Développer son propre DRIVER ... sur PEGASE

Arthur Bouché[1,2] Vincent Le Cam[1,2] Laurent Lemarchand[1]

January 31, 2024

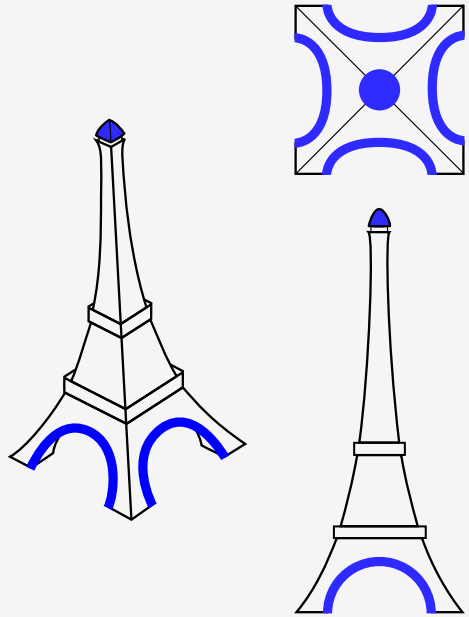
[1] Université Gustave Eiffel / COSYS / SII

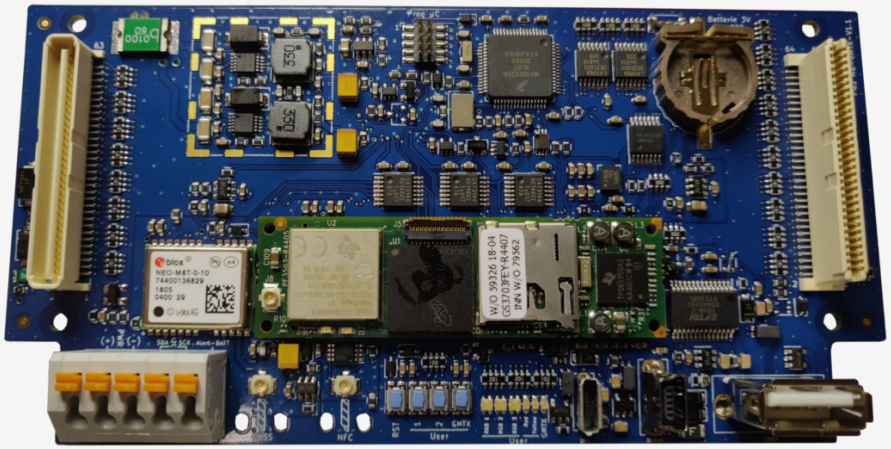
[2] INRIA Rennes / I4S

Outline

1. Théorie

2. Pratique sur PEGASE





Théorie

Contents

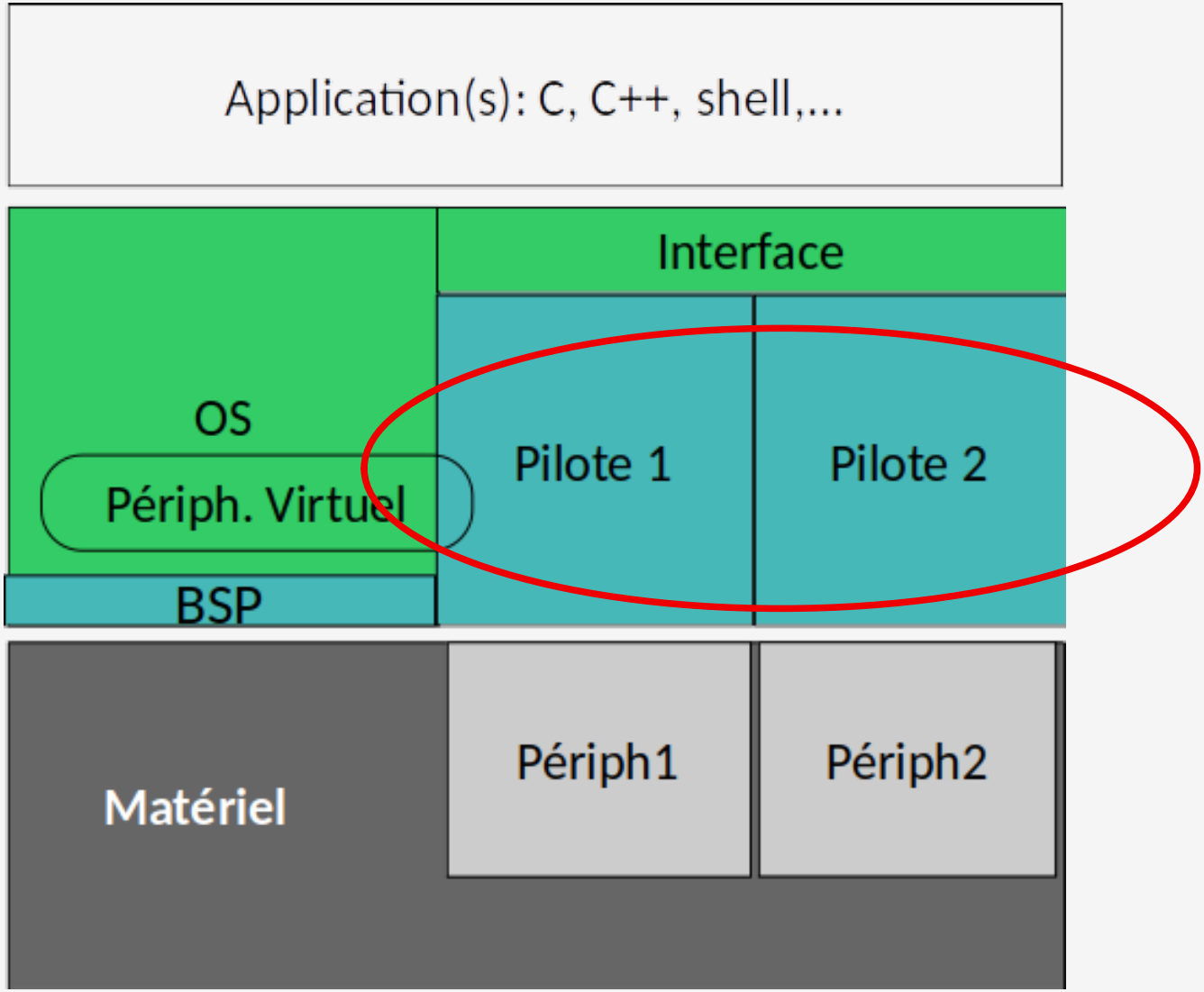
Théorie

Définition

Question : Pourquoi des drivers pour bien gérer les ports physiques ????
Réponse : Essayer sans !

