

Driver en mode caractères

Christophe BLAESS

christophe.blaess@logilin.fr

<https://www.blaess.fr/christophe/>



Ingénierie et formations sur Linux et les logiciels libres
<https://www.logilin.fr>

Enregistrement d'un driver caractères.....	3
Principe des pilotes de périphérique.....	3
Enregistrement d'un driver caractère.....	5
Méthodes du driver.....	7
Modèles de périphériques.....	9
Classes prédéfinies.....	13
Méthodes du driver.....	14
Méthodes de base.....	14
Travaux pratiques : écriture de driver caractère simple.....	16
Fonctions de paramétrage.....	17
Synchronisation des appels-système.....	19
Nécessité d'une synchronisation.....	19
Utilisation des mutex.....	20
Travaux pratiques : implémentation d'un ioctl().....	21

Ce support de formation est distribué sous licence **Creative Commons 4.0**



(Attribution - Partage dans les mêmes conditions).

Vous êtes libres de copier et partager ce document, en mentionnant son origine. Si vous l'intégrez dans un contenu plus vaste, ce dernier devra être distribué avec les mêmes droits.

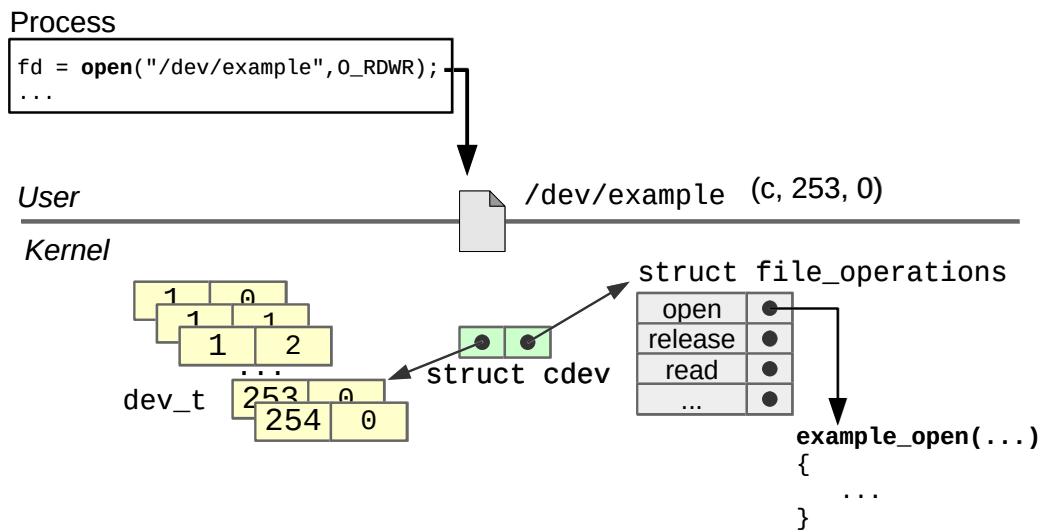
Ce cours a été rédigé en utilisant des logiciels libres sur système d'exploitation Linux :

- *LibreOffice Writer* pour le support et la mise en page
- *LibreOffice Draw* pour les dessins vectoriels
- *Gimp* pour les images bitmap

v. 9.12

Enregistrement d'un driver caractères

Principe des pilotes de périphérique



L'accès (lecture, écriture, etc.) à un **fichier spécial représentant un périphérique** est orienté par le noyau vers le driver associé.

Le fichier spécial est doté de deux numéros : majeur et mineur.

Le numéro **majeur** permet de sélectionner le pilote de périphérique.

Le numéro **mineur** équivaut à un numéro d'instance pour le pilote.

