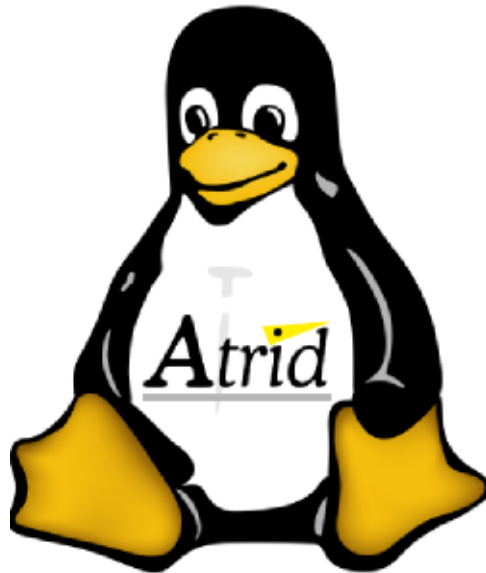


# Drivers et modules sous Linux



**ATRID**

**Mr. Gilles POLART-DONAT**

Directeur Technique  
Atrid systèmes

## **Drivers et modules sous Linux**

par ATRID

par Mr. Gilles POLART-DONAT

Copyright © janvier 2000 par ATRID Systèmes

Ce document peut être librement lu, stocké, reproduit, diffusé, traduit et cité par tous moyens et sur tous supports aux conditions suivantes:

- Tout lecteur ou utilisateur de ce document reconnaît avoir pris connaissance de ce qu'aucune garantie n'est donnée quant à son contenu, à tous points de vue, notamment véricité, précision et adéquation pour toute utilisation ;
- il n'est procédé à aucune modification autre que cosmétique, changement de format de représentation, traduction, correction d'une erreur de syntaxe évidente, ou en accord avec les clauses ci-dessous ;
- le nom, le logo et les coordonnées de l'auteur devront être préservés sur toutes les versions dérivées du document à tous les endroits où ils apparaissent dans l'original, les noms et logos d'autres contributeurs ne pourront pas apparaître dans une taille supérieure à celle des auteurs précédents, des commentaires ou additions peuvent être insérés à condition d'apparaître clairement comme tels ;
- les traductions ou fragments doivent faire clairement référence à une copie originale complète, si possible à une copie facilement accessible ;
- les traductions et les commentaires ou ajouts insérés doivent être datés et leur(s) auteur(s) doi(ven)t être identifiable(s) (éventuellement au travers d'un alias) ;
- cette licence est préservée et s'applique à l'ensemble du document et des modifications et ajouts éventuels (sauf en cas de citation courte), quelqu'en soit le format de représentation ;
- quel que soit le mode de stockage, reproduction ou diffusion, toute version imprimée doit contenir une référence à une version numérique librement accessible au moment de la première diffusion de la version imprimée, toute personne ayant accès à une version numérisée de ce document doit pouvoir en faire une copie numérisée dans un format directement utilisable et si possible éditable, suivant les standards publics, et publiquement documentés en usage ;

La transmission de ce document à un tiers se fait avec transmission de cette licence, sans modification, et en particulier sans addition de clause ou contrainte nouvelle, explicite ou implicite, liée ou non à cette transmission. En particulier, en cas d'inclusion dans une base de données ou une collection, le propriétaire ou l'exploitant de la base ou de la collection s'interdit tout droit de regard lié à ce stockage et concernant l'utilisation qui pourrait être faite du document après extraction de la base ou de la collection, seul ou en relation avec d'autres documents.

Toute incompatibilité des clauses ci-dessus avec des dispositions ou contraintes légales, contractuelles ou judiciaires implique une limitation correspondante : droit de lecture, utilisation ou redistribution verbatim ou modifiée du document.

