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 3

Description sommaire :

Dans ce laboratoire, séparé en trois parties, il vous sera demandez 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

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[1] (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* »

[1] Adaptation du sommaire du cours que l'ont trouve sur le site du département de génie électrique

Introduction

Dans la première partie du laboratoire il vous a été demandé de coder un pilote de type caractère simple. Par la suite, dans la deuxième partie, vous avez ajouté la couche USB ainsi que quelques fonctions de base servant à contrôler l'objectif de la caméra. Dans la troisième partie, vous devrez ajouter la fonction nécessaires pour récupérer des images de la caméra. Ceci complétera le pilote.

La fonction « *probe* »

Comme indiqué dans le livre de référence[2], un périphérique peut avoir plusieurs interfaces. De plus, chaque interface peut avoir plusieurs configurations possibles. Nous devrons donc sélectionner correctement la configuration qui nous intéresse afin que notre pilote fonctionne correctement. Pour ce faire, il faut ajouter, à la suite des commandes normales de la fonction `probe`, la commande `usb_set_interface` comme suit:

```
interface_to_usbdev(…)
usb_get_dev(…)
usb_set_intfdata(…)
usb_register_dev(…)
usb_set_interface(dev, 1, 4);
```

Ceci effectuera la sélection de la configuration #4 de l'interface #1. Le choix de cette configuration n'est pas arbitraire, il a été trouvé en effectuant du `reverse engineering` sur le code du module `uvc` officiel de Linux. De plus, sachez que ce choix est fonction de la grandeur de l'image que nous voulons obtenir. Si nous aurions voulue une image de plus grande dimension, une configuration différente aurait été choisie.

La fonction « *IOCTL_GRAB* »

En plus des commandes `IOCTL` déjà implémentées dans votre pilote, vous devrez ajouter la commande suivante :

```
IOCTL_GRAB    0x50
```

La commande `IOCTL_GRAB` devra initialiser 5 `URB` différents qui seront envoyés pour récupérer les données de la caméra. Étant donné que nous voulons utiliser le mode `isochronous` pour communiquer, nous n'avons pas de fonction comme `usb_control_msg(…)` pour nous aider à effectuer cette étape. Nous devrons donc effectuer cette initialisation à la main. Afin de faciliter la création de cette commande, voici ce que devrait avoir l'air l'initialisation des `URB`. Partout où vous voyez « `/** … */` » dans le code, vous devez ajouter des arguments. Faite des recherches dans le livre de référence[3] ou sur Internet pour avoir plus d'information sur les arguments à utiliser pour initialiser les `URB`. Cette partie de code se trouve dans le fichier `initUrb.c` dans le répertoire `/home/document` sur l'ordinateur de compilation.

```
cur_altsetting = intf->cur_altsetting;
endpointDesc = cur_altsetting->endpoint[0].desc;
```

[2] Linux Device Drivers, Third Edition, chapitre 13 page 331
[3] Linux Device Drivers, Third Edition, chapitre 13 toutes les pages !

```
nbPackets = 40;  // The number of isochronous packets this urb should contain
myPacketSize = le16_to_cpu(endpointDesc.wMaxPacketSize);
size = myPacketSize * nbPackets;
nbUrbs = 5;

for (i = 0; i < nbUrbs; i++) {
  usb_free_urb(/** ... */); // Pour être certain
  myUrb[i] = usb_alloc_urb(/** ... */);
  if (myUrb[i] == NULL) {
    //printk(KERN_WARNING "");
    return -ENOMEM;
  }

  myUrb[i]->transfer_buffer = usb_buffer_alloc(/** ... */);

  if (myUrb[i]->transfer_buffer == NULL) {
    //printk(KERN_WARNING "");
    usb_free_urb(myUrb[i]);
    return -ENOMEM;
  }

  myUrb[i]->dev = /** ... */;
  myUrb[i]->context = dev;
  myUrb[i]->pipe = usb_rcvisocpipe(dev, endpointDesc.bEndpointAddress);
  myUrb[i]->transfer_flags = URB_ISO_ASAP | URB_NO_TRANSFER_DMA_MAP;
  myUrb[i]->interval = endpointDesc.bInterval;
  myUrb[i]->complete = /** ... */;
  myUrb[i]->number_of_packets = /** ... */;
  myUrb[i]->transfer_buffer_length = /** ... */;

  for (j = 0; j < nbPackets; ++j) {
    myUrb[i]->iso_frame_desc[j].offset = j * myPacketSize;
    myUrb[i]->iso_frame_desc[j].length = myPacketSize;
  }
}

for(i = 0; i < nbUrbs; i++){
  if ((ret = usb_submit_urb(/** ... */)) < 0) {
    //printk(KERN_WARNING "");
    return ret;
  }
}
```

Bien que ce code forme 95 % de la commande, certaines lignes devront être ajoutées afin que le pilote fonctionne convenablement. Continuer la lecture pour avoir plus d'information sur ce sujet.

