

Pilotes de périphériques orientés caractères pour Linux

2.6

Cet article a pour but de vous apporter les connaissances de base requises pour l'écriture d'un pilote de périphériques (driver) sous Linux 2.6. Une fois les concepts suivants acquis, il vous sera alors facile de savoir ce que vous devez rechercher afin d'améliorer vos connaissances dans le domaine et de réaliser des pilotes de périphériques performants. Les seules connaissances requises pour suivre ce tutoriel sont celles du langage C et les connaissances de base de l'architecture Linux.

Concept des espaces de travail

Le principe et les principales caractéristiques des trois espaces de travail suivants doivent être assimilés :

- l'espace matériel (hardware space) ;
- l'espace noyau (kernel space) ;
- l'espace utilisateur (user space).

L'espace matériel (hardware space) correspond simplement aux périphériques. Ce sont les cartes que vous connectez à votre carte mère par l'intermédiaire des ports (slots) AGP, ISA, PCI, PCI Express ou les appareils que vous reliez à votre ordinateur par l'intermédiaire des différents ports accessibles à l'arrière ou à l'avant de votre unité centrale tels que PS/2 (clavier, souris), USB, firewire, parallèle (imprimantes) et série.

L'espace noyau (kernel space) est le cœur de votre système informatique. Dans notre cas, il s'agit de Linux, soit que du code en langage C. Afin de manipuler, de modifier et de créer des nouveaux programmes, vous devez posséder le mot de passe du super utilisateur (root). Cela implique que vous savez ce que vous faites et que la moindre erreur peut-être fatale à la stabilité du système. C'est l'espace dans lequel se situent les pilotes de périphériques.

L'espace utilisateur (user space) correspond à l'espace des applications auquel l'utilisateur a accès afin d'obtenir les résultats souhaités grâce aux différents logiciels (software) installés. Une mauvaise manipulation de sa part n'engendrera pas ici une instabilité du système, à condition que ses droits d'accès soient limités par l'administrateur.

Un pilote de périphérique utilise un certain nombre de structures et de fonctions provenant du noyau. Les principales structures employées sont décrites ci-dessous.

Dans les exemples suivants, remplacez l'expression " mon_module " par le nom que vous aurez choisi pour votre pilote, un nom se rattachant à votre programme

Principales structures du noyau à utiliser

- **FILE OPERATION** : se situe dans `<linux/fs.h>`

Il s'agit d'un ensemble de pointeurs de fonctions qui correspond à une liste d'opérations qu'une application peut invoquer sur un périphérique. La structure est représentée ci-dessous et ne contient que les fonctions principales devant être implémentées par le programmeur. Cette structure doit être définie en tête de votre fichier principal. (ex : `mon_module_main.c`)

```
struct file_operations mon_module_fops = {
    .owner = THIS_MODULE,
    .read = mon_module_read,
```

```
.write = mon_module_write,
.ioctl = mon_module_ioctl,
.open = mon_module_open,
.release = mon_module_release,
...
};
```

- **FILE STRUCTURE** : se situe dans `<linux/fs.h>`

Cette structure est créée par le noyau à chaque fois que le fichier spécial (special file, voir plus loin) correspondant est ouvert et est transmis à chaque fonction qui agit sur le fichier. Le programmeur ne doit pas créer cette structure, mais juste utiliser certains éléments en fonction de ses nécessités.

```
struct file {
    const struct file_operations    *f_op;
    atomic_t                        f_count;
    unsigned int                    f_flags;
    mode_t                          f_mode;
    loff_t                          f_pos;
    void                            *private_data;
    ...
};
```

- **CDEV STRUCTURE** : se situe dans `<linux/cdev.h>`

Le noyau représente un périphérique caractère (character device) par la structure `cdev`.

```
struct cdev {
    struct kobject kobj;
    struct module *owner;
    struct file_operations *ops;
    struct list_head list;
    dev_t dev;
    unsigned int count;
};
```

Les fonctions suivantes devront être utilisées dans la fonction d'initialisation et de sortie du module.

Dans la fonction d'initialisation :

```
void cdev_init(struct cdev *cdev, struct file_operations *fops);
int cdev_add(struct cdev *cdev, dev_t num, unsigned int count);
```

Dans la fonction de sortie :

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

- **mon_module_dev STRUCTURE** : que vous créer dans votre fichier ".h"

Chaque périphérique (device) est représenté par une structure :

```
struct mon_module_dev {
    struct cdev cdev; /*Char device structure*/
};
```

```
} .....  
}
```

Cette structure doit être allouée dynamiquement et sera désallouée à la fin de l'utilisation du périphérique.

On intègre à notre propre structure, la structure `cdev`.

Quelques outils pour la programmation

Valeurs pouvant être retournées par les fonctions

Les valeurs suivantes sont utilisées dans certaines fonctions du noyau, des fonctions que vous devez implémenter et des portions de codes que vous pouvez rajouter en fonction de vos souhaits (liste non exhaustive) :

- **EINVAL** : argument est invalide.
- **EFAULT** : pointeur pointant en dehors de l'espace d'adressage accessible.
- **ENOMEM** : mémoire insuffisante pour le noyau.
- **EINTR** : l'appel système a été interrompu par un signal avant d'avoir pu lire ou écrire quoi que ce soit.
- **ENOTTY** : erreur sur le descripteur de fichier.
- **EPERM** : droits insuffisants.

Allocation dynamique de variables et de structures

La fonction utilisée pour l'allocation dynamique de structures dans le noyau ressemble à l'allocation dynamique de structures dans l'espace utilisateur. La seule différence est l'argument de priorité à spécifier.

La fonction `kfree()` libère l'espace alloué précédemment.

Prototypes : dans `<linux/slab.h>`

```
void *kmalloc(size_t size, int priority)
```

- **size** : taille en octet de l'espace mémoire à allouer ;
- **priority** :
 - GFP_KERNEL** : allocation mémoire " normale du noyau
 - P_USER** : allocation mémoire pour l'espace utilisateur
 - GFP_ATOMIC** : allocation provenant du gestionnaire d'interruptions(liste non exhaustive) ;
- **return** un pointeur de la mémoire allouée ou `NULL` si non alloué.

```
void kfree(const void *ptr)
```

ptr : pointeur retourné par la fonction `kmalloc()`

Méthodes de débogage

Il existe plusieurs manières pour le débogage de votre programme.

