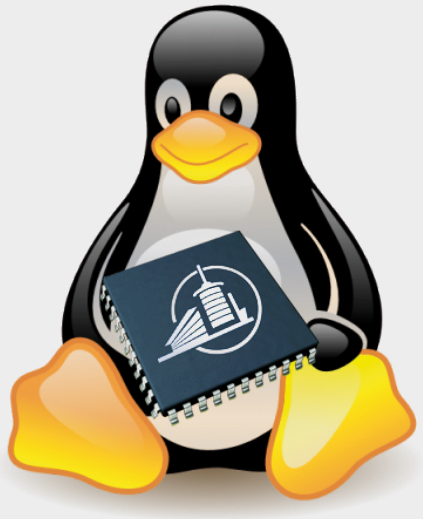


Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

MA-CSEL1 – Construction Systèmes Embarqués sous Linux

Noyau:
Pilotes de périphériques

HES-SO//Master TIC/TIN 2020-21

Daniel Gachet – HEIA-FR – Télécommunications

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz

Contenu

- Introduction
- Pilotes orientés mémoire
- Pilotes orientés caractère
- Plateforme
- Device Tree
- System file system – sysfs
- *Input/Ouput Control – ioctl*
- *Process file system – procfs*
- Opérations bloquantes
- Aspects pratiques

MASTER OF SCIENCE IN ENGINEERING

MSE

Introduction

MASTER OF SCIENCE IN ENGINEERING

MSE

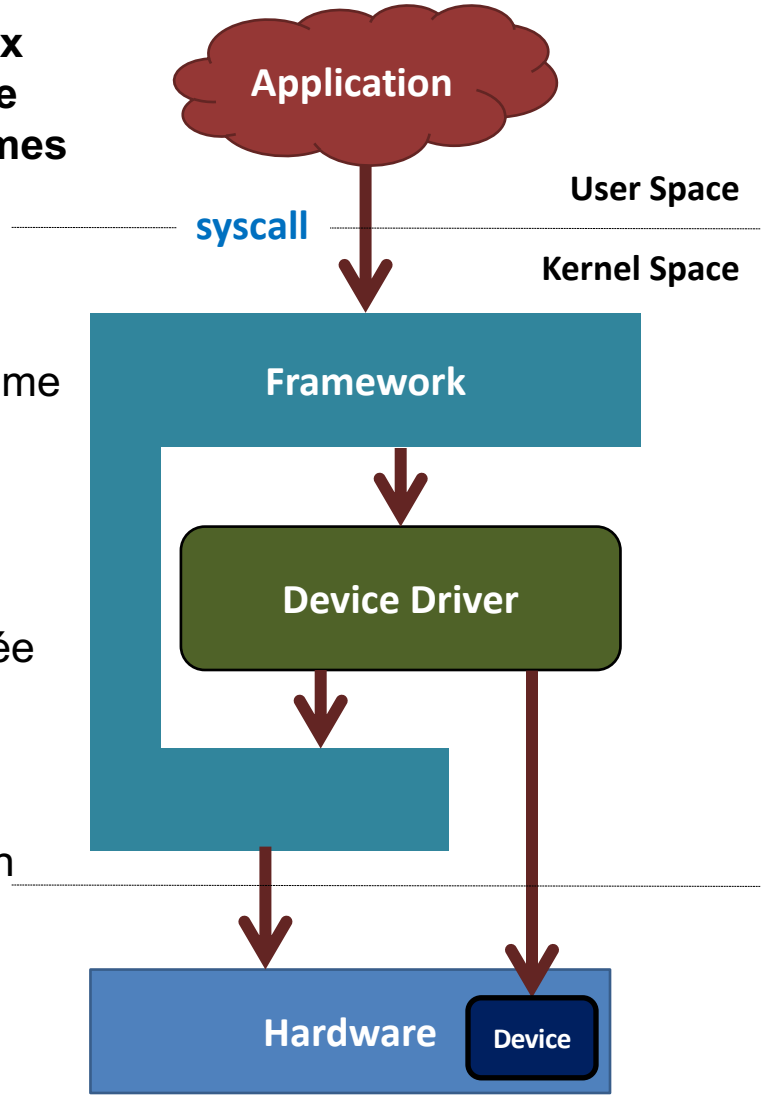
Types de pilote

- **Sous Linux pour accéder aux ressources matérielles, il est impératif de développer un pilote de périphérique (device driver)**
- **Linux distingue deux types principaux de pilotes de périphériques**
  - Pilotes orientés caractère (char device driver), visible sous /dev
    - Accès séquentiel des données (octet par octet)
    - Accès à des périphériques simples (port série, …)
  - Pilotes orientés bloc (block device driver), visible sous /dev
    - Accès aléatoire des données (par bloc)
    - Accès aux disques
- **Cette liste peut être complétée par deux types supplémentaires**
  - Pilotes réseau (network device driver), visible avec ifconfig
    - Accès aux interfaces réseau (Ethernet,…)
    - Accès aux piles de protocoles
  - Pilotes orientés mémoire (uio device driver)
    - Accès aux périphériques très simples avec accès aux registres mémoires (memory mapped devices)

MASTER OF SCIENCE IN ENGINEERING
MSE

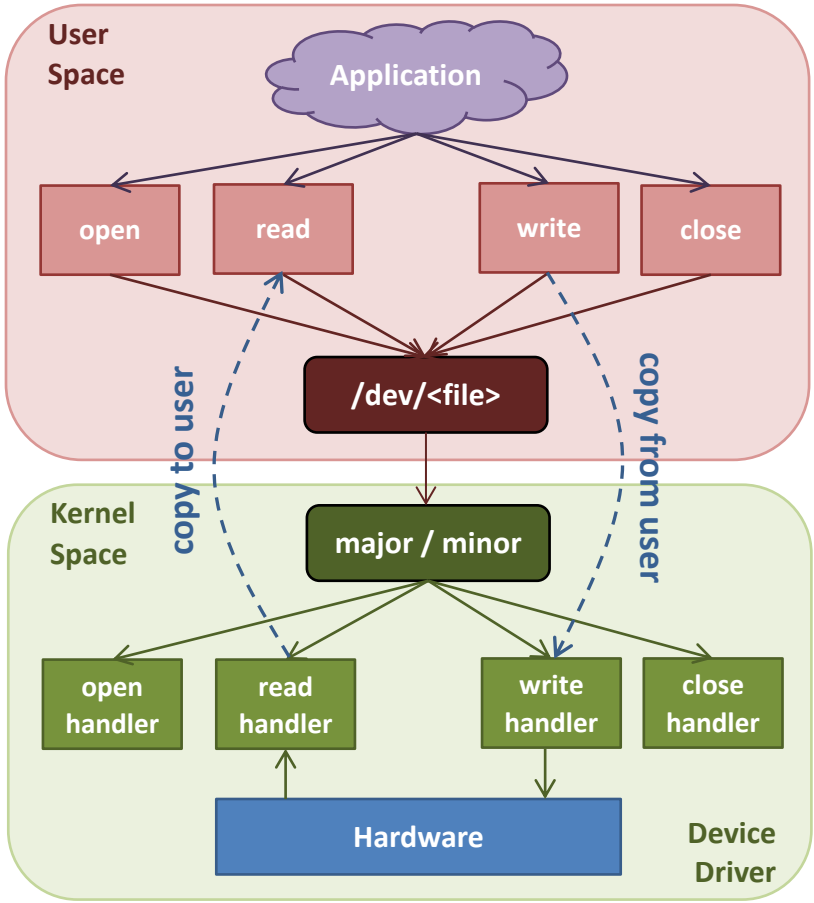
Device Model

- **Le "Device Model" proposé par le noyau Linux permet de maximaliser la réutilisation du code entre les différentes architectures et plateformes hardware**
- **3 structures importantes héritées de la structure `struct kobject`**
  - **`struct device`** : périphérique dans le système et généralement associé à un bus. Il est découvert de différentes manières (hot-plug, pilote de périphérique (driver), initialisation)
  - **struct device_driver** : entité logicielle associée au périphérique et permettant d'effectuer des opérations sur ce dernier
  - **`struct bus_type`** : canal de communication entre le µP et le périphérique d'entrée/sortie (quelques bus: I2C, SPI, USB, PCI,…)



Interface utilisateur

- **L'accès aux pilotes de périphériques se fait par l'intermédiaire de fichiers virtuels**
- **Espace utilisateur**
  - Le fichier, situé dans le répertoire `/dev`, permet d'interagir avec le pilote chargé dans le noyau Linux à l'aide des opérations standard sur les fichiers
- **Espace noyau**
  - Pour connaître, le pilote en charge de traiter les requêtes de l'application en espace utilisateur, le noyau utilise un numéro de périphérique *(device number)* associé au nom du fichier. Celui-ci est composé d'un numéro majeur associé au pilote (le code) et d'un numéro mineur associé au périphérique (instance du pilote)
  - Pour chaque opération en espace utilisateur, le pilote de périphérique implémente une méthode correspondante *(handler) dans le noyau*