Adapté de la licence Licence LLDD v1, octobre 1997, Libre reproduction © Copyright Bernard Lang [F1450324322014] URL :

<http://pauillac.inria.fr/~lang/licence/lldd.html>

### Historique des versions

Version 1.010 avril 2000

Version du noyau 2.2.14

---

## Table des matières

<b>1. Introduction</b> .....	<b>4</b>
1.1. Présentation .....	4
1.2. Intégration du driver au noyau .....	5
<b>2. Squelette en mode caractère</b> .....	<b>6</b>
2.1. Entête.....	6
2.2. skel_init .....	6
2.3. skel_open .....	6
2.4. skel_release .....	7
2.5. skel_read .....	7
2.6. skel_write .....	8
2.7. skel_ioctl .....	8
2.8. skel_llseek .....	9
2.9. skel_poll .....	9
2.10. Autre fonctions .....	10
<b>3. API noyau</b> .....	<b>11</b>
3.1. Passage de paramètres .....	11
3.2. Gestion des interruptions.....	11
3.2.1. request_irq.....	12
3.2.2. free_irq.....	12
3.2.3. Blocage des interruptions .....	12
3.2.4. init_bh et remove_bh.....	13
3.2.5. mark_bh.....	13
3.2.6. enable_bh et disable_bh .....	13
3.2.7. queue_task.....	13
3.3. Gestion des processus.....	14
3.3.1. sleep_on .....	14
3.3.2. wake_up .....	15
3.4. DMA.....	15
<b>Bibliographie</b> .....	<b>18</b>

## Chapitre 1. Introduction

Ce document présente le fonctionnement des pilotes de périphériques (*device drivers*) sous Linux et les fonctions du noyau disponibles pour le programmeur pour l'écriture des pilotes.

### 1.1. Présentation

Linux banalise l'accès aux périphériques en proposant une interface unifiée à travers des fichiers spéciaux pour permettre d'utiliser les mêmes commandes sur les périphériques que sur les fichiers standards. Les fichiers spéciaux sont, généralement, créés dans le répertoire `/dev`; ils ont une taille nulle (mais occupent un i-noeud) et des droits d'accès comme n'importe quel autre fichier.

Il existe deux types de fichiers spéciaux :

- les fichiers en mode caractère sont des fichiers à accès direct au périphérique. On écrit et on lit octet par octet.
- les fichiers en mode bloc utilisent le tampon mémoire et implémentent des accès par blocs de données

Les exemples ci-dessous montrent deux fichiers en mode caractère (`ttys0` et `ttys1`) et deux fichiers en mode bloc (`sda` et `sdb`). Ils sont différenciés par la lettre précédant les droits d'accès (`c` pour le mode caractère et `b` pour le mode bloc)

```
crw-r--r-- 1 root root 4, 64 May 5 1998 /dev/ttyS0
crw-r--r-- 1 root root 4, 65 May 5 1998 /dev/ttyS1
brw-rw---- 1 root disk 8, 0 May 5 1998 /dev/sda
brw-rw---- 1 root disk 8, 16 May 5 1998 /dev/sdb
```

Les deux valeurs numériques entre le groupe du propriétaire et la date sont le *major number* et le *minor number*. Le *major number* est un index dans une table interne du noyau contenant les structures d'accès aux fonctions du pilote. Il existe en fait deux tables, une pour chaque type de pilote. On peut donc avoir un pilote en mode bloc ayant le même *major number* qu'un pilote en mode caractère. Le *minor number* est une valeur de configuration pour le pilote qui caractérise le périphérique particulier pointé par ce fichier. Par exemple, pour le pilote des ports série (`major number = 4`), le port 0 a un *minor number* de 64 et le port 1 un *minor number* de 65.

Ecrire un pilote de périphérique sous Linux, consiste à implémenter les appels systèmes de gestion de fichiers (`open`, `close`, `read`, `write`, `ioctl`, ...). et à enregistrer les fonctions dans le noyau par l'intermédiaire d'une structure de description des points d'entrées du pilote.

Cette structure est du type donné ci-dessous :

```
struct file_operations {
    loff_t (*lseek) (struct file *, loff_t, int);
    ssize_t (*read) ( struct file *, char *, size_t, loff_t *);
    ssize_t (*write) ( struct file *, const char *, size_t, loff_t *);
    int (*readdir) (struct inode *, struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
```

```
int (*mmap) ( struct file *, struct vm_area_struct *);
int (*open) (struct inode *, struct file *);
int (*flush) (struct file *);
void (*release) (struct inode *, struct file *);
int (*fsync) ( struct file *, struct dentry *);
int (*fasync) (int, struct file *, int);
int (*check_media_change) (kdev_t dev);
int (*revalidate) (kdev_t dev);
int (*lock) ( struct file *, int, struct file_lock *);
};
```

Lorsqu'un processus effectue un appel système sur un fichier spécial, le noyau appelle la fonction correspondante du pilote en fonction du *major number* du pilote. Lorsque cette fonction n'existe pas (valeur nulle dans la structure) le noyau retourne une erreur.

