



Université du Québec

**École de technologie supérieure**

**Département de génie électrique**

## ELE784 - Ordinateurs et programmation système

### Laboratoire #2

#### Développement d'un pilote pour une caméra USB sous Linux

Partie 2

##### **Description sommaire :**

Dans ce laboratoire, séparé en trois parties, il vous sera demandé de coder un pilote pour une caméra USB répondant au standard UVC. Dans un premier temps, le squelette du module sera mis en place. Par la suite, certaines fonctions types d'un module USB seront ajoutées et finalement le cœur du module sera codé dans la troisième partie. Le résultat final sera un module capable d'envoyer des commandes de base à une caméra et un programme écrit en C utilisé pour communiquer avec ce module. Vous serez donc en mesure d'obtenir des images de la caméra et ces images seront utilisées dans le cadre du laboratoire #3 afin de mettre en évidence l'interaction matériel-logiciel.

**Professeur :** Bruno De Kelper

**Chargé de laboratoire :** Louis-Bernard Lagueux

Objectif .....	3
Introduction.....	4
La fonction d'initialisation.....	4
La fonction de sortie .....	4
La fonction « <i>probe</i> » .....	4
La fonction « <i>disconnect</i> » .....	5
La fonction « <i>open</i> » .....	5
La fonction « <i>IOCTL</i> » .....	6
IOCTL_STREAMON.....	6
IOCTL_STREAMOFF .....	6
IOCTL_PANTILT .....	7
IOCTL_PANTILT_RESEST.....	7

# Objectif

Le but ultime de la série de laboratoire de ce cours est de vous faire configurer un système informatique avec un noyau Linux, d'y charger un module (pilote) que vous aurez développé pour contrôler une caméra USB et d'utiliser les images générées par cette dernière afin d'effectuer certains tests sur le processeur. De cette manière, il vous sera possible d'étudier la structure fonctionnelle d'un ordinateur et ses différentes composantes avec un intérêt majeur sur l'interaction matériel-logiciel<sup>1</sup> (ceci est l'un des objectifs principales du cours ELE784).

L'ensemble du laboratoire sera divisé en trois parties:

1. Développement des composantes logicielles de base d'un système informatique. C'est dans cette partie que vous allez configurer le système informatique avec le noyau Linux et avec certains outils couramment utilisés.
2. Développement d'un pilote pour contrôler une caméra USB sous Linux
3. Traitement des données obtenues avec la caméra pour démontrer l'importance de l'interaction matériel-logiciel dans un système informatique.

Les objectifs du laboratoire #2 sont les suivants :

- Se familiariser avec la notion de module et de pilote sous Linux
- Se familiariser avec les différentes commandes utilisées pour travailler avec les modules sous Linux
- Se familiariser avec les différentes sections dans le code d'un module
- Se familiariser avec la notion de synchronisation dans un pilote
- Se familiariser avec le transfert de données entre le « *user space* » et le « *kernel space* »

---

<sup>1</sup> Adaptation du sommaire du cours que l'on trouve sur le site du département de génie électrique

## Introduction

Dans la seconde partie du deuxième laboratoire vous devrez ajouter certaines fonctions typiques d'un pilote USB nécessaires pour le bon fonctionnement du périphérique. Vous devrez, dans un premier temps, modifier les fonctions d'initialisation et de sortie du module que vous avez créées dans la première partie. Par la suite, Les fonctions *probe* et *disconnect* seront ajoutées. Finalement, nous modifierons la fonction *open* et la fonction *IOCTL* afin de pouvoir faire bouger l'objectif de la caméra.

## Les fonctions d'initialisation et de sortie (init, exit)

Bien qu'on a vu que les modules doivent avoir une fonction d'initialisation, qui sera exécutée au moment de l'installation du Pilote, ainsi qu'une fonction de sortie qui sera exécutée lors de la désinstallation, dans le cas des Pilotes USB, ce processus a été simplifié. Dans le cas des Pilotes USB, il suffit de déclarer les deux structures suivantes (voir diapos 20 à 22, cours # 5) :

