Ecriture de driver sous Linux grâce au Langage C

par Roux Benjamin (Retour aux articles)

Date de publication : 04/01/2007

Dernière mise à jour : 21/02/2007

Dans cet article, vous verrez les bases pour créer vos propres drivers sous Linux, grâce au Langage C.

Avant-Propos

Tout d'abord sous Linux, il existe 2 modes différents, le mode noyau et le mode utilisateur.

- Espace noyau : tout est permis même le pire, on peut vite tout casser.
- Espace utilisateur : tout est protégé, les possibilités sont bridées.


La création de driver et de module utilise le mode noyau, alors faites attention !!!!

1 - Mode noyau et module

1.1 - Débogage en mode noyau

Le débogage en mode noyau peut se faire via plusieurs outils.

- En mode console, via **printk()** et **dmesg** pour voir les messages;
- Avec **kgdb**, nécessite une seconde machine reliée par câble série et possédant GDB-client pour déboguer la cible;
- Avec **kdb**, un débogeur noyau embarqué.

Il existe d'autres méthodes mais, ne les ayant jamais utilisées, je n'en parlerai pas.

Prototype :

```
int printk(const char *fmt, ...)
```

Exemple :

```
printk("<1> Hello World !\n");
```

Plusieurs niveaux de débogage sont définis dans **<linux/kernel.h>**

```
#define KERN_EMERG    "<0>"  /* système inutilisable */
#define KERN_ALERT    "<1>"  /* action à effectuer immédiatement*/
#define KERN_CRIT     "<2>"  /* conditions critiques */
#define KERN_ERR      "<3>"  /* conditions d'erreurs */
#define KERN_WARNING  "<4>"  /* message d'avertissement */
#define KERN_NOTICE   "<5>"  /* normal mais significatif */
#define KERN_INFO     "<6>"  /* informations */
#define KERN_DEBUG    "<7>"  /* messages de débugging */
```

Du coup cela devient :

```
printk(KERN_ALERT "Hello World !\n");
```

La commande **dmesg** permet d'afficher les messages de printk()

1.2 - Chargement et déchargement de module

Un module possède un point d'entrée et un point de sortie.

- point d'entrée : **int xxx(void)**
- point de sortie : **void yyy(void)**

Où xxx et yyy sont ce que vous voulez. Pour dire au module que ces 2 fonctions sont le point d'entrée et le point de sortie, nous utilisons ces 2 macros.

- module_init(xxx);

- module_exit(yyy)

Dans la suite de l'article les points d'entrée et de sortie sont nommés **module_init()** et **module_exit()**.

Ces 2 fonctions sont automatiquement appelées, lors du chargement et du déchargement du module, avec **insmod** et **rmmod**.

Remarque : Un module ne possède pas de fonction **main()**.

La fonction **module_init()** doit s'occuper de préparer le terrain pour l'utilisation de notre module (allocation mémoire, initialisation matérielle...).

La fonction **module_exit()** doit quant à elle défaire ce qui a été fait par la fonction module_init().

Voici le source minimum d'un module

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

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

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

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

Vous pouvez ainsi appeler vos **init** et **cleanup** comme bon vous semble.

Pour compiler c'est très simple, il faut utiliser cette commande :

