# CSEL TP n°1

Repo git :
https://github.com/gaetan-passeri/csel-workspace

Gaetan Passeri, Glenn Muller

08 Avril 2022

# 1. Environnement Linux embarqué (du 25.02 au 04.03)

## Résumé

### Installation

- Installer docker
- Installer git => configurer git pour utiliser wsl2 au lieu de poweshell => depuis poweshell :

```
git config --global core.autocrlf false
```

- Installer vscode
- fork du projet du cours https://github.com/supcik/csel-workspace et ajout des collaborateurs
- ouvrir vscode, le projet contient un environnement de developpement docker => accepter de relancer le projet dans un container docker. => le projet se relance, le terminal de vscode est un linux.

### Buildroot

- depuis le terminal de vscode, lancer la commande suivante pour télécharger buildroot :

```
./get-buildroot.sh
```

- une fois télécharger, lancer la commande suivante pour configurer buildroot si nécessaire :

```
cd buildroot
make menuconfig
```

- et compiler buildroot avec :

```
make
```

  **NB** : la compilation dure environ une heure.

- actualiser le **root filesystem**

```
rm -Rf /rootfs/*
tar xf /buildroot/output/images/rootfs.tar -C /rootfs
```

### Gravure de la carte SD

- copier les images dans le répertoire synchronisé avec la machine de developpement

```
1 rsync -rlt --progress --delete /buildroot/output/images/ /
    workspace/buildroot-images
```

- copier les images de buildroot qui se trouvent dans le répertoire */buildroot/output/images/* dans le répertoire partagé avec l'ordinateur *workspace*

```
1 rsync -rlt --progress --delete /buildroot/output/images/ /
    workspace/buildroot-images
```

**NB** : la commande **rsync** propose plus d'option que **cp**.

- Pour flasher la carte SD, on utilise le logiciel **Balena Etcher** https://www.balena.io/etcher/
  - Insérer la carte SD dans la machine de développement
  - sélectionner l'image `buildroot-images/sdcard.img`
  - sélectionner le disque qui correspond à la carte SD et cliquer sur "Flash!". (carte SD de 16GB en l'occurence)

#### Communication avec la cible

- Installation du logiciel **Putty** sur Windows (il existe de nombreux autres programmes pour Linux) `putty-64bit-0.76-installer.msi`
- Installer la carte SD dans la cible
- Connecter la cible sur la machine de développement via port série-USB
- lancer Putty depuis windows =>
  - **Type de connection** : Serial
  - **Port** : COM6
  - **Vitesse de communication** : 115200 baud

    => `open`
- La cible démare l'image linux qui demande le **login : root**

#### Mise en place de l'infrastructure réseau

- On travaille avec des adresses IP fixes du côté de la cible comme du côté de la machine de développement.
  - **Cible : 192.168.0.14**

  - **Dev : 192.168.0.4**

- L'adresse IP de la cible est déjà configurée (vérification : ifconfig => ok)
- Configurer l'adresse IP fixe **de l'adaptateur Ethernet** sur la machine de développement (windows)
  - Settings => Network & Internet => onglet Status
  - sélectionner l'interface Ethernet (Ethernet 2 dans notre cas) => Properties => IP Settings => Edit
  - configurer une adresse IP manuelle de la façon suivante :

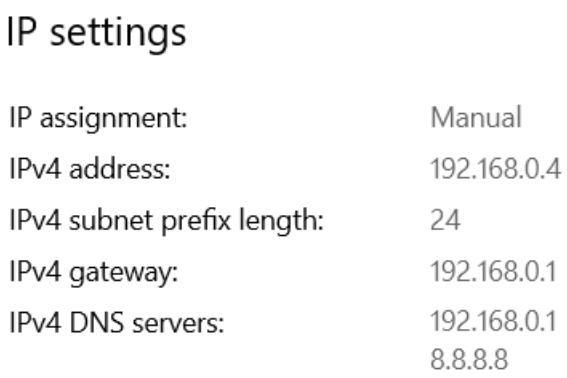


**Figure 1:** IP fixe cible

**NB :** IPv4 subnet prefix length = 24 = addresse de sous réseau de 24 bits= *255.255.255.0*

- Depuis la machine de développement, vérifier que la connexion fonctionne : `ping 192.168.0.14`
- Se connecter à la cible en SSH : `ssh root@192.168.0.14`
- Une fois connecter, lancer la commande `uname -a` pour connaitre le système d'exploitation de la cible =>

```
1 Linux csel 5.15.21 #1 SMP PREEMPT Fri Feb 25 20:44:19 UTC 2022
    aarch64 GNU/Linux
```