Répertoire des pilotes de périphériques

- Tous les pilotes orientés caractère ou bloc sont accessibles pour les utilisateurs par l'intermédiaire de fichiers virtuels situés dans le répertoire `/dev`

```
$ ls -l /dev
brw-------    1 root   root    179,   33 Jan  1 00:00 mmcblk0p1
brw-------    1 root   root    179,   34 Jan  1 00:00 mmcblk0p2
crw-------    1 root   root     10,   61 Jan  1 00:00 network_latency
crw-------    1 root   root     10,   60 Jan  1 00:00 network_throughput
crw-rw-rw-    1 root   root      1,    3 Jan  1 00:00 null
crw-rw-rw-    1 root   root      5,    2 Jan  1 00:00 ptmx
```

- Les fanions **b** et **c** indiquent le type de pilote (**b** ⇔ block, **c** ⇔ character)

- **1, 5, 10** et **179** sont les « **major numbers** »
- Les **numéros majeurs** permettent d'identifier les différentes pilotes de périphériques à l'intérieur du noyau Linux (le code)

- **2, 3, 33, 34, 60** et **61** sont les « **minor numbers** »
- Les **numéros mineurs** représentent une instance d'un pilote correspondant à un périphérique donné (les données)

Interfaces de configuration et de gestion

- **Afin de pouvoir effectuer des opérations de configuration, de gestion et/ou de maintenance sur un périphérique, les pilotes de périphériques nécessitent la mise en œuvre d'une interface supplémentaire**

- **Linux propose 3 interfaces**
  - ioctl – input/output control
    - interface du pilote mis à disposition au niveau du fichier d'accès accessible sous /dev
    - utilisée pour la configuration des pilotes de périphériques
  - procfs – process filesystem
    - fichiers d'état (configuration) accessible sous /proc
    - utilisée principalement pour la configuration et le monitoring du noyau Linux
  - sysfs – system filesystem
    - fichiers de configuration et de gestion accessible sous /sys
    - utilisée principalement pour la configuration et la gestion des modules noyaux et des pilotes de périphériques

MASTER OF SCIENCE IN ENGINEERING

MSE

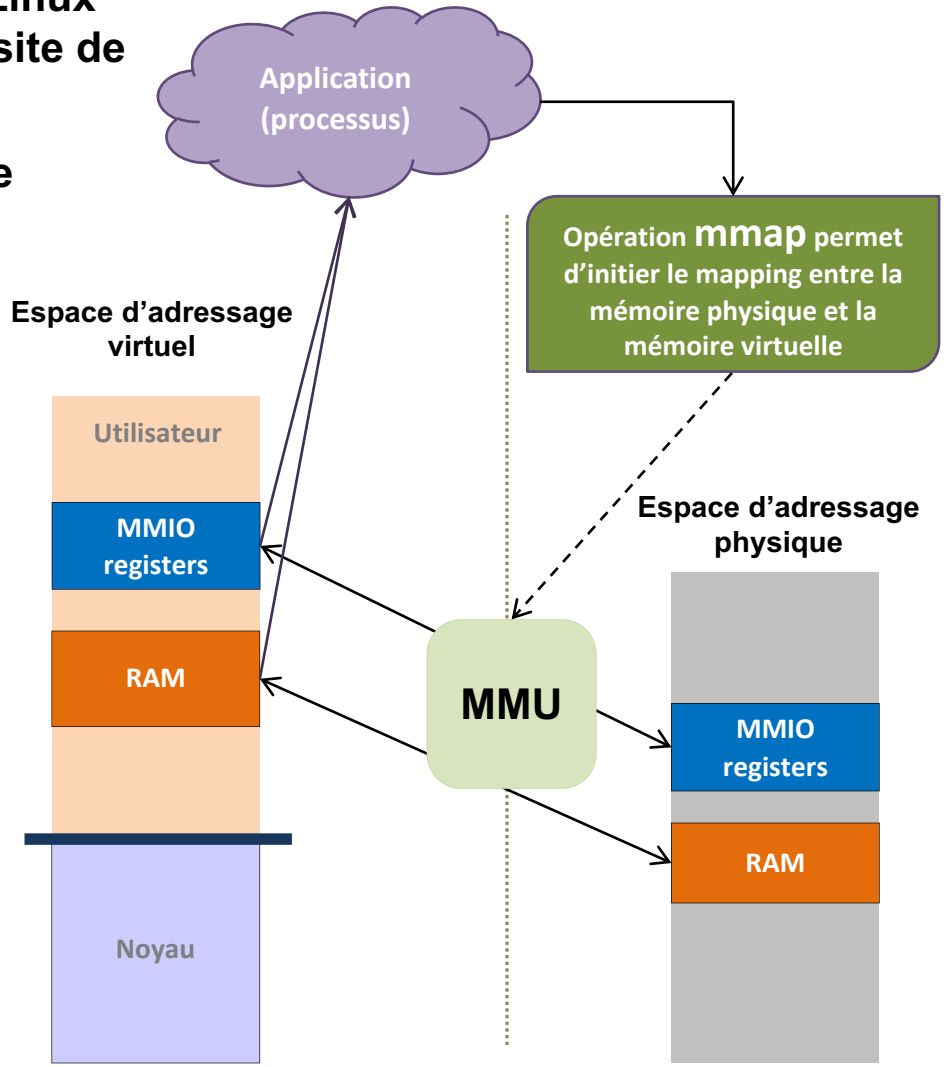
Pilotes orientés mémoire

MASTER OF SCIENCE IN ENGINEERING

MSE

Concept des pilotes orientés mémoire (uio-driver)

- **Le développement de pilotes sous Linux est relativement complexe et nécessite de bonnes connaissances du noyau**
- **Il souffre également de beaucoup de restrictions**
- **Les pilotes orientés mémoire permettent de mapper dans l'espace virtuel du processus d'application les registres et zones mémoire nécessaires au pilotage du périphérique**
  - Ce mapping est réalisé par l'appel de l'opération `mmap`
  - Le fichier `/dev/mem` offre ce service par défaut
  - Il est possible d'implémenter ce service avec un son propre pilote



Avantages et inconvénients

- **Les pilotes orientés mémoire (uio-driver) offrent une alternative intéressante lors de migration d'applications et de leurs pilotes d'un OS propriétaire vers Linux ou lorsque l'interface avec les périphériques à gérer est simple**
- **Avantages**
  - Pas (ou peu) d'adaptations en cas de mise à jour du noyau Linux
  - Choix du langage de programmation
  - Accès à toutes les librairies (p. ex. calcul en virgule flottante autorisé)
  - Accès aux registres plus efficace (pas d'appel système)
  - Pas de problèmes de licences liés à Linux (GPL)
- **Désavantages**
  - ***Pas de support pour le traitement des interruptions***
  - Fonctionnalité des pilotes est limitée au développement d'application en mode user
  - Séparation en *kernel space* et *user space* est abandonnée
  - Système peut devenir plus complexe (mauvaise abstraction / API)

Implémentation du pilote dans l'espace utilisateur

- **L'implémentation d'un pilote de périphériques orientés mémoire s'implémente à l'aide de 5 opérations** *(#include <sys/mman.h>)*
  - Ouverture du fichier correspondant au pilote `(int fd = open (…))`
  - Appel de l'opération mmap afin de placer dans la mémoire virtuelle du processus les registres du périphérique

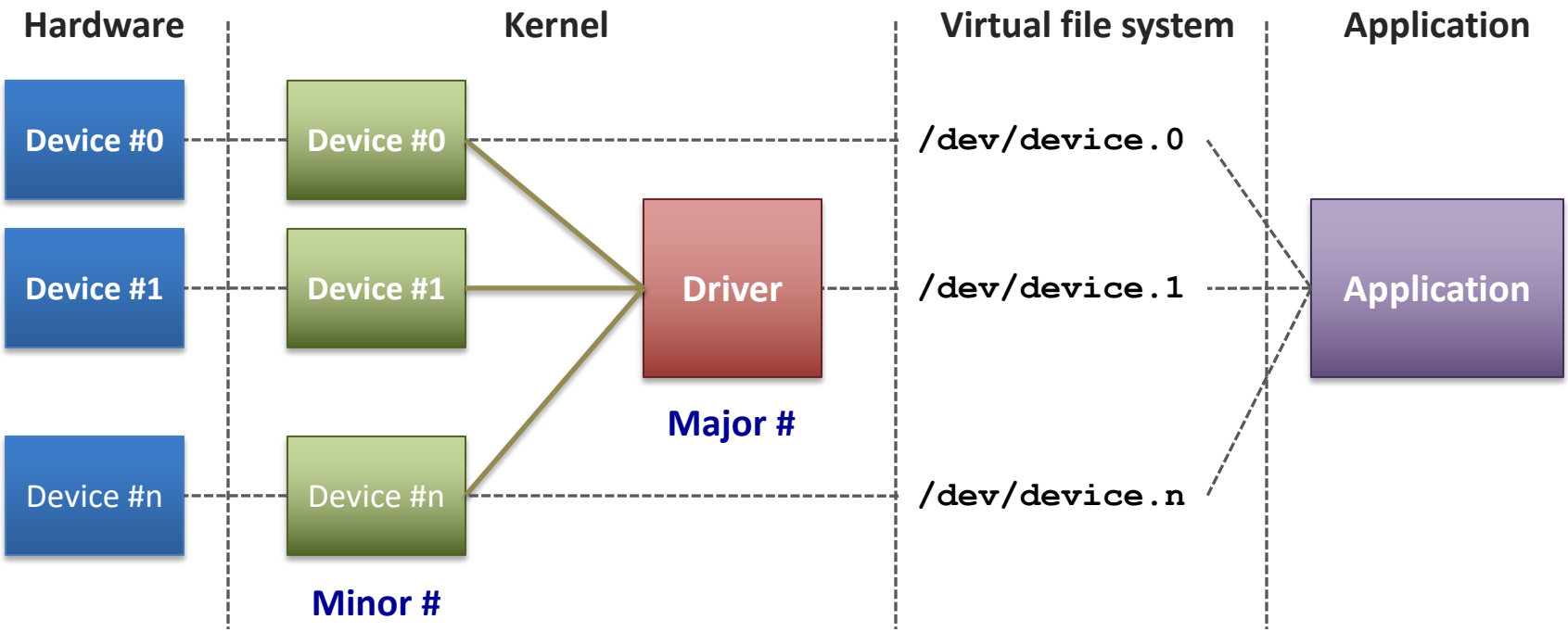
```
void* mmap (
  void* addr,     //généralement NULL, adresse de départ en mémoire virtuelle
  size_t length, //taille de la zone à placer en mémoire virtuelle
  int prot,       //droits d'accès à la mémoire: read, write, execute
  int flags,      //visibilité de la page pour d'autres processus: shared, private
  int fd,         //descripteur du fichier correspondant au pilote
  off_t offset); //offset des registres en mémoire
```

