en écriture. Nous allons également développer un module noyau et une application utilisateur qui exploitent les interruptions provenant des entrées/sorties bloquantes des boutons K1, K2 et K3 de la board. La fonction `select` permettra de surveiller le signal et d'afficher un message à chaque fois qu'un bouton est pressé. Ainsi, un message sera affiché à chaque pression de bouton détectée.

### 6.1 Travail à effectuer

#### 5.1. Ajoutez maintenant les opérations sur les fichiers définies à l'exercice #3

Pour cet exercice, nous avons choisi de créer un *miscdevice* à la place de réutiliser une *class* comme à l'exercice précédent. En premier lieu, il faut déclarer le misc device ainsi que les paramètres (ici le buffer) qui seront utilisés dans les fonctions `.read` et `.write`.

```
1  #define BUFFER_SZ 10000
2  static char s_buffer[BUFFER_SZ];
3
4  static struct file_operations skeleton_fops = {
5      .owner    = THIS_MODULE,
6      .open     = skeleton_open,
7      .read     = skeleton_read,
8      .write    = skeleton_write,
9      .release  = skeleton_release,
10 };
11 static struct miscdevice misc_device = {
12     .minor = MISC_DYNAMIC_MINOR,
13     .fops  = &skeleton_fops,
14     .name  = "my_misc_module",
15     .mode  = 0,
16 };
17
18 static int __init skeleton_init(void) {
19     int status = 0;
20     if (status == 0) status = misc_register(&misc_device);
21     if (status == 0) status =
22     ↪ device_create_file(misc_device.this_device, &dev_attr_val);
23     if (status == 0) status =
24     ↪ device_create_file(misc_device.this_device, &dev_attr_cfg);
25
26     return 0;
27 }
28
29 static void __exit skeleton_exit(void) {
30     misc_deregister(&misc_device);
31 }
```



```
29 | }  
30 |
```

Ensuite, comme on peut le voir sur l'extrait de code ci-dessus, les prochaines étapes sont d'initialiser et d'enregistrer notre misc device avec les deux opérations qui nous sont fournies `misc_register(&misc_device)` et `device_create_file(misc_device.this_device, &dev_attr_val)` qui va nous permettre de créer les attributs pour notre device.

Enfin, les accès aux fichiers `val` et `cfg` permettent de lire et de définir les valeurs contenues dans les variables déclarées dans le code.

```
1  # insmod mymodule.ko  
2  # dmesg | tail -n 1  
3  [ 3265.742497] Linux module skeleton loaded  
4  # cd /sys/class/misc/my_misc_module/  
5  # ls  
6  cfg dev power subsystem uevent val  
7  # ls -l /dev/my_misc_module  
8  crw-rw---- 1 root root 10, 125 Jan  1 00:54 /dev/my_misc_module  
9  # echo "20 30 my_name my_descr" > cfg  
10 # cat cfg  
11 20 30 my_name my_descr  
12
```

## 7. Développer un pilote et une application utilisant les entrées/sorties bloquantes

Pour gérer les interruptions, la première étape consiste à créer un gestionnaire d'IRQ et à l'enregistrer pour les 3 boutons. Le gestionnaire d'IRQ est conçu à être simple afin de pouvoir être traité en un minimum de cycles. Ce dernier modifie un compteur partagé de manière atomique et réveille les processus en userspace en attente dans une file d'attente. Ce comportement est montré par l'extrait de code ci-dessous :

```
1  irqreturn_t gpio_isr(int irq, void* handle) {
2      atomic_inc(&nb_of_interrupts);
3      wake_up_interruptible(&queue);
4
5      pr_info("interrupt %s raised...\n", (char*)handle);
6
7      return IRQ_HANDLED;
8  }
9
10 static unsigned int skeleton_poll(struct file* f, poll_table* wait) {
11     unsigned mask = 0;
12     poll_wait(f, &queue, wait);
13     if (atomic_read(&nb_of_interrupts) != 0) {
14         mask |= POLLIN | POLLRDNORM; /* read operation */
15         /* mask |= POLLOUT | POLLWRNORM; write operation */
16         atomic_dec(&nb_of_interrupts);
17         pr_info("polling thread waked-up...\n");
18     }
19     return mask;
20 }
```