**Mise en place de l'espace de travail (*workspace*) sous CIFS/SMB**

L'objectif ici est que la machine hôte mette à disposition un répertoire qui sera monté et accédé par la cible. Pour le faire de façon automatique il faut depuis la cible :

- Créer un répertoire

```
mkdir -p /workspace
```

- Ajouter la ligne suivante à la fin du fichier `/etc/fstab`

```
//192.168.0.4/workspace /workspace cifs vers=1.0,username=root,
    password=toor,port=1445,noserverino
```

- Activer la configuration

```
mount -a
```

#### Génération d'applications sur la machine de développement hôte

Pour faire une compilation croisée depuis la machine de développement, le **makefile** doit faire appeler les bons outils de la **toolchain** adaptée à la cible (gcc, gdb, ...). Les indications suivantes seront utilisées :

```
# Makefile toolchain part
TOOLCHAIN_PATH=/buildroot/output/host/usr/bin/
TOOLCHAIN=$(TOOLCHAIN_PATH)aarch64-linux-

# Makefile common part
CC=$(TOOLCHAIN)gcc
LD=$(TOOLCHAIN)gcc
AR=$(TOOLCHAIN)ar
CFLAGS+=-Wall -Wextra -g -c -mcpu=cortex-a53 -O0 -MD -std=gnu11
```

#### Debugging de l'application sur la cible (VS-Code)

Pour debugger de façon simple, on installe **gdbserver** sur la cible et **gdb** (complet) sur la machine hôte. Ceci permet de debugger le programme qui tourne sur la cible depuis VS-Code, sur la machine hôte.

NB : **gdbserver** prends peu de place et peut dont être installé sur la cible sans problème.

- VS-Code permet de debugger plusieurs projets séparés en ouvrant le même "folder" grâce à son système **multi-root workspaces**
- Pour debugger, il a besoin de trois fichiers :
  - Un fichier "workspace"
  - Un fichier "task.json"
  - Un fichier "launcher.json"

NB : les fichiers **task.json** et **launcher.json** se trouvent aux racines des répertoires **.vscode**, qui se trouvent eux à la racine de chaque projet.

Ex : `/workspace/src/fibonacci/.vscode/`

**Core dumps** les core dumps permettent de visualiser à quel endroit du code le programme sur la cible a crashé. Une portion du code sera enregistrée dans un fichier nommé `core` avec des indications sur le bug..

- Il est nécessaire d'autoriser le core dumps en spécifiant ou non une taille maximale de fichier à ne pas dépasser. Depuis la cible :

```
ulimit -c <size|unlimited>
```

**NB :** *unlimited* : pas de limite de taille.

- lorsqu'on lance un programme depuis la cible, utiliser core dumps :

```
gdb <program-name> <core-file>
```

**NB :** le core-file se nomme généralement **core**, il faut donner le path complet du fichier

- pour récupérer l'information sur l'hôte :

```
sudo <path><target>-gdb <program-name> <core-file>
```

ou automatiquement sur vs code (à vérifier…)

#### Mise en place de l'environnement pour le développement du noyau sous CIFS/SMB

- Créer un répertoire `boot-scripts` à la racine du workspace et y créer le fichier `boot-cifs.cmd` contenant le code suivant :

```
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,username=root
    ,password=toor,port=1445 ip=$myip:$serverip:$gatewayip:$netmask
    :$hostname::off

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

```
11
12 booti $kernel_addr_r - $fdt_addr_r
```

- Créer également un fichier `Makefile` dans ce même répertoire contenant :

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

- Depuis le terminal, entrer dans ce répertoire et lancer la commande `make`

  => le fichier `boot.cifs` est alors créé

- Copier ce fichier sur la partition `boot` de la carte SD (en la connectant directement avec un adaptateur USB)
- Redémarrer la cible avec sa carte SD et l'arrêter dans **U-Boot** en pressant une touche au démarrage
- Depuis **U-Boot**, changer le script de démarrage pour `boot.cifs` et sauvegarder la configuration :