Une structure `usb_device_id` :

```
static struct usb_device_id my_usb_id [] = {
    {USB_DEVICE(0x046d, 0x0837)},
    {USB_DEVICE(0x046d, 0x046d)},
    {USB_DEVICE(0x046d, 0x08c2)},
    {USB_DEVICE(0x046d, 0x08cc)},
    {USB_DEVICE(0x046d, 0x0994)},
    {},
};
MODULE_DEVICE_TABLE(usb, my_usb_id);
```

Une structure `usb_driver` :



```
static struct usb_driver udriver = {
    .name = "My_usb_driver",
    .probe = my_probe,
    .disconnect = my_disconnect,
    .id_table = my_usb_id,
};
```

Et d'ajouter la déclaration suivante :

```
module_usb_driver(udriver);
```

La déclaration ci-dessus s'assure d'incorporer tout le nécessaire pour l'installation et le retrait standards du Pilote USB.

Cependant, si des actions spécifiques sont requises pendant l'une ou l'autre de ces phases (installation, retrait), alors il sera nécessaire d'ajouter une fonction INIT et une fonction EXIT, en bonne et due forme, plutôt que d'avoir recours à la déclaration ci-dessus mentionnée. Et dans ce cas, ces fonctions devront absolument faire appel aux fonctions suivantes du « Noyau USB », pour que l'inscription et la désinscription du Pilote USB auprès du Noyau USB se fasse correctement (voir diapo 23, cours # 5) :

INIT		usb_register (...)
EXIT		usb_deregister (...)

Et toujours dans ce dernier cas, les deux structures `usb_device_id` et `usb_driver` sont toujours requises et identiques à celles décrites ci-dessus.

## La fonction « *probe* »

Avant de pouvoir coder cette fonction, vous devez définir la structure de type `usb_class_driver` ainsi que la structure de type `file_operations` pour votre pilote (voir diapo 29, cours #5, et l'exemple dans le livre de référence chapitre #13), tel que :

```
static const struct file_operations fops = {
    .owner = THIS_MODULE,
    .read = my_read,
    .open = my_open,
    .unlocked_ioctl = my_ioctl,
};

static struct usb_class_driver class_stream_driver = {
    .name = "camera_stream",
    .fops = &fops,
    .minor_base = DEV_MINOR,
};

static struct usb_class_driver class_control_driver = {
    .name = "camera_control",
    .fops = &fops,
    .minor_base = DEV_MINOR,
};
```

Pour l'interface  
Video Streaming

Pour l'interface  
Video Control

**Note :** Nous n'utiliserons pas vraiment l'interface Video Control et celle-ci est ajoutée uniquement pour une utilisation future éventuelle.

La fonction Probe, quant à elle, sera appelée par le Noyau USB, une fois pour chaque « interface » qui apparaît; c'est-à-dire dans notre cas trois (3) fois, une fois pour l'interface Video Control (interface 0), une fois pour l'interface Video Streaming (interface 1) et une fois pour l'interface Human Interface Device (interface 2).

À chaque appel de la fonction Probe, celle-ci reçoit la structure `usb_interface` et la structure `usb_device_id` correspondant à l'interface pour laquelle la fonction Probe est appelée. Dans ceux-ci, on retrouve l'information nécessaire pour identifier cette interface et plus spécifiquement pour les deux interfaces qui nous intéressent :

	Video Control	Video Streaming
<code>interface-&gt;cur_altsetting-&gt;desc.bInterfaceClass</code>	CC_VIDEO	CC_VIDEO
<code>interface-&gt;cur_altsetting-&gt;desc.bInterfaceSubClass</code>	SC_VIDEOCONTROL	SC_VIDESTREAMING
<code>interface-&gt;cur_altsetting-&gt;desc.bInterfaceNumber</code>	0	1

et dans les deux cas :

<code>id-&gt;idVendor</code>	0x046d
<code>id-&gt;idProduct</code>	0x0837

Avec ces informations, la fonction Probe peut identifier avec assez de précision les interfaces que le Pilote est prêt à prendre en charge. Cela étant fait, la fonction Probe s'assure, entre autres :

- D'indiquer au Noyau USB que le Pilote va prendre en charge cette interface :

```
usb_register_dev(...)
```

- De choisir l'interface « alternative » appropriée :

```
usb_set_interface (...)
```

Dans notre cas, ça sera l'interface  
alternative 0, dans les deux cas

- D'attacher toute « structure personnelle » du Pilote à la structure de cette interface :

```
usb_set_intfdata(...)
```

## La fonction « *disconnect* »

La fonction Disconnect, à l'instar de la fonction Probe, est aussi appelée par le Noyau USB pour chaque interface qui disparaît. Dans cette fonction vous aurez tout simplement à utiliser la fonction `usb_deregister_dev(...)` pour indiquer au Noyau USB que votre pilote n'est plus associé à cette interface.