- **Définition** : Un pilote de périphérique (i.e. driver) est responsable de la **gestion du ou des périphériques associés**. Par définition, il connaît leur fonctionnement et doit permettre aux autres composants logiciels (applications) d'exploiter le périphérique sans se préoccuper de détails liés à la mise en œuvre.
- **Frontière** entre applications et matériel
- **Encapsulation** du périphérique (abstraction)
- **Protection** des accès physiques
- **Interface** standard

Vision en couches



Théorie

Rôle du driver

Gestion du périphérique, mais aussi :

- Uniformisation des interfaces
  - Les E/S d'un driver sont standards
- Permet une indépendance des aspects matériels
  - Portage des codes métier haut niveau rapide
  - Portage des codes liés aux périphériques (driver) plus rapide (que sans driver)
- Protection du système
  - Confidentialité (codes sources fournis ou non)
  - Intégrité
  - Inocuïté
  - Accès concurrents

Théorie

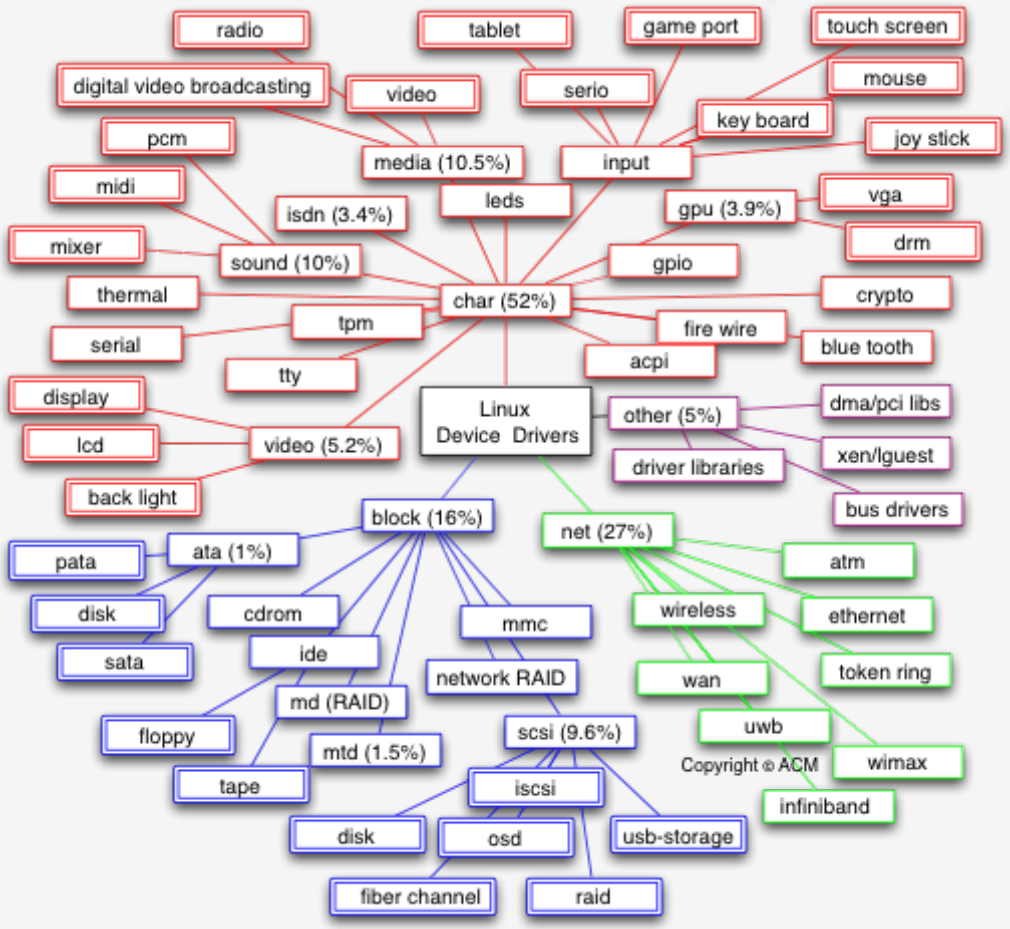
Classes de driver

Université Gustave Eiffel

Classe = interface

Classification répandue :

- périphériques orientés caractère
- périphériques orientés bloc
- périphériques orientés réseau



Théorie

Vu de l'application

Université Gustave Eiffel

Périphérique généralement vu comme un fichier

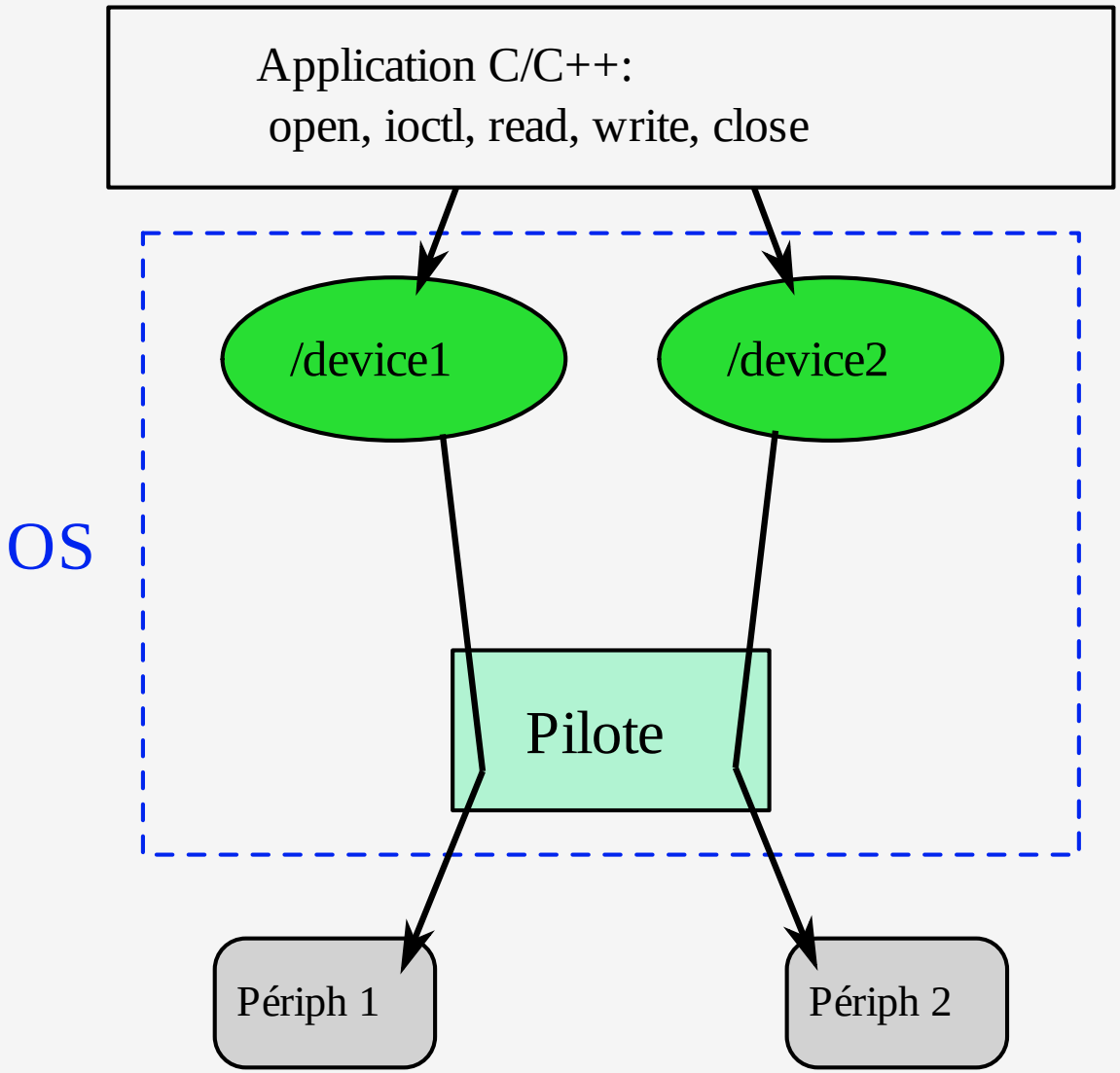
- "En Linux tout est fichier..." : fichiers classiques, répertoire, périphérique d'E/S, port TCP/IP (socket), pipe...
- "Fichier sous Linux" → "file descriptor" → `int fd = open("/dev/ttyS0"...);`

Fonctions d'accès aux fichiers possibles sur les périphériques (open, read, write, close)

- `int fd = open ("/dev/ttyS0"...);`
- `read(fd, buffer, sizeof(buffer));`
- `write(fd, buffer, sizeof(buffer));`
- `close(fd);`

Fonctions supplémentaires

(dépend de la classe choisie), généralement ioctl voire d'autres (seek, flush,...)