- Lecture directe des messages du noyau

Pour ce faire, vous devez intégrer dans vos fonctions l'équivalent du `printf()` pour l'espace utilisateur, c'est-à-dire le `printk()`.

Ensuite, ouvrez une console et tapez `tail -f /var/log/messages`. Vous aurez en " temps réel " l'affichage de vos messages.

- La spécification des flags du noyau dans certains `printk()` :
 - **KERN_EMERG** : système inutilisable ;
 - **KERN_ALERT** : action devant être effectuée immédiatement ;
 - **KERN_CRIT** : condition critique ;
 - **KERN_ERR** : condition d'erreur ;
 - **KERN_WARNING** : condition d'avertissement ;
 - **KERN_NOTICE** : normal, mais condition significative ;
 - **KERN_INFO** : information ;
 - **KERN_DEBUG** : messages de débogage.

L'affichage de ces niveaux de messages (les niveaux de priorités dépendent de votre noyau et du démon `klogd`). Les définitions des niveaux de priorités de ces types de messages sont dans le fichier `<linux/kernel.h>`. La modification des messages que vous souhaitez afficher en fonction des priorités s'effectue dans le fichier `proc/sys/kernel/printk`.

Exemple : `printk(KERN_INFO "MON_MODULE: Device removed\n")`

- L'utilisation de logiciels tels que `kdb` ou `kgdb`, voire autres...

Étapes de codage

Remarque

Une macro est une fonction noyau permettant d'identifier des valeurs particulières.

Enregistrement du pilote de périphérique en tant que module

Sous Linux, un module est une partie de code qui peut être inséré dynamiquement (le système étant allumé) dans le noyau sans recompiler l'ensemble de ce dernier. Il n'est pas nécessaire de redémarrer le système afin que notre rajout soit opérationnel. Cependant, il est également possible d'effectuer une configuration afin que notre module soit chargé dès le démarrage du système. Un module peut utiliser toute autre fonction, avoir un accès aux entrées/sorties (Input/Output) et exécuter des parties du noyau.

Concrètement, un module est un fichier binaire d'extension `.ko` qui signifie " kernel object ".

Un même pilote de périphérique peut être utilisé quel que soit le nombre de périphériques identiques connectés à votre ordinateur, ainsi que pour plusieurs types de périphériques différents.

Nous verrons par la suite comment se distingue l'utilisation du pilote.

Un module est constitué de trois parties :

- le point d'entrée ;
- le programme (pilote de périphérique) ;
- le point de sortie.

Paramètres de votre module

La commande **lsmod** vous permet d'afficher la liste des modules insérés dans le noyau. Si vous souhaitez par la suite accéder aux informations de base d'un module, il vous suffit d'utiliser la commande **modinfo nom_du_module**.

Lors de la création de votre propre module, vous pouvez donner la possibilité aux utilisateurs de connaître les informations sur votre programme.

Voici les quatre principales informations à écrire :

```
MODULE_AUTHOR("Jeremie Pilette");
MODULE_DESCRIPTION("Device driver");
MODULE_SUPPORTED_DEVICE("Reference(s) of peripheral(s)");
MODULE_LICENSE("GPL/BSD");
```

Paramètres d'enregistrement du module au sein du noyau

Chaque pilote est enregistré au sein du noyau à l'aide de deux numéros.

- le " major number " ;
- le " minor number " .

Le " major number " permet au système de fichier virtuel (Virtual File System – VFS) d'identifier le pilote. Le " minor number " permet au pilote d'identifier le périphérique utilisé.

Le programmeur décide d'allouer statiquement ou dynamiquement le " major number " pour son module. Dans le cas de l'allocation statique, il faut d'abord vérifier que le numéro que vous souhaitez affecter n'est pas déjà utilisé par d'autres modules. La liste des numéros déjà alloués est visualisable dans le fichier **documentation/devices**. L'allocation dynamique est recommandée. Elle laisse le noyau allouer le numéro en prenant compte de ceux déjà existants.

Exemple :

```
#define N_DEV 1
mon_module_major = 0;
mon_module_minor = 0;
static dev_t devno; /*To store Major and minor numbers*/

static int __init mon_module_init(void) {
    int result;
    /*Get the device numbers*/
    if(mon_module_major) {
        devno = MKDEV(mon_module_major, mon_module_minor);
        result = register_chrdev_region(devno, N_DEV, "mon_module");
        printk("MAJOR: %d\nMINOR%d\n", mon_module_major, mon_module_minor);
    }
    else {
        result = alloc_chrdev_region(&devno, mon_module_minor, N_DEV,
"mon_module");
        mon_module_major = MAJOR(devno);
        printk(KERN_INFO "MON_MODULE: MAJOR: %d\nMON_MODULE: MINOR:%d\n",
mon_module_major, mon_module _minor);
    }
}
```

```

}
if(result < 0) {
    printk(KERN_WARNING "MON_MODULE: can't get major %d\n", mon_module_major);
    return result;
}
...
}

```

Variables :

- **mon_module_major** : initialisation du major
- **mon_module_minor** : initialisation du minor
- **devno** : variable noyau de type **dev_t** contenant le " major number ", ainsi que le " minor number "
- **N_DEV** : le nombre de périphériques qui vont utiliser le pilote de périphérique.

Macro :

- **MKDEV** : macro qui associe le " major " et " minor " dans une seule variable **devno**.
- **MAJOR** : macro qui extrait le " major " de la variable **devno**.

Prototypes :

```

/*Allocation statique*/
int register_chrdev_region(dev_t devno, unsigned int count, const char *name)

```

- **devno** : la variable contenant les deux numéros ;
- **count** : le nombre de périphériques utilisés ;
- **name** : le nom de votre périphérique ;
- **return 0** si succès, valeur négative si échec.

```

/*Allocation dynamique*/
int alloc_chrdev_region(dev_t *devno, unsigned int baseminor, unsigned int
count, const char *name)

```

- **devno** : la variable contenant les deux numéros ;
- **baseminor** : la première valeur que vous souhaitez attribuer à votre périphérique. En général " 0 " ;
- **count** : le nombre de périphériques utilisés ;
- **name** : le nom de votre périphérique ;
- **return 0** si succès, valeur négative si échec.