  - Opérations sur le périphérique à l'aide de l'adresse virtuelle retournée par l'opération mmap
  - Après utilisation, appel de l'opération munmap pour libérer l'espace mémoire `(munmap (…))`
  - Fermeture du fichier `(close (…))`

Pilotes orientés caractère

Structure d'un pilote orienté caractère

- **Un pilote de périphériques orienté caractère est constitué de deux parties essentielles**
  - Le pilote (driver)
    - *code permettant de piloter un ou plusieurs périphériques du même type*
  - Le périphérique (device)
    - *instance du pilote (l'objet) permettant à une application d'échanger des données avec un périphérique au travers du système de fichiers virtuels*



Etapes d'implémentation

- **L'implémentation d'un pilote orienté caractère peut être décomposer en 5 étapes principales**

1. Implémentation des opérations sur les fichiers (handler) correspondantes aux appels système qu'une application en espace utilisateur pourra utiliser
2. Définition de la structure `struct file_operations` (appelée **`fops`**) permettant d'associer les opérations à leur implémentation dans le pilote
3. Réservation du numéro de pilote (numéro majeur et numéro mineur) permettant d'identifier le pilote et le périphérique dans le noyau
4. Association des opérations sur le fichier au numéro de pilote dans le noyau Linux
5. Intégration du code du pilote de périphérique dans le squelette d'un module noyau

1. Opérations

- La structure `struct file_operations` est générique à tous les fichiers traités par le noyau Linux. Elle contient énormément d'opérations, mais il n'est pas nécessaire de toutes les implémenter pour un pilote orienté caractère.
- Le tableau ci-dessous présente cinq opérations principales et en gris-brun trois optionnelles

Opérations sur les fichiers Espace noyau	Appels système Espace utilisateur
.open	open
.release	close
.write	write
.read	read
.llseek	lseek
.mmap	mmap
.poll	select, poll, epoll
.unlocked_ioctl	ioctl
…	…

MASTER OF SCIENCE IN ENGINEERING
MSE

1. Opérations – open et release

- **La fonction `int (*open) (struct inode* i, struct file* f);` est appelée quand l'application en espace utilisateur ouvre le fichier correspondant au périphérique**
  - `i` pointe sur la structure `struct inode` qui représentent de façon unique un fichier dans le noyau Linux (que ce soit un fichier régulier, un répertoire, un lien symbolique, un pilote de périphérique orienté caractère ou bloc, etc.)
  - `f` pointe sur la structure de fichier `struct file` qui est créée à chaque fois qu'un fichier est ouvert. Il peut exister plusieurs structures de fichiers attachées au même `inode`.
    - Elle contient des informations telles que la position actuelle dans le fichier ou le mode d'ouverture
    - Elle contient également un pointeur `void * private_data` que l'on peut utiliser librement. Cet attribut est passé à toutes les autres opérations sur les fichiers
- **La fonction `int (*release) (struct inode* i, struct file* f);` est appelée quand l'application en espace utilisateur ferme le fichier**

1. Opérations – read et write

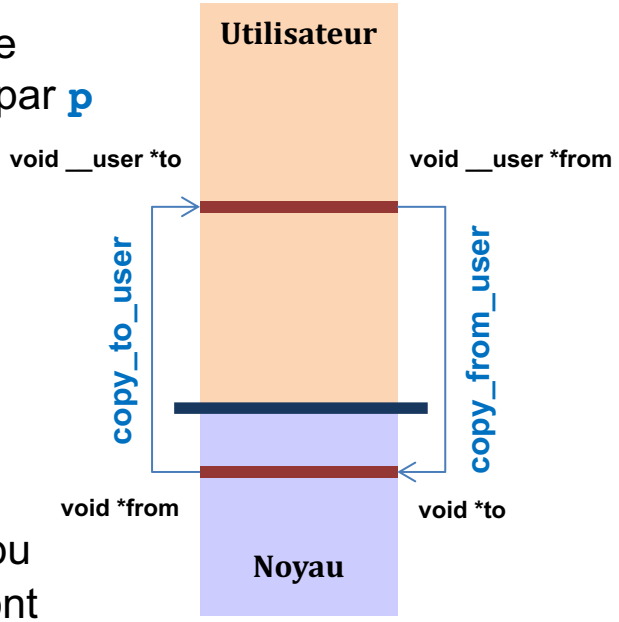
- **La fonction** `ssize_t (*read) (struct file* f, char* __user buf, size_t count, loff_t* off);` **est appelée quand l'application en espace utilisateur utilise la méthode** `read()` **sur le fichier correspondant au périphérique**
  - Elle permet de lire les données du périphérique.
  - Elle copie au maximum `count` octets du périphérique dans le tampon `buf` en espace utilisateur. Une fois l'opération terminée, elle met à jour la position `off` du fichier et retourne le nombre d'octets lus.
  - `f` est le pointeur sur la structure de fichier qui a été passé lors de l'opération `open()`

- **La fonction** `ssize_t (*write) (struct file* f, const char* __user buf, size_t count, loff_t* off);` **est appelée quand l'application en espace utilisateur utilise la méthode** `write()` **sur le fichier correspondant au périphérique**
  - Elle permet d'écrire des données dans le périphérique.
  - Elle copie `count` octets du tampon `buf` en espace utilisateur dans le périphérique. Une fois l'opération terminée, elle met à jour la position `off` du fichier et retourne le nombre d'octets copiés.
  - `f` est le pointeur sur la structure de fichier qui a été passé lors de l'opération `open()`

MASTER OF SCIENCE IN ENGINEERING
MSE

1. Opérations – Echange de données (read & write)

- **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 déréférenciation du pointeur `buf`. Pour garder le code portable sur différentes architectures, on préférera utiliser les services du noyau disponible dans l'interface `<linux/uaccess.h>`.**
- **Pour copier une seule valeur**
  - `int get_user (v, p)` permet de copier dans la variable `v` du noyau le contenu pointé par `p` en espace utilisateur.
  - `int put_user (v, p)` permet de copier le contenu de la variable `v` du noyau vers l'espace utilisateur pointé par `p`
  - Ces méthodes retournent zéro (0)en cas de succès ou -EFAULT en cas d'erreur
- **Pour copier une grande quantité de données**
  - `unsigned long copy_to_user (void __user *to, const void* from, unsigned long n);`
  - `unsigned long copy_from_user (void* to, const void __user *from, unsigned long n);`
  - Ces méthodes retournent le nombre d'octets n'ayant pu être copiés. Zéro (0) indique que toutes les données ont bien été copiées.



MASTER OF SCIENCE IN ENGINEERING
MSE

1. Opérations – mmap

- **La fonction `int (*mmap)(struct file *f, struct vm_area_struct *vma)` permet de mapper dans l'espace utilisateur une zone mémoire ou les registres d'un périphérique. Elle est appelée quand l'application en espace utilisateur utilise la méthode `mmap()`.**
  - `f` est le pointeur sur la structure de fichier qui a été passé lors de l'opération `open()`
  - `vma` est un pointeur sur la structure de la mémoire virtuelle d'un processus

- **Pour mapper en la zone souhaitée on utilisera la fonction `remap_pfn_range` disponible dans l'interface `<linux/mm.h>`**

```
int remap_pfn_range (
 struct vm_area_struct *vma,// pointeur reçu lors l'appel de mmap
 unsigned long virt_addr,   // adresse virtuelle utilisateur de départ
 unsigned long pfn,  // numéro de page de l'adresse physique de départ
 unsigned long size, // taille de la zone à mapper
 pgprot_t pprot);    // permissions sur la page
```

- **Le numéro de page de l'adresse physique de départ se calcule comme suit**

```
pfn = <PHY_ADDR> >> PAGE_SHIFT;
```

2. Définition de la structure de fichier

- La structure `struct file_operations` permettant de définir les opérations supportées par le pilote de périphérique est disponible depuis l'interface `<linux/fs.h>`. Les opérations sont déclarées comme pointeurs de fonction.