Application C/C++:
open, ioctl, read, write, close
/device1
/device2
OS
Pilote
Périph 1
Périph 2

Théorie

Techniques d'implémentation

Méthodes read/write :

Coté application :

- Bloquants : lecture/écriture de X octets terminée
- Non bloquants... mais pooling déconseillé
- Bloquants avec chien de garde...utile pour mécanisme de callback
- Attente multiple (poll) de plusieurs évènements:
  - Données lues/écrites sur le port
  - Et/ou timeout
  - Et/ou erreur

Coté implémentation

- Scrutation (en mode noyau), utilisé notamment pour l'attente multiple (`select`)
- Interruption $\rightarrow$ **c'est dans le driver qu'on récupère l'interruption prévue par le processeur pour un port donné.**
- Challenge: **relier des IT physiques (bas niveau du driver) à des IT logicielles** dans le code C/C++ utilisant le driver

Méthode ioctl :

On peut lire, écrire, mais aussi configurer notre driver :

Exemple

Sur un port série on peut :

- Lire et écrire des données,
- Configurer le port :
  - la vitesse de transmission (baudrate),
  - la parité, contrôle de flux,
  - bit de data, bit de stop,
  - ...

Implémentation

Pour configurer un driver on utilise la fonction `ioctl`.

Missions du driver :

Gestion des accès concurrents (tâches + interruptions)

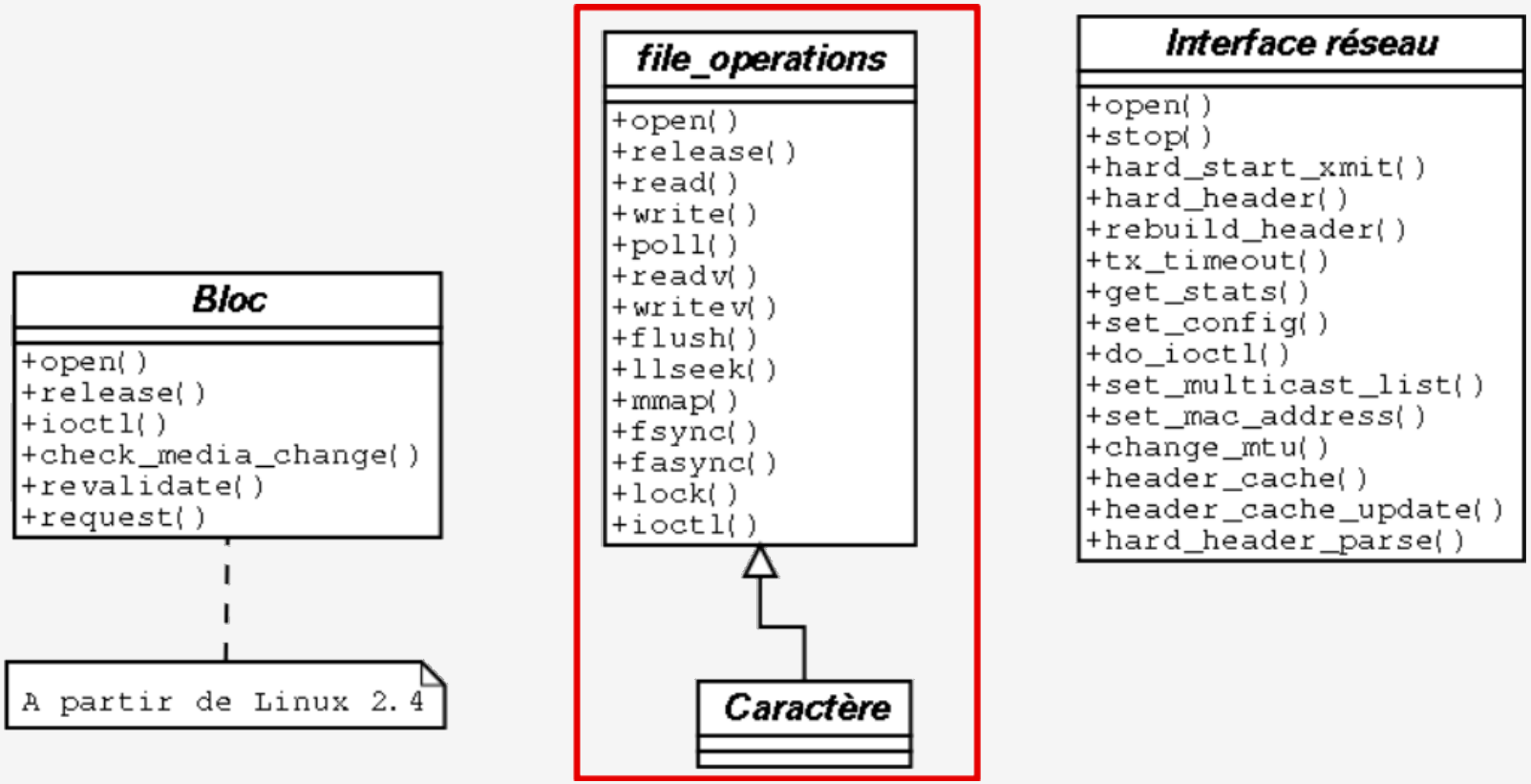
- Utilisation de verrous :
  - généralement verrou associé à une interruptions, débloquant le read/write sur évènement
- Utilisation de sémaphores et mutex pour **réserver l'utilisation d'un périphérique**
- N'autoriser qu'un processus/tâche à ouvrir le périphérique à un instant

Gestion des espaces mémoires:

- Rappel : espace driver et espace user sont distincts
- Le passage de données doit en tenir compte
  - `copy_to_user(dest, src, 1024);`
  - `copy_from_user(dest, src, 1024);`

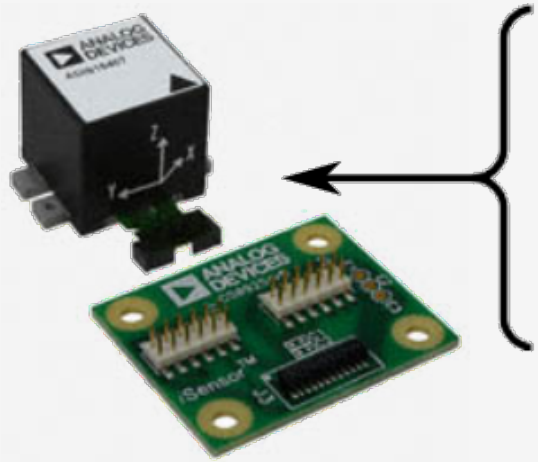
Université Gustave Eiffel

Fonctions à implémenter dans le driver :



Attention : point de vue du noyau (contenu d'un driver), pas d'une application **Liberté :** à l'intérieur d'un driver on peut créer autant de sous-fonctions que nécesaires

2 façons d'envisager le développement de driver radicalement opposées:



- un GPIO "data ready"
- un bus SPI
- un GPIO on/off de la centrale

Que faire ???

