Pilote de périphérique

Objectif général

L'objectif de la séance est de commander les LEDS et le bouton poussoir (BP) et en passant par des pilotes installé(s) dans le noyau sur la RaspberryPi 1.

Vous savez déjà contrôler les LEDS et accéder au BP depuis l'espace utilisateur. Pour cela, vous avez dû mapper dans l'espace virtuel du processus utilisateur la zone de l'espace d'adressage physique permettant l'accès aux GPIO. Mais, il vous fallait avoir les droits du root. En passant par un pilote, les LED et le BP seront accessibles par un utilisateur standard.

Driver pour les LEDs et le bouton poussoir

> **Nous allons donc créer un pilote pour le périphérique LED et un pour BP.**
>
> Ce pilote sera accessible dans par un pseudo-fichier comme `/dev/led0_XY` pour la commande
> **XY** correspond aux initiales de votre binôme pour éviter les conflits avec vos camarades
> Par exemple, pour **A**lmada et **F**omentin, il faudrait créer /dev/ledbp**AF**
>
> **Dans le texte de TP, nous n'avons pas toujours fait apparaître les lettres XY, vous devez les ajouter vous-même en les remplaçant par vos initiales.**

Par exemple, vous pourrez écrire un programme test (cf fichier `test.c` ci-dessous) qui accède aux LED et BP en s'exécutant entièrement en mode utilisateur.

> **Le comportement proposé ici du pilote est le suivant :**
>
> - **Pour les LEDS**
>   - Le driver LED ne commande qu'une seule LED dont il reçoit le numéro au moment de l'enregistrement.
>   - On éteint la led si on écrit '0', on l'allume en écrivant '1'.
>   - Si le device led est accessible par /dev/led0_XY
>
> ```
> char led = '0';
> int fd = open("/dev/led0_XY", O_WR);
> write( fd, &led, 1);
> ```
>
> - **Pour le bouton**
>   - Le driver BP contrôle l'unique bouton dont il reçoit le numéro au moment de l'enregistrement.
>   - Quand le bouton est relâché, le pilote met le caractère `'0'`.
>   - Quand le bouton est enfoncé, le pilote met une valeur différente de `'0'`.
>   - Si le device bp est accessible par /dev/bp_XY
>
> ```
> char bp;
> int fd = open("/dev/bp_XY", O_RD);
> read( fd, &bp, 1);
> ```

C'est une proposition, vous pouvez faire comme bon vous semble. On peut imaginer d'autres manières, mais celle-ci me semble plus simple. Ce programme (de principe) est censé faire clignoter la led `'0'` jusqu'à ce qu'on appuie sur le bouton.

```
#include <stdio.h>

int main()
```

```
{
   char led, bp;
   int fdled0 = open("/dev/led0_XY", O_WR);
   int fdbp = open("/dev/bp_XY", O_RD);
   if ((fdled0 < 0)||(fdbp < 0)) {
      fprintf(stderr, "Erreur d'ouverture des pilotes LED ou
Boutons\n");
      exit(1);
   }
   do {
      led = (led == '0') ? '1' : '0';
      write( fd, &led, 1);
      sleep( 1);
      read( fd, &bp, 1);
   } while (bp == '0');
   return 0;
}
```

Références

- Vous trouverez pas mal d'informations sur internet :
  - http://doc.ubuntu-fr.org/tutoriel/tout_savoir_sur_les_modules_linux
  - http://pficheux.free.fr/articles/lmf/drivers/
  - https://broux.developpez.com/articles/c/driver-c-linux/
- si vous voulez allez plus loin, il y a le livre :
  - Linux Device Drivers, 2nd Edition

Étape 1 : création et test d'un module noyau

Code du module

- La première étape consiste à créer un module noyau, l'insérer puis l'effacer du noyau.
- Le module minimal se compose d'une fonction d'initialisation et d'une fonction de cleanup, dans le fichier `module.c` suivant:

```
#include <linux/module.h>
#include <linux/init.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Charlie, 2015");
MODULE_DESCRIPTION("Module, aussitot insere, aussitot efface");

static int __init mon_module_init(void)
{
   printk(KERN_DEBUG "Hello World <votre nom> !\n");
   return 0;
}

static void __exit mon_module_cleanup(void)
{
   printk(KERN_DEBUG "Goodbye World!\n");
}

module_init(mon_module_init);
module_exit(mon_module_cleanup);
```

- **Questions**
  - Quelle fonction est exécutée lorsqu'on insère le module du noyau ?
  - Quelle fonction est exécutée lorsqu'on enlève le module du noyau ?