```
1 setenv boot_scripts boot.cifs
2 saveenv
```

- Redémarrer la cible à l'aide de la commande `boot` depuis U-Boot
- Lors du premier redémarrage, la cible crée une nouvelle paire de clés SSH

  => se connecter sur la console `root` et lancer la commande `chmod go= /etc/ssh/*_key`

  NB : Il faut accepter la nouvelle clé lors de la première connexion

## Questions

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

   Le bootloader U-Boot est généré par Buildroot en même temps que le kernel, le rootfs ainsi que la toolbox.

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

   Pour générer un package *custom* dans le Buildroot, il faut créer un nouveau répertoire dans `buildroot/`**`package`**. Ce répertoire doit contenir au moins deux fichiers qui sont `myCustomPackage.mk` et `Config.in`. Enfin, il faut référencer ce nouveau package dans le fichier `buildroot/`**`package`**`/Config.in`. Le package est alors disponible dans l'interface de configuration `make menuconfig`, depuis lequel il pourra être ajouter au noyau Linux.

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

   On peut modifier la configuration du noyau Linux en exectant la commande `make menuconfig` depuis la racine du répertoire de buildroot. un menu de configuration interactif se lance alors dans le terminal. Il est ensuite nécessaire de recompiler le noyau à l'aide de la commande `make`

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

   On peut configurer la structure de son *rootfs* depuis le menu de configuration de buildroot à l'aide de la commande `make menuconfig`.

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

   On pourrait sans problème utiliser l'eMMC à la place de la carte SD. En revanche, il faudrait flasher le circuit à distance, via le port série par exemple.

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 ?*

   Même si le travail de développement se limite à des applications en espace utilisateur, il serait préférable d'opter pour un environnement de développement où le *rootfs* de la cible serait partagé par la machine de développement via *nfs*. En effet, cela permettrait d'utiliser un IDE tel que *VS Code* proposant des outils de debug ergonomiques et de tester les applications en cours de développement. En revanche, le noyau pourrait très bien être stocké dans la mémoire flash de la cible si il n'a pas besoin d'être recompiler pendant le développement.

## Etat d'avancement / compréhension

La première partie de ce laboratoire consistait à se servir de *Buildroot* pour générer les outils nécessaires à faire fonctionner Linux sur un système embarqué et permettre la compilation croisée depuis la machine de développement pour la cible. Cette partie a été acquise. Dans un deuxième temps, il a fallut configurer l'environnement de développement afin de simplifier le développement d'applications pour la cible. Là encore, les différentes méthodes ainsi que la justification de la direction prise est claire pour les membres du groupe. Enfin, il a été question de tester différentes méthodes de débogage. Cette dernière partie mériterait d'être exercée d'avantage. Cela dit, le mini projet sera l'occasion de le faire.

## 2. Modules Noyaux (du 11.03 au 18.03)

### Résumé

Ce laboratoire est constitué de 8 exercices. Ces exercices nous permettent de nous familiariser avec plusieurs aspects des modules noyaux.

1. Exercice 01 : Cet exercice nous permet de voir la structure basique d'un module ainsi que son makefile associé

    1. Un module a besoin, au minimum, d'une fonction **Init** qui sera exécutée lors de son chargement dans le noyau de ce dernier et d'une fonction **exit** qui, elle, sera exécutée au moment de sa désinstallation.
    2. La commande modinfo permet d'extraire les métadonnées du module.
    3. Voici les messages que nous voyons après installation et désinstallation du module :

    ```
    [  704.967401] Linux module 01 loaded
    [  710.109996] Linux module skeleton unloaded
    ```

    4. Les deux commandes nous affichent tous les modules installés sur la machine.
    5. Pour permettre l'installation d'un module avec la commande modprobe, il faut ajouter les deux lignes suivante au Makefile :

    ```
    MODPATH := $(HOME)/workspace/buildroot/output/target # production mode install:
    $(MAKE) -C $(KDIR) M=$(PWD) INSTALL_MOD_PATH=$(MODPATH) modules_install
    ```

2. Exercice 02 :

    - Pour pouvoir passer des paramètres au module, il faut ajouter les lignes suivantes :

```
#include <linux/moduleparam.h>
static char* text= "dummy help";
module_param(text, charp, 0);
static int elements= 1; module_param(elements, int, 0);
```

    La librairie <linux/moduleparam.h> donne accès à la macro **module_param**. Cette macro permet de définir les paramètres du module.

    - Pour passer des paramètres au module lors de l'installation :

    ```
    insmod mymodule.ko elements=-1 'text="bonjour le monde"'
    ```

3. Exercice 03 :

```
cat /proc/sys/kernel/printk
7       4       1       7
```

Cette commande affiche les niveaux de log courant de la console. Le tableau ci-dessous nous montre la signification des ces valeurs :

| NOM | String | Function |
|---|---|---|
| KERN_EMERG | 0 | pr_emerg() |
| KERN_ALERT | 1 | pr_alert() |
| KERN_CRIT | 2 | pr_crit() |
| KERN_ERR | 3 | pr_err() |
| KERN_WARNING | 4 | pr_warn() |
| KERN_NOTICE | 5 | pr_notice() |
| KERN_INFO | 6 | pr_info() |
| KERN_DEBUG | 7 | pr_debug() |

Pour modifier ces valeurs, on peut exécuter la commande suivante :

```
echo 8 > /proc/sys/kernel/printk
cat /proc/sys/kernel/printk
8       4       1       7
```

4. Exercice 04 :

Le code suivant permet de créer une liste :