- **mono-driver :** un driver spécifique qui gère chaque port physique (GPIO, SPI...) et implémente l'automate d'état de la centrale ADIS
- **multi-driver :** développer plusieurs drivers génériques pour chaque type de port (GPIO, SPI...) et ramener la complexité dans le code métier (C / User Space)

Choix : placer le caractère et la complexité *métier* du device en espace noyau ou en espace user

Théorie

Exemple

On veut piloter des LEDS via un port d'entrée/sorties

- LED 1 sur port A (bit 0)
- LED 2 sur port B (bit 0)

adresse du registre du port A (sur bus uP) : 0xFFFF1002
adresse du registre du port B (sur bus uP) : 0xFFFF1004

Que doit-on faire ?

Il faut initialiser le pilote

- Déclarer le pilote
- Déclarer le(s) fichier(s) de périphérique
- Faire le lien entre le pilote et le(s) fichier(s) de périphérique
- Initialiser le périphérique (matériel et informations logicielles associées)
- Déclarer les ressources

Il faut respecter l'interface standard (classe)

- Fonction "`open`"
  - Initialiser le périphérique ?
  - Éviter les accès concurrents ?
  - Vérifier les droits d'accès
  - Déclarer utilisation
- Fonction "`close`"
  - Réinitialiser le périphérique ?
  - Déclarer fin d'utilisation
- Fonction "`read`" (un sens ?) , "`write`" et "`ioctl`"
  - Gestion accès concurrents
  - Accès au périphérique
  - Gestion espaces de privilèges

Théorie

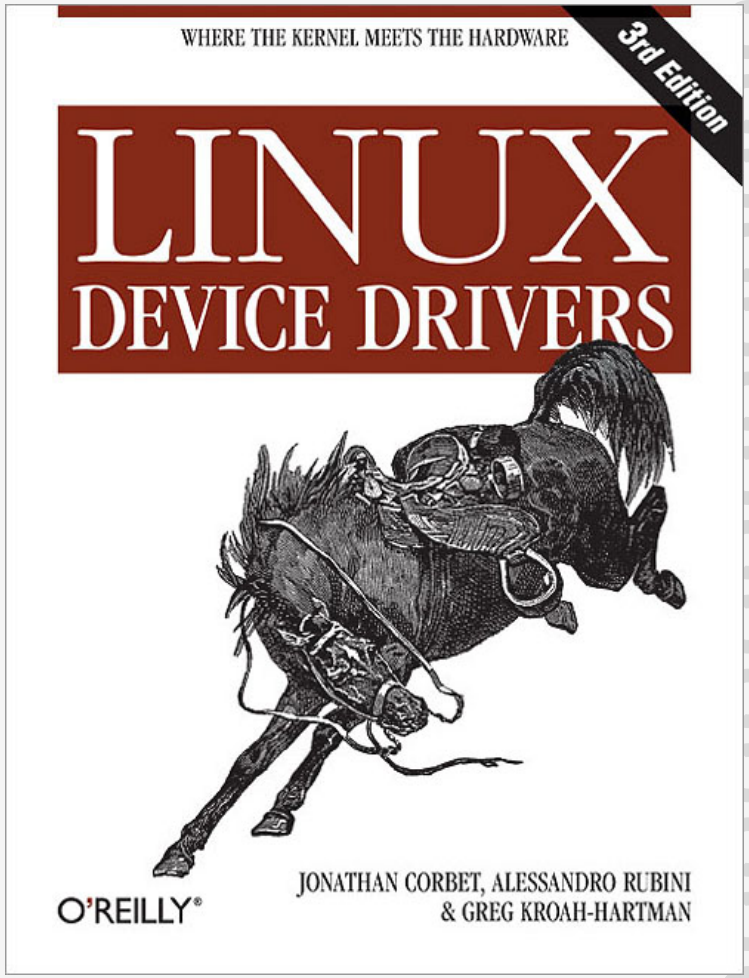
Références

Université Gustave Eiffel

Le **web** pullule de drivers pour **Linux**

- stackoverflow
- code source driver Linux (https://elixir.bootlin.com)
- google

Une référence pour éléments basiques des divers : livre “Linux Device Drivers”

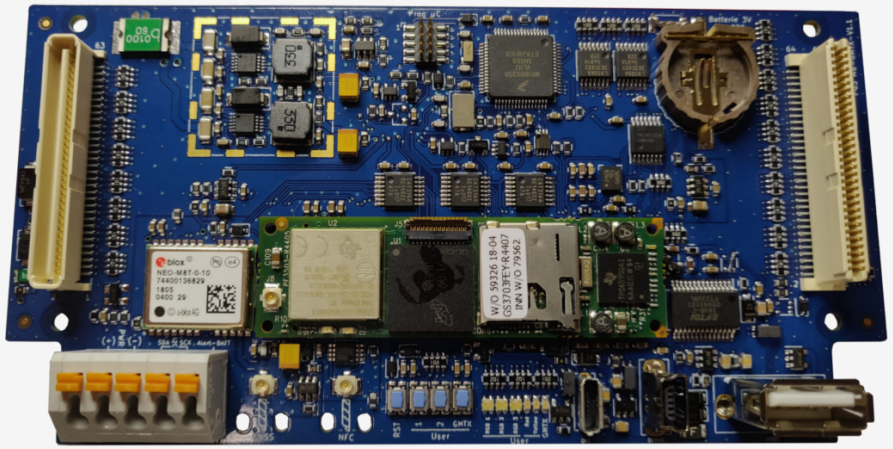


Pratique sur PEGASE

Contents

Université Gustave Eiffel

Développons brique par brique notre propre driver sur la cible PEGASE 3



Pratique sur PEGASE

Compiler un driver

Sur PC on va compiler les sources d'un driver "presque vide". C'est le squelette de tout driver de type "`char`". Ouvrir le fichier "ExampleDriver.c" et regarder le contenu du driver, voici le code minimum d'un driver.

ajoutez le répertoire "ExampleDriver" du driver dans
`/pegase3-sources/app`

Rappel:

Avant de compiler, lancez : `./open-terminal.sh`

`./open-terminal.sh`

Positionnez vous dans le répertoire du driver

`/pegase3-sources/app`

Une ligne de compilation particulière ... et à retenir !!!

```
make -C [chemin kernel build ] SUBDIRS=[chemin sources modules] modules
make -C /lib/modules/4.9.3-custom/build
SUBDIRS=/pegase3-sources/app/ExampleDriver modules
```

1. Compiler
2. Obtenir un fichier ExampleDriver.ko... le binaire d'un driver sous Linux
3. L'uploader sur la carte PEGASE avec scp ou *Nautilus* dans `/lcpc`

Pratique sur PEGASE

Charger/décharger un driver

Consulter les drivers chargés à un instant T sur PEGASE

```
lsmod
```

Examinée le résultat. Quels drivers déjà chargés ? Quelles informations sont données ?
Charger votre driver (depuis le répertoire `/lcpc` où il est déposé) :

```
insmod ExampleDriver.ko
```

Vérifiez...

```
lsmod
```

Déchargez le driver

```
rmmod ExampleDriver.ko
```

Consultez le log noyau:

```
dmesg
```

Assez joué avec ce driver : désormais, nous nous focaliserons sur le contenu du dossier "ifsttar-mydriver" ! Comme pour ExampleDriver, copiez-le dans ifsttar-driver

Pratique sur PEGASE

Créer des entrées dans un driver

Ouvrez le fichier “ifsttar-mydriver.c”, repérez et comprenez les points suivants dans les fonctions `__init` et leurs opposés dans la fonction `__exit` :

Permet d'avoir un numéro de majeur et de mineur pour le fichier de périphérique :

```
dev_t m_oDev // contient le major et le minor
MAJOR(dev_t m_oDev) // extrait le majeur
MINOR(dev_t m_oDev) // extrait le mineur
```

Comparez avec les informations de `/proc/devices` :

```
alloc_chrdev_region(&m_oDev , 0, 1,"myDevice");
```

Permet de créer une entrée dans `/sys/class` :

```
class_create(THIS_MODULE, "myClass");
```

Permet de créer une entrée dans `/dev` :

```
device_create(cl, NULL, m_oDev, NULL, "myDevice");
```

(1) Compiler, charger sur P3 et valider par un encadrant en retrouvant les numéros de majeur/mineur.