Ajout du module au sein du noyau

Il s'agit du point d'entrée du module. Le programmeur implémente la fonction **mon_module_init()** et précise, à la fin du programme, que cette fonction correspondra au point d'entrée du module par l'intermédiaire de la macro **module_init()**. L'argument étant le nom de la fonction.

Exemple :

```

module_init(mon_module_init);

```

L'exemple suivant correspond à l'initialisation et l'ajout de la structure **cdev** pour le noyau.

Exemple :

```
static int __init mon_module_init(void) {
    struct mon_module_dev *sdev = NULL;
    /*Allocate a private structure and reference it as driver's data*/
    sdev = (struct mon_module_dev *)kmalloc(sizeof(struct mon_module_dev),
GFP_KERNEL);
    if(sdev == NULL) {
        printk(KERN_WARNING "MON_MODULE: unable to allocate private structure\n");
        return -ENOMEM;
    }
    /*Get the device numbers*/
    ...
    /*Init the cdev structure*/
    cdev_init(&sdev->cdev, &mon_module_fops);
    sdev->cdev.owner = THIS_MODULE;
    ...
    /*Add the device to the kernel*/
    cdev_add(&sdev->cdev, mon_module_minor, 1);
    ...
    return 0;
}
```

Variables :

- **Sdev** : la structure de votre périphérique que vous complétez en fonction de vos besoins.
- **THIS_MODULE** : ceci est la variable permettant de dire au noyau que cette structure ne sera utilisé que par votre module.

Prototypes :

```
void cdev_init(struct cdev *cdev, struct file_operations *fops)
```

- **cdev** : la structure **cdev** ajoutée à votre noyau, que vous avez placé dans votre structure **sdev** ;
- **fops** : correspond à **mon_module_fops**, il s'agit de votre file operation.

```
int cdev_add(struct cdev *cdev, dev_t num, unsigned int count)
```

- **cdev** : la structure **cdev** ajoutée à votre noyau, que vous avez placé dans votre structure **sdev** ;
- **num** : ceci est simplement le numéro mineur **mon_module_minor** ;
- **count** : correspond au nombre total de périphériques utilisant le pilote ;
- **return 0** si réussi, valeur négative si erreur.

Suppression du pilote au sein du noyau

Il s'agit du point de sortie module. Le programmeur implémente la fonction **mon_module_exit()** et précise, à la fin du programme, que cette fonction correspondra au point de sortie du module par

l'intermédiaire de la macro **module_exit()**. L'argument étant le nom de la fonction.

mon_module_exit() correspond à l'inverse de la fonction **mon_module_init()** et doit désallouer tout ce qui a été alloué durant l'initialisation.

Exemple :

```
module_exit(mon_module_exit);
```

Exemple :

```
static void __exit mon_module_exit(void) {
    struct mon_module_dev *sdev;
    /*Delete the cdev structure*/
    cdev_del(&sdev->cdev);
    /*Free the allocated memory*/
    kfree(sdev);
    /*Unregister Driver from the kernel*/
    unregister_chrdev_region(devno, N_DEV);
}
```

Variables :

- **sdev** : la structure de votre périphérique que vous complétez en fonction de vos besoins.

Prototypes :

```
void unregister_chrdev_region(dev_t devno, unsigned int N_DEV)
```

- **devno** : la variable contenant les deux numéros ;
- **N_DEV** : le nombre de périphériques qui ont été utilisés par le pilote de périphérique.

Ouverture du périphérique

La méthode **open()** est appelée dès la première utilisation du périphérique. C'est une étape d'initialisation du périphérique. Elle identifie le périphérique qui doit être ouvert, doit initialiser la structure **filp->private_data** (par le programmeur), qui correspond à un pointeur de structure utilisé comme " sauvegarde " de la structure principale **sdev** pour les fonctions principales du module.

L'argument **inode** est utilisé pour effectuer le lien avec la structure **cdev**. Cependant, puisqu'il est nécessaire d'obtenir la structure complète de votre périphérique (**mon_module_dev**), incluant la structure **cdev**, on utilise la macro **container_of** afin d'obtenir avec précision la structure voulue.

Exemple :

```
static int mon_module_open(struct inode *inode, struct file *filp) {
    int minor;
    struct mon_module_dev *sdev = NULL; /*Device information*/
    /*Allocate and fill any data structure to be put in filp->private_data*/
    sdev = container_of(inode->i_cdev, struct mon_module_dev, cdev);
    filp->private_data = sdev;
    return 0;
}
```

```
}
```

Variables :

- **sdev** : la structure de votre périphérique que vous complétez en fonction de vos besoins ;
- **filp->private_data** : pointeur de sauvegarde de votre structure initialisée.

Macro :

container_of(pointer, container_type, container_field) : retourne la structure de type **container_type** dont le champ **container_field** contient la valeur **pointer**.

Prototypes :

```
int mon_module_open(struct inode *inode, struct file *filp)
```

- **inode** : structure interne au noyau, non utilisée par le programmeur ;
- **filp** : correspond à **mon_module_fops**, il s'agit de votre file operation ;
- **return 0**

Fermeture du périphérique

La méthode **release()** est appelée pour la fermeture du ou des périphériques à la fin de leur utilisation. En fait, il s'agit de l'inverse de la méthode **open()**. Elle doit désallouer tout ce qui a été alloué par la méthode **open**. Rappelez-vous bien que dans l'espace noyau, tout ce qui est alloué doit être désalloué en fin d'utilisation.