- Pour définir les opérations du pilote, il suffit de fournir seulement les méthodes qui ont été implémentées.

```
static struct file_operations skeleton_fops = {
   .owner = THIS_MODULE,
   .open = skeleton_open,
   .read = skeleton_read,
   .write = skeleton_write,
   .release = skeleton_release,
};
```

  - L'attribut `owner` doit impérativement être initialisé à l'aide de la macro `THIS_MODULE`

3. Le numéro de pilote

- **Le type `dev_t` permettant de définir le numéro de pilote est disponible dans l'interface `<linux/kdev_t.h>`.**
  - Le numéro de pilote est constitué
    - d'un numéro majeur (12 bits) et
    - d'un numéro mineur (20 bits)
  - La macro `MKDEV (int major, int minor)` permet d'initialiser la variable contenant le numéro de pilote
  - Les macros `MAJOR(dev_t dev)` et `MINOR(dev_t dev)` permettent d'extraire le numéro majeur et respectivement le numéro mineur de cette variable

- **En espace utilisateur, on peut obtenir la valeur du numéro majeur à l'aide de la commande**

```
$ cat /proc/devices
      Character devices:
        1 mem
        2 pty
        3 ttyp
        4 /dev/vc/0
        4 tty
        4 ttyS
        5 /dev/tty
        5 /dev/console
```

3. Réservation statique du numéro de pilote

- **Une fois défini, le numéro de pilote doit être enregistré dans le noyau à l'aide de la méthode**

```
int register_chrdev_region (dev_t from, unsinged count,
                            const char* name);
```

  - `from` spécifie le numéro de pilote (numéro majeur et 1er numéro mineur)
  - `count` indique le nombre de numéros mineurs consécutifs du pilote
  - `name` indique le nom du pilote de périphérique
  - La fonction retourne 0 si la réservation s'est effectuée avec succès

MASTER OF SCIENCE IN ENGINEERING

MSE

3. Réservation dynamique du numéro de pilote

- **Linux offre également le service `alloc_chrdev_region` permettant d'allouer dynamiquement les numéros de pilote**

```
int alloc_chrdev_region (dev_t* dev, unsinged baseminor,
                         unsinged count, const char* name);
```

  - `dev` paramètre de retour avec la 1er numéro assigné au pilote
  - `baseminor` spécifie le 1er numéro mineur du pilote
  - `count` indique le nombre de numéros mineurs requis par le pilote
  - `name` indique le nom du pilote de périphérique
  - La fonction retourne 0 si la réservation s'est effectuée avec succès

- **Ce service est à préférer, car il permet d'éviter tout conflit avec d'autres pilotes**

4. Enregistrement du pilote

- **Un pilote orienté caractère est représenté dans le noyau à l'aide de la structure `struct cdev` laquelle est disponible dans l'interface `<linux/cdev.h>`**
  - Cette structure doit être déclarée globalement dans le module
    ```
    static struct cdev skeleton_cdev;
    ```
  - Elle doit ensuite être initialisée à l'aide de la méthode cdev_init
    ```
    cdev_init (&skeleton_cdev, &skeleton_fops);
    ```
  - Une fois initialisé, l'attribut `owner` doit être assigné à `THIS_MODULE`

- **Le pilote doit ensuite être enregistré dans le noyau**
  ```
  int cdev_add (stuct cdev *p, dev_t dev, unsigned count);
  ```
  - `p` pointeur sur la structure du pilote
  - `dev` numéro du pilote
  - `count` indique le nombre de périphériques
  - La fonction retourne 0 si la réservation s'est effectuée avec succès

- **Après enregistrement, le noyau connaît l'association entre le numéro du pilote (numéro majeur et numéro mineur) et les opérations de fichier attachées à ce pilote. Le pilote est finalement prêt à être opéré depuis l'espace utilisateur.**

4. Libération du pilote

- **La libération d'un pilote se fait en deux étapes**
  - Elimination du pilote dans le noyau

```
void cdev_del (struct cdev* p);
```

  - Libération des numéros de pilote

```
void unregister_chrdev_region (dev_t from, unsigned count);
```

5. Fichier d'accès

- La création du fichier d'accès dans le répertoire `/dev` n'est pas automatique. Celle-ci doit être effectuée en utilisant la commande `mknod` en mode root, p.ex.

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

  - `c:` character device
  - `511:` major number
  - `0:` minor number

- Si beaucoup de périphériques doivent être créés, l'utilisation de la commande ci-dessus n'est que très peu efficace. Dès la version 2.6.13, le noyau Linux a introduit un nouvel utilitaire « udev » pour la gestion des périphériques. Sous les systèmes équipés de BusyBox, « udev » est remplacé par une version plus simple, « mdev ».

sysfs

system file system

sysfs – principe

- **sysfs (system filesystem), accessible sous `/sys`, est un système de fichiers virtuels créés pour rendre le débogage de pilotes de périphériques plus simples. Aujourd'hui, sysfs va bien au-delà et est utilisé pour représenter l'architecture et l'état d'un système dans l'espace utilisateur.**
- **sysfs permet de représenter des objets du noyau Linux, leurs attributs et leurs relations les uns envers les autres, comme suit**

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

- **Des outils très simple, tels que `ls`, `cat` ou `echo`, offrent un moyen pour accéder aux informations stockées dans le sysfs sous forme ascii, p. ex.**
  - **`cat /sys/class/tty/ttyS0/dev`**
    retourne le device number de la première interface série du NanoPi

sysfs – principe (II)

- **sysfs est construit sous forme d'arborescence**

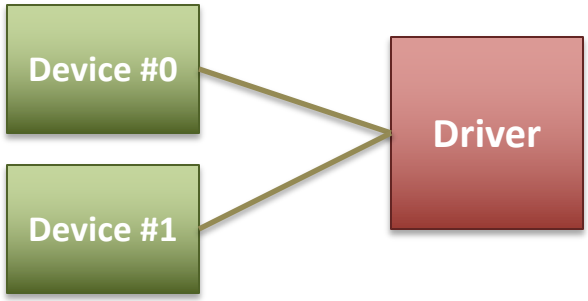
```
/sys/
   |-- block
   |-- bus
   |-- class
   |-- dev
   |-- devices
   |-- firmware
   |-- fs
   |-- kernel
   |-- module
   |-- power
```

- **Ceci permet de voir le système sous différents points de vue, p. ex.**
  - Depuis les périphériques existants dans le système **`/sys/devices`**
  - Depuis la structure du bus système **`/sys/bus`**
  - Depuis les pilotes disponibles **`/sys/module`**
  - Depuis différentes "classes" de périphériques **`/sys/class`**

- **La documentation est disponible dans les sources du noyau sous `Documentation/filesystems/sysfs.txt`**

sysfs – principe (III)

- **L'interface `<include/device.h>` fournit des services facilitant la représentation et la gestion des pilotes et de leurs périphériques dans le `sysfs`**



  - La structure `struct device_driver` et les fonctions **`driver_register`** et **`driver_unregister`** permettent de créer et d'instancier un pilote de périphérique dans le **`sysfs`**. Le pilote et ses attributs seront ainsi visible sous `/sys/module`.
  - La structure `struct device` et les fonctions **`device_register`** et **`device_unregister`** permettent de créer et d'instancier un périphérique dans le **`sysfs`**. Le périphérique et ses attributs seront ainsi visible sous `/sys/devices`.

- **Cependant, avant de commencer le développement d'un pilote, il est important de savoir dans quel bus le pilote doit être inséré (i2c, pci, usb, …) et, le cas échéant, utiliser les structures et services spécifiques fournis par ces bus.**

sysfs – attributs du pilote (driver)

- **La structure `struct driver_attribute` permet de spécifier des méthodes d'accès (lecture et/ou écriture) pour la échange d'information avec le pilote d'un périphérique.**

```
struct driver_attribute {
  struct attribute attr;
  ssize_t (*show)  (struct device_driver *drv, char *buf);
  ssize_t (*store) (struct device_driver *drv,
                    const char *buf, size_t count);
};
```

  - Il est impératif de créer/instancier, pour chaque attribut du pilote, une telle structure avec des méthodes d'accès propre.
  - Les valeurs de l'attribut sont passées sous forme `ascii` par l'intermédiaire de l'argument `buf`.