make

Avec le makefile suivant :

```
obj-m += module.o

default:
 make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
 make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Chargement et déchargement :

$ insmod ./module.ko

$ lsmod
$ rmmod module.ko
$ dmesg

1.3 - Description de module

Vous pouvez décrire vos modules à l'aide des différentes macros mises à votre disposition dans **<linux/module.h>**.

MODULE_AUTHOR(nom) : place le nom de l'auteur dans le fichier objet

MODULE_DESCRIPTION(desc) : place une description du module dans le fichier objet

MODULE_SUPPORTED_DEVICE(dev) : place une entrée indiquant le périphérique pris en charge par le module.

MODULE_LICENSE(type) : indique le type de licence du module

On peut obtenir ces informations avec la commande **modinfo nom_module**.

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

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


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

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

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

1.4 - Passage de paramètres

Comme vous vous en doutez, il serait bien utile de passer des paramètres à notre module.

une fonction et une macro sont donc disponibles pour cela

module_param(nom, type, permissions)

MODULE_PARM_DESC(nom, desc)

Plusieurs types de paramètres sont actuellement supportés pour le paramètre type : short (entier court, 2 octet), int (entier, 4 octets), long (entier long) et charp (chaînes de caractères).

Dans le cas de tableau, vous devez utiliser la fonction : module_param_array(name, type, addr, permission);

addr est l'adresse d'une variable qui contiendra le nombre d'élément initialisés par la fonction.

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

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

static int param;

module_param(param, int, 0);
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;
}

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

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

Test:

$insmod ./module.o param=2

Voilà, maintenant que vous connaissez les bases d'un module, nous allons voir la création d'un driver et les interactions possibles avec le mode utilisateur, et le matériel.

2 - Driver en mode caractère

Un driver en mode caractère permet de dialoguer avec le périphérique, en échangeant des informations. Il existe d'autre driver, dit en mode bloc, qui échangent des données uniquement par bloc de données (disque dur par exemple).

Dans cet article, nous nous intéresserons uniquement aux drivers en mode caractère.

2.1 - Ajout d'un driver au noyau

Lors de l'ajout d'un driver au noyau, le système lui affecte un nombre majeur. Ce nombre majeur a pour but d'identifier notre driver.

L'enregistrement du driver dans le noyau doit se faire lors de l'initialisation du driver (c'est à dire lors du chargement du module, donc dans la fonction **init_module()**), en appelant la fonction : **register_chrdev()**.

De même, on supprimera le driver du noyau (lors du déchargement du module dans la fonction **cleanup_module()**), en appelant la fonction : **unregister_chrdev()**.

Ces fonctions sont définies dans <linux/fs.h>

Prototypes :

```
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 indique que l'on souhaite une affectation dynamique.
- **name** : nom du périphérique qui apparaîtra dans /proc/devices
- **fops** : pointeur vers une structure qui contient des pointeurs de fonction. Ils définissent les fonctions appelées lors des appels systèmes (open, 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**

```
static struct file_operations fops =
{
    read : my_read_function,
    write : my_write_function,
    open : my_open_function,
    release : my_release_function /* correspond a close */
};
```

Une autre façon de déclarer la structure **file_operations**, qui reste plus portable et "correcte" est la suivante

```
struct file_operations fops =
{
    .read = my_read_function,
    .write = my_write_function,
    .open = my_open_function,
    .release = my_release_function  /* correspond a close */
};
```

Certaines méthodes non implémentées sont remplacées par des méthodes par défaut. Les autres méthodes non implémentées retournent **-EINVAL**.

2.2 - Implémentation des appels systèmes

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

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

Ces fonctions renvoient **0** (ou **>0**) en cas de succès, une valeur négative sinon.

La structure file définie dans **<linux/fs.h>** représente un fichier ouvert et est créee par le noyau sur l'appel système **open()**. Elle est transmise à toutes fonctions qui agissent sur le fichier, jusqu'à l'appel de **close()**.

Les champs les plus 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_NONBLOCK...)
- **struct file_operations *f_op** : opérations associées au fichier
- **void *private_data** : le driver peut utiliser ce champ comme il le souhaite

Un fichier disque sera lui représenté par la structure inode. On n'utilisera généralement qu'un seul champ de cette structure :

kdev_t i_rdev : le numéro du périphérique actif

Ces macros nous permettrons d'extraire les nombres majeurs et mineurs d'un numéro de périphérique :

int minor = MINOR(inode->i_rdev);

int major = MAKOR(inode->i_rdev);

2.3 - Méthodes open et 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 qui sera placée dans 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

Dans les noyaux actuels, le compteur d'utilisation est ajusté automatiquement, pas besoin de s'en occuper.

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

MODULE_AUTHOR("skyrunner");
MODULE_DESCRIPTION("premier driver");
MODULE_SUPPORTED_DEVICE("none");
MODULE_LICENSE("none");

static int major = 254;

module_param(major, int, 0);
MODULE_PARM_DESC(major, "major number");

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

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

static struct file_operations fops =
{
    read : my_read_function,
    write : my_write_function,
    open : my_open_function,
    release : my_release_function /* correspond a close */
};

static int __init mon_module_init(void)
{
    int ret;

    ret = register_chrdev(major, "mydriver", &fops);

    if(ret < 0)
    {
        printk(KERN_WARNING "Probleme sur le major\n");
        return ret;
    }

    printk(KERN_DEBUG "mydriver chargé avec succès\n");
    return 0;
}

static void __exit mon_module_cleanup(void)
{
    int ret;

    ret = unregister_chrdev(major, "mydriver");

    if(ret < 0)
    {
        printk(KERN_WARNING "Probleme unregister\n");
    }

    printk(KERN_DEBUG "mydriver déchargé avec succès\n");
}

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

Une fois le driver compilé et chargé, il faut tester son lien, avec le mode utilisateur, et plus particulièrement les appels systèmes.

