Les modules Linux
premier pilote

http://doc.ubuntu-fr.org/tutoriel/tout_savoir_sur_les_modules_linux
http://pficheux.free.fr/articles/lmf/drivers/

Qu'est-ce qu'un Module ?

- Morceau de code permettant d'ajouter des fonctions au noyau
  - pilotes de périphériques
  - appels systèmes
  - protocole réseau
- Un module est chargé dynamiquement dans le noyau sans recompilation et sans redémarrage
- Un module peut aussi être déchargé
- Un module s'exécute avec les droits du noyau
- Nous allons utiliser les modules pour l'ajout d'un pilote de périphérique

Connaitre les modules installés

Pour connaitre les modules installés :

```
pi@raspberrypi ~ $ lsmod
Module                  Size  Used by
snd_bcm2835            21342  0
snd_pcm                93100  1 snd_bcm2835
snd_seq                61097  0
snd_seq_device          7209  1 snd_seq
snd_timer              23007  2 snd_pcm,snd_seq
snd                    67211  5
snd_bcm2835,snd_timer,snd_pcm,snd_seq,..
i2c_bcm2708             6200  0
spi_bcm2708             6018  0
evdev                  11000  2
joydev                  9766  0
8192cu                569633  0
uio_pdrv_genirq         3666  0
uio                     9897  1 uio_pdrv_genirq
```

Les modules peuvent dépendre les uns des autres,
→ l'ordre de chargement est important

Avoir des informations des modules installés

Pour avoir des informations sur un modules :

```
pi@raspberrypi ~ $ modinfo i2c_bcm2708
filename:       /lib/modules/3.18.7+/kernel/drivers/i2c/busses/i2c-bcm2708.ko
alias:          platform:bcm2708_i2c
license:        GPL v2
author:         Chris Boot <bootc@bootc.net>
description:    BSC controller driver for Broadcom BCM2708
srcversion:     0ACD78A7932FB3F3042F78B
alias:          of:N*T*Cbrcm,bcm2708-i2c*
depends:
intree:         Y
vermagic:       3.18.7+ preempt mod_unload modversions ARMv6
parm:           baudrate:The I2C baudrate (uint)
parm:           combined:Use combined transactions (bool)
```

En plus des commentaires, il y a des informations sur les paramètres, leur fonction et leur type

Charger / décharger un module

Commandes de chargement :

insmod <module> [module parameters]

module est le cheminom complet

modprobe -a <module> [module parameters]

Cette commande est plus intelligente, elle gère les dépendances et les alias pour ne pas avoir à taper le nom complet

Commandes de déchargement :

rmmod <module>

modprobe -r <module>

La commande **dmesg** qui affiche les messages du système informe du travail réalisé.

Passer des paramètres aux modules

Les paramètres peuvent être donné au moment de leur chargement

```
pi@raspberrypi ~ $ modinfo i2c_bcm2708
filename:       /lib/modules/3.18.7+/kernel/drivers/i2c/busses/i2c-bcm2708.ko
[...]
parm:           baudrate:The I2C baudrate (uint)
[...]
pi@raspberrypi ~ $ sudo rmmod i2c_bcm2708

pi@raspberrypi ~ $ sudo insmod /lib/modules/3.18.7+/kernel/drivers/i2c/
                                  /busses/i2c-bcm2708.ko baudrate=100000
pi@raspberrypi ~ $ dmesg
[...]
[23865.018959] bcm2708_i2c_init_pinmode(1,2)
[23865.018997] bcm2708_i2c_init_pinmode(1,3)
[23865.022374] bcm2708_i2c 20804000.i2c: BSC1 Controller at 0x20804000 (irq 79)
(baudrate 100000)
```

Charger des modules au boot

On peut demander le chargement des modules au moment du boot en mettant leur nom dans le fichier

`/etc/modules`

L'ordre est important, si le module A dépend du module B alors B doit être chargé avant A

Écrire un module

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

Contient les macros définit dans les sources du noyaux

```
MODULE_AUTHOR("skyrunner");
MODULE_DESCRIPTION("exemple de module");
MODULE_SUPPORTED_DEVICE("none");
MODULE_LICENSE("none");
```

informations récupérable par modinfo

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

fonction appelée lors du chargement du module

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

fonction appelée au déchargement du module

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

informe le noyau du nom des fonctions de chargement et de déchargement

compilation

Pour compiler, il faut les sources du noyau.

```
cc -0 -DMODULE -D__KERNEL__ -c module.c
```

Il est déconseillé de compiler sur les modules sur la raspberry, il est préférable de crosscompiler.

Makefile

```
KERNELDIR=/users/enseig/franck/peri/linux-rpi-3.18.y
CROSS_COMPILE ?= bcm2708hardfp-
obj-m += module.o

make    -C $(KERNELDIR) ARCH=arm\
        CROSS_COMPILE=$(CROSS_COMPILE) M=$(PWD) modules
```

Passage des paramètres

Les paramètres d'un module sont de type :

- short (entier court, 2 octet),
- int (entier, 4 octets),
- long (entier long) et
- charp (chaînes de caractères).