Exemple :

```
int mon_module_release(struct inode *inode, struct file *filp) {  
    struct mon_module_dev *sdev;  
    filp->private_data = NULL;  
    sdev = NULL;  
    return 0;  
}
```

Prototypes :

```
int mon_module_release(struct inode *inode, struct file *filp)
```

- **inode** : structure interne au noyau, non utilisé par le programmeur ;
- **filp** : correspond à **mon_module_fops**, il s'agit de votre file operation ;
- **return 0**

Portes d'accès entre espaces

a) La liaison entre l'espace matériel et l'espace noyau s'effectue par l'intermédiaire d'un gestionnaire de mémoire (memory management). Le système d'adressage du périphérique étant

différent de celui du noyau, le gestionnaire de mémoire traduit les adresses physiques du périphérique en adresses virtuelles pour l'espace noyau. Vous devez savoir si votre périphérique effectue la liaison avec votre ordinateur par l'intermédiaire de ports d'entrées/sorties ou par l'allocation mémoire d'entrées/sorties. Le code sera alors différent en fonction de cette caractéristique.

Ports d'entrées/sorties

La fonction `request_region()` permet de demander au noyau que vous souhaitez l'utilisation de ports d'entrées/sorties pour votre périphérique.

Exemple :

```
static int __init mon_module_init(void) {
    struct mon_module_dev *sdev = NULL;
    int *port = NULL;
    int i;
    /*Allocate a private structure and reference it as driver's data*/
    ...
    /*Get the device numbers*/
    ...
    /*Init the cdev structure*/
    ...
    /*Requested I/O ports*/
    port = request_region(FIRST_PORT, PORT_NBER, 'my_device');
    if(port == NULL)
        printk(KERN_ERR 'MON_MODULE: Error port attribution\n');
    /*Add the device to the kernel*/
    ...
    return 0;
}
```

Une ressource étant allouée lors de l'initialisation, il faut donc la désallouer lors de la fin d'utilisation.

Exemple :

```
static void __exit mon_module_exit(void) {
    struct mon_module_dev *sdev;
    int i;
    /*Delete the cdev structure*/
    ...
    /*Free the allocated memory*/
    ...
    /*Release I/O ports*/
    release_region(FIRST_PORT, PORT_NBER);
    /*Unregister Driver from the kernel*/
    ...
}
```

Vous pouvez utiliser les commandes `cat`, `more` ou encore `less` pour obtenir les informations concernant l'adressage virtuel de votre système dans les fichiers `/proc/ioports` s'il s'agit de ports d'entrées/sorties ou `/proc/iomem` s'il s'agit de plages mémoire utilisées.

Variables :

- **FIRST_PORT** : numéro de port que vous aurez attribué dans un fichier .h en hexadécimal ;
- **PORT_NBER** : nombre de ports que vous aurez attribués dans un fichier .h en hexadécimal.

Prototypes :

```
<linux/ioport.h>
struct ressource *request_region(unsigned long first, unsigned long n, const
char *name)
```

- **first** : numéro du premier port souhaité ;
- **n** : nombre de ports nécessaires à votre périphérique ;
- **name** : nom de votre périphérique ;
- **return NULL** si échec.

```
void release_region(unsigned long start, unsigned long n)
```

- **start** : numéro du premier port ;
- **n** : nombre de port.

Mémoires d'entrées/sorties

La fonction **request_mem_region()** permet de demander au noyau une attribution de plage mémoire pour votre périphérique.

Exemple :

```
static int __init mon_module_init(void) {
    struct mon_module_dev *sdev = NULL;
    int *mem = NULL;
    int i;
    /*Allocate a private structure and reference it as driver's data*/
    ...
    /*Get the device numbers*/
    ...
    /*Init the cdev structure*/
    ...
    /*Requested I/O memories*/
    mem = request_mem_region(BaseAddr, LenAddr, 'my_device');
    if(mem == NULL)
        printk(KERN_ERR "'MON_MODULE: Error memory allocation\n'");
    /*Add the device to the kernel*/
    ...
    return 0;
}
```

De même, la ressource allouée doit pouvoir être désallouée.

Exemple :

```
static void __exit mon_module_exit(void) {
    struct mon_module_dev *sdev;
    int i;
    /*Delete the cdev structure*/
    ...
    /*Free the allocated memory*/
    ...
    /*Release I/O memories*/
    release_mem_region(BaseAddr, LenAddr);
    /*Unregister Driver from the kernel*/
}
```

```
...  
}
```

Variables :

- **BaseAddr** : adresse de base que vous aurez attribuée dans un fichier .h en hexadécimal ;
- **LenAddr** : longueur de la plage d'adresse souhaitée que vous aurez attribuée dans un fichier .h en hexadécimal.

Prototype :

```
<linux/ioport.h>  
struct resource *request_mem_region(unsigned long start, unsigned long length,  
char *name)
```

- **start** : adresse de base, de départ ;
- **length** : longueur de l'allocation requise ;
- **name** : nom de votre périphérique ;
- **return NULL** si échec.

Remarque:

Dans le cas d'un pilote de périphérique PCI, les fonctions **request_region()** et **request_mem_region()** sont remplacées par la fonction **ioremap()**. De même, les fonctions **release_region()** et **release_mem_region()** seront remplacées par la fonction **iounmap()**.

Prototype : <asm/io.h>

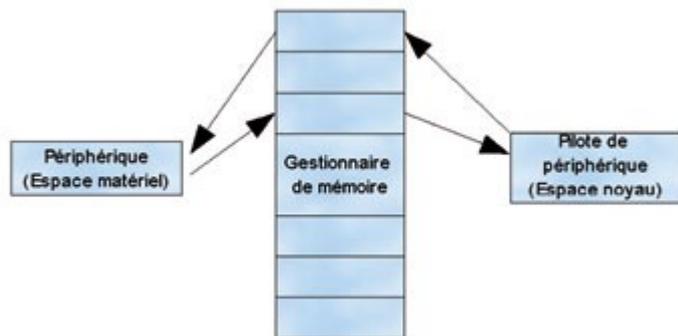
```
void * ioremap(unsigned long addr, unsigned long size)
```

- **addr** : correspond à l'adresse physique de base de votre périphérique ;
- **size** : correspond à la longueur des adresses physiques du périphérique ;
- **return** un pointeur sur les adresses virtuelles.

```
void iounmap(void *addr)
```

- **addr** : adresse virtuelle attribuée

Vous pouvez utiliser les commandes **cat**, **more** ou encore **less** pour obtenir les informations concernant l'adressage virtuel de votre système dans les fichiers **/proc/ioports** s'il s'agit de ports d'entrées/sorties ou **/proc/iomem** s'il s'agit de plages mémoires utilisées.