IMPORTANT : A RETENIR avant d'aller plus loin

- **Charger et décharger un driver**
  1. `int __init myDriver_init(void)` $\Longleftrightarrow$ `insmod` /chargement du driver
  2. `void __exit myDriver_exit(void)` $\Longleftrightarrow$ `rmmod` /déchargement du driver

- **Les "fichiers" dans `/dev` sont représentés par 2 numéros:**
  - Le MAJEUR (MAJOR) qui identifie le pilote.
  - Le MINEUR (MINOR) qui représente une sous-adresse en cas de présence de plusieurs périphériques identiques, contrôlés par un même pilote.
  - **Exemple** : le major des 8 Timers de PEGASE , un minor pour chaque Timer.
  - **Exécuter** et analyser sur P3 le résultat de *ls -l /dev*. Retrouvez d'autres banques avec 1 MAJOR $\Longleftrightarrow$ X MINOR

Pratique sur PEGASE

Interaction application/driver

Ajout des fonctions suivantes (dans le driver) pour les utiliser depuis le userspace (signature normée):

1. `open()` : ouverture du périphérique
2. `close()` : fermeture (libération) du périphérique
3. `read()` : lecture de données en provenance du périphérique
4. `write()` : écriture de données vers le périphérique
5. `ioctl()` : contrôle du périphérique (configuration, paramétrage...)
6. autres fonctions : `poll()`...

Ces fonctions sont appelées au travers des fichiers du répertoire `/dev` et définit dans le driver à l'aide de la structure standard (à ajouter dans le driver) :

```
static const struct file_operations my_fops = {
    .owner = THIS_MODULE,
    .read = my_read,
    .write = my_write,
    .release = my_close,
    .open = my_open,
    .unlocked_ioctl = my_ioctl,
};
```

1ère étape : initialiser le char device dans la fonction `__init`

```
static struct cdev m_oCDev; // char device
```

```
cdev_init(&m_oCDev, &my_fops);
cdev_add(&m_oCDev, m_oDev, 1);
```

`cdev_init` : sert à lier la structure `file_operations` (vu à la diapo 23) à la structure `cdev`

`cdev_add` : sert à lier la structure cdev à la structure `dev_t` → `/dev/myDevice`

2ème étape : supprimer le char device dans la fonction `__exit`

```
cdev_del(&m_oCDev);
```

3ème étape : ajoutez les fonctions `open`, `close` ... (fonction de la diapo 23) notez qu'il s'agit ici uniquement des prototypes des fonctions.

```
static int my_open(struct inode *inod, struct file *fil)
```

```
static int my_close(struct inode *inod, struct file *fil)
```

```
static ssize_t my_read(struct file *filp, char *buff, size_t len,
loff_t *off)
```

```
static ssize_t my_write(struct file *filp, const char *buff, size_t len,
loff_t *off)
```

```
static long my_ioctl (struct file *file, unsigned int cmd,
unsigned long arg)
```

Coté driver :

Dans le driver, ajoutez `printk("driver open\n");` dans la fonction `open` du driver et un `printk("driver close\n");` dans la fonction `close`. Compilez ifsttar-mydriver une première fois, des erreurs vont apparaître, corrigez-les. Une fois corrigées, installez le driver sur votre carte PEGASE.

Coté application :

Récupérez sur votre PC local, le répertoire **"MyDriverTestBasic"** (code source de base de l'application). Modifiez **MyTesterDriver.cpp** : ouvrir et fermer un des device montés par le insmod.

```
fd_MyDriver = open("/dev/XXXXXX", O_RDWR);
close(...);
```

Ensuite :

1. Téléchargez les 2 binaires : `ifsttar-mydriver.ko` et `myDriverTestBasic.bin` sur la PEGASE.
2. Chargez le driver (`insmod...`).
3. Lancez MyDriverTestBasic.
4. Repérez les différentes étapes : `open`, `close`...

(2) Faites valider l'affichage "driver open" et "driver close" par un encadrant

Nous allons dans un premier temps, lire la **date** en seconde epoch (seconde depuis 1970) à laquelle est la carte PEGASE à travers la fonction **read** du driver.
Important : se rappeler que espace driver et espace user sont des espaces mémoires différents. Nécessité de copier les données d'un espace à l'autre !

Coté driver : Complétez la fonction `read` pour qu'elle récupère la date de la carte à l'aide de la fonction :

```
ktime_get_real_ts64(struct timespec64 *tv);
```

et qu'elle renvoie dans buff le champ `.tv_sec` de la structure `timespec64` à l'aide de :

```
copy_to_user(..., ..., ...);
```

Coté application : Appelez la fonction `read` avec du code C++ pour lire un `64 bits`.

```
test = read(fd_MyDriver, (void*) &dateToRead, sizeof(dateToRead));
printf("Date read est %llu\n",dateToRead);
```

(3) Faites valider l'affichage "Data read..." par un encadrant et vérifier la validité de la date

Idem pour le `write...` : Nous allons changer la date de la carte en venant écrire dans le driver cette nouvelle date.

Coté driver :

Complétez la fonction `write` pour qu'elle affiche la valeur reçu à l'aide de :

```
copy_from_user(...,...,...)
```

Changer la date de la carte avec la fonction (pensez à mettre le champ tv_nsec à 0):

```
do_settimeofday64(const struct timespec64 *ts);
```

Coté application :

Appelez la fonction `write` avec du code C/C++ pour écrire une date au format seconde epoch (www.epochconverter.com pour récupéré un exemple)

```
test = write(... , ... , ...);
```

(4) Faites valider l'affichage "Date write..." par un encadrant et le changement de date de la carte

Passer/récupérer des paramètres au driver

Nous allons utiliser la fonction **ioctl** pour changer la fonctionnalité du driver. Dans notre cas, l'ioctl va permettre de changer la date renvoyée par le `read`, au lieu de renvoyer la date à laquelle est la carte, le read va renvoyer le temps écoulé depuis le boot à l'aide de la fonction :

```
ktime_get_ts64(struct timespec64 *ts);
```

Coté driver :

Créer une variable **globale** "`modeRead`" qui en fonction de sa valeur permet au **read** de renvoyer soit la **date actuelle** (`ktime_get_real_ts64(struct timespec64 *ts)`) ou le temps depuis le **boot** (`ktime_get_ts64(struct timespec64 *ts)`).

Nous allons maintenant implémenter la fonction `ioctl` pour modifier cette variable global modeRead et ainsi changer le fonctionnement du read.

Passer/récupérer des paramètres au driver

Coté driver :

Créer un `.h` du driver pour y ajouter les `#define` des `IOCTL_XXXX`. Les numéros de commande `ioctl` doivent être uniques dans tout le système afin d'éviter les erreurs dues à l'envoi de la bonne commande au mauvais driver. Il faut donc utiliser les macros suivantes :

```
#define IOC_MAGIC 'k'
#define IOCTL_XXXX _IO(IOC_MAGIC, int number)
#define IOCTL_XXXX _IOW(IOC_MAGIC, int number, data_type)
#define IOCTL_XXXX _IOR(IOC_MAGIC, int number, data_type)
#define IOCTL_XXXX _IOWR(IOC_MAGIC, int number, data_type)
```

number : numéro de commande (entier de 8 bits), dans un driver, des numéros distincts doivent être choisis pour chaque commande `ioctl` différent que le driver fournit
data_type : le nom du type échangés (Exemple `int`, `int16_t`, `struct myStruct`), le noyau l'utilise pour calculer le nombre d'octets échangés entre le client et le pilote.