```
static LIST_HEAD (my_list);
```

La structure suivante permet de stocker son numéro unique (id), le texte (str), ainsi que le lien vers le prochain élément de la liste :

```
struct element
{
 char *str;
 int id;
 struct list_head node;
};
```

Le code suivant permet l'allocation mémoire dynamique des éléments ainsi que son insertion

dans la liste :

```
for(i;i < elements;i++){
     struct element* ele;
     ele = kmalloc(sizeof(*ele), GFP_KERNEL); // create a new
        element if (ele != NULL)
     ele->id = i;
     ele->str = text;
     list_add_tail(&ele->node, &my_list); // add element at the
        end of the list }
     pr_debug(" element with id %d and str %s has been added to
        the list",ele->id,ele->str);
 }
```

La libération mémoire des éléments se fait avec le code ci-dessous :

```
while(!list_empty(&my_list)){
     ele = list_entry(my_list.next, struct element, node);
     list_del(&ele->node);
     kfree(ele);
     pr_debug("element free");
 }
```

5. Exercice 05 :

Pour pouvoir accéder aux registres d'un périphérique il faut demander l'accès mémoire à une région au noyeau linux. Pour cela il faut utiliser la fonction suivante :

```
res[0] = request_mem_region(CHIP_ID_ADD, 0x1000, "allwiner sid");
res[1] = request_mem_region(TEMP_CPU_ADD, 0x1000, "allwiner h5 ths
   ");
res[2] = request_mem_region(MACC_ADD, 0x1000, "allwiner h5 emac");
```

Une fois cette demande effectuée il faut transférer les adresses mémoire du noyau sur la mémoire virtuel.

```
reg_chipid = ioremap(CHIP_ID_ADD, 0x1000);
reg_temp_sensor = ioremap(TEMP_CPU_ADD, 0x1000);
reg_mac = ioremap(MACC_ADD, 0x1000);
```

Une fois avoir mappé les addresses dans la mémoire virtuel, il est possible de lire et écrire à ces addresses. Dans ce labo on ne fait que de la lecture.

```
chipid[0] = ioread32(reg_chipid+0x200);
chipid[1] = ioread32(reg_chipid+0x204);
chipid[2] = ioread32(reg_chipid+0x208);
chipid[3] = ioread32(reg_chipid+0x20c);
```

Dans l'exemple au-dessus on vient lire les 4 registres du Chip ID.

Une fois que on a fini d'utiliser les registres périphérique il faut libérer les espaces mémoire. Cela se fait lors de la désinstallation du module de la manière suivante :

```
release_mem_region(CHIP_ID_ADD, 0x1000);
release_mem_region(TEMP_CPU_ADD, 0x1000);
release_mem_region(MACC_ADD, 0x1000);
```

6. Exercice 06 :

   L'instanciation d'un thread se fait de la manière suivante :

   ```
   kthread_run(thread, NULL, "EX06");
   ```

   La fonction exécutée dans le thread est la suivante :

   ```
   static int thread(void* data){
       while (!kthread_should_stop())
       {
           pr_info("See you in 5 sec");
           ssleep(5);
       }
       return 0;
   }
   ```

   L'arrêt du thread se fait à la désinstallation du module de la façon suivante :

   ```
    kthread_stop(k);
   ```

7. Exercice 07 :

   Dans cette exercice, l'utilisation des waitqueues était obligatoire. Pour cela, deux waitqueues sont utilisées. La première est initialisée statiquement, tandis que la deuxième est initialisée dynamiquement.

   La première waitqueues est utilisée par le premier thread pour attendre son réveil par le deuxième thread.

   ```
   DECLARE_WAIT_QUEUE_HEAD(queue);

   static int thread(void* data){

       pr_info("Thread 1 is active");
       while (!kthread_should_stop())
       {
           pr_info("Thread 1: is waiting for thread 2");
           wait_event(queue, atomic_read(&start) != 0 ||
              kthread_should_stop());

           pr_info("Thread 1: is active");
           atomic_set(&start,0);
   ```

```
13      }
14      return 0;
15  }
```

**NB :** L'utilisation du variable atomic est requise, car les deux threads accèdent à cette dernière.

La deuxième waitqueues est utilisée en mode timeout pour se réveiller toute les 5 secondes.