N'oublions pas, une fois de plus de désallouer les ressources à la fin de l'utilisation du périphérique. On utilisera alors la fonction **ionunmap()** dans la fonction **mon_module_exit()**.

Exemple :

```
Zeus:/proc# cat ioprots
0000-001f : dma1
0020-0021 : pic1
0040-0043 : timer0
0050-0053 : timer1
0060-006f : keyboard
0070-0077 : rtc
0080-008f : dma page reg
00a0-00a1 : pic2
00c0-00df : dma2

Zeus:/proc# cat iomem
00000000-0009fbff : System RAM
00000000-00000000 : Crash kernel
0009fc00-0009ffff : reserved
000a0000-000bffff : Video RAM area
000c0000-000ccfff : Video ROM
000cd000-000d17ff : Adapter ROM
000f0000-000fffff : System ROM
00100000-3ffbffff : System RAM
00200000-003e1f04 : Kernel code
```

b) La liaison entre l'espace noyau et l'espace utilisateur s'effectue par l'intermédiaire d'un fichier spécial (special file ou inode). Les fichiers spéciaux sont visualisables dans le répertoire **/dev/**. En listant le contenu du répertoire, nous obtenons les principales informations relatives aux périphériques.

Exemple :

```
Zeus:/dev# ls -l
crw-rw-rw- 1 root root 5, 0 2007-08-28 11:50 tty
crw-rw-rw- 1 root root 4, 0 2007-08-28 11:50 tty0
crw-rw-rw- 1 root root 4, 1 2007-08-28 11:50 tty1
crw-rw-rw- 1 root root 4, 10 2007-08-28 11:50 tty10
crw-rw-rw- 1 root root 4, 11 2007-08-28 11:50 tty11
crw-rw-rw- 1 root root 4, 12 2007-08-28 11:50 tty12
crw-rw-rw- 1 root root 4, 13 2007-08-28 11:50 tty13
crw-rw-rw- 1 root root 4, 14 2007-08-28 11:50 tty14
crw-rw-rw- 1 root root 4, 15 2007-08-28 11:50 tty15
crw-rw-rw- 1 root root 4, 16 2007-08-28 11:50 tty16
brw-rw-rw- 1 root disk 3, 0 2007-08-28 11:50 hda
brw-rw-rw- 1 root disk 3, 1 2007-08-28 11:50 hda1
brw-rw-rw- 1 root disk 3, 2 2007-08-28 11:50 hda2
brw-rw-rw- 1 root disk 3, 3 2007-08-28 11:50 hda3
brw-rw-rw- 1 root disk 3, 5 2007-08-28 11:50 hda5
brw-rw-rw- 1 root disk 3, 6 2007-08-28 11:50 hda6
brw-rw-rw- 1 root disk 3, 64 2007-08-28 11:50 hdb
brw-rw-rw- 1 root cdrom 22, 0 2007-08-28 11:50 hdc
```

La première lettre correspond au type du périphérique. " c " pour un périphérique de type caractère, " b " pour un périphérique de type bloc. Les nombres au milieu correspondent respectivement au numéro majeur (major number) et au numéro mineur (minor number). Comme précisé précédemment, un même numéro majeur permet d'identifier le pilote utilisé et le numéro mineur identifie le périphérique. Enfin, la dernière colonne correspond au nom du fichier spécial rattaché à la paire de numéro.

Transfert de données entre espace matériel et espace noyau

Afin de coder cette partie, vous devez posséder une bonne documentation de votre périphérique pour connaître la liste des registres accessibles en lecture et/ou écriture, ainsi que le nombre de lignes d'interruption que le périphérique peut mettre à disposition.

C'est la fonction d'interruption qui gère ces transferts de données.

Le périphérique génère une interruption sur le processeur de votre machine lorsque le périphérique

souhaite communiquer des informations à l'utilisateur ou lorsque le périphérique reçoit ou émet une information. Cela dépend du périphérique. Là où les lignes d'interruption doivent être déclarée(s) lors de l'initialisation du module. (<linux/sched.h>)

Exemple :

```
static int __init mon_module_init(void) {
    struct mon_module_dev *sdev = NULL;
    int req;
    /*Allocate a private structure and reference it as driver's data*/
    ...
    /*Get the device numbers*/
    ...
    /*Init the cdev structure*/
    ...
    /*Requested I/O ports or I/O memories*/
    ...
    /*Add the device to the kernel*/
    ...
    /*Declare the interrupt line*/
    req = request_irq(sdev->irq, mon_module_interrupt, IRQF_SHARED,
"Device_name", sdev);
    if(req < 0) {
        printk(KERN_WARNING "MON_MODULE: Can't get assigned irq %d\n", sdev->irq);
        return 0;
    }
}
```

Prototype :

```
int request_irq(unsigned int irq, mon_module_interrupt, unsigned long
flags, const char *dev_name, void *dev_id)
```

- **irq** : numéro de la ligne d'interruption ;
- **mon_module_interrupt** : nom du prototype de la fonction d'interruption ;
- **flags** : permet de définir le type de l'interruption ;
 - **SA_INTERRUPT** : pour interruption "rapide"
 - **SA_SHIRQ** : pour ligne d'interruption partagée
 - (liste non exhaustive)
- **dev_name** : le nom de votre périphérique ;
- **dev_id** : numéro d'identification du périphérique (pour interruption partagée).

Voici les principales exigences de votre fonction d'interruption :

- Identifier la raison de l'interruption.
Les raisons peuvent être les suivantes : arrivée d'une donnée, fin d'une transmission, indication d'erreurs...
- Changer la valeur du bit signalant l'interruption afin de permettre d'autres interruptions.
Il s'agit ici de modifier la valeur d'un bit du principal registre de configuration des interruptions du périphérique.
- Appeler les fonctions du noyau pour la lecture ou l'écriture de données en fonction de la source d'interruption.
Les fonctions appelées dépendent de la taille d'adressage des registres. Il peut s'agir de 8, 16 ou 32 bits. Dans ce cas, la valeur est précisée dans la fonction à utiliser.