Exemple :

```
#define IOCTL_SET_MODE_READ _IOW(IOC_MAGIC,0x01,int)
```

Passer des paramètres au driver

Coté driver :

Complétez la fonction `ioctl` pour qu'elle gère la sollicitation des différents `ioctl` via un `switch`. Afficher le paramètre passé par l'ioctl et implémenter le changement de mode du read.

```
switch(cmd) {
        case IOCTL_SET_MODE_READ:
        //changement mode read
        printk("Ioctl called with cmd = %d et arg= %d.\n",cmd,num);
        break;
```

Coté application :

Appelez la fonction `ioctl` avec du code C/C++ pour passer le **nouveau** mode de read en paramètre. L'application doit lire chaque seconde, la date actuelle de la carte et le temps écoulé depuis le boot.

```
int newModeRead ...
ioctl(fd_MyDriver, IOCTL_SET_MODE_READ, &newModeRead);
```

(5) Faites valider l'affichage "Ioctl called..." et l'affichage des deux différentes dates par un encadrant

Récupérer des paramètres au driver

La fonction ioctl permet aussi de récupérer la configuration du driver

Coté driver :

Complétez la fonction `ioctl` pour qu'elle gère la sollicitation des différents `ioctl` via un `switch` :

```
switch(cmd) {
        case IOCTL_GET_MODE_READ:
            copy_to_user( arg, &modeRead,sizeof(int));
            ...
            break;
```

Coté application :

Appelez la fonction `ioctl` avec du code C/C++ pour récupérer le mode de read dans lequel est le driver.

```
ioctl(fd_MyDriver, IOCTL_GET_MODE_READ, &mode);
printf("mode = %d\n",mode);
```

(6) Faites valider l'affichage "mode = ..." par un encadrant

IMPORTANT : A RETENIR avant d'aller plus loin

- Rôles des fonctions `open/close`
- Rôles des fonctions `read/write`
- Partage les différents commandes `ioctl` possibles via un fichier commun driver / code C (`ifsttar-mydriver.h`)
- Notion d'espaces mémoires distincts (kernel / user-space): `copy_to/from_user`

Pratique sur PEGASE

Piloter un vrai port physique

Nous allons maintenant contrôler un port physique à l'aide d'un driver. Pour cet exemple nous allons contrôler une LED (GPIO). Les drivers permettent de contrôler toute sorte de ports physiques : SPI, I2C, UART, GPMC....

Sur la carte PEGASE nous allons utiliser le port GPIO **176** pour allumer une LED.

Mais avant cela, reprenons depuis le début.

Qu'est ce qu'un device tree :
Une Arborscence Matérielle (DT = Device Tree) est une façon de **décrire le matériel présent dans un système**. Il doit inclure le nom de la CPU (microprocesseur) de base, sa configuration mémoire et les périphériques (internes et externes) qui y sont reliés.

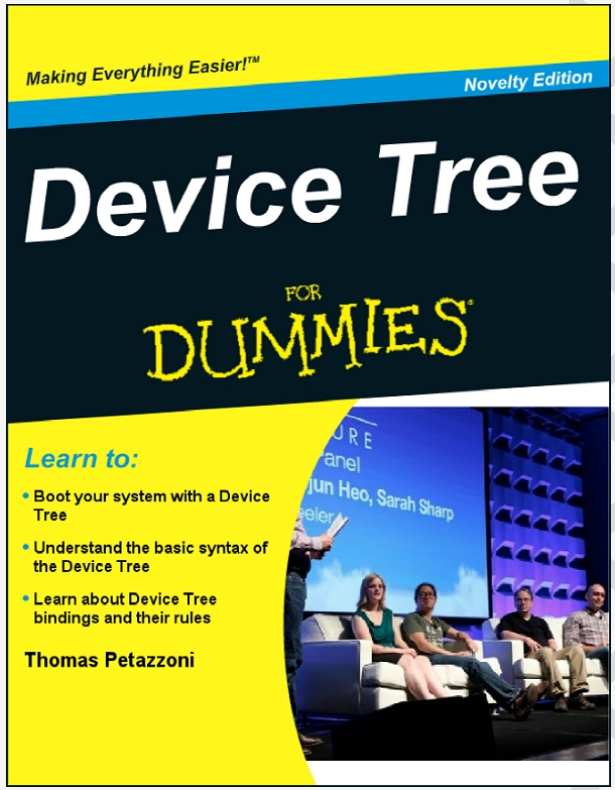
Pourquoi les device tree :

- **Mutualiser** le code kernel ARM
- Tendre vers un kernel **générique** qui exploite une **description du hardware**
- Décrire le matériel dans une structure de données **DTS (device tree source)**
- Le kernel généricisé exploite cette structure de données compilée:
  - Fichier **DTS** (compilé) $\rightarrow$ Fichier **DTB (device tree blob)**
- Définition dans des fichiers texte (dts et dtsi) sous forme d'arbre selon une syntaxe normée à base de noeuds et de propriétés

Analogie :
Un exécutable qu'on lance avec des paramètres $\rightarrow$ `main(void* argv, int argc)`

Exemple dts :

```
/{
  node1 {
      a-string-property = "A string";
      a-string-list-property = "first", "string";
      a-byte-data-property = [01 23 34 56];
      child-node1 {
      first-child-property = <1>;
    };
    child-node2 {
      a-string-property = "Hello, world";
    };
};
```



Pour visualiser le **dts complet de pegase** compiler le DTS puis décompiler le DTB avec `fdtdump`, dans le docker :

```
apt-get install device-tree-compiler
cd kernel/dts/
make && fdtdump /pegase3/kernel/dts/pegase3-mere.dtb > myDts.txt
```

Ajoutez les lignes suivantes dans le fichier `/kernel/dts/pegase3-mere.dts` après **model = "pegase-mere";** :

```
ifsttar_mydriver@4 {
  compatible = "ifsttar,myDriver";
};
```

`compatible` : ce champ permet de lier votre driver au dtb, il est par convention de la forme “entreprise, nomDuDriver”

Compliler le dts (`make`) et mettre le nouveau pegase3-mere.dtb dans le répertoire `/usr/lib/linux-image-4.19.4-custom/` de la PEGASE. Rebootez la carte pour que le nouveau dtb soit chargé.

Sur l’ordinateur, décompiler (`fdtdump`) le nouveau dtb et retrouver le node ifsttar_mydriver.

Sur PEGASE3, retrouver le node ajouté par le nouveau dtb sur lequel elle a booté:

```
ls -l /proc/device-tree/
```

(7) Faites valider par un encadrant en montrant la node ifsttar_mydriver

Dans le driver, déclarer une structure de type *platform_driver* :

```
static struct platform_driver ifsttar_mydriver_platform_driver = {
    .driver = {
      .name = "myDriver",
      .owner = THIS_MODULE,
      .of_match_table = ifsttar_mydriver_of_match,
    },
    .probe = myDriver_probe,
    .remove = myDriver_remove,
}
```

Cette structure est appelée par les fonctions suivantes (à ajouter à la fin des fonctions: `__init` et le `__exit`)

```
platform_driver_register(&ifsttar_mydriver_platform_driver);
platform_driver_unregister(&ifsttar_mydriver_platform_driver);
```

Nous allons maintenant définir les structures et fonctions `ifsttar_mydriver_of_match`, `myDriver_probe` et `myDriver_remove`.