## 1.2. Intégration du driver au noyau

Pour permettre l'utilisation du pilote et la génération du noyau avec les outils standards de Linux, il faut modifier quelques fichiers sources de Linux :

- `drivers/char/mem.c` : il faut ajouter le code suivant dans la fonction `__initfunc(int chr_dev_init(void))` :

```
#if defined (CONFIG_SKEL)
skel_init();
#endif
```

- `drivers/char/Config.in` : ce fichier contient les paramètres pour les questions lors de la configuration du noyau. Il faut ajouter une ligne du type :

```
tristate "Skell driver support" CONFIG_SKEL
```

- `driver/char/Makefile` : il faut ajouter le fichier objet à la liste des drivers ou des modules :

```
ifeq ($(CONFIG_SKEL),y)
L_OBJS += skel_char.o
else
ifeq ($(CONFIG_SKEL),m)
M_OBJS += skel_char.o
endif
endif
```

Le fichier source du pilote doit être copié dans le répertoire `drivers/char` pour être compilé par la procédure de génération standard.

## Chapitre 2. Squelette en mode caractère

Cette section présente le squelette d'un pilote en mode caractère. Toutes les fonctions sont présentées, avec les appels les plus courants des fonctions du noyau.

### 2.1. Entête

Le pilote commence par les inclusions des fichiers systèmes :

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

### 2.2. skel\_init

Cette fonction est appelée par le noyau lors de son initialisation pour initialiser le pilote dans le cas où le pilote est lié statiquement au noyau. Elle doit procéder à l'installation du pilote dans la table du noyau. Cette fonction initialise le pilote et vérifie la présence du matériel. Généralement elle affiche quelques messages concernant son travail.

```
unsigned long skel_init()
{
    printk("skel_init() Skel driver, no hardware detected\n");
    major = register_chrdev(SKEL_MAJOR, "skel", &skel_fops);
}
```

La fonction `register_chrdev` installe les fonctions du pilote dans la table `chrdevs` en fonction du *major number*. Si celui-ci est nul, elle alloue un numéro dynamiquement. Le nom donné permet d'accéder aux informations du pilote par le système de fichiers `/proc`. Une entrée est créée dans `/proc/devices` avec le nom donné.

```
printk("skel_init() major number = %d\n", major);
return 0;
}
```

### 2.3. skel\_open

Cette fonction effectue l'initialisation du périphérique lors de l'ouverture du fichier spécial par l'appel système `open`. Elle reçoit la description du i-noeud et peut donc obtenir les informations sur le périphérique. Cette fonction permet d'allouer les ressources de fonctionnement pour une instance du pilote.

```
static int skel_open(struct inode * inode, struct file * file)
{
    unsigned int minor = MINOR(inode->i_rdev);
}
```

La macro `MINOR` retourne le *minor number* du fichier spécial. Cela permet de connaître le périphérique effectivement utilisé.

```
}
```

La structure `file` est surtout utilisée lors de l'écriture de systèmes de fichiers.

Si cette fonction n'est pas présente pour le pilote et que le fichier spécial existe, l'appel système réussira.

## 2.4. `skel_release`

Cette fonction est appelée lorsqu'un processus utilise l'appel système `close`. Elle permet d'effectuer les opérations de remise à zéro du périphérique et de libérer la mémoire utilisée.

```
static int skel_release(struct inode * inode, struct file * file)
{
    unsigned int minor = MINOR(inode->i_rdev);
}
```

Cette fonction n'est appelée que sur le dernier `close` d'un processus si celui-ci a ouvert plusieurs fois le périphérique.

Si cette fonction n'est pas présente pour le pilote, le fichier sera fermé normalement.

## 2.5. `skel_read`

Cette fonction est appelée sur une lecture par un processus. Elle doit déclencher une lecture de données sur le périphérique et copie les données dans l'espace utilisateur. Il n'est pas possible de copier des données de l'espace d'adressage du noyau (*kernel space*) à l'espace utilisateur (*user space*) par une affectation simple. Pour cela il faut utiliser des fonctions qui vérifient les adresses passées en paramètre et font le transfert entre les deux espaces d'adressage.