Ils sont déclarés dans le module et on informe le noyau par des macros

```
static type nom;
module_param(nom, type, permissions)
MODULE_PARM_DESC(nom, desc)
```

Exemple :

```
static int param;

module_param(param, int, 0); // 0 : ne pas exposer param dans sysfs
MODULE_PARM_DESC(param, "Un paramètre de ce module");

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

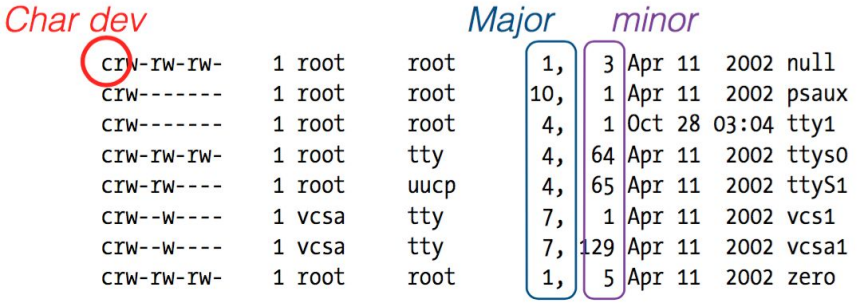
```
$ insmod ./module.o param=2
```

Driver le retour

- Un driver est un module qui permet d'échanger avec un périphérique.
- Il existe plusieurs types de driver, nous nous intéressons aux drivers de type caractère dont les échanges se font avec une granularité caractère.
- Un driver doit d'abord être enregistré dans le noyau au chargement du module, puis devra être supprimé du noyau au déchargement.
- à l'enregistrement,
  - on associe un numéro au driver (numéro majeur), on peut choisir ce numéro, ou le laisser choisir par le système.
  - on associe un liste de méthodes standards (open, read, write, ioctl,...) qui définissent le comportement du driver
- On ajoute une entrée dans le système de fichier dans le répertoire /dev en utilisant le numéro majeur qui permet de le nommer afin que l'utilisateur puisse l'ouvrir (open) et l'utiliser (read, write, ioctl)

Accès aux pilotes

- Ils sont accessibles dans le répertoire /dev
  → chaque pilote est associé à un fichier avec 2 numéros d'identifications : major & minor



Standardisation des numéros major/minor

Le fichier Documentation/devices.txt contient la liste des numéros major et minor dans linux

```
  0             Unnamed devices (e.g. non-device mounts)
                  0 = reserved as null device number
                See block major 144, 145, 146 for expansion areas.

  1 char    Memory devices
                  1 = /dev/mem          Physical memory access
                  2 = /dev/kmem         Kernel virtual memory access
                  3 = /dev/null         Null device
                  4 = /dev/port         I/O port access
[...]

  1 block       RAM disk
                  0 = /dev/ram0         First RAM disk
                  1 = /dev/ram1         Second RAM disk
                    ...
                250 = /dev/initrd       Initial RAM disk

                Older kernels had /dev/ramdisk (1, 1) here.
                /dev/initrd refers to a RAM disk which was preloaded
                by the boot loader; newer kernels use /dev/ram0 for
                the initrd.

  2 char    Pseudo-TTY masters
                  0 = /dev/ptyp0 First PTY master
                  1 = /dev/ptyp1 Second PTY master
                    ...
                255 = /dev/ptyef 256th PTY master

[...]

  2 block       Floppy disks
                  0 = /dev/fd0          Controller 0, drive 0, autodetect
                  1 = /dev/fd1          Controller 0, drive 1, autodetect
```

```
 4 char TTY devices
                  0 = /dev/tty0         Current virtual console

                  1 = /dev/tty1         First virtual console
                    ...
                 63 = /dev/tty63 63rd virtual console
                 64 = /dev/ttyS0 First UART serial port
                    ...
                255 = /dev/ttyS191      192nd UART serial port

[...]

 10 char    Non-serial mice, misc features
                  0 = /dev/logibm       Logitech bus mouse
                  1 = /dev/psaux PS/2-style mouse port
                  2 = /dev/inportbm     Microsoft Inport bus mouse
                  3 = /dev/atibm ATI XL bus mouse
                  ...

                240-254                 Reserved for local use
                255                     Reserved for MISC_DYNAMIC_MINOR

[...]

240-254 char    LOCAL/EXPERIMENTAL USE

240-254 block   LOCAL/EXPERIMENTAL USE
                Allocated for local/experimental use.  For devices not
                assigned official numbers, these ranges should be
                used in order to avoid conflicting with future assignments.
```

Enregistrement / Déchargement

```
int register_chrdev(     unsigned char major, const char *name,
                         struct file_operations *fops);
int unregister_chrdev(   unsigned int major, const char *name);
```

Ces fonctions renvoient **0** ou **>0** si tout se passe bien.

register_chrdev

- major : numéro majeur du driver, **0** si l'on souhaite une affectation dynamique.
- name : nom du périphérique qui apparaîtra dans le fichier **/proc/devices** avec le numéro majeur
- fops : pointeur vers la structure des pointeurs de fonction. Ils définissent les fonctions appelées lors des appels systèmes (read…) du côté utilisateur.

unregister_chrdev

- major : numéro majeur du driver, le même qu'utilisé dans register_chrdev
- name : nom du périphérique utilisé dans register_chrdev

Opérations