Tout d'abord, nous devons créer son fichier spécial : **mknod /dev/mydriver c 254 0**

Nous pouvons désormais effectuer notre premier test : **$ cat mydriver.c > /dev/mydriver**

Maintenant vous pouvez vérifier en tapant dmesg, et voir les appels à **open()**, **write()** et **release()**.

Vous pouvez aussi créer votre propre programme qui ouvre le périphérique et envoi une donnée, puis le referme.

Exemple:

```
#include <stdio.h>
```

```c
#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;
}
```

2.4 - Allocation mémoire

En mode noyau, l'allocation mémoire se fait via la fonction **kmalloc**, la désallocation, elle se fait via **kfree**.

Ces fonctions, sont très peu différentes des fonctions de la bibliothèque standard. La seule différence, est un argument supplémentaire pour la fonction **kmalloc()** : la priorité.

Cet argument peut-être :

- **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;
```

2.5 - Méthodes read et write

Ces 2 méthodes renvoient le nombre d'octets, lus pour **read** et écrits pour **write**.

```c
static int buf_size = 64;
static char *buffer;

static ssize_t my_read_function(struct file *file, char *buf, size_t count, loff_t *ppos)
{
 int lus = 0;

 printk(KERN_DEBUG "read: demande lecture de %d octets\n", count);
 /* Check for overflow */
```

```
 if (count <= buf_size - (int)*ppos)
  lus = count;
 else  lus = buf_size - (int)*ppos;
 if(lus)
  copy_to_user(buf, (int *)p->buffer + (int)*ppos, lus);
 *ppos += lus;
 printk(KERN_DEBUG "read: %d octets reellement lus\n", lus);
 printk(KERN_DEBUG "read: position=%d\n", (int)*ppos);
 return lus;
}

static ssize_t my_write_function(struct file *file, char *buf, size_t count, loff_t *ppos)
{
 int ecrits = 0;
 int i = 0;

 printk(KERN_DEBUG "write: demande ecriture de %d octets\n", count);
 /* Check for overflow */
 if (count <= buf_size - (int)*ppos)
  ecrits = count;
 else  ecrits = buf_size - (int)*ppos;

 if(ecrits)
  copy_from_user((int *)p->buffer + (int)*ppos, buf, ecrits);
 *ppos += ecrits;
 printk(KERN_DEBUG "write: %d octets reellement ecrits\n", ecrits);
 printk(KERN_DEBUG "write: position=%d\n", (int)*ppos);
 printk(KERN_DEBUG "write: contenu du buffer\n");
 for(i=0;i<buf_size;i++)
  printk(KERN_DEBUG " %d", p->buffer[i]);
 printk(KERN_DEBUG "\n");
 return ecrits;
}
```

L'allocation mémoire de **buffer** se fera lors du chargement du module.

```
buffer = kmalloc(buf_size, GFP_KERNEL);
```

Pour comprendre l'utilisation de **buffer** et de **ppos**, vous n'avez qu'à faire plusieurs essais en mode utilisateur, avec par exemple des appels à **lseek**.

Les fonctions **copy_from_user** et **copy_to_user** servent à transférer un buffer depuis/vers l'espace utilisateur.

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

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

2.6 - Méthode ioctl

Dans le cadre d'un pilote, la méthode **ioctl** sert généralement à contrôler le périphérique.

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

Prototype :

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

Cette fonction est définie dans **<linux/fs.h>**.

La fonction **ioctl**, côté utilisateur a ce prototype:

```
int ioctl(int fd, int cmd, char *argp);
```

Une fonction type ressemble à :

```
int my_ioctl_function(struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg)
{
    int retval = 0;

    switch(cmd)
    {
        case ... : ... break;
        case ... : ... break;
        default : retval = -EINVAL; break;
    }
    return retval;
}
```

Ne pas oublier de rajouter cette entrée dans la structure **file_operations**.