- **La macro `DRIVER_ATTR` permet d'instancier très simplement cette structure**

```
DRIVER_ATTR (name, mode, show, store);
```

sysfs – installation des méthodes d'accès (driver)

- **Pour installer les méthodes d'accès d'un attribut du pilote dans `sysfs`, on utilisera la fonction**

```
int driver_create_file (struct device_driver *drv,
                        struct driver_attribute *attr);
```

- **Pour éliminer une entrée dans `sysfs`, on utilisera la méthode**

```
void driver_remove_file (struct device_driver *drv,
                         struct driver_attribute *attr);
```

MASTER OF SCIENCE IN ENGINEERING

MSE

sysfs – attributs d'un périphérique (device)

- **La structure `struct device_attribute` permet de spécifier des méthodes d'accès (lecture et/ou écriture) pour l'échange d'information avec le périphérique.**

```
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);
};
```

  - Il est impératif de créer/instancier, pour chaque attribut d'un périphérique, une telle structure avec des méthodes d'accès propre.
  - Les valeurs de l'attribut sont passées sous forme `ascii` par l'intermédiaire de l'argument `buf`.

- **La macro `DEVICE_ATTR` permet d'instancier très simplement cette structure**

```
DEVICE_ATTR (name, mode, show, store);
```

sysfs – installation des méthodes d'accès (device)

- **Pour installer les méthodes d'accès d'un attribut du périphérique dans `sysfs`, on utilisera la fonction**

```
int device_create_file (struct device * dev,
                        struct device_attribute * attr);
```

- **Pour éliminer une entrée dans `sysfs`, on utilisera la méthode**

```
void device_remove_file (struct device * dev,
                         struct device_attribute * attr);
```

MASTER OF SCIENCE IN ENGINEERING

MSE

sysfs – création d'un device sous le répertoire class

- **Les attributs d'un pilote (driver) ou d'un périphérique (device) peuvent être accessible sous différents répertoires de l'arborscence sysfs. La bibliothèque "`platform_device`" permet de créer assez facilement une interface pour y accéder.**
- **Les méthodes ci-dessous sa propre "`class`" à laquelle on pourra ensuite attacher les fichiers d'accès aux attributs d'un device.**
  - Création d'une nouvelle "class"

```
struct class * class_create (struct module * owner, // THIS_MODULE
                             const char * name);
```

  - Destruction de la "class"

```
void class_destroy (struct class * cls);
```

  - Création d'un "device"

```
struct device * device_create (struct class * class,
                               struct device * parent,
                               dev_t devt,
                               const char * fmt, ...);
```

  - Destruction du "device"

```
void device_destroy (struct class * class, dev_t devt);
```

MASTER OF SCIENCE IN ENGINEERING

MSE

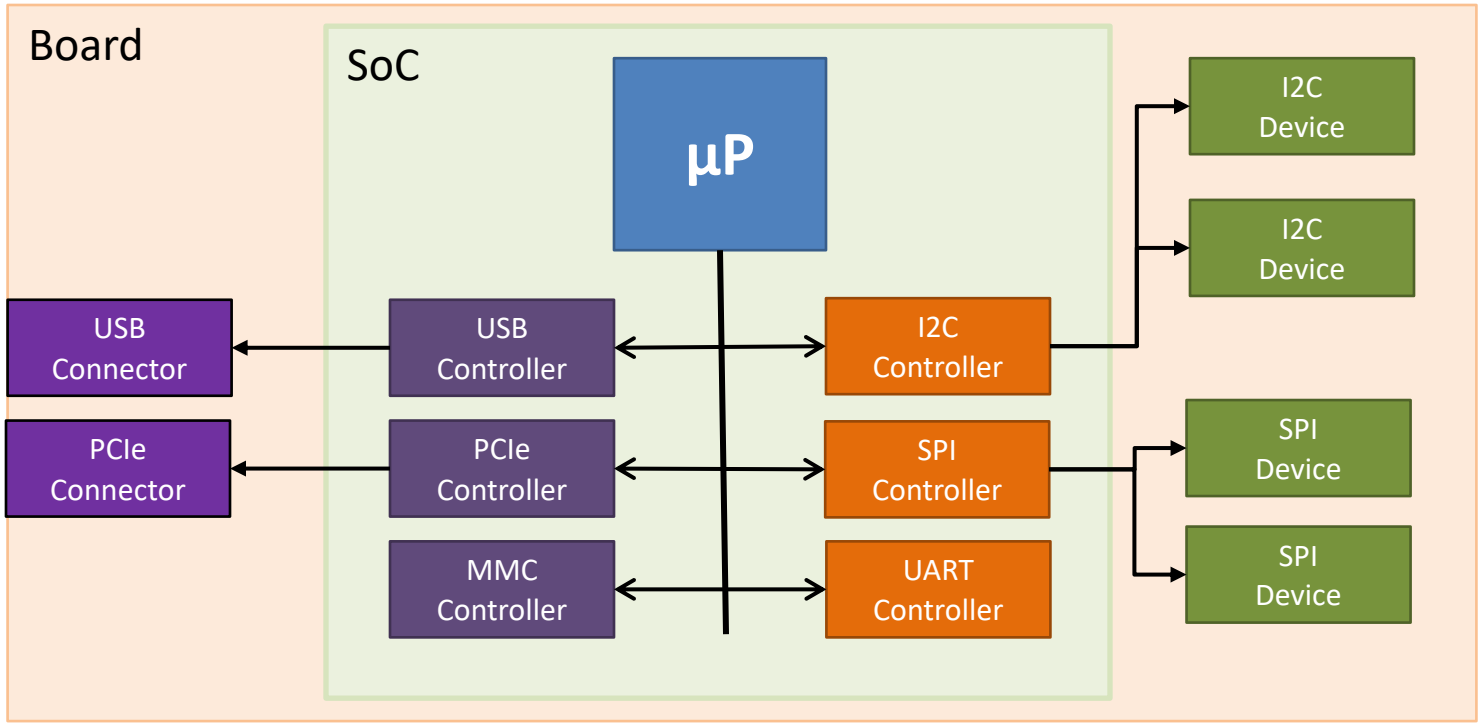
Device Tree

MASTER OF SCIENCE IN ENGINEERING

MSE

Device Tree – Introduction

- **Sur les systèmes à µP, certains bus permettent de découvrir dynamiquement les périphériques qui y sont connectés, p.ex. le PCIe ou l'USB, ce qui permet d'éviter de connaître à l'avance leur présence et leurs caractéristiques.**



MASTER OF SCIENCE IN ENGINEERING

Device Tree – Introduction (II)

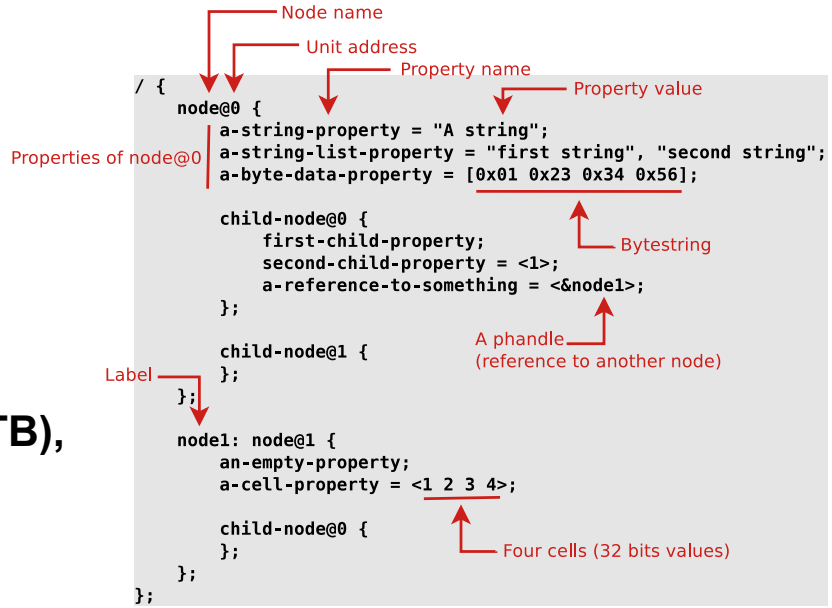
- Cependant, beaucoup d'autres bus du µP, tels que I2C ou SPI, n'ont pas cette capacité. Il est donc primordiale que le système, le noyau Linux, soit informé de la description du matériel à l'avance.

- Jusqu'au début des années 2010, le noyau Linux contenait dans son code une description du matériel disponible sur le système.

- Dès 2011, d'abord pour les µPs PowerPC, le noyau Linux ne contient plus de description du matériel. Celle-ci lui est fournie par un fichier binaire, nommé le « Device Tree Blob » (DTB), lequel est passé comme argument au noyau Linux lors de son lancement. Ce fichier contient toutes les informations sur le matériel disponible sur le système. Une fois le noyau Linux lancé, on peut obtenir cette description des périphériques sous `/proc/device-tree` ou sous `/sys/firmware/devicetree/base`.