IMPORTANT : c'est la structure de type `struct of_device_id`, qui assure le **lien entre le driver (.ko) et le device tree (.dts)**. Il faut que ça matche !

```
static const struct of_device_id ifsttar_mydriver_of_match[] = {
  { .compatible = "XXXXXXXXXXX" }, //pour que le driver match avec dts
  {}
};
```

Lors de l'appel de la fonction "`platform_driver_register`" (insmod) Linux va chercher le champ **"compatible"** dans la structure `of_device_id`. Si Linux retrouve ce champ dans le **device tree**, alors la fonction `probe` **sera appelée automatiquement**.

Université Gustave Eiffel

Il faut maintenant définir la fonction `probe` et `remove` :

```
static int myDriver_probe(struct platform_device *pdev){
  printk("start probe\n");
  return 0;
}
static int myDriver_remove(struct platform_device *pdev){
  printk("start remove\n");
  return 0;
}
```

Les informations concernant le device tree se trouveront dans la structure "`platform_device`" en paramètre de la fonction `probe`. Compilez le driver et le mettre sur la carte.

(8) Faites valider par un encadrant en affichant "start probe/remove"

IMPORTANT : A RETENIR avant d'aller plus loin

Comment est activée la fonction `probe` :

1. `platform_driver_register(&ifsttar_mydriver_platform_driver)`
2. Dans la structure `platform_driver`, nous avons défini une fonction `probe` et une structure `of_device_id (of_match_table)`
3. Le Linux va chercher dans la structure `of_device_id` le champ **compatible** et va essayer de retrouver le même dans le **device tree**
4. Si le Linux retrouve ce champ compatible dans le device tree, alors la fonction `probe` **sera appelée** avec en paramètre les **information du node correspondant (`struct platform_device`)**

Nous allons maintenant enrichir le dts et venir récupérer ces informations depuis le driver.

Nous allons maintenant récupérer un paramètre "charTest" dans le dts

Coté device tree :

Ajouter la ligne **charTest = "ma premiere propriete char";** dans votre dts :

```
ifsttar_mydriver@4 {
  compatible = "ifsttar,mydriver";
  charTest = "ma premiere propriete char";
};
```

Coté driver :

Dans la fonction `probe`, nous allons venir chercher la propriété **charTest**

```
struct device_node *np = pdev->dev.of_node;
const char* test_char_property;
test_char_property = of_get_property(np, XXXXXXXXX, NULL);
printk("test_char_property: %s\n", test_char_property);
```

np : correspond au node de notre device tree

XXXXXXXXX : correspond au nom de la propriété cherchée

(9) Faites valider par un encadrant en affichant "test_char_..."

Nous allons maintenant contrôler un GPIO : le 176 ou 64 (pegase3 version $>$ 1.41)

Coté device tree :

Ajouter la ligne **ifsttar-led...** dans votre dts :

```
ifsttar-led = <&gpio6 16 GPIO_ACTIVE_LOW>; /* GPIO176: (6-1)*32 + 16 */
          ou <&gpio3 0 GPIO_ACTIVE_LOW>; /* GPIO64: (3-1)*32 + 0 */
```

Coté driver :

Dans la fonction probe, nous allons venir chercher ce **GPIO**

```
int ledGpio;
ledGpio = of_get_named_gpio(np, "ifsttar-led", 0); //recupere GPIO
printk("ledGpio = %d\n", ledGpio);
devm_gpio_request_one(&pdev->dev, ledGpio, GPIOF_OUT_INIT_LOW,
                                                "ifsttar-led");
gpio_set_value(ledGpio, 1); // ledGPpio a 1
```

devm_gpio_request_one : alloue la ressource GPIO, elle est désallouée au déchargement du driver

Inclure le header: **"linux/of_gpio.h"**

(10) Faites valider par un encadrant en allumant la LED

Application bilan LED :

Application

- Dans **ifsttar-mydriver.c** :
  - faites en sorte que sur un `write` 1 ou 0 on allume ou éteigne la LED
- Dans **MyDriverTestBasic** :
  - Faire clignoter la LED toutes les secondes

(11) Faites valider par un encadrant en faisant clignoter la LED

Et passons à la **lecture** d'un port physique...l'état d'un simple bouton (**GPIO 65**)

Coté device tree :

Ajouter la ligne **ifsttar-led...** dans votre dts :

```
ifsttar-bouton = <&gpioXXX YYY 0>; /* GPIO_65 */
```

Coté driver :

Dans la fonction `probe`, récupéré le GPIO comme vu précédemment mais en utilisant GPIOF_IN au lieu de GPIOF_OUT_INIT_LOW

Dans la fonction `read`, renvoyer la valeur du GPIO avec la fonction :

```
gpio_get_value(boutonGpio);
```

Application bilan bouton :

Application

- Dans **MyDriverTestBasic** :
  - Lire la valeur du bouton toutes les 2 secondes

(12) Faites valider par un encadrant en renvoyant la valeur du bouton toutes les deux secondes

Application bilan bouton :

La lecture du bouton fonctionne mais...

Est temporisé par le `sleep(2);` → **manque de réactivité du programme**

Pour plus de réactivé :

- Essayer en enlevant `sleep()` et afficher le `printf()` **uniquement** sur changement d'état du bouton;
- Lancer le programme en parallèle (avec un & à la fin de l'appel du programme)
- Observer l'activité CPU de PEGASE 3 avec la commande `top`

(13) Faites valider par un encadrant en montrant le temps CPU de votre application

IMPORTANT : A RETENIR avant d'aller plus loin

- Les GPIO utilisés dans le driver sont défini dans le dts et le dts est lié au driver à l'aide du
  - compatible = "ifsttar,mydriver";
- Pour récupérer les GPIOs et paramètres du dts avec les fonctions suivantes :
  - of_get_property
  - of_get_named_gpio
  - of_property_read_u32...
- Même principe pour tous les ports: SPI, Timers, UART,...

On lit l'état d'un port (BP) OK... mais quel est l'inconvénient ???

Gestion d'interruption : comprendre cette étape est un pas important dans la maîtrise du Linux Embarqué.
Lier hardware et Software:

- Hardware $\Rightarrow$ interruption
- Software $\Rightarrow$ CallBack

Permettre au codes utilisateurs de développer des applications, des SDK... au **fonctionnement événementiel** :

- Rappel / Objectif d'une application : une boucle principale qui dort
- Des traitements réalisés sur évènement. Exemples :
  - Réception d'une trame TCP/IP
  - Acquisition bus SPI
  - Bouton Pressé
  - Mouse Move
  - Timer
  - etc...

Et passons à la **lecture** d'un port Physique...l'état d'un simple bouton (**GPIO 65**)

Coté device tree :

Ajouter la ligne **ifsttar-led...** dans votre dts :

```
interrupt-parent = <&gpio3>;
interrupts = <1 IRQ_TYPE_EDGE_RISING>; /* GPIO_65 */
```

Coté driver :

Dans la fonction probe, récupérez le numéro d'IRQ et liez avec la fonction d'IRQ :

```
irq = platform_get_irq(pdev, 0); // recupere numero d'irq
devm_request_threaded_irq(&pdev->dev, irq, NULL, my_fonction_irq,
            IRQF_TRIGGER_RISING | IRQF_ONESHOT, "driver name", NULL);
```

Dans le driver, créez la **fonction d'IRQ** et ajoutez le #include:

```
#include <linux/interrupt.h>
static irqreturn_t my_fonction_irq (int irq, void *dev_id){
  printk("Rising!\n");
  return IRQ_HANDLED;
}
```

(14) Faites valider par un encadrant en affichant "Rising" sur appuis du bouton

Nous allons maintenant débloquer un mutex sur interruption pour réaliser un read bloquant :

Coté driver :

Déclarer un **mutex** et le **débloquer** dans la **fonction d'interruption** :

```
DECLARE_WAIT_QUEUE_HEAD(MyDriverReadMutex);

static irqreturn_t my_fonction_irq (int irq, void *dev_id)
{
  bDone = 1;
  wake_up_interruptible(&MyDriverReadMutex);
  return IRQ_HANDLED;
}
```

Cette fonction, appelée lors d'une IT physique sur le périphérique débloque un mutex MyDriverRead.