```
static struct file_operations fops =
{
    read : my_read_function,
    write : my_write_function,
    open : my_open_function,
    release : my_release_function, /* correspond a close */
    ioctl : my_iotl_function;
};
```

Si la commande n'existe pas pour le pilote, l'appel système retournera **EINVAL**.

C'est au programmeur de choisir les numéros qui correspondent aux commandes.

Les numéros de commande, doivent être uniques dans le système, afin d'éviter l'envoi d'une commande au "mauvais" périphérique, il existe pour cela une méthode sûre pour choisir ses numéros.

Depuis la version 2.4, les numéros utilisent quatre champs de bits : **TYPE**, **NOMBRE**, **SENS** et **TAILLE**.

Le fichier header **<asm/ioctl.h>**, inclus dans **<linux/ioctl.h>**, définit des macros qui facilitent la configuration des numéros de commande.

_IO(type, nr) où type sera le nombre magique (voir le fichier ioctl-number.txt)

_IOR(type, nr, dataitem) // sens de transfert : Lecture

_IOW(type, nr, dataitem) // sens de transfert : Écriture

_IOWR(type, nr, dataitem) // sens de transfert : Lecture / Écriture

Une définition des numéros de commande ressemblera ainsi à ça :

driver.h

```
...
#define SAMPLE_IOC_MAGIC 't'
#define GETFREQ _IOR(SAMPLE_IOC_MAGIC, 0, int)
#define SETFREQ _IOW(SAMPLE_IOC_MAGIC, 1, int)
```

Ceci peut être défini dans un fichier .h qui sera inclus, dans le code du driver, et dans le code du programme pilotant le driver.

driver.c

```
#include "driver.h"
...

int my_ioctl_function(struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg)
{
    int retval = 0;

    switch(cmd)
    {
        case SETFREQ : ... break;
        case GETFREQ : ... break;
        default : retval = -EINVAL; break;
    }
    return retval;
}
```

utilisateur.c

```
#include "driver.h"
...

ioctl(file, SETFREQ, 100);
ioctl(file, SETACK, 10);
```

2.7 - Gestion d'interruptions

Une interruption est un signal envoyé par un matériel et qui est capable d'interrompre le processeur.

Notre pilote, met donc en place un gestionnaire d'interruptions pour les interruptions de son périphérique.

Un gestionnaire d'interruptions est déclaré par cette fonction : **request_irq()**.

Il est libéré par cette fonction : **free_irq()**.

Les prototypes définis dans **<linux/sched.h>** sont les suivants:

```
int request_irq(unsigned int irq,
  void (*handler) (int, void , struct pt_regs *),
  unsigned long flags /* SA_INTERRUPT ou SA_SHIRQ */,
  const char *device, void *dev_id);
```

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

Quand l'interruption qui a pour numéro **irq** se déclenche, la fonction **handler()** est appelée.

Le champ **dev_id** sert à identifier les périphériques en cas de partage d'IRQ (SA_SHIRQ).

La liste des IRQ déjà déclarées est disponible dans **/proc/interrupts**.

Le gestionnaire d'interruptions peut-être déclaré à 2 endroits différents :

- au chargement du pilote (**module_init()**)
- à la première ouverture du pilote par un programme (**open()**), il faut alors utiliser le compteur d'utilisation et désinstaller le gestionnaire au dernier **close()**.

Le driver peut décider d'activer ou de désactiver le suivi de ses interruptions. Il dispose de 2 fonctions pour cela, définies dans **<linux/irq.h>**.

```
void enable_irq(int irq);
void disable_irq(int irq);
```

Cela peut-être utile pour ne pas être interrompu par une interruption, lors d'une interruption.

3 - Conclusion

Vous venez donc de voir que la conception de driver sous Linux, n'est pas des plus difficiles, à vous maintenant d'apprendre plus tant au niveau informatique qu'électronique pour concevoir des drivers intéressants.

4 - Bibliographie

Pour écrire ce tutoriel, je me suis inspiré d'un cours que j'ai reçu lors de mes études.

5 - Documentation

http://lwn.net/Kernel/LDD3/

Le kit de survie du developpeur Linux

6 - Exemple

Voici l'exemple d'un driver, pour une carte d'interfaçage Cigal fonctionnant sur bus ISA, fait lors d'un TP de cours d'Interface (ce code n'est pas exempt de bug, mais le principe est là)

cigal.h