Device Tree – Exemple de la syntaxe du DT

- Le «Device Tree» (DT) est un arbre de nœuds modélisant la hiérarchie des périphériques d'un système allant des périphériques internes au processeur aux périphériques de la carte.
- Chaque nœud contient un certain nombre de propriétés décrivant le périphérique, p.ex. les adresses, les interruptions, les horloges, etc.
- Lors du lancement, une version compilée, le «Device Tree Blob» (DTB), est passée au noyau Linux pour instancier toutes les descriptions des périphériques sous **/proc/device-tree**




```
ths: thermal-sensor@1c25000 {
        compatible = "allwinner,sun50i-h5-ths";
        reg = <0x01c25000 0x400>;
        interrupts = <GIC_SPI 31 IRQ_TYPE_LEVEL_HIGH>;
        resets = <&ccu RST_BUS_THS>;
        clocks = <&ccu CLK_BUS_THS>, <&ccu CLK_THS>;
        clock-names = "bus", "mod";
        nvmem-cells = <&ths_calibration>;
        nvmem-cell-names = "calibration";
        #thermal-sensor-cells = <1>;
};
```

Device Tree – Modification et/ou extension du DT

- **Pour les µP ARMv7 et ARMv8, les descriptions des différentes cartes supportées par le noyau Linux sont stockées dans l'arborescence des sources du noyau sous `arch/arm/boot/dts` et `arch/arm64/boot/dts`**

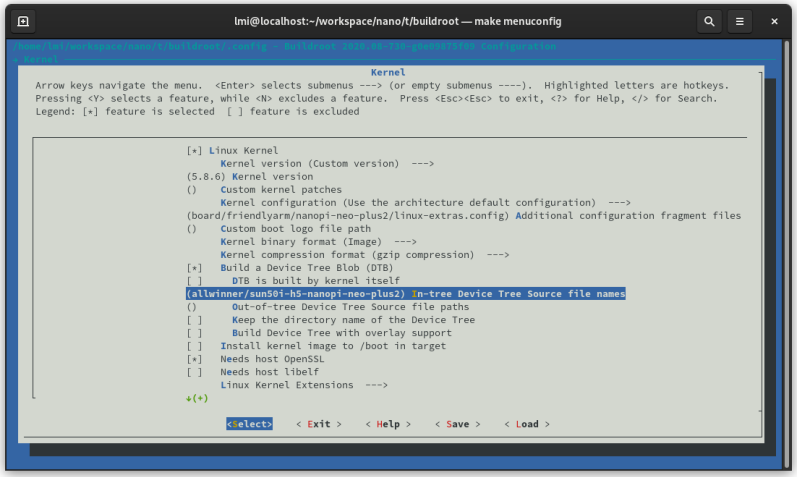
- **Si le système nécessite des modifications ou des extensions par rapport au DT «standard» fourni par le noyau Linux, plusieurs solutions sont à disposition du développeur:**
  - Un nouveau DT peut être ajouté dans l'arborescence du noyau Linux
  - Un patch peut être appliqué au DT « standard »
  - Un nouveau DT hors de l'arborescence du noyau peut être créé

- **Si l'on fait le choix d'un nouveau DT, alors il est possible de**
  - Ecrire un DT complètement neuf
  - Ecrire un nouveau DT en utilisant les descriptions mises à disposition
  - Inclure le DT « standard » et apporter les modifications/extensions nécessaires

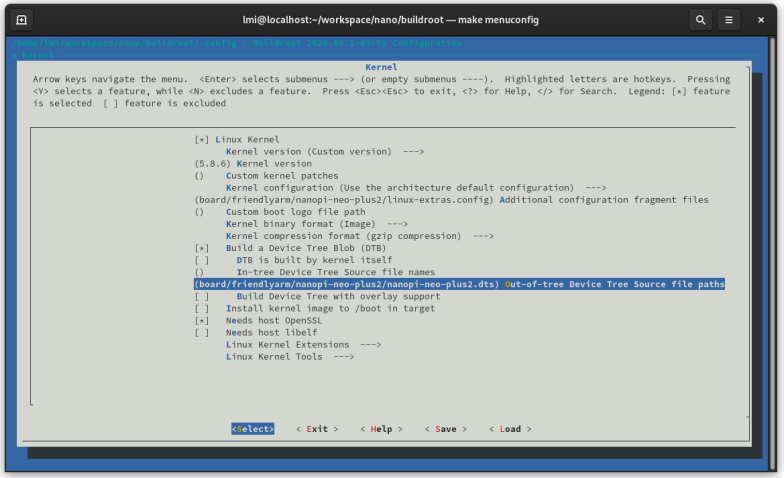
MASTER OF SCIENCE IN ENGINEERING
MSE

Device Tree – Génération du DTB

- **Selon le choix effectué pour la création du DT pour le système, différentes variantes sont à disposition pour la génération du DTB**
  - Si le DT est placé dans l'arborescence du noyau, il suffit de choisir le nouveau DT lors de la configuration du noyau. Ce choix se laisse facilement configurer avec Buildroot.
  - Si le DT est placé dans l'arborescence de Buildroot, p.ex. sous «board/<name>», il suffit de configurer Buildroot afin qu'il génère le nouveau DTB.
  - Si le DT est placé en dehors des 2 arborescences précédentes, il faudra alors développer son propre Makefile.



Arborescence du noyau



Arborescence de Buildroot

Device Tree – String de compatibilité

- **Le lien entre le pilote traitant un périphérique dans le noyau Linux et la description dans le « Device Tree » est garanti par le string de compatibilité, mot clef « `.compatible` »**

```
static const struct of_device_id of_ths_match[] = {
  { .compatible = "allwinner,sun50i-a64-ths", .data = &sun50i_a64_ths },
  { .compatible = "allwinner,sun50i-h5-ths",  .data = &sun50i_h5_ths },
  { .compatible = "allwinner,sun50i-h6-ths",  .data = &sun50i_h6_ths },
  { /* sentinel */ },
};
MODULE_DEVICE_TABLE(of, of_ths_match);
```

- **L'attribut « `.of_match_table` » de la structure « `struct device_driver` » contient la liste des strings compatibles avec le pilote.**

```
static struct platform_driver ths_driver = {
  .probe = sun8i_ths_probe,
  .remove = sun8i_ths_remove,
  .driver = {
    .name = "sun8i-thermal",
    .of_match_table = of_ths_match,
  },
};
```

- **Cette compatibilité est contrôlée lors de l'instanciation du pilote.**

Device Tree – Références

[1] **Device Tree for Dummies,**
https://bootlin.com/pub/conferences/2013/elce/petazzoni-device-tree-dummies/petazzoni-device-tree-dummies.pdf

[2] **Power.orgTM Standard for Embedded Power Architecture Platform Requirements (ePAPR),**
http://www.power.org/resources/downloads/Power_ePAPR_APPROVED_v1.0.pdf

[3] **DeviceTree.org website,** http://www.devicetree.org

[4] **Device Tree documentation in the kernel sources,**
Documentation/devicetree

[5] **The Device Tree kernel mailing list,**
http://dir.gmane.org/gmane.linux.drivers.devicetree

[6] **Device Tree Reference,** https://elinux.org/Device_Tree_Reference

[7] **Device Tree Usage,** https://elinux.org/Device_Tree_Usage

Plateforme

MASTER OF SCIENCE IN ENGINEERING

MSE

Introduction

- Pour les périphériques, le noyau Linux a développé un bus plateforme, lequel supporte des pilotes pour la gestion de ces périphériques ne faisant pas partie d'un type de bus avec une détection dynamique.
- L'interface `<linux/platform_device.h>` offre des méthodes/services pour créer simplement des fichiers d'accès aux attributs des pilotes et périphériques

MASTER OF SCIENCE IN ENGINEERING

MSE

Plateforme – les pilotes (driver)

- **La structure `struct platform_driver` permet de spécifier les méthodes et services d'un pilote plateforme**

```
struct platform_driver {
    int  (*probe)    (struct platform_device *);
    int  (*remove)   (struct platform_device *);
    void (*shutdown) (struct platform_device *);
    int  (*suspend)  (struct platform_device *, pm_message_t);
    int  (*resume)   (struct platform_device *);
    struct device_driver driver;
    // ...
};
```

  - probe méthode pour l'instanciation du périphérique. Cette méthode n'est appelée que si lors de l'enregistrement du pilote, celui-ci trouve un string de compatibilité dans le « Device Tree »
  - remove méthode pour la destruction du périphérique. Cette méthode est appelée lors du dé-enregistrement du pilote
  - shutdown méthode pour la destruction du périphérique. Cette méthode est appelée lorsque le système Linux est éteint (shutdown)

Plateforme – les pilotes (II)

- suspend et resume sont des méthodes appelées lors de la mise en sommeil et réveil du pilote.
- driver attribut décrivant le pilote
  - *name attribut pour spécifier le nom du pilote, lequel sera visible sous /sys*
  - *of_match_table pointe sur la table des strings de compatibilité*

- **Les deux méthodes ci-dessous permettent d'enregistrer/de libérer le pilote.**

```
int  platform_driver_register   (struct platform_driver *);
void platform_driver_unregister (struct platform_driver *);
```

Ces méthodes doivent simplement être appelées dans les méthodes `init` et `exit` du module.

Plateforme – les périphériques (device)

- **Dans le cas d'un périphérique appartenant à la plateforme, on peut utiliser les méthodes de l'interface `<linux/platform_device.h>`**

```
struct platform_device {
   const char* name;
   int id;
   struct device dev;  //→ a release method should be attached
   u32 num_resources;
   struct resource *resource;
};
```

  - Cette structure permet de spécifier le nom du périphérique, le numéro d'instance (**`id`**, -1 s'il n'existe qu'une instance), ainsi que les ressources utilisées par le périphérique.