```
1   static int thread2(void* data){
2
3       wait_queue_head_t queue_2;
4       init_waitqueue_head(&queue_2);
5
6       pr_info("Thread 2: is active");
7       while (!kthread_should_stop())
8       {
9           int ret = wait_event_timeout(queue, kthread_should_stop(),
                5*HZ);
10          if(ret == 0){
11              pr_info("Thread 2: Timeout elapsed");
12          }
13          atomic_set(&start,1);
14          wake_up(&queue);
15          pr_info("Thread 2: See you in 5 sec");
16      }
17      return 0;
18  }
```

8. Exercice 08 :

   Dans cet exercice, il était demandé de capturer l'interruption des trois switches de la carte d'extension.

   Tout d'abord, nous devons obtenir le port GPIO de la manière suivante :

```
1  int status;
2  status = gpio_request(K1,k1_label);
3  status = gpio_request(K2,k2_label);
4  status = gpio_request(K3,k3_label);
```

   Une fois le port récupéré, nous devons attacher la fonction d'interruption sur le bon port. Pour cela, il faut obtenir le vecteur d'interruption, puis attacher la callback à ce vecteur tout en spécifiant le fanion de gestion des interruptions.

```
1  request_irq(gpio_to_irq(K1) ,gpio_callback, IRQF_TRIGGER_FALLING,
       k1_label, k1_label);
2  request_irq(gpio_to_irq(K2) ,gpio_callback, IRQF_TRIGGER_FALLING,
       k2_label, k2_label);
3  request_irq(gpio_to_irq(K3) ,gpio_callback, IRQF_TRIGGER_FALLING,
       k3_label, k3_label);
```

La callback d'interruption ne fait qu'afficher le switche sur lequel l'appuis a été fait.

```
1  irqreturn_t gpio_callback(int irq, void *dev_id){
2
3      pr_info("interrupt %s: rise",(char*) dev_id);
4
5      return IRQ_HANDLED;
6  }
```

### Questions

Ce laboratoire ne possédait aucune question en plus des exercices vu dans la section précédente.

### Etat d'avancement / compréhension

L'utilisation et l'implémentation des modules ont été parfaitement comprises par le groupe. Les notions de multi threading, mise en sommeil à l'aide de waitqueues et listes d'éléments, ont été parfaitement comprises. Cependant, la lecture de registre (exercice 5) devra être plus approfondie de notre côté.

### Retour personnel sur le laboratoire

Les exercices proposés lors de ce laboratoire étaient très intéressants. Avoir la correction rapidement nous a permis d'améliorer notre compréhension des différents sujets abordés.

## 3. Pilotes de périphériques (du 25.03 au 01.04)

### Résumé

#### Pilotes orientés mémoire

Les pilotes orientés mémoire permettent de mapper des zones de la mémoire physique du processeur sur une zone de mémoire virtuelle nécessaire au pilotage de périphériques. Ces pilotes se développent en zone utilisateur. Pour ce faire, on emploi l'opération `mmap`, disponible en intégrant la bibliothèque `<sys/mman.h>`.

Une fois la zone mémoire mappée, elle est accessible à l'emplacement `/dev/mem`.

### Mise en place du pilote

- inclusion de la bibliothèque

```c
#include <sys/mman.h>
```

- ouverture du fichier `/dev/mem`

```c
int fd = open("/dev/mem", O_RDWR);
if (fd < 0) {
    printf("Could not open /dev/mem: error=%i\n", fd);
  return -1;
}
```

- *mapping* de la mémoire afin de rendre des registres du processeur accessibles depuis la mémoire virtuelle (les commentaires détaillent les opérations effectuées)

```c
size_t psz     = getpagesize();       // récupération de la taille d'une page mémoire
off_t dev_addr = 0x01c14200;      // adresse du début des registres que l'on souhaite récupérer
off_t ofs      = dev_addr % psz;  // nombre d'octets d'écart entre l'adresse de nos registres et le début d'une nouvelle page
                                  // (on souhaite récuperer une page entiére!)
off_t offset   = dev_addr - ofs;  // adresse du début de la page dans laquelle se trouvent ces registres

volatile uint32_t* regs = mmap (
    0,                          // void* addr       => généralement NULL, adresse de départ en mémoire virtuelle
    psz,                  // size_t length    => taille de la zone à placer en mémoire virtuelle ( => taille d'une page entiére)
    PROT_READ | PROT_WRITE,   // int prot=> droits d'accés à la mémoire: read, write, execute
    MAP_SHARED,               // int flags        => visibilité de la page pour d'autres processus: shared, private
    fd,                       // int fd           => descripteur du fichier correspondant au pilote
    offset                    // off_t offset     => début de la zone mémoire à placer en mémoire virtuelle
);
```

- opérations souhaitées sur le périphérique à l'aide de l'adresse virtuelle retournée par `mmap`

```c
// contrôle du pointeur retourné par `mmap`
if (regs == MAP_FAILED) {  // (void *)-1
    printf("mmap failed, error: %i:%s \n", errno, strerror(errno));
    return -1;
```