Deux types de périphériques :

- **bloc** : accessible par blocs, supporte un système de fichiers
- **caractère** : tous les périphériques accessibles octet par octet

```
$ ls -l /dev/hda2  
brw-rw---- 1 root disk      8, 0    jan 30  07:38 /dev/sda2  
  
$ ls -l /dev/ttyS4  
crw-rw---- 1 root dialout  4, 68   jan 30  07:38 /dev/ttyS4
```

Voir aussi

[RUBINI 2005] : Alessandro Rubini, Jonathan Corbet, Greg Kroah-Hartman - *Linux Device Drivers* - Troisième édition - O'Reilly & Associates, 2005.

À vous...

Le fichier `/dev/urandom` est un fichier spécial représentant un périphérique virtuel en mode caractère, de numéros majeur 1 et mineur 9. Lorsqu'on lit ce fichier, le noyau nous renvoie des octets aléatoirement choisis entre 0 et 255.

Observez le contenu de ce fichier avec une commande comme :

```
# hexdump /dev/urandom
```

Créez un fichier spécial manuellement, puis consultez son contenu :

```
# mknod /dev/alea c 1 9  
# hexdump /dev/alea
```

Le comportement est-il le même ?

Essayez de créer ce fichier spécial dans votre répertoire personnel, puis dans `/proc`. Y parvenez-vous ? Pourquoi ?

Enregistrement d'un driver caractère

```
<linux/fs.h>
```

Pour manipuler le numéro d'un périphérique indépendamment de la représentation binaire des numéros majeur et mineur, on utilise un type **dev_t** :

```
unsigned int MAJOR(dev_t num);  
unsigned int MINOR(dev_t num);  
dev_t MKDEV(unsigned int major, unsigned int minor);
```

```
int alloc_chrdev_region(dev_t *dev,  
                        unsigned int first_minor,  
                        unsigned int nb, const char *name);
```

```
int register_chrdev_region (dev_t first, unsigned int nb,  
                           const char *name);
```

```
void unregister_chrdev_region(dev_t first,  
                             unsigned int nb);
```

Si le numéro majeur est connu à l'avance, on peut réserver une plage de numéros mineurs avec **register_chrdev_region()**. Si le numéro majeur n'est pas connu à l'avance, on demande au noyau d'en attribuer un avec **alloc_chrdev_region()**. Le *name* apparaît alors dans `/proc/devices` avec son numéro majeur.

On peut créer un fichier spécial avec la commande shell : **mknod** *nom* b|c *major minor*

Voir aussi

- fichier `Documentation/admin-guide/devices.txt` dans les sources du noyau.

À vous...

Chargez le module **example-IV-01**.

Vérifiez s'il apparaît dans `/proc/devices`.

Créez un fichier spécial avec ce numéro majeur (et le numéro mineur zéro). Essayez d'y accéder avec **cat**. Que se passe-t-il ?

Méthodes du driver

```
<linux/cdev.h>
```

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

```
int cdev_add (struct cdev *, dev_t first,  
              unsigned int nb);  
void cdev_del (struct cdev *);
```

```
struct file_operations {  
    struct module * owner;  
    ...  
    int      (*open)    (struct inode *, struct file *);  
    int      (*release) (struct inode *, struct file *);  
    loff_t   (*llseek) (struct file *, loff_t, int);  
    ssize_t  (*read)    (struct file *, void *, size_t, loff_t *);  
    ssize_t  (*write)   (struct file *, void *, size_t, loff_t *);  
    long     (*ioctl)   (struct file *, int, long);  
    int      (*poll)    (struct file *, poll_table_struct *);  
    ...  
}
```

La structure **cdev** représente un périphérique caractère en associant un identifiant **dev_t** avec une structure **file_operations** regroupant les méthodes proposées par le driver.

La structure **file_operations** regroupe les opérations possibles sur un fichier. Pour un périphérique caractère, on implémente les méthodes **open()**, **release()**, **read()**, **write()**, et éventuellement **ioctl()**, **poll()**, et **mmap()**.

Le champ **owner** de la structure est souvent initialisé statiquement avec la constante symbolique **THIS_MODULE**. Ainsi on ne pourra pas retirer le module tant que le fichier est ouvert.