Pour mieux comprendre ce problème, il peut être nécessaire de revoir le fonctionnement de la mémoire avec Linux. Les processus Linux utilisent un espace d'adressage virtuel, c'est à dire que chaque processus pense détenir l'ensemble des ressources mémoires adressables par le processeur (4Go pour les machines x386). Cet espace est découpé en zones (code, données, pile) et est positionné en mémoire physique par le noyau en fonction des besoins et de la mémoire physique disponible. Le noyau maintient à jour une table d'équivalence entre la mémoire virtuelle de chaque processus et la mémoire physique avec une granularité d'une page mémoire (4 Ko sur les x86). Si le noyau a besoin de place dans la mémoire physique, il peut sauvegarder des pages, sur le disque dur, dans la zone d'échanges (*swap*). Lorsque le pilote effectue un accès aux données de l'espace utilisateur, il doit utiliser des fonctions spéciales qui interprètent les adresses virtuelles de l'espace utilisateur.

La fonction de lecture est du type :

```
ssize_t (*skel_read) ( struct file * file, char * buf, size_t length, loff_t * off-
set)
{
    ssize_t nb_bytes_read = 0;
    // read device to local_buf
    if (copy_to_user (buf, local_buf, length) )
    {
        return -EFAULT;
    }
    nb_bytes_read = length;
    return nb_bytes_read;
}
```

Le paramètre *offset* est utilisé par l'appel système `pread` qui permet de lire un fichier à partir d'une position donnée. Ce paramètre peut être ignoré dans un pilote de périphérique.

Si cette fonction n'est pas présente pour le pilote, l'appel système échouera avec une erreur `EINVAL`.

## 2.6. skel\_write

Cette fonction est appelée pour une écriture sur le périphérique. Son fonctionnement est identique à la fonction de lecture.

```
ssize_t (*skel_write) ( struct file * file, char * buf, size_t length, loff_t * off-
set)
{
    ssize_t nb_bytes_write = 0;
    if (copy_from_user (local_buf, buf, length) )
    {
        return -EFAULT;
    }
    // write device from local_buf
    nb_bytes_write = length;
    return nb_bytes_write;
}
```

Le paramètre *offset* est utilisé par l'appel système `pwrite` qui permet d'écrire un fichier à partir d'une position donnée. Ce paramètre peut être ignoré dans un pilote de périphérique.

Si cette fonction n'est pas présente pour le pilote, l'appel système échouera avec une erreur `EINVAL`.

## 2.7. skel\_ioctl

Cette fonction permet de passer des commandes particulières au périphérique. Les commandes sont codées par un entier et peuvent avoir un argument. Cet argument peut être un entier ou un pointeur sur une structure de

données. Les commandes FIOCLEX, FIONCLEX, FIONBIO et FIOASYNC sont gérées directement par le noyau. Cela correspond respectivement aux options `close-on-exec`, `O_NONBLOCK` et `O_SYNC` des fichiers.

Une fonction type ressemble à :

```
int (*ioctl) (struct inode * inode, struct file * file , unsigned int cmd , unsigned long arg)
{
    int retval;
    switch(cmd)
    {
        case ... :
            ...
            break;
        case ... :
            ...
            break;
        default :
            retval = -EINVAL;
            break;
    }
    return retval;
}
```

Si la fonction n'existe pas pour le pilote, l'appel système échouera avec une erreur `EINVAL`.

## 2.8. `skel_llseek`

Cette fonction permet de positionner le pointeur de lecture ou d'écriture. Elle n'est généralement pas utilisée pour les pilotes de périphérique. Si cette fonction n'est pas présente, le noyau réalisera l'option par défaut qui consiste à positionner la valeur `f_pos` de la structure `file`.

```
loff_t (*llseek) (struct file * file, loff_t offset, int whence)
{
    loff_t offs;
    ...
    return offs;
}
```

## 2.9. `skel_poll`

Cette fonction permet d'implémenter l'appel système `poll` (`select`) pour le pilote.

```
unsigned int (*poll)(struct file *, struct poll_table_struct *);
```

## 2.10. Autre fonctions

Les autres fonctions de la structure `file_operations` ne sont généralement pas implémentées pour un pilote de périphérique en mode caractère.