```
#ifndef CIGAL_H_
#define CIGAL_H_
#include <linux/ioctl.h>

#define SAMPLE_IOC_MAGIC 't'
#define GETFREQ _IOR(SAMPLE_IOC_MAGIC, 0, int)
#define SETFREQ _IOW(SAMPLE_IOC_MAGIC, 1, int)
#define SETACK _IOW(SAMPLE_IOC_MAGIC, 2, int)
#define START _IOW(SAMPLE_IOC_MAGIC, 3, int)
#define STOP _IOW(SAMPLE_IOC_MAGIC, 4, int)

#define FREQ_MAX 10000
#define FREQ_MIN 1

#define IRQ 5

/* From datasheet */
#define ADBASE 0x320

#define PPI 0x000
#define DAC 0x004
#define ADC 0x008
#define TIMER 0x00C
#define ECIR 0x010
#define LEI 0x014

#endif /* guard */
```

cigal.c

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <asm/io.h>
#include "cigal.h"
#include <asm/irq.h>
#include <linux/signal.h>
#include <linux/sched.h>
#include <linux/interrupt.h>

static int lire_can(void);

static int major = 252;

MODULE_DESCRIPTION("Driver carte CIGAL");
MODULE_AUTHOR("skyrunner");
MODULE_SUPPORTED_DEVICE("Carte Cigal");
MODULE_LICENSE("No");
MODULE_PARM_DESC(major, "major number");
module_param(major, int, 0);

typedef struct
{
 int *buffer;
 int freq;
```

cigal.c

```
 int n_ack;
} s_driver;

static unsigned int buf_size = 101*sizeof(int);
static s_driver *tab[8]; // le driver peut gérer 8 périphériques (/dev/cigal, /dev/cigal2, ...,
 /dev/cigal8)
static int n_ack; // variable globale pour connaitre le nombre d'acquisitions quand on est dans
 l'interruption
static int irq_on = 0; // variable pour savoir si l'irq est on ou off

static irqreturn_t interruption(int irq, void *dev_id, struct pt_regs *regs)
{
 static int nb = -1;

 if(nb == -1)
 {
  nb++;
  lire_can();
 }
 else
 {
  tab[0]->buffer[nb++] = lire_can();
  nb++;
  if(nb == n_ack)
  {
   if(irq_on)
   {
    free_irq(IRQ, NULL);
    irq_on = 0;
   }
   nb = -1;
  }
 }
 return 0;
}

/* voir datasheet de carte Cigal pour adresse */
static void timer(int freq)
{
 char ctrlT0 = 0x34;
 char ctrlT1 = 0x74;

 char initT0_L = 0x00;
 char initT0_H = 0x01;

 char initT1_L = (char)((int)((FREQ_MAX / freq) & 0xFF));
 char initT1_H = (char)((int)((FREQ_MAX / freq) >> 16));

 printk(KERN_INFO "freq = %d 0x%x 0x%x\n", freq, initT1_L, initT1_H);

 outb(ctrlT0, ADBASE + TIMER + 3);
 outb(initT0_L, ADBASE + TIMER);
 outb(initT0_H, ADBASE + TIMER);

 outb(ctrlT1, ADBASE + TIMER + 3);
 outb(initT1_L, ADBASE + TIMER + 1);
 outb(initT1_H, ADBASE + TIMER + 1);
}

static void init_timer_interrupt(int freq)
{
 int ret;
 timer(freq);
 if((ret = request_irq(IRQ, interruption, SA_INTERRUPT, "cigal", NULL)) != 0)
 {
  printk(KERN_DEBUG "erreur request_irq : %d \n",ret);
  return;
```