```
5  }
6
7  // copie des valeurs en RAM pour utilisation dans le programme
8  uint32_t chipid[4] = {
9      [0] = *(regs + (ofs + 0x00) / sizeof(uint32_t)),
10     [1] = *(regs + (ofs + 0x04) / sizeof(uint32_t)),
11     [2] = *(regs + (ofs + 0x08) / sizeof(uint32_t)),
12     [3] = *(regs + (ofs + 0x0c) / sizeof(uint32_t)),
13 };
```

- libération de l'espace mémoire avec la fonction `munmap`

```
1  munmap((void*)regs, psz);
```

- fermeture du fichier virtuel

```
1  close(fd);
```

### Pilotes orientés caractères

Un pilote orienté caractères permet d'interagir avec des périphériques de façon assez simple. Il faut faire la distinction entre le pilote, qui est le programme permettant de piloter un ou plusieurs périphériques de même type, et le périphérique ou *device*, qui est une instance du pilote dédiée à la gestion d'un objet matériel. L'échange de données entre l'instance du pilote et le périphérique matériel se fait au travers de fichiers virtuels créés par le pilote.

### Structure du pilote

- inclusion des bibliothèques nécessaires

```
1  #include <linux/cdev.h>          /* needed for char device driver */
2  #include <linux/fs.h>            /* needed for device drivers */
3  #include <linux/init.h>          /* needed for macros */
4  #include <linux/kernel.h>        /* needed for debugging */
5  #include <linux/module.h>        /* needed by all modules */
6  #include <linux/moduleparam.h> /* needed for module parameters */
7  #include <linux/uaccess.h>       /* needed to copy data to/from user
       */
```

- déclaration de l'objet `dev_t` permettant de définir le numéro de pilote

```
1  static dev_t skeleton_dev;
```

  Le numéro de pilote est constitué de :

  - numéro majeur de 12 bits

  - numéro mineur de 20 bits

  Il est réservé dynamiquement lors du chargement du pilote, à l'aide de la fonction `alloc_chrdev_region`

- déclaration de la structure `skeleton_cdev` permettant l'enregistrement du pilote caractère dans le noyau

```
1  static struct cdev skeleton_cdev;
```

  Lors du chargement du pilote, la structure `cdev` est :

  - initialisée à l'aide de la méthode `cdev_init`
  - enregistrée dans le noyau à l'aide de la méthode `cdev_add`

- définition de la structure de fichier

```
1  static struct file_operations skeleton_fops = {
2      .owner   = THIS_MODULE,
3      .open    = skeleton_open,
4      .read    = skeleton_read,
5      .write   = skeleton_write,
6      .release = skeleton_release,
7  };
```

  Chaque attribut que l'on souhaite utiliser dans le pilote doit être initialisé à l'aide d'un pointeur vers une fonction implémentée dans le code du pilote.

- Lors de l'initialisation du module (`skeleton_init`), il faut réserver dynamiquement le numéro du pilote et enregistrer le pilote dans le noyau :

```
1  static int __init skeleton_init(void) {
2      // Réservation dynamique du numéro de pilote
3      int status = alloc_chrdev_region(
4          &skeleton_dev,    // instance dev_t
5          0,                // base_minor : premier numéro mineur du
       pilote
6          1,                // count : le nombre de numéros mineurs
       requis par le pilote
7          "mymodule"        // nom du pilote de périphérique
8      );
9
10     if (status == 0) {
11         // enregistrement du pilote dans le noyau
12         //    => association entre les numéros majeurs / mineurs
       et les opérations de fichies attachées au pilote
13         cdev_init(&skeleton_cdev, &skeleton_fops);
14         skeleton_cdev.owner = THIS_MODULE;
15         status = cdev_add(
```

```
16              &skeleton_cdev,   // pointeur sur la structure du
                    pilote
17              skeleton_dev, // numéro du pilote
18              1             // count : indique le nombre de périphé
                    riques
19          );
20      }
21
22      pr_info("Linux module skeleton loaded\n");
23      return 0;
24  }
```

- Lors du déchargement du pilote, il faut éliminer le pilote dans le noyau et libérer les numéros majeur et mineur du pilote.

```
1  static void __exit skeleton_exit(void) {
2      // élimination du pilote dans le noyau
3      cdev_del(&skeleton_cdev);
4      // libération des numéros (majeurs/mineurs) du pilote
5      unregister_chrdev_region(skeleton_dev, 1);
6      pr_info("Linux module skeleton unloaded\n");
7  }
```

**Méthode Read** La méthode `skeleton_read` implémentée dans le code du pilote écrit les données nécessaires dans le buffer de l'espace utilisateur. Elle prends en paramètre :

- le descripteur du fichier ouvert dans l'espace utilisateur
- le pointeur vers le buffer de destination
- la quantité d'octets maximale à écrire dans l'espace utilisateur
- offset d'écriture (les *n* premiers octets avant ceux qu'on envoi à l'utilisateur)

La méthode met à jour la valeur de l'offset afin de ne pas envoyer plusieurs fois les mêmes données à l'utilisateur. Elle retourne ensuite le nombre d'octets qu'elle y a écrit.