## Chapitre 3. API noyau

### 3.1. Passage de paramètres

Le noyau Linux permet de passer des paramètres à un module lors de son installation pour définir des valeurs propres à une utilisation. Un module peut ainsi récupérer une adresse d'E/S, un numéro d'IRQ ou un canal DMA.

Lors de l'installation, on écrit :

```
insmod skel_module param1=10 param2=toto
```

Pour décoder ces valeurs, on doit déclarer les noms des paramètres grâce à une macro dans l'entête du module :

```
MODULE_PARM(param1, "5");  
MODULE_PARM(param2, "titi");
```

Cette déclaration définit les variables `param1` et `param2` avec leurs valeurs par défaut.

Dans la routine d'initialisation du module on peut faire appel à ces variables et tester les valeurs pour voir si elles sont acceptables.

```
int init_module()  
{  
    ...  
    if ( (param1 <=0) && (param1 > 10) )  
    {  
        printk("param1 value must be between 1 and 10\n");  
        return -EINVAL;  
    }  
    ...  
}
```

### 3.2. Gestion des interruptions

Les interruptions permettent de gérer les périphériques de manière asynchrone, sans avoir besoin de boucler en attendant qu'un registre soit à la bonne valeur ou qu'une opération soit terminée. Lorsqu'un périphérique a besoin de signaler un événement au noyau (par exemple une touche du clavier est appuyée), il valide un signal électrique sur le circuit contrôleur d'interruption, celui-ci interrompt le processeur qui lance la procédure associée à cet événement (par exemple lecture de la touche actionnée). Chaque pilote peut enregistrer une fonction pour la gestion des interruptions du périphérique qu'il gère, si celui-ci peut en générer.

Lorsqu'une interruption est traitée par la fonction gestionnaire d'interruptions, le noyau bloque l'arrivée de nouvelles interruptions pour le niveau considéré ou pour tous les niveaux selon le type de fonction déclaré (voir plus

loin). Le gestionnaire d'interruption doit donc effectuer le minimum de choses pour ne pas perturber le fonctionnement du système.

Linux propose un système à deux niveaux pour la gestion des interruptions :

- le premier niveau (`request_irq` et `free_irq`) enregistre le gestionnaire d'interruption privilégié fonctionnant avec les interruptions masquées
- le deuxième niveau est appelé par le noyau lorsque la gestion des interruption a été revalidée.

### 3.2.1. `request_irq`

Cette fonction permet d'enregistrer un gestionnaire pour le traitement d'une interruption :

```
int request_irq(unsigned int irq, void (*handler)(int, void *, struct pt_regs *), unsigned long irqflags, constant char *devname, void dev_id);
```

La valeur `irqflags` permet de spécifier le type de gestionnaire d'interruption que l'on installe :

- `SA_INTERRUPT` signifie que la fonction est du type *fast interrupt handler*, c'est à dire qu'elle sera appelée avec toutes les interruptions bloquées.
- `SA_SHIRQ` signifie que le niveau d'interruption peut être partagé par d'autres périphériques, le paramètre `dev_id` permettant de différencier les périphériques.

Le paramètre `devname` permet de spécifier le nom dans `/proc/interrupts`

La fonction de gestion reçoit en paramètres le numéro d'interruption, l'identificateur du périphérique et une structure contenant les registres au moment de l'appel.

### 3.2.2. `free_irq`

Cette fonction permet de détacher une fonction de gestion d'un niveau d'interruption. Elle est généralement utilisée dans la fonction `cleanup_module`.

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

### 3.2.3. Blocage des interruptions

Plusieurs macros permettent de gérer l'arrivée des interruptions. Elles sont définies dans le fichier `asm/system.h`. Ci dessous un exemple d'utilisation de ces macros.

```
unsigned long flags;  
...  
save_flags(flags); cli();
```

```
...  
// partie critique  
...  
restore_flags(flags);
```

### 3.2.4. `init_bh` et `remove_bh`

La fonction `init_bh`registre un gestionnaire d'interruption de second niveau. Le premier paramètre est le numéro de gestionnaire pour le noyau. Il doit être unique pour le système et peut être ajouté à l'énumération anonyme dans le fichier `include/linux/interrupts.h`.

```
void init_bh(int nr, void (*routine)(void));
```