À vous...

Chargez le module **example-IV-02**.

Vérifiez que son numéro majeur soit bien le même que celui du fichier spécial créé précédemment.

Essayez de lire le contenu de ce fichier spécial avec un cat.

Que se passe-t-il ? Pourquoi ? Comment vérifier le comportement du driver ?

Modèles de périphériques

Avec l'augmentation du nombre de périphériques supportés par Linux, la gestion classique basée sur les numéros majeurs et mineurs posaient plusieurs problèmes d'administration.

- Le numéro majeur ne fournit aucune information quant au type de périphérique représenté. La répartition en modes *caractère* et *blocs* n'est pas suffisante.
- Il est difficile de savoir quels périphériques sont présents sur un système à un moment donné.
- Suivant l'ordre de reconnaissance, un même périphérique peut changer de numéro majeur si celui-ci est attribué dynamiquement entre deux démarrages du système.
- Le répertoire `/dev` d'un système générique devrait contenir plusieurs milliers d'entrées.

Le projet *Devfs* permettait de créer automatiquement les entrées de `/dev` lorsqu'un numéro majeur était attribué.

Linux offre un *modèle de périphériques* permettant d'améliorer leur visibilité depuis l'espace utilisateur.

Le noyau gère les objets suivants :

- les **périphériques** (*devices*)
- les **bus** qui rattachent les périphériques au système
- les **pilotes** (*drivers*) qui gèrent les périphériques
- les **classes** qui regroupent les périphériques par fonctionnalités
- les **sous-systèmes** sont des vues spécifiques sur l'organisation du système.

Le système de fichiers virtuel *Sysfs*, monté normalement sur `/sys`, fournit un accès depuis l'espace utilisateur au modèle des périphériques.

Le projet *Udev* s'appuie sur *Sysfs* pour fournir des fonctionnalités améliorées par rapport à *Devfs*.

Une option de compilation permet de disposer d'un *devtmpfs* : ram-disque monté automatiquement par le kernel sur `/dev` et dans lequel il crée automatiquement les fichiers spéciaux. *Udev* peut servir à modifier les noms et les droits des fichiers spéciaux.

Le but du *modèle de driver* est d'obtenir une structuration des données internes, permettant de représenter précisément le système :

- quels périphériques sont présents et à quels bus ils sont reliés ;
- quels bus sont disponibles ainsi que leurs interconnexions (*bridges*) ;
- les types (*classes*) des périphériques présents ;
- l'état d'alimentation (marche, arrêt, veille, etc.) des périphériques.

Attention, l'*API* de *Sysfs* est exportée avec le status `GPL_ONLY`, elle ne peut être employée directement que par des modules sous licence *GPL*.

Utilisation du modèle de devices

Création / destruction d'une classe personnalisée :

```
<linux/device.h>

struct class *class_create (const char *name);

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

On teste le retour de `class_create()` avec la macro `IS_ERR()`.

Ajout / suppression de périphériques dans une classe :

```
<linux/device.h>

struct device *device_create (struct class *cls,
                               struct device *parent,
                               dev_t dev,
                               void *driver_data,
                               const char *format...);

void device_destroy (struct class *cls,
                     dev_t dev);
```

Avant Linux 6.4, la fonction `class_create()` prenait deux arguments : la structure du module qui la créait (`THIS_MODULE`) et le nom de la classe.

La fonction `device_create()` prend en argument un *format* semblable à `printf` suivi d'arguments correspondant aux indicateurs de conversion (`%d` etc.)

À vous...

Chargez le module **example-IV-03** qui réalise le même travail que l'exemple précédent, tout en s'inscrivant dans le modèle des drivers.

Vérifiez que la classe se trouve bien dans `/sys/class` et qu'on y trouve un répertoire pour notre driver.

Si le démon `udev` est actif ou si l'option `DevTmpFs` est présente à la compilation du kernel, vous verrez également une entrée apparaître dans le répertoire `/dev`.

Classes prédéfinies

Il existe plusieurs classes prédéfinies, l'une des plus utilisées pour les périphériques caractères est « misc ».