Remarque:

Pour connaître la ou les lignes d'interruptions de l'ensemble de vos périphériques, vous pouvez visualiser le fichier `/proc/interrupts`.

Exemple :

```
irqreturn_t mon_module_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    struct mon_module_dev *sdev = (mon_module_dev *)dev_id;
    /*Le code dépend de votre périphérique*/
    /*Identifier la raison de l'interruption*/
    /*Changer la valeur du bit signalant l'interruption*/
    /*Appeler les fonctions du noyau pour la lecture ou l'écriture des données*/

    return IRQ_HANDLED;
}
```

Prototypes :

Interruption

```
irqreturn_t mon_module_interrupt(int irq, void *dev_id, struct pt_regs *regs)
```

- **irq** : le numéro de l'interruption concernée ;
- **dev_id** : utilisé pour les lignes d'interruption partagées. Identifiant complémentaire de l'interruption lorsque la ligne d'interruption est partagée ;
- **Regs** : rarement utilisé. Non utilisé dans notre cas (registres stockés sur la pile) ;
- **return IRQ_HANDLED** (si l'interruption a bien été gérée).

Lecture/écriture

Si vous utilisez des pointeurs d'adresses :

Lecture:

```
int ioread8(void __iomem *addr)
int ioread16(void __iomem *addr)
int ioread32(void __iomem *addr)
addr: pointeur de l'adresse mémoire concernée
Ecriture:
void iowrite8(u8 val, void __iomem *addr)
void iowrite16(u16 val, void __iomem *addr)
void iowrite32(u32 val, void __iomem *addr)
    val: Valeur à copier à l'adresse mémoire "addr"
    addr: pointeur de l'adresse mémoire concernée
```

Si vous utiliser les adresses directement :

Lecture:

```
unsigned int readb(addr)
unsigned int readw(addr)
unsigned int readl(addr)
addr: pointeur de l'adresse mémoire concernée
Ecriture:
void writeb(unsigned value, addr)
void writew(unsigned value, addr)
```

```
void writel(unsigned value, addr)
val: Valeur à copier à l'adresse mémoire "addr"
addr: pointeur de l'adresse mémoire concernée
```

Là où les lignes d'interruptions sont une ressource allouée, il ne faut pas oublier de la désallouer lors de l'arrêt du périphérique.

Exemple :

```
static void __exit mon_module_exit(void) {
    struct mon_module_dev *sdev;
    /*Delete the cdev structure*/
    ...
    /*Free the allocated memory*/
    ...
    /*Free the interrupt line*/
    free_irq(sdev->irq, sdev);
    /*Free the virtual memory addresses*/
    ...
    /*Release I/O ports or I/O memories*/
    ...
    /*Unregister Driver from the kernel*/
    ...
}
```

Prototype :

```
void free_irq(unsigned int irq, void *dev_id)
```

- **irq** : numéro de la ligne d'interruption ;
- **dev_id** : identifiant complémentaire de la ligne d'interruption.

Transfert de données entre espace noyau et espace utilisateur

Le programmeur doit coder les deux méthodes `mon_module_read()` et `mon_module_write()` en y intégrant les appels aux fonctions suivantes, respectivement `copy_to_user()` et `copy_from_user()`, qui effectuent la copie des données et non le transfert.

Prototypes :

```
ssize_t mon_module_read(struct file *filp, char __user *buffer, size_t count,
loff_t *offp)
```

- **filp** : correspond à `mon_module_fops`, il s'agit de votre file operation ;
- **buffer** : mémoire tampon de l'espace utilisateur où seront placées les données ;
- **count** : taille des données à transférer ;
- **offp** : pointeur des données en cours de traitement ;
- **return** le nombre d'octets lus.

```
ssize_t mon_module_write(struct file *filp, const char __user *buffer, size_t
count, loff_t *offp)
```

- **filp** : correspond à **mon_module_fops**, il s'agit de votre file operation ;
- **buffer** : mémoire tampon de l'espace utilisateur où sont les données à copier dans l'espace noyau ;
- **count** : taille des données à transférer ;
- **offp** : pointeur des données en cours de traitement ;
- **return** le nombre d'octets écrits.

```
unsigned long copy_to_user(void __user *to, const void *from, unsigned long
count)
```

- **to** : mémoire tampon de l'espace utilisateur ;
- **from** : mémoire tampon de l'espace noyau ;
- **count** : nombre d'octets à copier en une fois, à l'appel de la fonction **copy_to_user** ;
- **return** la quantité de mémoire restante à être copiée.

```
unsigned long copy_from_user(void *to, const void __user *from, unsigned long
count)
```

- **to** : mémoire tampon de l'espace noyau ;
- **from** : mémoire tampon de l'espace utilisateur ;
- **count** : nombre d'octets à copier en une fois, à l'appel de la fonction **copy_from_user** ;
- **return** la quantité de mémoire restante à être copiée.

Création des fichiers spéciaux

Les fichiers spéciaux sont créés à l'aide de la commande **mknod**.