La fonction `Callback` pour les `URB`

Comme indiqué dans le livre de référence[4], un URB nécessite une fonction qui sera appelée lorsque la tâche de ce dernier sera terminée. Dans notre cas, les étapes effectuées par cette fonction sont légèrement compliquées. Pour cette raison, le code de cette fonction vous sera fourni. Vous pouvez le trouver dans le fichier `callback.c` dans le répertoire `/home/document` sur l'ordinateur de compilation.

Par contre, vous devrez ajouter la partie de code nécessaire pour indiquer que vous avez bel et bien reçue toutes les données et que la fonction `read` peut retourner le résultat obtenu au programme de test. L'endroit où vous devez ajouter cette partie de code est clairement indiqué dans le fichier `callback.c`. La méthode suggérée pour synchroniser la fonction `read` de votre pilote avec la fonction `callback` est l'interface `completion`. Vous pouvez obtenir plus d'information sur cette interface dans le livre de référence[5]. De plus veillez noter que certaines

[4] Linux Device Drivers, Third Edition, chapitre 13 page 339
[5] Linux Device Drivers, Third Edition, chapitre 5 page 115

variables sont utilisées dans cette fonction et sont propre à votre pilote. Vous devez donc les créer et les initialiser. Voici la liste de ces variables :

Variable	Type	Note
`myStatus`	`unsigned int`	Doit être initialisé à 0 avant la création des URB
`myLength`	`unsigned int`	Ne change jamais et égale à 42666
`myLengthUsed`	`unsigned int`	Doit être initialisé à 0 avant la création des URB
`myData`	`char`	Tableau de longueur `myLength`

La fonction `read`

La fonction `read` de votre pilote est utilisée pour retourner les données récupérées sur la caméra à votre programme de test. Cette fonction est relativement simple et ne devrait pas vous poser de problème. Voici les étapes de cette fonction :

1. Récupérer la référence à votre structure `usb_device` normalement contenue dans le pointeur `file->private_data`.
2. Attendre que la fonction `callback` nous indique que tous les `URB` ont terminés leur tâche avec l'interface `completion`.
3. Copier les données au programme de test de façon sécuritaire.
4. Sur chaque URB envoyé par votre pilote, effectuer les fonctions suivantes :
    a. `usb_kill_urb(…)` sur le `URB` courant
    b. `usb_buffer_free(…)` sur le transfer_buffer du `URB` courant
    c. `usb_free_urb(…)` sur le `URB` courant
5. retourner le nombre de données transférées.

Le programme de test

Dans votre programme de test, vous devrez utiliser la nouvelle commande `IOCTL_GRAB` de votre pilote pour lancer l'acquisition d'une image. Les étapes pour obtenir une image sont les suivantes :

1. ouvrir le fichier dans lequel vous enregistrerez l'image
2. Effectuer la commande IOCTL_STREAMON
3. Effectuer la commande IOCTL_GRAB
4. Utiliser la commande `read` du pilote pour récupérer les données de l'image
5. Effectuer la commande IOCTL_STREAMOFF
6. Effectuer les modifications sur les données (voir exemple plus bas)
7. Écrire le résultat final dans le fichier ouvert à l'étape #1
8. Fermer le fichier ouvert à l'étape #1

Afin de faciliter cette étape, voici ce que devrait avoir l'air cette partie de code (les étapes 2, 3, 4 et 5 ont été supprimées, vous devez ajouter par vous-même ces parties de code). Notez que vous trouverez la définition des variables `HEADERFRAME1`, `DHT_SIZE` et `dht_data` dans le fichier `dht_data.h` situé dans le répertoire `/home/document` de l'ordinateur de compilation. De plus, sachez que la variable `mySize` est égale au nombre de données qui a été retourné par la fonction `read` du pilote. Finalement, le tampon `inBuffer` est utilisé pour sauvegarder les données retournées par cette même fonction.

```
FILE *foutput;
unsigned char * inBuffer;
```

```
unsigned char * finalBuf;

inBuffer = malloc((42666)* sizeof(unsigned char));
finalBuf = malloc((42666 * 2)* sizeof(unsigned char));

if((inBuffer == NULL) || (finalBuf == NULL)){
  return -1;
}

foutput = fopen("/lien/vers/fichier.jpg", "wb");
if(foutput != NULL){
  // Etape #2
  // Etape #3
  // Etape #4
  // Etape #5
  memcpy (finalBuf, inBuffer, HEADERFRAME1);
  memcpy (finalBuf + HEADERFRAME1, dht_data, DHT_SIZE);
  memcpy (finalBuf + HEADERFRAME1 + DHT_SIZE,
          inBuffer + HEADERFRAME1,
          (mySize -     HEADERFRAME1));

  fwrite (finalBuf, mySize + DHT_SIZE, 1, foutput);
  fclose(foutput);
}
```