```
1  static ssize_t skeleton_read(struct file* f, char __user* buf, size_t
       count, loff_t* off) {
2      // compute remaining bytes to copy, update count and pointers
3      ssize_t remaining = BUFFER_SZ - (ssize_t)(*off);
4
5      // init. read ptr
6      char* ptr = s_buffer + *off;
7
8      // protect reading against buffer overflow
9      if (count > remaining) count = remaining;
10
11     // compute future return count
12     *off += count;
```

```
13
14      // copy required number of bytes
15      if (copy_to_user(buf, ptr, count) != 0) count = -EFAULT; // EFAULT
            = bad adress
16      // NB : copy_to_user retourne le nombre d'octets qui n'ont pas pu ê
            tres copiés!
17
18      return count; // retourne le nombre d'octets écris dans l'espace
            utilisateur
19  }
```

**Méthode Write** La méthode `skeleton_write` copie les données de l'espace utilisateur vers le buffer instancié dans le code du pilote.

La méthode doit prendre en considération la taille maximale du buffer du pilote afin d'éviter un éventuel débordement. Elle retourne le nombre d'octets copiés dans le buffer.

```
1  static ssize_t skeleton_write(struct file* f, const char __user* buf,
       size_t count, loff_t* off) {
2
3      ssize_t remaining = BUFFER_SZ - (ssize_t)(*off);
4      pr_info("skeleton: at%ld\n", (unsigned long)(*off));
5
6      // check if still remaining space to store additional bytes
7      if (count >= remaining) count = -EIO;
8
9      if(count > 0){
10         char* ptr = s_buffer + *off;
11         *off += count;
12         ptr[count] = 0; // make sure string is null terminated
13         if(copy_from_user(ptr, buf, count)) count = -EFAULT;
14     }
15
16     pr_info("skeleton: write operation... written=%ld\n", count);
17
18     return count; // retourne le nombre d'octets copiés
19 }
```

**Création du fichier d'accès au périphérique dans le répertoire `/dev`** Après avoir chargé le pilote, il faut créer le fichier à l'aide de la commande `mknod`

```
1  mknod /dev/mymodule c 511 0
```

- **c** : type de périphérique caractère
- **511** : numéro majeur
- **0** : numéro mineur

**données privées d'instance** le descripteur de fichier contient un pointeur `private_data` qui peut être initialisé pour pointer sur un espace de données dans le pilote. Ceci permet d'attribuer un espace de donnée propre à chaque instance du pilote.

Pour ce faire, il faut :

- déclarer **en global** un pointeur de pointeur pour le tableau de buffers

```
#define BUFFER_SZ 1000
static char** buffers = 0;
```

- définir un nombre d'instance, un paramètre de module peut être utiliser pour rendre ce nombre configurable

```
static int instances = 3;
module_param(instances, int, 0);
```

- lors de l'initialisation du pilote, créer un tableau de buffers dynamique en fonction du nombre d'instances :

```
if(status == 0){
    buffers = kzalloc(instances * sizeof(char*), GFP_KERNEL);
    for(i=0; i<instances; i++){
        buffers[i] = kzalloc (BUFFER_SZ, GFP_KERNEL);
    }
}
```

- dans la méthode `open`, il faut initialiser le pointeur `private_data` du descripteur de fichier pour le faire pointer sur le buffer correspondant à l'instance que l'utilisateur a ouvert :

```
f->private_data = buffers[iminor(i)];
pr_info("skeleton: private_data=%p\n", f->private_data);
```

**NB** : la méthode `iminor(struct inode* i)` renvoi le numéro mineur de l'instance (entre *0* et *instances*)

- lors d'opérations de lecture / écriture, il faut se référer au pointeur configuré pour l'instance lors de l'ouverture du fichier

```
char* ptr = (char*) f->private_data + *off;
```

- enfin, lors du déchargement du module, il faut libérer la mémoire allouée pour le tableau de buffers

```
for(i=0; i<instances; i++) kfree(buffers[i]);
kfree(buffers);
```

**NB** : il ne faut pas oublier d'initialiser le nombre correct d'instances dans la structure `dev_t skeleton_dev` lors du chargement et du déchargement du module.

### Sysfs

le sysfs est un système de fichiers virtuels permettant de représenter les objets du noyau dont les périphériques. Il est accessible via le répertoire `/sys`.

L'objectif de cette dernière partie est de construire des pilotes sous forme de classes de périphériques qui seront accessibles dans `/sys/class`.

### Structure du pilote

- Pour chaque attribut que l'on souhaite présenter dans le *sysfs*, il faut :
  - déclarer l'attribut en tant que **variable globale** dans le code du pilote. Les attributs peuvent êtres de n'importe quel type, y compris des structures. Ex :

```
struct skeleton_config {
    int id;
    long ref;
    char name[30];
    char descr[30];
};
static struct skeleton_config config;
```

  - créer une méthode d'accès en lecture / présentation dans le *sysfs*. Cette méthode retournera le contenu à afficher lorsqu'une commande `cat` sera effectuée sur le fichier correspondant à l'attribut.