Aussi, comme nous verrons plus tard, il sera aussi nécessaire d'éliminer toute activité en lien avec cette interface qui serait encore en cours. Par exemple, toute requête « urb » non encore complétée.

## La fonction « *open* »

Pour cette fonction, vous n'aurez rien à faire, car le code vous est donné dans ce qui suit (petit cadeau gratuit). Prenez quand même le temps de comprendre les étapes qui sont effectuées.

```
int my_open(struct inode *inode, struct file *file) {
    struct usb_interface *interface;
    int subminor;

    printk(KERN_WARNING "ELE784 -> Open\n");

    subminor = iminor(inode);
    interface = usb_find_interface(&udriver, subminor);
    if (!interface) {
        printk(KERN_WARNING "ELE784 -> Open: Ne peux ouvrir le peripherique\n");
        return -ENODEV;
    }

    file->private_data = usb_get_intfdata(interface);
    return 0;
}
```

## La fonction « *IOCTL* »

Finalement, il vous est demandé, pour la fonction IOCTL, d'ajouter le code pour les 4 commandes suivantes :

<u>Commandes à implémenter</u>		<u>Commandes à faire plus tard</u>	
IOCTL_GET	0x10	IOCTL_STREAMON	0x30
IOCTL_SET	0x20	IOCTL_STREAMOFF	0x40
IOCTL_PANTILT_RELATIVE	0x50		
IOCTL_PANTILT_RESET	0x60		

Dans les sections qui suivent, vous trouverez une description de ce que vous devez faire pour chacune d'entre elles. De plus, l'implémentation de la commande `IOCTL_SET` vous est donné en guise d'exemple, en annexe et dans le code « squelette » qui vous est fourni pour le lab.

### ***IOCTL\_GET***

Cette commande est utilisée pour récupérer des valeurs ou configurations internes de la caméra. Cette commande est envoyée à un *EndPoint* de type contrôle. Voici les informations nécessaires pour acheminer cette commande avec la fonction `usb_control_msg`:

Argument	Information
usb_device	Votre device USB
Pipe	Endpoint #0 de type RCV
Request	GET_CUR (0x81) GET_MIN (0x82) GET_MAX (0x83) GET_RES(0x84) <b>(choix fournit par l'utilisateur)</b>
requestType	<b>USB_DIR_IN</b>   USB_TYPE_CLASS   USB_RECIP_INTERFACE
Value	<b>(choix fournit par l'utilisateur)</b> << 8
Index	<b>(choix fournit par l'utilisateur)</b> << 8   (numéro de l'interface)
Data	<b>(tableau fournit par l'utilisateur, s'il y a lieu)</b>
Size	<b>(taille du tableau Data, fournit par l'utilisateur)</b>
Timeout	<b>(choix fournit par l'utilisateur)</b>

## IOCTL\_SET

Cette commande est utilisée pour affecter des valeurs ou des configurations à la caméra. Cette commande est envoyée à un *EndPoint* de type contrôle. Voici les informations nécessaires pour acheminer cette commande avec la fonction `usb_control_msg`:

Argument	Information
usb_device	Votre device USB
pipe	Endpoint #0 de type SND
request	SET_CUR (0x01) <b>(fournit par l'utilisateur)</b>
requestType	<b>USB_DIR_OUT</b>   USB_TYPE_CLASS   USB_RECIP_INTERFACE
value	<b>(choix fournit par l'utilisateur)</b> << 8
index	<b>(choix fournit par l'utilisateur)</b> << 8   (numéro de l'interface)
data	<b>(tableau fournit par l'utilisateur, s'il y a lieu)</b>
size	<b>(taille du tableau Data, fournit par l'utilisateur)</b>
timeout	<b>(choix fournit par l'utilisateur)</b>

Pour ces deux commandes, les valeurs pour les champs *value*, *index*, *data*, *size*, *Timeout* et *request* (dans le cas de *IOCTL\_GET*) devront être passées au pilote par votre programme de test.