cigal.c

```
    }
    irq_on = 1;
    outb(0x02, ADBASE+ECIR);
   }

   static int lire_can(void)
   {
    int tmp,tmp1;

    tmp = inb(ADBASE + ADC);   // lancer la conversion
    tmp1 = inb(ADBASE + ADC + 1);  // lire les 4 derniers
    tmp = inb(ADBASE + ADC);               // lire les 8 premiers bits
    tmp1 &= 0x0F;
    tmp1 <<= 8;
    tmp += tmp1;
    return tmp;
   }

   static int cigal_ioctl(struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg)
   {
    int retval = 0;
    int freq_tmp = (int)arg;
    s_driver *p = (s_driver*)file->private_data;
    switch(cmd)
    {
     case GETFREQ :
     {
         copy_to_user((int**)arg, (int*)p->freq, sizeof(p->freq));
               break;
     }

     case SETFREQ :
     {
         if(freq_tmp > FREQ_MAX)
       p->freq = FREQ_MAX;
               else if(freq_tmp < FREQ_MIN)
       p->freq = FREQ_MIN;
      else
                p->freq = freq_tmp;
      break;
     }

     case SETACK :
     {
               p->n_ack = arg;
               break;
     }

     case START :
     {
         n_ack = p->n_ack;
               init_timer_interrupt(p->freq);
               break;
     }

           case STOP :
           {
               if(irq_on)
               {
                   free_irq(IRQ, NULL);
                   irq_on = 0;
               }
               break;
           }

           default :
           {
```

cigal.c

```
                retval = -EINVAL;
                break;
            }
        }
        return retval;
    }

    static ssize_t cigal_read(struct file *file, char *buf, size_t count, loff_t *ppos)
    {
     int lus = 0;

     s_driver *p = (s_driver*)file->private_data;
     printk(KERN_DEBUG "read: demande lecture de %d octets\n", count);
     /* Check for overflow */
     if (count <= buf_size - (int)*ppos)
      lus = count;
     else  lus = buf_size - (int)*ppos;
     if(lus)
      copy_to_user(buf, (int *)p->buffer + (int)*ppos, lus);
     *ppos += lus;
     printk(KERN_DEBUG "read: %d octets reellement lus\n", lus);
     printk(KERN_DEBUG "read: position=%d\n", (int)*ppos);
     return lus;
    }

    static ssize_t cigal_write(struct file *file, char *buf, size_t count, loff_t *ppos)
    {
     int ecrits = 0;
     int i = 0;
     s_driver *p = (s_driver*)file->private_data;
     printk(KERN_DEBUG "write: demande ecriture de %d octets\n", count);
     /* Check for overflow */
     if (count <= buf_size - (int)*ppos)
      ecrits = count;
     else  ecrits = buf_size - (int)*ppos;

     if(ecrits)
      copy_from_user((int *)p->buffer + (int)*ppos, buf, ecrits);
     *ppos += ecrits;
     printk(KERN_DEBUG "write: %d octets reellement ecrits\n", ecrits);
     printk(KERN_DEBUG "write: position=%d\n", (int)*ppos);
     printk(KERN_DEBUG "write: contenu du buffer\n");
     for(i=0;i<buf_size;i++)
      printk(KERN_DEBUG " %d", p->buffer[i]);
     printk(KERN_DEBUG "\n");
     return ecrits;
    }

    static int cigal_open(struct inode *inode, struct file *file)
    {
     unsigned int i = iminor(inode);
     file->private_data = (s_driver *)tab[i];
     printk(KERN_DEBUG "open()\n");
     return 0;
    }


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

    static struct file_operations cigal_fops =
    {
     read: cigal_read,
     write: cigal_write,
```

cigal.c

```
    open: cigal_open,
    release:cigal_release,
    ioctl:cigal_ioctl
   };

   static int __init cigal_init(void)
   {
    int ret;
    int i;

    ret = register_chrdev(major, "cigal", &cigal_fops);
    if(ret <0)
    {
     printk(KERN_WARNING "probleme major sur cigal\n");
     return ret;
    }

    for(i = 0; i<8; i++)
    {
     tab[i] = kmalloc(sizeof(*tab[i]), GFP_KERNEL);
     tab[i]->buffer = kmalloc(buf_size, GFP_KERNEL);
     tab[i]->freq = 1000;
    }
    printk(KERN_INFO "cigal sucess load !\n");

    return 0;
   }

   static void __exit cigal_exit(void)
   {
    int ret,i;

    for(i = 0; i<8; i++)
    {
     kfree(tab[i]->buffer);
     kfree(tab[i]);
    }

    ret = unregister_chrdev(major, "cigal");
    if(ret < 0)
     printk(KERN_WARNING "probleme unregister sur cigal\n");

    printk(KERN_INFO "cigal success unload !\n");
   }

   module_init(cigal_init);
   module_exit(cigal_exit);
```

7 - Remerciements

Je tiens à remercier gege2061, Yogui et Michaël pour leur relecture et corrections.