La fonction `remove_bh` permet de supprimer l'enregistrement du gestionnaire.

```
void init_bh(int nr);
```

### 3.2.5. `mark_bh`

Cette fonction permet de demander au noyau de faire tourner le gestionnaire de second niveau aussitôt que possible. Celui-ci effectue l'activation des gestionnaires de second niveau dans l'ordonnanceur ou en retour d'un appel système.

```
void mark_bh(int nr);
```

### 3.2.6. `enable_bh` et `disable_bh`

Ces fonctions permettent d'invalider et de valider (respectivement) un gestionnaire de second niveau.

```
void disable_bh(int nr);
```

```
void enable_bh(int nr);
```

### 3.2.7. `queue_task`

L'utilisation des gestionnaires de second niveau pose le problème du numéro unique qui doit être déclaré dans le fichier `include/linux/interrupts.h`. Pour palier ce problème, le noyau propose l'utilisation de *files de tâches* associées à un numéro donné. Il existe quatre files prédéfinies (`tq_timer`, `tq_immediate`, `tq_scheduler` et

`tq_disk`) et il est possible de définir sa propre file. Cela permet de plus de raccrocher plusieurs tâches de second niveau.

Il faut déclarer une variable du type `tq_struct` et l'initialiser :

```
static void skel_isr_2(void *);
static struct tq_struct task = {
    NULL,
    /* élément suivant dans la liste */
    0,
    /* le drapeau signifiant que l'on est insérer dans une file */
    skel_isr_2,
    /* fonction de traitement */
    NULL
    /* données pour la fonction */
};
```

Dans le gestionnaire d'interruption, il faut enregistrer la tâche et appeler le gestionnaire de second niveau :

```
void skel_isr(int irq, void * data, struct pt_regs * regs)
{
    ...
    queue_task(&task, &tq_immediate);
    mark_bh(IMMEDIATE_BH);
}
```

## 3.3. Gestion des processus

Un pilote de périphérique peut être amené à faire attendre un processus tant qu'une opération n'est pas exécutée. Le noyau fournit quelques fonctions pour mettre en sommeil et réveiller un processus.

### 3.3.1. `sleep_on`

Plusieurs fonctions permettent de mettre en attente un processus :

```
void sleep_on(struct wait_queue** condition);
```

```
long sleep_on_timeout(struct wait_queue** p, signed long timeout);
```

```
void interruptible_sleep_on(struct wait_queue** p);
```

```
long interruptible_sleep_on_timeout(struct wait_queue** p, signed long timeout);
```

Ces fonctions prennent en paramètre un pointeur sur un pointeur sur une structure `wait_queue`. Pour mettre en sommeil un processus, il faut exécuter la séquence ci-dessous :

```
struct wait_queue *wait_cond = NULL;
...
sleep_on(&wait_cond);
// Ce code sera exécuté au réveil
...
```

Lorsqu'un processus est en sommeil, il ne peut pas être interrompu. Pour permettre l'interruption, il faut utiliser la fonction `interruptible_sleep_on`. Dans le cas où un appel système bloquant est interrompu, il ne faut pas oublier de positionner l'erreur à `EINTR`.

Ces fonctions peuvent être appelées en précisant un délai maximum d'attente. Ce délai est exprimé en multiple de la constante `HZ` qui donne le nombre de tics horloge par seconde. Par exemple, pour un délai d'attente de 500 ms, on aura :

```
long timeout = 500 * HZ / 1000;
timeout = interruptible_sleep_on_timeout(&wait_cond, timeout);
```

La valeur en retour donne le temps restant à courir pour l'attente si celle-ci est interrompue avant son terme.

Avant de mettre un processus en attente, il faut vérifier si le drapeau `O_NONBLOCK` n'est pas positionné pour le processus dans `file->f_flags`.

### 3.3.2. `wake_up`

Cette fonction permet de réveiller un processus en sommeil. Elle est appelée en passant le pointeur sur la condition d'attente. Cette fonction est généralement appelée à partir du gestionnaire d'interruption pour débloquer un processus dans un appel système.