Compilation du module

- Ce programme est cross compilé, puis copié sur la Raspberry Pi cible avec le fichier `Makefile` ci-après.
- Ce Makefile a besoin des sources compilées du noyau présent sur la RaspberryPi. Comme elles sont volumineuses, elles sont copiées dans le répertoire `/dsk/l1/misc/linux-rpi-3.18.y`.
- Si vous voulez le faire chez vous, il faut que vous preniez les sources de votre distribution. Vous pouvez suivre le tutoriel suivant : Compilation croisée d'un module linux pour Raspberry Pi.
- Sur votre PC, vous allez commencer par regarder si le répertoire `/dsk/l1/misc/linux-rpi-3.18.y` existe avec `ls -d /dsk/l1/misc/linux-rpi-3.18.y`.
  s'il existe, c'est bon, il n'y a rien à faire.
  s'il n'existe pas vous allez le créer :
  - en téléchargeant l'archive linux-rpi-3.18.y.tbz2 (192Mb) **dans le répertoire** `/dsk/l1/misc`
  - puis en la décompressant `tar xjf /dsk/l1/misc/linux-rpi-3.18.y.tbz2 -C /dsk/l1/misc` (1.1Gb après décompression)
  - puis autorisant ces sources à tous `chmod -R 755 /dsk/l1/misc/linux-rpi-3.18.y`

```
CARD_NUMB        = 2X
ROUTER           = peri
LOGIN            = nom1-nom2
LAB              = lab2

MODULE           = module

CROSSDIR         = /users/enseig/franck/IOC
KERNELDIR        = /dsk/l1/misc/linux-rpi-3.18.y
CROSS_COMPILE    = $(CROSSDIR)/arm-bcm2708hardfp-linux-
gnueabi/bin/bcm2708hardfp-

obj-m            += $(MODULE).o
default:;        make -C $(KERNELDIR) ARCH=arm
CROSS_COMPILE=$(CROSS_COMPILE) M=$(PWD) modules
clean:;          make -C $(KERNELDIR) ARCH=arm
CROSS_COMPILE=$(CROSS_COMPILE) M=$(PWD) clean
upload:;         scp -P622$(CARD_NUMB) $(MODULE).ko
pi@$(ROUTER):$(LOGIN)/$(LAB)
```

Travail à faire

- Sur votre compte enseignement, vous devez:
  - **Créer** ces fichiers, vous pouvez les commenter en cherchant dans les manuels (ou Google).
  - **Changer la valeur des variables** `CARD_NUMB`, `LOGIN` et `LAB` afin de les adapter respectivement à votre numéro carte, au nom du répertoire créé par vous sur la !RaspberryPI et au nom du sous-répertoire créé par vous pour ce TP. Les répertoires et sous-répertoires doivent exister et vous devez donc commencer par vous logger sur votre carte !RaspberryPI avec `ssh` pour les créer.
  - **Compiler** le module avec la commande `make`.
  - **Copier** sur la RaspberryPi avec scp avec la commande `make upload`.

- Sur la carte RaspberryPi, vous devez:
  - Dans le répertoire `$(LOGIN)/$(LAB)` (par exemple `alamada-fromentin/lab2`) où vous avez copié votre module

```
$ sudo insmod ./module.ko

$ lsmod
$ dmesg
$ sudo rmmod module
$ lsmod
```

```
$ dmesg
```

- Les commandes `lsmod` et `dmesg` permettent de voir les actions du module.
- **Résumez dans le CR** ce que vous avez fait et ce que vous observez.