La seconde étape à réaliser est de créer le *miscdevice*. Cela est fait en utilisant la fonction `misc_register()`. On peut voir cela sur l'extrait de code ci-dessous :

```
1  struct miscdevice misc_device = {
2      .minor = MISC_DYNAMIC_MINOR,
3      .fops = &skeleton_fops,
4      .name = "mymodule",
5      .mode = 0777,
6  };
7  static int __init skeleton_init(void) {
8      int status = 0;
9
10     atomic_set(&nb_of_interrupts, 0);
11
12     status = misc_register(&misc_device);
13
14     // Register GPIO
```

```
15 |     ...
16 | }
17 |
18 | static void __exit skeleton_exit(void) {
19 |     misc_deregister(&misc_device);
20 |     pr_info("Linux module skeleton unloaded\n");
21 | }
```

Nous observons dans les dossiers `/sys` et `/dev` que notre module a été créé avec succès :

```
1 | # insmod mymodule.ko
2 | # dmesg | tail -n 1
3 | [ 436.490419] Linux module skeleton loaded(status=0)
4 | # ls /sys/class/misc/mymodule/
5 | dev power subsystem uevent
6 | # ls -l /dev/mymodule
7 | crw-rw---- 1 root root 10, 125 Jan  1 00:07 /dev/mymodule
8 |
```

Afin de pouvoir communiquer avec le module, il est nécessaire de créer une application. Cette dernière doit utiliser des fonctions de multiplexage d'I/O tels que `select`, `poll` ou encore `epoll`.

Pour cet exercice, `select` a été utilisé. On peut voir l'implémentation faite ci-dessous :

```
/workspace/src/03_drivers/exercice07/main.c

1 | #include <sys/select.h>
2 | int main(int argc, char* argv[])
3 | {
4 |     if (argc <= 1) return 0;
5 |
6 |     /* open memory file descriptor */
7 |     int fdw = open(argv[1], O_RDWR);
8 |     fd_set read_fds;
9 |     FD_ZERO(&read_fds);
10 |
11 |     int counter = 0;
12 |     int status = 0;
13 |     int nbErrors = 0;
14 |     printf("Waiting for button press... \n");
15 |     while (1) {
16 |         FD_SET(fdw, &read_fds);
17 |         status = select(fdw + 1, &read_fds, NULL, NULL, NULL);
18 |         if (status == -1) {
19 |             nbErrors++;
```

```
20         printf("Error: %d times \n", nbErrors);
21     } else if (FD_ISSET(fdw, &read_fds)) {
22         printf("Button pressed %d times \n", ++counter);
23     } else {
24         printf("No data.\n");
25     }
26 }
27 return 0;
28 }
29
```

Finalement l'extrait de console ci-après démontre le bon fonctionnement. On peut voir qu'à chaque fois que l'on appuie sur un bouton, le compteur du nombre de pression s'incrémente et est affiché.

```
1  # ./app /dev/mymodule
2  Waiting for button press...
3  Button pressed 1 times
4  Button pressed 2 times
5  Button pressed 3 times
6  ...
7  # rmmod mymodule
8  # dmesg | tail -n1
9  [ 3220.222312] Linux module skeleton unloaded
10
```

## 6.2 Synthèse sur ce qui a été appris/exercé

— Non acquis :

-

— Acquis, mais à exercer :

1. Mieux comprendre comment sysfs interagit avec le reste du noyau

— Parfaitement acquis :

1. Distinction entre les deux types de drivers orientés mémoire, caractère
2. Implémentation d'un driver
3. Création d'un programme pour interagir avec le driver

## 6.3 Remarques et choses à retenir

Cette session a permis de comprendre comment implémenter des *miscdevice*. La structure importante est `struct miscdevice`. Cette structure simplifie l'instanciation du périphérique avec la création d'un fichier d'accès sous `/dev`. Finalement les fonction `misc_register()` et `misc_unregister()` permettent d'enregistrer et de retirer un *miscdevice*.