Par exemple, pour la gestion du clavier, un module peut enregistrer une fonction de gestion d'interruption qui sera activée par appui d'une touche. Lorsqu'un processus attend une entrée clavier, il effectue l'appel système `read` qui appelle la fonction de lecture du module. Celle-ci vérifie qu'il n'y a pas de caractère en attente, et s'endort en attendant que l'utilisateur tape une touche. Dès qu'une touche est pressée, la routine d'interruption est activée : elle lit le caractère et appelle la fonction `wake_up`. Le processus termine l'appel système en récupérant son caractère.

Cet exemple est simplifié (pour ne pas dire simpliste), mais le principe est là !

## 3.4. DMA

Le DMA (*Direct Memory Access*) désigne un transfert direct entre un périphérique et la mémoire du système. Il est géré par un circuit particulier de la machine, collaborant avec le processeur pour se partager les cycles d'accès

à la RAM du système. Ce type de transfert est intéressant pour des transferts de gros paquets de données vers ou en provenance des périphérique. Prenons l'exemple d'une carte son : pour un transfert en stéréo avec une bonne qualité sonore, on a besoin de  $2 \times 16 \text{ bits} \times 40 \text{ KHz}$  (environ) soit 80 000 mots de 16 bits par seconde. Si le processeur devait gérer cela en transfert mot à mot, on aurait une interruption toute les 12 microsecondes. Cela est irréaliste, à la fois en terme d'occupation CPU et de déterminisme du système. Un transfert DMA va permettre l'envoi de N secondes de musique en *une seule opération* vers la carte pendant que le processeur effectue d'autres opérations et que la carte son restitue les N secondes précédemment envoyées.

Le principe des opérations consiste à :

- copier les données dans une zone de la mémoire physique allouée à cet effet
- armer le transfert DMA
- demander l'émission d'une interruption en fin de transfert
- lancer le transfert.

L'architecture du PC entraine quelques limitations dans l'utilisation du DMA :

- un transfert DMA ne peut s'effectuer qu'à partir d'une zone de mémoire physique située dans les premiers 16 MB de la mémoire
- un transfert ne peut pas traverser une limite de page (64Ko pour les transfert 8 bits et 128 Ko pour les transferts 16 bits)
- les canaux 0 à 3 sont pour les DMA 8bits (le canal 4 n'est pas utilisable)
- les canaux 5 à 7 sont pour les DMA 16 bits, les adresses doivent être alignées sur une frontière de mot.

L'allocation d'un tampon de mémoire pour le DMA s'effectue avec la fonction `kmalloc` d'allocation dynamique de mémoire en précisant que l'on veut de la mémoire pour un DMA.

```
void *dma_buf;  
dma_buf = kmalloc(buffer_size, GFP_BUFFER | GFP_DMA);
```

La mémoire obtenue devra être libérée avec un appel à `kfree` :

```
kfree(dma_buf);
```

La fonction `set_dma_mode` permet de définir le type de DMA que l'on veut exécuter :

```
void set_dma_mode(unsigned int dmanr, char mode);
```

avec `mode` pouvant prendre les valeurs :

- `DMA_MODE_READ` pour un DMA en lecture (périphérique vers mémoire)
- `DMA_MODE_WRITE` pour un DMA en écriture.

La fonction `set_dma_addr` permet de spécifier l'adresse du tampon de mémoire utilisé :

```
void set_dma_addr(unsigned int dmanr, unsigned int a);
```

La fonction `set_dma_count` permet de spécifier la taille du transfert :

```
void set_dma_count(unsigned int dmanr, unsigned int count);
```

Le lancement du transfert DMA s'effectue en déclenchant la carte d'E/S. En fin de transfert, une interruption doit être générée par la carte, et il faut vérifier que le transfert s'est bien passé en testant le nombre de mots non transférés :

```
void get_dma_residue(unsigned int dmanr);
```

Cette fonction retourne 0 si le DMA est terminé, sinon elle donne le nombre d'octets restant.

L'ensemble des opérations d'initialisation du DMA doit être exécuté en dévalidant les interruptions.

## **Bibliographie**

*Les sources de Linux.*

*Linux Kernel Module Programming Guide*, ORI POMERANTZ, Version 1.1.0.

*Writing Character Device Driver for Linux*, R. Baruch et C. Schroeter, Version 1.0.

*The Linux Kernel*, David A. Rusling , Version 0.8.2.