```
<linux/miscdevice.h>
```

```
struct miscdevice {  
    int      minor;  
    char     *name;  
    struct file_operations *fops;  
    [...]   
};
```

```
int misc_register (struct miscdevice *misc);  
int misc_deregister (struct miscdevice *misc);
```

Si aucune autre action n'est nécessaire lors de l'initialisation du module, on peut utiliser :

```
module_misc_device(struct miscdevice misc)
```

À vous...

Chargez le module **exemple-IV-04**.

Voyez-vous une nouvelle entrée dans `/sys/class/misc/` ?

Quel fichier spécial est apparu dans `/dev` ?

Méthodes du driver

Méthodes de base

```
int      open  (struct inode *ind, struct file *filp);
int      release(struct inode *ind, struct file *filp);

struct inode {
    unsigned long    i_ino;
    uid_t            i_uid;
    gid_t             i_gid;
    struct timespec  i_atime;
    struct timespec  i_mtime;
    struct timespec  i_ctime;
    ...
}
```

```
ssize_t read (struct file *filp, char __user *buffer,
              size_t lg, loff_t *off);
ssize_t write(struct file *filp, const char __user *buffer,
              size_t lg, loff_t *off);

struct file {
    ...
    void * private_data;
}
```

Si la méthode `open()` n'est pas définie (le champ vaut `NULL`), l'ouverture réussit toujours. De même la méthode `release()` peut rester indéfinie, elle réussira toujours.

La structure **inode** (<linux/fs.h>) contient des informations sur le fichier spécial. Pour en connaître les numéros majeur et mineur, on utilise les macros **imajor(inode)** et **iminor(inode)**.

La structure **file** que l'on retrouve dans toutes les méthodes de **file_operations** contient un champ **private_data** que le driver peut utiliser pour enregistrer des informations.

Le *buffer* des méthodes `read()` et `write()` est situé dans l'**espace utilisateur**. Il faut utiliser les fonctions `copy_to_user()` ou `copy_from_user()` vues dans le chapitre précédent.

Le paramètre de type `loff_t *` des méthodes `read()` et `write()` permet d'organiser des lectures/écritures successives.

À vous...

Regardez le code source du module **example-IV-05.c** et déterminez ce qu'il doit afficher lors d'un appel à la méthode `read()`.

Chargez le module `example-IV-05.ko` et invoquez `cat` sur le fichier spécial correspondant.

Vérifiez qu'il fonctionne comme vous l'attendez.

Travaux pratiques : écriture de driver caractère simple

Énoncés et solutions sur <https://www.logilin.fr/tp/ild>.

Exercice IV-1 : méthode `read()` d'un *driver* en mode *character*.

Exercice IV-2 : méthode `write()` d'un *driver* en mode *character*.

Fonctions de paramétrage

```
<sys/ioctl.h>
```

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

```
<asm/ioctl.h>
```

```
_IO (magic, num);           /* pas d'argument          */  
_IOR(magic, num, type_arg); /* lecture d'un parametre */  
_IOW(magic, num, type_arg); /* ecriture d'un parametre */
```

```
struct file_operations {  
    ...  
    long (*unlocked_ioctl)(struct file *filp,  
                           unsigned int cmd,  
                           unsigned long arg);
```