La seconde chose que a été vue, est l'implémentation d'un driver qui permet l'utilisation de multiplexeur d'IO bloquant tels que `select`, `poll` ou encore `epoll`. Ceci se fait simplement en utilisant la fonction de callback `.poll` de la structure `struct file_operations`.

Toutes les autres remarques où choses importantes à retenir se trouvent dans le chapitre 6.1

## 6.4 Feedback personnel sur le laboratoire

**Louka Yerly** Ce laboratoire a été très intéressant. La notion de *miscdevice* ne m'était pas connue. Je trouve cette solution très bien pour de petits drivers qui doivent communiquer avec le userspace.

Finalement, l'implémentation d'un device driver qui permet l'utilisation de multiplexeur d'IO a été ma partie favorite. En effet, j'aime tout particulièrement utiliser `epoll` lors de mes projets.

**Luca Srdjenovic**

Au cours de ce laboratoire, j'ai appris des concepts avancés dans la création de pilotes, tels que les modules du noyau, les périphériques de caractères et les périphériques de blocs, que je dois encore explorer pour acquérir une compréhension plus complète de ce qui se cache sous le userland.

## 7 Conclusion

Ce premier rapport a permis de grouper les six premières séances de travaux pratiques.

Lors de la première et de la seconde session, l'environnement de développement a été mis en place. Puis, les travaux pratiques ont débutés.

Durant la troisième session, les bases de l'installation d'un module ont été vues (`insmod`, `modprobe`). Le passage de paramètres à un module et l'utilisation de listes chaînées a également été mis en place.

La quatrième session a permis d'aller plus en profondeur dans la création de modules noyau avec notamment l'utilisation de threads et des GPIOs. La cinquième séance a permis de montrer comment accéder à la RAM depuis le userspace. Puis d'autres exercices ont montrés comment utiliser le sysfs. Finalement, la sixième séance a permis d'utiliser un *miscdevice* et d'intégrer à un module la possibilité de traiter les multiplexeurs d'IO bloquant comme `select`, `poll` ou encore `epoll`.

Globalement les TPs étaient très intéressants et nous avons pris du plaisir à les réaliser.

## Références

- [1] Jonathan Corbet. Gpio in the kernel : an introduction. <https://lwn.net/Articles/532714/>, 2023.
- [2] FRIENDLY ELEC. Schematic\_nanohat\_oled\_v1.pdf. [https://mse-csel.github.io/website/documentation/assets/Schematic\\_NanoHat\\_OLED\\_v1.4\\_1804.pdf](https://mse-csel.github.io/website/documentation/assets/Schematic_NanoHat_OLED_v1.4_1804.pdf), 2023.
- [3] The kernel development community. Message logging with printk. <https://www.kernel.org/doc/html/next/core-api/printk-basics.html>, 2023.
- [4] man7.org. mem(4) — linux manual page. <https://man7.org/linux/man-pages/man4/mem.4.html>, 2023.
- [5] Rafael J. Wysocki <rjw@rjwysocki.net>. /sys/devices/.../power/. <https://www.kernel.org/doc/Documentation/ABI/testing/sysfs-devices-power>, 2023.
- [6] Haute école d'ingénierie et d'architecture of Fribourg. sysfs : system file system. <https://mse-csel.github.io/website/lecture/programmation-noyau/pilotes/sysfs/>, 2023.
- [7] Haute école d'ingénierie et d'architecture of Fribourg. Threads dans le noyau. <https://mse-csel.github.io/website/lecture/programmation-noyau/modules/threads/#creation-de-threads-dans-le-noyau>, 2023.