  - Si le périphérique implémente les services d'un « char device », le numéro de périphérique obtenu lors de l'instanciation du « cdev » devra être assigné à l'attribut « .dev.devt ». Cette opération permettra au noyau Linux de créer le fichier d'accès dans /dev

- **Les deux méthodes ci-dessous d'enregistrer/de libérer le périphérique.**

```
int  platform_device_register   (struct platform_device *);
void platform_device_unregister (struct platform_device *);
```

Plateforme – miscdevice

- **La structure** struct **miscdevice** **simplifie l'instanciation du périphérique avec la création d'un fichier d'accès sous /dev.**

```
struct miscdevice {
   int minor;
   const char *name;
   const struct file_operations *fops;
   // ...
   struct device *this_device;
   umode_t mode;
};
```

  - minor minor device number du périphérique, utiliser MISC_DYNAMIC_MINOR pour obtenir dynamiquement un numéro mineur
  - name nom du périphérique, ce nom sera également le nom du fichier d'accès
  - fops pointeur sur la structure contenant les opérations sur le fichier d'accès (struct **file_operations**)
  - mode définit les droits sur fichier d'accès sous /dev

- **Les deux méthodes ci-dessous d'enregistrer/de libérer le périphérique.**

```
int  misc_register   (struct miscdevice *);
void misc_unregister (struct miscdevice *);
```

ioctl

input/output control

ioctl – principe

- ioctl (Input/Output Control) a été introduit dans les systèmes Unix vers la fin des années 1970. Elle est supportée par la plupart des systèmes Unix, dont Linux et Mac OS X. Windows fournit une interface similaire connue sous le nom de DeviceIoControl.

- ioctl fournit au niveau du mode utilisateur la méthode `ioctl` avec le prototype suivant:

```
int ioctl (int fd, unsigned long cmd, ...);

fd  = descripteur de fichier obtenu lors de son ouverture
cmd = commande/opération devant être exécutée
... = paramètres optionnels

return = valeur de retour si positif, sinon erreur
         -EINVAL (invalid argument)
```

ioctl – principe (II)

- **La fonction `long (*unlocked_ioctl) (struct file *f, unsigned int cmd, unsigned long arg)` est appelée quand l'application en espace utilisateur utilise la méthode `ioctl()` sur le fichier correspondant au pilote**
  - Elle permet d'échanger les données de configuration/états avec le pilote du périphérique sans bloquer le noyau
  - **`f`** est le pointeur sur la structure de fichier qui a été passé lors de l'opération **`open()`**
  - **`cmd`** correspond au mot/numéro de commande passé par l'application
  - **`arg`** est le paramètre optionnel de la commande ioctl() en espace utilisateur. Dans le cas où celui-ci n'est pas spécifié par l'application lors de l'appel, son contenu est indéterminé.

ioctl – mot de commande

- **Les mots de commande `cmd` sont des nombres uniques, codés sur 32 bits, permettant d'identifier les opérations que le driver devra exécuter.**

- **Le mot de commande a la structure suivante:**
  - **`type`**: nombre magique unique (magic number) codé sur 8 bits, lequel doit est défini après consultation de la liste « Documentation/ioctl-number.txt » fournie dans la documentation de Linux
  - **`number`**: numéro de la commande/opération codé sur 8 bits.
  - **`direction`**: défini le type d'opération devant être exécutée
    - **`_IOC_NONE`**: pour une commande
    - **`_IOC_READ`**: pour une lecture de données du pilote vers l'application
    - **`_IOC_WRITE`**: pour une écriture de données de l'application vers le pilote
    - **`_IOC_READ|_IOC_WRITE`**: pour une écriture et lecture
  - **`size`**: taille des données de l'utilisateur impliquées dans l'opération (13 ou 14 bits)

MASTER OF SCIENCE IN ENGINEERING
MSE

ioctl – mot de commande (II)

- **L'interface `<linux/ioctl.h>` fournit une série de macros facilitant la définition des numéros de commande**
  - `__IO (type, nr)` pour une commande
  - `__IOR (type, nr, datatype)` pour une opération de lecture
  - `__IOW (type, nr, datatype)` pour une opération d'écriture
  - `__IOWR (type, nr, datatype)` pour une opération d'écriture et lecture
- **Pour décoder une commande, l'interface fournit des macros**
  - `__IOC_DIR (cmd)` pour la direction
  - `__IOC_TYPE (cmd)` pour le type (magic number)
  - `__IOC_NR (cmd)` pour le numéro de la commande/opération
  - `__IOC_SIZE (cmd)` pour la taille des données
- **Exemples**

```
struct SKELETON_RW { char str[100]; };
#define SKELETON_IOMAGIC 'g'
#define SKELETON_IO_RESET    _IO  (SKELETON_IOMAGIC, 0)
#define SKELETON_IO_WR_REF   _IOW (SKELETON_IOMAGIC, 1, struct RW)
#define SKELETON_IO_RD_REF   _IOR (SKELETON_IOMAGIC, 2, struct RW)
#define SKELETON_IO_WR_VAL   _IOW (SKELETON_IOMAGIC, 3, int)
```

ioctl – paramètre optionnel

- **Le paramètre optionnel `arg` permet à l'application d'échanger, selon les architectures, jusqu'à 16K octets de données avec le pilote.**
- **Le paramètre est passé au pilote sous la forme d'un `unsigned long`. Celui-ci peut aussi bien représenter une valeur entière qu'un pointeur.**
- **Si les données sont passées par référence (pointeur), celles-ci doivent être copiées à l'aide des méthodes définies dans l'interface `<linux/uaccess.h>`**
  - `copy_from_user`
  - `copy_to_user`
- **Il existe également d'autres méthodes pour effectuer le transfert:**
  - `acces_ok`: pour vérifier la validité de l'adresse (impératif)
  - `put_user`: pour transférer des données du pilote vers l'utilisateur
  - `get_user`: pour transférer des données de l'utilisateur vers le pilote

procfs

process file system

MASTER OF SCIENCE IN ENGINEERING

MSE

procfs – principe

- **procfs (process file system), accessible sous le répertoire `/proc`, est un pseudo-système de fichiers, créé sous Linux pour accéder, au moyen d'utilitaires très simple, tels que `ls, cat ou echo`, aux informations du noyau**
  sur les processus, p. ex.

  - **`cat /proc/devices`**
    retourne la liste des pilotes installés, le type et le major number

  - **`cat /proc/modules`**
    retourne la liste des modules installés dans le noyau

- **Les pilotes désirant échanger des données avec des applications, le font simplement en exportant des fichiers virtuels sous `/proc.` Ceux-ci permettent l'échange de ces informations sous forme `ascii`.**

- **Cependant aujourd'hui, il est plutôt recommandé d'implémenter cette fonctionnalité sous `sysfs` que sous `procfs`.**

procfs – méthodes d'accès et leur installation

- Au niveau du noyau, l'interface `<linux/proc_fs.h>` offre une série de méthodes permettant au pilote d'instancier des méthodes de lecture et d'écriture.
- Depuis la version 3.10 du noyau, procfs utilise les `fops` employés par les méthodes d'accès aux pilotes de périphériques devront être utilisées.
- Pour installer les méthodes d'accès dans `procfs`, on utilisera la fonction

```
struct proc_dir_entry *proc_create (const char *name, mode_t mode,
        struct proc_dir_entry *parent, struct proc_ops *fops);
```

- Si l'on désire créer des sous-répertoires, on utilisera la méthode

```
struct proc_dir_entry *proc_mkdir (const char *name,
      struct proc_dir_entry *parent);
```

- Pour éliminer une entrée dans `procfs`, on utilisera la méthode

```
void remove_proc_entry (const char *name,
                        struct proc_dir_entry *parent);
```

MASTER OF SCIENCE IN ENGINEERING
MSE

Opérations bloquantes

Opérations d'entrées/sorties bloquantes

- Quand une opération de lecture ou d'écriture doit attendre sur des données ou sur la disponibilité du périphérique, le pilote doit bloquer le thread jusqu'à ce que les données soient disponibles. Ceci peut être réalisé en le mettant en sommeil jusqu'à ce que la requête puisse être satisfaite à l'aide de waitqueues.

- Déclaration de la waitqueue (`<linux/wait.h>, <linux/sched.h>`) et d'un fanion de signalisation

```
wait_queue_head_t my_queue;
int request_can_be_processed  = 0;
```

- Initialisation de la waitqueue

```
init_waitqueue_head (&my_queue);
```

- Attendre jusqu'à ce que l'opération puisse être satisfaite (read/write)

```
wait_event_interruptible
  (&my_queue, request_can_be_processed != 0);
```

MASTER OF SCIENCE IN ENGINEERING

MSE

Opérations d'entrées/sorties bloquantes (II)

- Lorsque les ressources sont disponibles, le pilote peut réveiller le thread et lui signaler que la requête peut finalement être traitée

```
request_can_be_processed  = 1;
wake_up_interruptible(&my_queue);
```

- L'application en espace utilisateur sera notifiée s'il utilise un des appels système pour la scrutation (`select, poll` ou `epoll`). Ces opérations utilisent l'opération `poll` du pilote de périphérique dont les services sont disponibles dans l'interface `<linux/poll.h>`.