```
_IOC_TYPE(cmd); /* extrait le numero magique */  
_IOC_DIR(cmd)   /* sens : _IOC_READ ou _IOC_WRITE */  
_IOC_NR(cmd);   /* extrait le numéro de commande */
```

Les *ioctl* sont des commandes que l'on peut transmettre à un driver, afin de paramétrer le fonctionnement d'un périphérique. Certains *ioctl* prennent un argument (généralement un pointeur), d'autres n'en prennent pas. La liste des *ioctl* reconnus par un driver est stockée dans un fichier d'en-tête, et les valeurs sont construites à partir des macros `_IO()`, `_IOR()`, `_IOW()`.

L'argument `cmd` intègre :

- un *numéro magique* représentant le driver (valeur sur un octet choisie parmi celles libres du fichier `linux/Documentation/ioctl-numbers.txt`) ;
- un numéro de commande interne au driver ;
- un indicateur de direction (lecture/écriture) ;
- la taille de l'argument à transférer.

Dans l'implémentation de la méthode `ioctl()`, on décodera la valeur de commande reçue à l'aide des macros `_IOC_TYPE()`, `_IOC_DIR()` et `_IOC_NR`.

À vous...

Observez le code source, puis chargez le module **example-IV-06**.

Examinez le contenu du fichier `example-IV-06.h`, et regardez l'implémentation du fichier `ioctl-example-IV-06.c` qui assurera l'invocation de `ioctl()` depuis l'espace utilisateur.

Observez l'affichage lors d'un `cat` sur le fichier spécial associé au module `example-IV-06`.

Modifiez le paramètre d'affichage du *PPID* avec `ioctl-example-IV-06`.

Vérifiez à nouveau l'affichage obtenu lors d'un `cat`.

Synchronisation des appels-système

Nécessité d'une synchronisation

L'accès incontrôlé à des variables globales peut conduire à une corruption des données dans plusieurs cas :

- modifications simultanées de la même variable globale ;
- modification d'une variable globale durant une lecture non-atomique de cette même variable.

Cette situation peut se présenter dans plusieurs situations :

- sur les systèmes multi-processeurs, les mêmes données sont accessibles simultanément par différents CPU ;
- sur un système uni-processeur dont le noyau est compilé en mode *préemptible*, une commutation de tâche peut intervenir durant l'exécution d'un appel-système ;
- sur tous systèmes, une interruption peut survenir durant un appel-système et le gestionnaire de cette interruption peut accéder aux mêmes données que l'appel-système.

La protection des variables globales se fait différemment en fonction des accès possibles.

À vous...

Chargez le module **exemple-IV-07**.

Exécutez dans deux terminaux différents, deux `cat` sur le fichier spécial. Voyez-vous apparaître des caractères '#' indiquant une collision d'accès à la variable globale ?

Dans quelle condition de fonctionnement vous trouvez-vous ? Multi-processeurs (multi-coeurs) ou uni-processeur ? Noyau préemptible, non préemptible ?

Utilisation des mutex

```
<linux/mutex.h>
```

```
DEFINE_MUTEX (name);
```

```
void mutex_init(struct mutex *name);
```

```
void mutex_lock (struct mutex *name);
```

```
int mutex_lock_interruptible (struct mutex *name);
```

```
int mutex_trylock (struct mutex *name);
```

```
int mutex_is_locked (struct mutex *name);
```

Attention, `mutex_lock()` et `mutex_lock_interruptible()` sont susceptibles d'endormir la tâche `current`, aussi ne doit-on pas les utiliser dans des contextes d'interruption.

Si `mutex_lock_interruptible()` renvoie une valeur non-nulle, il faut quitter l'appel-système en renvoyant l'erreur -ERESTARTSYS.

```
void mutex_unlock (struct mutex *name);
```

`mutex_lock()` endort le processus appelant en sommeil ininterrompible (profond)

`mutex_lock_interruptible()` endort ce processus en sommeil interrompible (léger).

À vous...

Chargez le module **exemple-IV-08**, et exécutez les mêmes tests que pour le module précédent.

Voyez-vous toujours des collisions ?

Travaux pratiques : implémentation d'un ioctl()

Énoncés et solutions sur <https://www.logilin.fr/tp/ild>.

Exercice IV-3 : méthode `ioctl()` d'un *driver* en mode *character*.

Exercice IV-4 : multiples instances d'un *driver* en mode *character*.