Étape 2 : ajout des paramètres au module

- Votre driver devra être paramétré pour lui indiquer le numéro de ports utilisés pour les LEDS et les boutons. Dans un premier temps vous allez vous contenter d'indiquer le numero du bouton pour le module de test.
- **Vous devez ajouter** dans module.c

```
static int btn;
module_param(btn, int, 0);
MODULE_PARM_DESC(btn, "numéro du port du bouton");

static int __init mon_module_init(void)
{
    printk(KERN_DEBUG "Hello World !\n");
    printk(KERN_DEBUG "btn=%d !\n", btn);
    return 0;
}
```

- Le paramètre est défini au moment de l'insertion.

```
$ sudo insmod ./module.ko btn=18
```

- Pour les numéros de GPIO de LEDs, comme il pourrait y en avoir plusieurs (si vous voulez commander les deux leds en même temps), vous pouvez utiliser `module_param_array`.

```
#define NBMAX_LED 32
static int leds[NBMAX_LED];
static int nbled;
module_param_array(leds, int, &nbled, 0);
MODULE_PARM_DESC(LEDS, "tableau des numéros de port LED");

static int __init mon_module_init(void)
{
    int i;
    printk(KERN_DEBUG "Hello World !\n");
    for (i=0; i < nbled; i++)
       printk(KERN_DEBUG "LED %d = %d\n", i, leds[i]);
    return 0;
}
```

- Le paramètre est défini au moment de l'insertion.

```
$ sudo insmod ./module.ko leds=4,17
```

- **Questions** :
  - Comment **voir** que le paramètre a bien été lu ?

Étape 3 : création d'un driver qui ne fait rien, mais qui le fait dans le noyau

Création du driver

- Votre driver va être intégré dans un module. Vous allez donc créer un module **nommé** `led` (et non plus `module`) paramétré avec les numéros de ports pour les LEDS et le bouton. Vous utiliserez un nouveau répertoire. Vous modifierez le Makefile en conséquence.
- Vous ajoutez dans le fichier `.c` du module `led` :

```c
#include <linux/fs.h>

static int
open_led_XY(struct inode *inode, struct file *file) {
    printk(KERN_DEBUG "open()\n");
    return 0;
}

static ssize_t
read_led_XY(struct file *file, char *buf, size_t count, loff_t *ppos)
{
    printk(KERN_DEBUG "read()\n");
    return count;
}

static ssize_t
write_led_XY(struct file *file, const char *buf, size_t count, loff_t
*ppos) {
    printk(KERN_DEBUG "write()\n");
    return count;
}

static int
release_led_XY(struct inode *inode, struct file *file) {
    printk(KERN_DEBUG "close()\n");
    return 0;
}

struct file_operations fops_led =
{
    .open       = open_led_XY,
    .read       = read_led_XY,
    .write      = write_led_XY,
    .release    = release_led_XY
};
```

- Vous allez **enregistrer** ce driver dans ce module en ajoutant la fonction d'enregistrement dans la fonction init du module. Vous devez aussi prendre en compte les paramètres. C'est à vous de décider comment.
    - Au **début du fichier c du module**, vous déclarez une nouvelle variable statique.

      ```c
      static int major;
      ```

    - et **dans la fonction d'initialisation du module**, vous ajoutez l'enregistrement du driver,

      ```c
      major = register_chrdev(0, "led", &fops_led); // 0 est le numéro
      majeur qu'on laisse choisir par linux
      ```

    - et **dans la fonction exit du module**, vous allez décharger le driver dans ce module en ajoutant :

      ```c
      unregister_chrdev(major, "led");
      ```

Compilation

- Vous devez compiler, déplacer le module (upload du Makefile) et le charger (insmod) dans la RaspberryPi.
- Vous allez chercher dans le fichier `/proc/devices` le numéro `major` choisi par linux.
- vous allez maintenant créer le noeud dans le répertoire `/dev` et le rendre accessible par tous.
    - Dans les lignes qui suivent pensez à remplacer `major` par sa valeur !