Exemple :

```
mknod /dev/mon_module
```

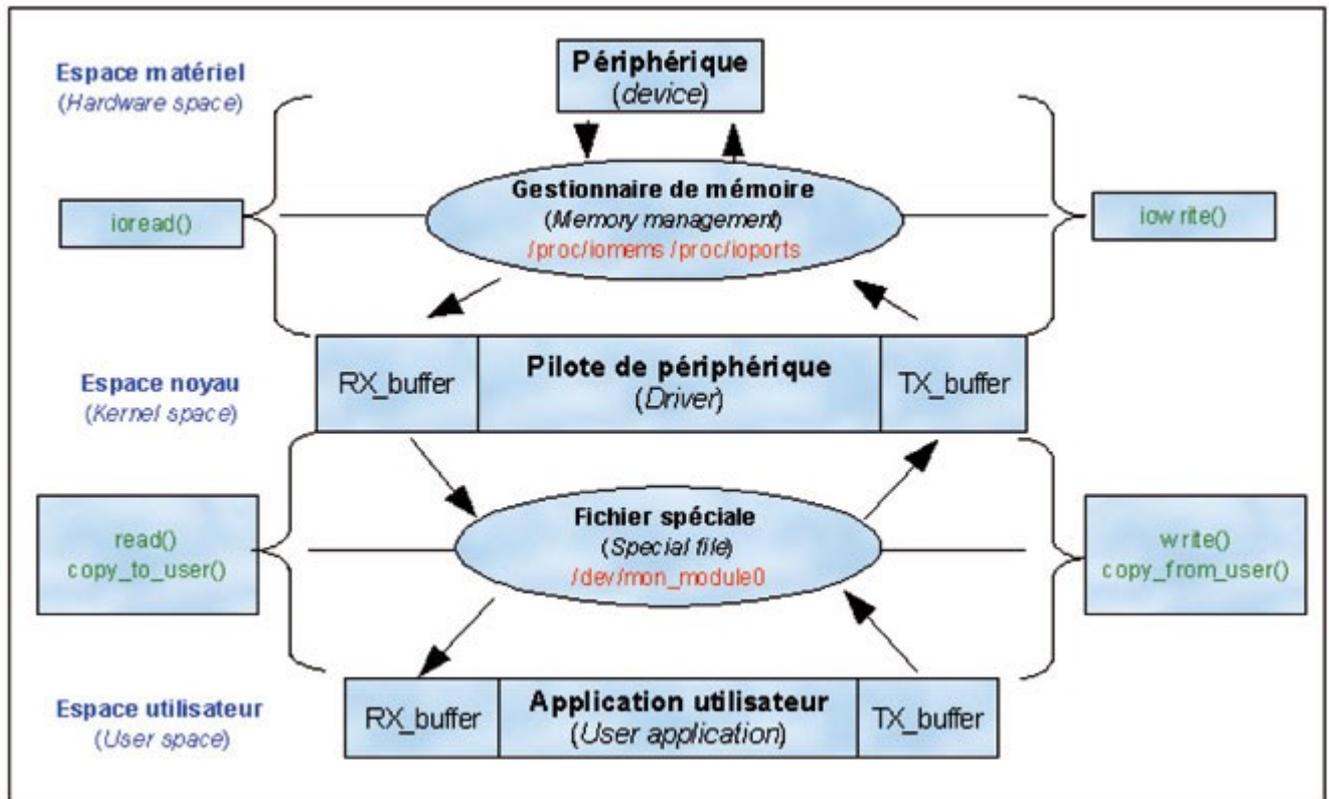
Remarque

La commande **rmnod** supprime le fichier spécial.

Afin que la création de vos fichiers spéciaux soit automatique, vous devez utiliser un script shell. Il faut créer deux fichiers **.sh** permettant la création de **mon_module_mkdev.sh** et la suppression de **mon_module_rmdev.sh** du fichier spécial. Pour cela, vous devez connaître les bases du langage shell (ou vous servir d'un script déjà existant en l'adaptant à votre module).

Synthèse

Le schéma suivant synthétise le fonctionnement global du pilote de périphérique.



Les mémoires tampons (buffers) "RX" et "TX" des espaces noyau et utilisateur doivent être créés par l'intermédiaire de structures que vous aurez définies au préalable vous-même dans un fichier .h.

Contrôle du périphérique

La fonction `ioctl()` sert au contrôle de votre périphérique.

Le codage de cette fonction requiert une bonne connaissance de votre périphérique concernant les possibilités que celui-ci met à votre disposition. Dans le cadre d'un lecteur de CD-ROM, vous pouvez, par la commande `eject` ouvrir votre lecteur depuis votre espace utilisateur (dans la console). Cette ouverture est gérée par la fonction `ioctl()`. Cette fonction est également utilisée pour l'implémentation de programmes qui vous permettent de lire ou d'écrire sur les registres de votre périphérique.

L'implémentation correspond à un `switch()` définissant les différents cas voulus.

Il faut attribuer ce qu'on appelle un "magic number" qui correspond à l'identification de contrôle de votre module. Ce numéro magique doit être unique parmi l'ensemble des numéros magiques de tout les modules du noyau. Par conséquent, vous devez vérifier si le numéro que vous souhaitez attribuer n'est pas déjà réservé dans le fichier `Documentation/ioctl-number.txt` du répertoire de votre noyau.

Ensuite, chaque commande souhaitée doit correspondre à un numéro additionnel.

Dans votre fichier .h, vous devez attribuer un nom de variable pour chaque commande que vous souhaitez avoir la possibilité d'effectuer sur votre périphérique, en précisant s'il s'agit d'une lecture et/ou écriture à l'aide des macros suivantes :

- `_IO(magic number, numero1)`

- `_IOW(magic number, numero2, type)`
- `_IOR(magic number, numero3, type)`
- `_IORW(magic number, numero4, type)`

Exemple :

```
/*Magic number*/
#define MON_MODULE_MAGIC 0xcd /*A vous de voir pour cette valeur
hexadécimale*/
/*ioctl commands*/
#define OPEN_CD          _IO(MON_MODULE_MAGIC, 100)
#define READ_IR         _IOR(MON_MODULE_MAGIC, 101, int)
...
```

Exemple :

```
int mon_module_ioctl(struct inode *inode, struct file *filp, unsigned int cmd,
unsigned long arg) {
    struct pcl04plus_dev *sdev = filp->private_data;
    ...
    switch(cmd) {
        /*Ouverture tiroir CD-ROM*/
        case OPEN_CD:
            ...
            break;
        /*Lecture registre d'interruption*/
        case READ_IR:
            ...
            break;
    }
    return -ENOTTY;
}
```

Prototypes :

```
int ioctl(struct inode *inode, struct file *fops, unsigned int cmd, unsigned
long arg)
```

- **inode** : structure interne au noyau, non utilisée par le programmeur ;
- **fops** : correspond à `mon_module_fops`, il s'agit de votre file operation ;
- **cmd** : il s'agit de la variable attribuée pour chaque commande dans le fichier .h ;
- **arg** : argument optionnel de validité ;
- **return** `-ENOTTY`.