Un bon test serait de récupérer la valeur minimum, maximum et par défaut de certaines options tel que la luminance, le contraste et le gain. Par la suite, vous pouvez, avec votre programme de test, configurer ces options à votre guise.

## IOCTL\_PANTILT\_RELATIVE

Cette commande est utilisée pour modifier la position verticale et horizontale de l'objectif de la caméra, par rapport à sa position actuelle (relatif). Cette commande est envoyée à l'interface Video Control, visible pour l'utilisateur sous la forme du « nœud » : `/dev/camera_control` et au seul *EndPoint* de cette interface, soit le *EndPoint* 0 qui est toujours de type contrôle. Voici les informations nécessaires pour acheminer cette commande avec la fonction `usb_control_msg`:

Argument	Information
usb_device	Votre device USB
pipe	Endpoint #0 de type SND
request	SET_CUR (0x01)
requestType	<b>USB_DIR_OUT</b>   USB_TYPE_CLASS   USB_RECIP_INTERFACE
value	0x0100
index	0x0B00
data	int16_t data[2] <b>(fournit par l'utilisateur)</b>

	où      data[0] = Pan    (Horizontale) data[1] = Tilt    (Verticale)
size	4
timeout	500    (recommandé)

**Note :** Les courses limites, à partir de la position de « reset », pour les deux directions sont :

**Horizontale : -4480 à 4480**

**Verticale : -1920 à 1920**

Il est préférable d'éviter de dépasser ces limites, au risque d'endommager la caméra.

## ***IOCTL\_PANTILT\_RESET***

Cette commande est utilisée pour initialiser la position de l'objectif de la caméra (replacer au centre de la caméra). Cette commande est envoyée à un *EndPoint* de type contrôle. Voici les informations nécessaires pour acheminer cette commande avec la fonction `usb_control_msg`:

Argument	Information
usb_device	Votre device USB
pipe	Endpoint #0 de type SND
request	SET_CUR (0x01)
requestType	<b>USB_DIR_OUT</b>   USB_TYPE_CLASS   USB_RECIP_INTERFACE
value	0x0200
index	0x0B00
data	uint8_t data <b>(fournit par l'utilisateur)</b>  Valeurs possibles : Reset Pan            = 0x01 Reset Tilt          = 0x02 Reset Pan-Tilt = 0x03
size	1
timeout	500    (recommandé)



## ANNEXE

### Code d'exemple pour la commande IOCTL\_SET

#### **Note**

Ce code est incomplet et est fourni uniquement à titre d'exemple.

```
long my_ioctl(struct file *file, unsigned int cmd, unsigned long arg) {
    struct orbit_driver *driver = (struct orbit_driver *) file->private_data;
    struct usb_interface *interface = driver->interface;
    struct usb_device *udev = interface_to_usbdev(interface);

    struct usb_request user_request;
    uint8_t request, data_size;
    uint16_t value, index, timeout;
    uint8_t *data;

    switch(cmd) {

    case IOCTL_SET:
        printk(KERN_INFO "ELE784 -> IOCTL_SET\n");

        copy_from_user(&user_request, (struct usb_request *)arg, sizeof(struct usb_request));

        data_size = user_request.data_size;
        request = user_request.request;
        value = (user_request.value) << 8;
        index = (user_request.index) << 8 | interface->cur_altsetting->desc.bInterfaceNumber;
        timeout = user_request.timeout;
        data = NULL;

        if (data_size > 0) {
            data = kmalloc(data_size, GFP_KERNEL);
            copy_from_user(data, ((uint8_t __user *) user_request.data),
                          data_size*sizeof(uint8_t));
        }

        retval = usb_control_msg(udev,
                                usb_sndctrlpipe(udev, 0x00),
                                request,
                                USB_DIR_OUT | USB_TYPE_CLASS | USB_RECIP_INTERFACE,
                                value, index, data, data_size, timeout);

        if (data) {
            kfree(data);
            data = NULL;
        }

        break;
    }

    return retval;
}
```