- Le numéro mineur est 0 car il n'y a qu'une seule instance.

```
sudo mknod /dev/led0_XY c major 0
sudo chmod a+rw /dev/led0_XY
```

Questions

- Dans votre CR, je vous suggère d'expliquer chaque étape.
- **Comment savoir** que le device a été créé ?
- Le test de votre driver peut se faire par les commandes suivantes (avant de faire un vrai programme): dites ce que vous observez, en particulier, quelles opérations de votre driver sont utilisées.

```
$ echo "rien" > /dev/led0_XY
$ dd bs=1 count=1 < /dev/led0_XY
$ dmesg
```

- Nous pouvons automatiser le chargement du driver et son effacement en créant deux scripts shell:
  - Dans un fichier `insdev`

```
#!/bin/sh
module=$1
shift
/sbin/insmod ./$module.ko $* || exit 1
rm -f /dev/$module
major=$(awk "\$2==\"$module\" {print \$1;exit}" /proc/devices)
mknod /dev/$module c $major 0
chmod 666 /dev/$module
echo "=> Device /dev/$module created with major=$major"
```

  - Dans un fichier `rmdev`

```
#!/bin/sh
module=$1
/sbin/rmmod $module || exit 1
rm -f /dev/$module
echo "=> Device /dev/$module removed"
```

- Ces deux scripts doivent être copiés dans votre répertoire de la RaspberryPi. Ils doivent être exécutables et exécutés avec sudo.

```
chmod u+x insdev rmdev
```

- Pour les exécuter :

```
$ sudo ./insdev ledbp LED=2
  => Device /dev/ledbp created with major=237
$ sudo ./rmdev ledbp LED=2
  => Device /dev/ledbp removed
```

Question:

- Expliquer comment `insdev` récupère le numéro `major`

Étape 4 : accès aux GPIO depuis les fonctions du pilote

Création du driver qui accède aux GPIO

- Nous devons pouvoir accéder aux registres de configuration des GPIO.
  - Pour l'accès aux GPIOs, vous allez reprendre le principe des écritures dans les registre en passant par une structure

registre en passant par une structure.
    - Vous noterez que l'adresse physique de base des GPIO est mappée dans l'espace virtuel du noyau à l'adresse **io_addresse** et récupérer avec la macro du noyau `__io_address()`.
    - GPIO_BASE est prédéfini dans les includes à 0x20200000.

```
#include <linux/module.h>
#include <linux/init.h>
#include <asm/io.h>
#include <mach/platform.h>

static const int LED0 = 4;

struct gpio_s
{
    uint32_t gpfsel[7];
    uint32_t gpset[3];
    uint32_t gpclr[3];
    uint32_t gplev[3];
    uint32_t gpeds[3];
    uint32_t gpren[3];
    uint32_t gpfen[3];
    uint32_t gphen[3];
    uint32_t gplen[3];
    uint32_t gparen[3];
    uint32_t gpafen[3];
    uint32_t gppud[1];
    uint32_t gppudclk[3];
    uint32_t test[1];
}
volatile *gpio_regs = (struct gpio_s *)__io_address(GPIO_BASE);
```

- Les deux fonctions `gpio_fsel()` et `gpio_write()` possibles sont données juste après. Vous pouvez voir comment exploiter la structure.
- Vous devez écrire `gpio_read()`, puis invoquer ces fonctions dans les fonctions `open_ledbp()`, `read_ledbp()` et write_ledbp`.

```
static void gpio_fsel(int pin, int fun)
{
    uint32_t reg = pin / 10;
    uint32_t bit = (pin % 10) * 3;
    uint32_t mask = 0b111 << bit;
    gpio_regs->gpfsel[reg] = (gpio_regs->gpfsel[reg] & ~mask) | ((fun
<< bit) & mask);
}

static void gpio_write(int pin, bool val)
{
    if (val)
        gpio_regs->gpset[pin / 32] = (1 << (pin % 32));
    else
        gpio_regs->gpclr[pin / 32] = (1 << (pin % 32));
}
```

Travail à faire

- Ecrivez le driver complet pour les leds et le BP
- Un script de chargement (vu en cours)
- un programme de validation utilisant le driver.