Coté driver :

Modifiez la fonction **my_read** pour être **BLOQUANTE**

```
wait_event_interruptible(MyDriverReadMutex, bDone==1);
bDone = 0;
```

Cette fonction est bloquée sur le mutex **MyDriver_ReadMutex** ⇒ Lorsque le mutex débloque, on lit la valeur et `read` la renvoie au code appelant.

Coté application :

Dans la boucle principale provoquez une lecture en boucle MAIS bloquante sur le BP1:

```
int ValueButton = 0;
do {
  test = read(fd_myDevice, &ValueButton, sizeof(int));
  printf("Button state = %d\n", ValueButton);
} while (m_bRunningMain > 0);
```

Faites fonctionner l'ensemble sur PEGASE et constatez que vous affichez l'état du bouton **uniquement lors d'un appui bouton** $\Rightarrow$ sur **interruption** !

Vérifier l'activité CPU avec top.

(15) Faites valider par un encadrant

Vous avez maintenant une application qui est bloquée sur le read du bouton.
Cependant, votre main est **bloqué** et ne peut plus effectuer d'autre action, ex : contrôler la LED, lire un autre bouton, lire un port série...

Coté application :

Mettre en place un mécanisme pour faire clignoter la led à 1 Hz tous en continuant à lire l'état du bouton sur un read bloquant.
Penser à utiliser la librairie **pthread**

(16) Faites valider par un encadrant

Votre application ne renvoie la valeur du bouton que lors de son relâchement.

Application

Modifiez votre code pour que la valeur soit lu à **chaque changement d'état** du bouton (appuis et relâchement du bouton). De plus, faire clignoter la LED à 1 Hz quand le bouton est **relâché** et à 10 Hz quand celui-ci est **enfoncé**.

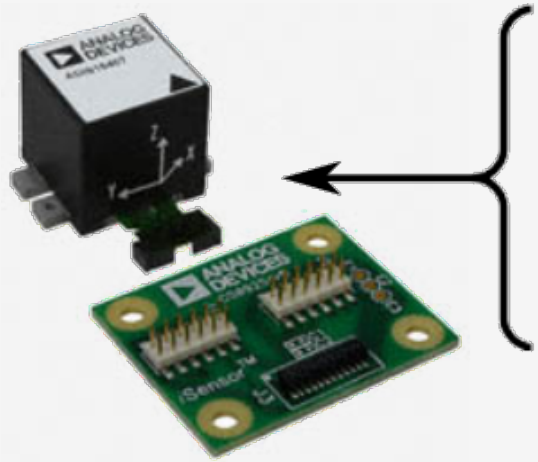
(17) Faites valider par un encadrant

IMPORTANT : A RETENIR avant d'aller plus loin

- Bien comprendre le fonctionnement et l'intérêt du mode "interruption" au sein des drivers...
  - **Stabilité** : pas de pooling
  - **Energie** : endormissement, réduction du process

- Intérêt **rendre tous les fonctions read** de chaque driver (gpio, spi, uart...) **bloquantes**

- **Question** : si le programme doit lire plusieurs ports (i.e. faire appel à plusieurs **read)** comment gérer le fait que chacune soit bloquante ??? Use case exemple:



- un GPIO "data ready"
- un bus SPI
- un GPIO on/off de la centrale

Pour aller plus loin, ajouter à votre driver les fonctionnalités suivantes :

Communication inter-driver... TRÈS simple !

Utile pour passer dynamiquement des données d'un driver à un autre

- Un premier (ifsttar-mydriver.ko) driver va exporter une variable ou une fonction
- Un seconds (ifsttar-mydriver2.ko) driver va pouvoir utiliser cette variable ou cette fonction

Solution : voir ce lien.
Observez les dépendances et l'ordre de chargement des drivers avec `lsmod`.

Passez un paramètre à un driver

Cette fonctionnalité permet de passer un paramètre lors du `insmod` :
`insmod ifsttar-mydriver.ko parameter=123456`
Solution : voir ce lien.
Ajoutez un `ioctl` ou un `printk` permettant d'afficher la valeur du paramètre...

Dernière fonction à ajouter :

Information du driver...

Créez une entrée du driver dans `/proc` pour pouvoir récupéré des informations en exécutant la commande `cat /proc/MyDriverProcName`

```
cat /proc/MyDriverProcName
Hello il y a eu 17235 interruptions !
```

Solution : voir ce lien.

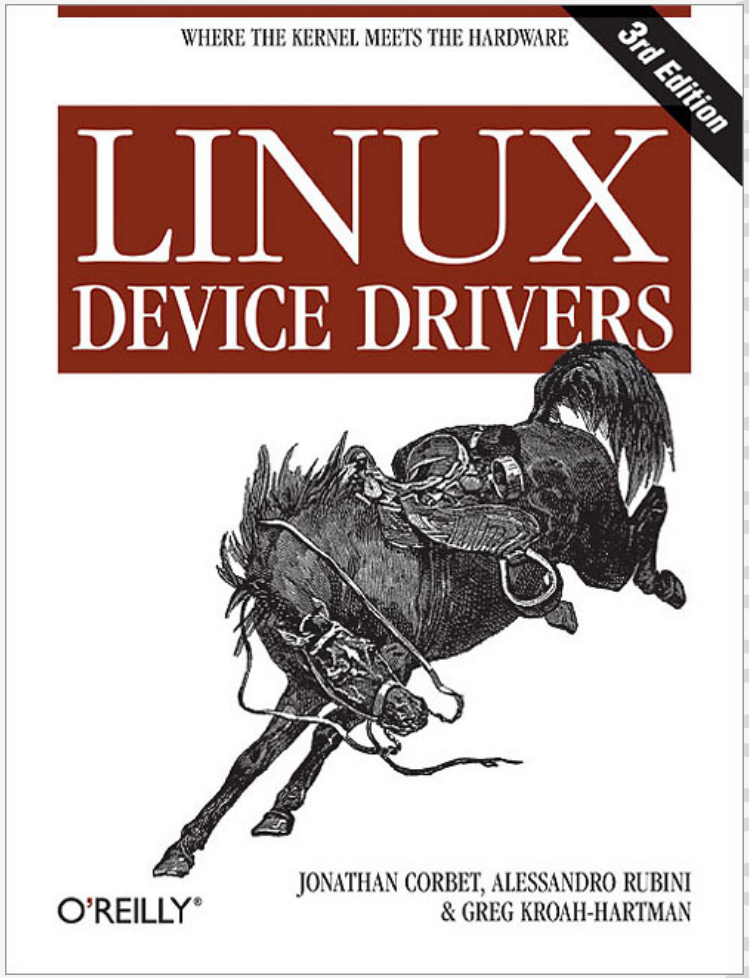
Pratique sur PEGASE

Références

Le **web** pullule de drivers pour **Linux**

- stackoverflow
- code source driver Linux (https://elixir.bootlin.com)
- google

Une référence pour éléments basiques des divers : livre “Linux Device Drivers”



Questions?