- Cette méthode permet d'attendre sur les ressources en mode non bloquant et peut être pour de opérations de lecture comme d'écriture

```
static unsigned int skeleton_poll (struct file *f,
                                   poll_table *wait) {
    unsigned mask = 0;
    poll_wait (f, &my_queue, wait);
    if (request_can_be_processed  != 0)
      mask |= POLLIN | POLLRDNORM;   // read operation
  //  mask |= POLLOUT | POLLWRNORM;  // write operation
    return mask;
}
```

ASPECTS PRATIQUES

Pilotes orientés caractère

MASTER OF SCIENCE IN ENGINEERING

MSE

Exemple – open et release

```c
static int skeleton_open (struct inode *i, struct file *f)
{
    pr_info ("skeleton : open operation... major:%d, minor:%d\n",
        imajor (i), iminor(i));

    if ((f->f_mode & (FMODE_READ | FMODE_WRITE)) != 0) {
        pr_info ("skeleton : opened for reading & writing...\n");
    } else if ((f->f_mode & FMODE_READ) != 0) {
        pr_info ("skeleton : opened for reading...\n");
    } else if ((f->f_mode & FMODE_WRITE) != 0) {
        pr_info ("skeleton : opened for writing...\n");
    }

    return 0;
}

static int skeleton_release (struct inode *i, struct file *f)
{
    pr_info ("skeleton: release operation...\n");
    return 0;
}
```

Exemple – read

```
#define BUFFER_SZ 10000
static char s_buffer[BUFFER_SZ];

static ssize_t skeleton_read(struct file* f,
                             char __user* buf,
                             size_t count,
                             loff_t* off)
{
    // compute remaining bytes to copy, update count and pointers
    ssize_t remaining = BUFFER_SZ - (ssize_t)(*off);
    char* ptr         = s_buffer + *off;
    if (count > remaining) count = remaining;
    *off += count;

       // copy required number of bytes
    if (copy_to_user(buf, ptr, count) != 0) count = -EFAULT;

    pr_info("skeleton: read operation... read=%ld\n", count);

    return count;
}
```

Exemple – write

```
static ssize_t skeleton_write(struct file* f,
                              const char __user* buf,
                              size_t count,
                              loff_t* off)
{
    // compute remaining space in buffer and update pointers
    ssize_t remaining = BUFFER_SZ - (ssize_t)(*off);

    // check if still remaining space to store additional bytes
    if (count >= remaining) count = -EIO;

    // store additional bytes into internal buffer
    if (count > 0) {
        char* ptr = s_buffer + *off;
        *off += count;
        ptr[count] = 0;  // make sure string is null terminated
        if (copy_from_user(ptr, buf, count)) count = -EFAULT;
    }

    pr_info("skeleton: write operation... written=%ld\n", count);
    return count;
}
```

Exemple – enregistrement et libération

```
static dev_t skeleton_dev;
static struct cdev skeleton_cdev;

static int __init skeleton_init(void)
{
    int status = alloc_chrdev_region (&skeleton_dev, 0, 1, "mymodule");
    if (status == 0) {
        cdev_init (&skeleton_cdev, &skeleton_fops);
        skeleton_cdev.owner = THIS_MODULE;
        status = cdev_add (&skeleton_cdev, skeleton_dev, 1);
    }
    pr_info ("Linux module skeleton loaded\n");
    return 0;
}

static void __exit skeleton_exit(void)
{
    cdev_del (&skeleton_cdev);
    unregister_chrdev_region (skeleton_dev, 1);
    pr_info ("Linux module skeleton unloaded\n");
}
```

ASPECTS PRATIQUES

Device Tree

MASTER OF SCIENCE IN ENGINEERING

MSE

Device Tree – Génération de son propre DTB hors arborescence

- **Définition d'une extension du Device Tree**

```
/dts-v1/;

#include "allwinner/sun50i-h5-nanopi-neo-plus2.dts"

/ {
        /delete-node/ leds;

        mydevice {
          compatible = "mydevice";
          attribute = "on";
        };
};
```

- **Génération du DTB (extension du Makefile)**

```
DTB = mydt.dtb
DTS = $(DTB:.dtb=.dts)
INCL+=-I. -I$(KDIR)/include -I$(KDIR)/arch/arm64/boot/dts

dtb: $(DTB)

$(DTB) : $(DTS)
    ln -s $(KDIR)/arch/arm/boot/dts arm
    -cpp $(INCL) -E -P -x assembler-with-cpp $(DTS) | dtc -I dts -O dtb -o $(DTB) -
    rm arm
```

MASTER OF SCIENCE IN ENGINEERING

MSE

Device Tree – Lecture des attributs

```c
int skeleton_drv_probe(struct platform_device * pdev)
{
    struct device_node *dt_node = pdev->dev.of_node;

    if (dt_node) {
        int ret = 0;
        const char *prop_str = 0;

        ret = of_property_read_string(dt_node, "attribute", &prop_str);
        if (prop_str && ret == 0)
            pr_info("attribute=%s (ret=%d)\n", prop_str, ret);
    }

    // ....

    return 0;
}
```

Device Tree – Exemple avec des sous-nœuds

```
/dts-v1/;

    #include "allwinner/sun50i-h5-nanopi-neo-plus2.dts"

/ {
     /delete-node/ leds;

     mydevice {
          compatible = "mydevice";
          #address-cells = <1>;
          #size-cells = <0>;

          attribute = "idle";

          mydevice@0 {
               reg = <0x0>;
               attribute = "on";
          };

          mydevice@1 {
               reg = <0x1>;
               attribute = "off";
          };
     };
};
```

MASTER OF SCIENCE IN ENGINEERING

MSE

Device Tree – Exemple avec des sous-nœuds – Lecture

```
int drv_probe(struct platform_device* pdev)
{
    struct device_node* dt_node = pdev->dev.of_node;

        if (dt_node) {
         const unsigned int* prop_reg = 0;
         struct device_node* child    = 0;

         for_each_available_child_of_node(dt_node, child)
         {
             pr_info("child found: name=%s, fullname=%s\n",
                     child->name,
                     child->full_name);
             prop_reg = of_get_property(child, "reg", NULL);
             if (prop_reg != 0) {
                 unsigned long reg = of_read_ulong(prop_reg, 1);
                 pr_info("reg:%lu\n", reg);
             }
         }
    }

    // ....

     return 0;
}
```

ASPECTS PRATIQUES

sysfs

MASTER OF SCIENCE IN ENGINEERING

MSE

sysfs – exemple : méthodes d'accès aux attributs du périphérique

```c
#include <linux/device.h>          /* needed for sysfs handling */
#include <linux/platform_device.h> /* needed for sysfs handling */

static char sysfs_buf[1000];
ssize_t sysfs_show_attr(struct device* dev, struct device_attribute* attr,
                        char* buf)
{
    strcpy(buf, sysfs_buf);
    return strlen(buf);
}
ssize_t sysfs_store_attr(struct device* dev, struct device_attribute* attr,
                         const char* buf, size_t count)
{
    int len = sizeof(sysfs_buf) - 1;
    if (len > count) len = count;
    strncpy(sysfs_buf, buf, len);
    sysfs_buf[len] = 0;
    return len;
}
DEVICE_ATTR(attr, 0664, sysfs_show_attr, sysfs_store_attr);
```

sysfs – exemple : définition des structures et installation

```
static void sysfs_dev_release(struct device* dev) {}

static struct platform_device sysfs_device = {
    .name         = "mymodule",
    .id           = -1,
    .dev.release = sysfs_dev_release,
};

static int __init skeleton_init(void)
{
    int status = 0;
    if (status == 0)
        status = platform_device_register(&sysfs_device);
    if (status == 0)
        status = device_create_file(&sysfs_device.dev, &dev_attr_attr);
    return status;
}

static void __exit skeleton_exit(void)
{
    device_remove_file(&sysfs_device.dev, &dev_attr_attr);
    platform_device_unregister(&sysfs_device);
}
```

MASTER OF SCIENCE IN ENGINEERING

MSE

ASPECTS PRATIQUES

procfs

MASTER OF SCIENCE IN ENGINEERING

MSE

procfs – exemple : installation des opérations

```
static struct file_operations fops_value = {
    .read  = skeleton_read_value,
    .write = skeleton_write_value,
};
static struct proc_dir_entry* procfs_dir = 0;

static int __init skeleton_init(void)
{
   int status = 0;

   procfs_dir = proc_mkdir ("mymodule", NULL);
   proc_create ("value", 0, procfs_dir, &fops_value);

   if (procfs_dir == 0) status = -EFAULT;

   return status;
}

static void __exit skeleton_exit(void)
{
    remove_proc_entry ("value", procfs_dir);
    remove_proc_entry ("mymodule", NULL);
}
```