Compilation du programme (fichier Makefile)

Depuis la version du noyau 2.6, le fichier Makefile est différent d'un fichier Makefile classique d'un simple programme de l'espace utilisateur.

Voici un exemple contenant les éléments principaux :

Exemple :

```
#Makefile for Linux driver 2.6.x
```

```

OBJS= mon_module.o
mon_module_OBJS= mon_module_main.o \
                  mon_module_xxxx.o \
                  mon_module_yyyy.o

#=====
# Select kernel source folder
#=====
KERNEL_SRC=/lib/modules/$(shell uname -r)/build
#=====
# Compilation
#=====
#If KERNELRELEASE is defined, we've been invoked from the kernel
#build system and can use its language.
ifneq ($(KERNELRELEASE),)
    obj-m := $(OBJS)
    mon_module-objs := $(mon_module_OBJS)
#else otherwise we were called directly from the command line;
#invoke the kernel build system
else
    KERNELDIR ?= $(KERNEL_SRC)
    PWD := $(shell pwd)
default:
    $(MAKE) -C $(KERNELDIR) SUBDIRS=$(PWD) modules
endif
#=====
# Delete files
#=====
clean:
    $(MAKE) -C $(KERNELDIR) SUBDIRS=$(PWD) clean
    rm -f Module.symvers

```

OBJS représente le nom de votre module après compilation. Ici, votre module sera le fichier **mon_module.ko**.

À chaque nouveau fichier .c, vous devez rajouter le même nom de fichier en remplaçant l'extension .c par .o. Ici, il s'agit des fichiers **mon_module_xxxx.o** et **mon_module_yyyy.o**.

Configuration du système pour l'insertion du module

Il existe deux commandes pour insérer son module dans le noyau.

La commande **insmod mon_module** permet uniquement d'insérer votre module sans se préoccuper des possibles dépendances avec d'autres modules du noyau. Pour retirer le module, on utilise alors la commande **rmmod mon_module**.

La commande **modprobe mon_module** permet non seulement d'insérer le module, mais de satisfaire les dépendances. On retire alors le module avec la commande **modprobe -r mon_module**. Nous avons vu précédemment que la création des fichiers spéciaux était effectuée par un fichier script. Lors de l'insertion du module, le fichier script n'est pas appelé. La configuration suivante va vous permettre de faire le lien entre l'insertion du module et l'appel automatique au fichier script de création des fichiers spéciaux.

Après avoir compilé votre pilote avec succès, le fichier **mon_module.ko** représentant votre module est le fichier devant être inséré.

1) Vous devez créer un lien symbolique dans le répertoire **/lib/modules/version_du_noyau/** :

```
ln -s /chemin_depuis_la_racine/mon_module.ko /lib/modules/version_du_noyau/
```

2) Tapez la commande **depmod** afin que le noyau prenne en compte le lien symbolique :

```
depmod
```

3) Le module et le lien symbolique doivent être autorisés en exécution :

```
chmod +x mon_module.ko  
chmod +x /lib/modules/version_du_noyau/mon_module.ko
```

4) Éditez un fichier **nom_repertoire** dans **/etc/modprobe.d/** et insérez-y les lignes suivantes :

```
install mon_module /sbin/modprobe --ignore-install mon_module  
remove mon_module /sbin/modprobe -r --ignore-remove mon_module
```

5) Créez les liens symboliques pour les deux scripts **mon_module_mkdev.sh** et **mon_module_rmdev.sh**.

```
ln -s /chemin_depuis_la_racine/mon_module_mkdev.sh /etc/nom_repertoire/  
ln -s /chemin_depuis_la_racine/mon_module_rmdev.sh /etc/nom_repertoire/
```

Récapitulatif de quelques fichiers " header " à inclure :

```
#include <linux/module.h> /*Pour la création du module*/  
#include <linux/init.h> /*Pour l'utilisation du module*/  
#include <linux/kernel.h> /*Pour l'utilisation de printk()*/  
#include <linux/fs.h> /*Pour la structure file_operations*/  
#include <linux/types.h> /*Pour l'utilisation de types comme size_t*/  
#include <linux/cdev.h> /*Pour la structure cdev*/  
#include <linux/errno.h> /*Pour la liste de erreurs*/  
#include <linux/sched.h> /*Pour la déclaration de la ligne d'interruption*/  
#include <linux/interrupt.h> /*Pour la fonction d'interruption*/  
#include <linux/slab.h> /*Pour l'allocation mémoire kmalloc()/kfree()*/  
#include <asm/io.h> /*Pour les fonctions ioremap/iounmap*/  
#include <asm/uaccess.h> /*Pour les fonctions copy_{from,to}_user()*/  
#include <asm/ioctl.h> /*Pour la fonction ioctl()*/
```

Remarques générales :

N'oubliez pas de faire un fichier **README** afin d'y préciser ce que vous jugerez utile comme les améliorations possibles, les capacités de votre pilote, les tests effectués, la date de création, la méthode d'installation, etc.

Je précise une fois de plus que cet article a pour but de vous donner les bases de l'écriture d'un driver en mode caractère. Il serait possible d'écrire un livre pour chaque étape décrite, afin d'y préciser tous les détails nécessaires.

Afin que votre pilote puisse être modifié ou amélioré par un maximum d'autres programmeurs, il

est nécessaire d'y insérer un maximum de commentaires, afin d'expliquer clairement vos codes. Et ce, bien sûr, en anglais.

Il existe différents types de pilotes de périphérique orientés caractères tels que les pilotes PCI, les pilotes USB, les pilotes ISA, les pilotes pour ports parallèles... Il y a également d'autres critères pouvant être utilisés en fonction de votre architecture, tels qu'un processeur 64 bits ou encore des processeurs multicœur (ex : AMD X2, Intel core 2 Duo).

J'espère que cet article vous sera bénéfique pour vous lancer dans la conception de programmes pouvant améliorer le noyau Linux, des programmes malheureusement encore réputés trop difficiles pour se lancer dans leur développement.