```
ssize_t sysfs_show_cfg(struct device* dev, struct
    device_attribute* attr, char* buf){
    sprintf(
        buf, "%d %ld %s %s\n",
        config.id,
        config.ref,
        config.name,
        config.descr
    );
    return strlen(buf);
}
```

    La méthode copie le contenu à afficher dans le buffer pointé en argument et en retourne la taille en octets.

- créer une méthode d'accès en écriture (store). Cette méthode prendra en argument un pointeur vers un buffer contenant la donnée à stocker dans l'attribut.

```
ssize_t sysfs_store_cfg(struct device* dev, struct
    device_attribute* attr, const char* buf, size_t count){
     sscanf(
         buf,             // src string
         "%d %ld %s %s",  // string data format
         &config.id,      // pointers to data
         &config.ref,     // |
         config.name,     // |
         config.descr     // |
     );
     return count;
}
```

La méthode copie le contenu du buffer pointé en argument dans l'attribut. Dans l'exemple ci-dessus, l'attribut étant une structure, la méthode `sscanf` est utilisée afin de respecter la structure des données.

- utiliser la macro `DEVICE_ATTR` pour instancier la structure `device_attribute` qui spécifie les méthodes d'accès de l'attribut.

```
DEVICE_ATTR(config, 0664, sysfs_show_cfg, sysfs_store_cfg);
```

la structure `device_attribute` contient les éléments suivants:

```
struct device_attribute {
    struct attribute attr;
    ssize_t (*show) (struct device *dev, struct
       device_attribute *attr, char *buf);
    ssize_t (*store) (struct device *dev, struct
       device_attribute *attr, const char *buf, size_t count);
};
```

Les derniers arguments sont des pointeurs vers les méthodes implémentées plus haut.

- en global, il faut déclarer les deux structures suivantes représentant respectivement la classe et le device :
  - structure de classe `sysfs_class`

```
static struct class* sysfs_class;
```

  - structure device `sysfs_device`

```
static struct device* sysfs_device;
```

- lors du chargement du module (dans la méthode `skeleton_init`), il faut :

- instancier la classe préalablement déclarée à l'aide de la méthode `class_create`

```
sysfs_class  = class_create(
    THIS_MODULE,    // struct module * owner => le propriétaire de la classe
    "my_sysfs_class"// const char * name     => le nom de la classe
);
```

- instancier le *device* préalablement déclaré à l'aide de la méthode `device_create`

```
sysfs_device = device_create(
    sysfs_class,        // structure classe
    NULL,               // device parent
    0,                  // dev_t devt (définition du numéro de pilote)
    NULL,               // format (const char *)
    "my_sysfs_device"   // nom du device
);
```

- Installer les méthodes d'accès pour chaque attribut

```
if (status == 0) status = device_create_file(
    sysfs_device,       // device
    &dev_attr_config    // device_attribute struct address
);
```

- Lors du déchargement du module (dans la méthode `skeleton_exit`), il faut :
  - désinstaller les méthodes d'accès pour chaque attribut

```
device_remove_file(sysfs_device, &dev_attr_config);
```

  - détruire le *device*

```
device_destroy(sysfs_class, 0);
```

  - détruire la classe

```
class_destroy(sysfs_class);
```

### Etat d'avancement / compréhension

Dans l'état d'avancement actuel des travaux, les pilotes orientés mémoire, les pilotes orientés caractères ainsi que le sysfs ont étés correctement assimilés. En revanche, les travaux pratiques optionnels n'ont malheureusement pas pu être réalisés, faute de temps. Nous aurons probablement l'occasion de les

utiliser dans le cadre de notre mini projet ou du moins d'étudier en quoi ils pourraient améliorer son développement.

### Retour personnel sur le laboratoire

Ce laboratoire était très intéressant et complète tout-à-fait le précédent en nous ouvrant les portes vers le développement d'un pilote complet avec lequel une application peut interagir depuis l'espace utilisateur. Ceci dit, bien que le cours soit très bien structuré, je trouve que le volume de matière à assimiler est un peu trop important. Ceci peut créer de la confusion quand il s'agit de faire les bons choix pour un développement parmis les nombreux outils vus en cours.