```
struct file_operations fops =
{
    .open    = my_open_function,
    .read    = my_read_function,
    .write   = my_write_function,
    .release = my_release_function  /* appelée par le dernier close */
};
```

Les prototypes sont bien sûr imposés

Il en existe beaucoup d'autres :

- `ioctl()`, `select()`, `llseek()`, `aioread()`, `aiowrite()`, `readdir()`, `poll()`, `unlocked_ioctl()`, `compar_ioctl()`, `mmap()`, `flush()`, `fsync()`, `aio_fsync()`, `aio_fsync()`, `fasync()`, `lock()`, `sendpage()`, `get_unmapped_area()`, `check_flags()`, `dir_notify()`, `flock()`, `splice_write()`, `splice_read()` `setlease()`
- Celles qui ne sont pas redéfinies sont initialisées avec une méthode par défaut qui rend `EINVAL`

Principales opérations

open
: Ouverture du périphérique. Cette méthode effectuera le plus souvent la détection et l'initialisation du hardware lorsque cela est nécessaire.

read
: Lecture des données sur le périphérique (dans l'espace du noyau) puis de faire passer ces données à l'espace utilisateur appelant.

write
: Ecriture des données de l'espace utilisateur à l'espace noyau puis d'envoyer les données au périphérique.

close
: Fermeture de l'accès. Cette méthode pourra exécuter des actions matérielles nécessaires à la fermeture du périphérique.

ioctl
: Paramètrage du périphérique depuis un programme utilisateur.

select
: permet à un programme utilisateur de se mettre en attente d'évènements (read/write/signal) sur un ensemble de descripteurs de fichiers

Implémentation des fonctions

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

static ssize_t
my_read_function(struct file *file, char *buf, size_t count, loff_t *ppos) {
    printk(KERN_DEBUG "read()\n");
    return 0;
}

static ssize_t
my_write_function(struct file *file, const char *buf, size_t count, loff_t *ppos) {
    printk(KERN_DEBUG "write()\n");
    return 0;
}

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

Structure file

Elle est définie dans **<linux/fs.h>**

La structure **file** est passée à toutes les opérations.
Les champs importants sont :

- `mode_t f_mode` — indique le mode d'ouverture du fichier
- `loff_t f_pos` — position actuelle de lecture ou d'écriture
- `unsigned int f_flags` — flags de fichiers (O_RDONLY, ...)
- `struct file_operations *f_op` — opérations associées au fichier
- `void *private_data` — pointeur vers des données privés du pilote (état)

open / release

En général, la méthode **open** réalise ces différentes opérations :

- Incrémentation du compteur d'utilisation
- Contrôle d'erreur au niveau matériel
- Initialisation du périphérique
- Identification du nombre mineur
- Allocation et remplissage de la structure privée file->private_data

Le rôle de la méthode **release** est tout simplement le contraire

- Libérer ce que open a alloué
- Éteindre la périphérique
- Décrémenter le compteur d'utilisation

création d'un noeud dans le /dev

mknod /dev/mydriver c major minor

Le numéro major est celui attribué lors de l'enregistrement,
si c'est le noyau qui l'a choisi, il se trouve dans /proc/device

```
pi@raspberrypi ~ $ cat /proc/devices
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  5 ttyprintk
  [...]
```

Le numéro minor est un numéro d'instance.

L'effacement est un simple : **sudo rm**

Test d'un driver

echo "bonjour" > /dev/mydriver permet d'invoquer write
dd bs=1 count=1 < /dev/mydriver permet d'invoquer read

```c
#include <stdio.h>
#include <unistd.h>
#include <errno.h>

int main(void)
{
     int file = open("/dev/mydriver", O_RDWR);

     if(file < 0) {
          perror("open");
          exit(errno);
     }

     write(file, "hello", 6);
     close(file);

     return 0;
}
```

Allocation mémoire

En mode noyau, l'allocation mémoire utilise **kmalloc**, la désallocation, **kfree**.
Ces fonctions ont un argument de priorité supplémentaire pour la fonction **kmalloc()**

- GFP_KERNEL : allocation normale de la mémoire du noyau
- GFP_USER : allocation mémoire pour le compte utilisateur (faible priorité)
- GFP_ATOMIC : alloue la mémoire à partir du gestionnaire d'interruptions

```c
#include <linux/slab.h>
buffer = kmalloc(64, GFP_KERNEL);
if(buffer == NULL) {
    printk(KERN_WARNING "problème kmalloc !\n");
    return -ENONEM;
}
kfree(buffer), buffer = NULL;
```

Pour déplacer des données entre l'espace noyau et utilisateur

```c
copy_from_user(unsigned long dest, unsigned long src, unsigned long len);
copy_to_user(unsigned long dest, unsigned long src, unsigned long len);
```

TME

L'objectif du TME va être de contrôler les leds et les boutons poussoir

avec un pilote

- les paramètres définiront le nombre de leds et de boutons
- write pour écrire les leds
- read pour lire les boutons