Programmation de l'API Video for Linux

Pierre Ficheux (pierre.ficheux@openwide.fr)

Mars 2005

Résumé

Cet article décrit l'interface de programmation *Video for Linux* (plus communément appelée V4L) destinée à l'utilisation de périphériques de capture vidéo comme les cartes d'acquisition ou les caméras. Outre l'utilisation des pilotes existants, l'article décrit également les bases de l'écriture d'un pilote de périphériques V4L (caméra virtuelle) dans le cas des noyaux 2.4 et 2.6.

Introduction

Le problème du pilotage des périphériques vidéos est assez complexe car il existe une multitude de possibilités tant au niveau du type de périphérique (caméra, carte d'acquisition), des caractéristiques de ces périphériques (couleur ou noir et blanc, taille d'image, résolution) de l'architecture matérielle (type de circuit d'acquisition utilisé) ou des modes de connexion (bus PCI, USB, parallèle). Pour simplifier les choses, les développeurs du noyau Linux ont défini une interface appelée *Video for Linux* (Video4Linux ou V4L). Cette interface présente une API unique aux applications ce qui permet à ces dernières de fonctionner quel que soit le périphérique utilisé. En fait, la complexité est reportée au niveau du pilote de périphérique qui doit être conçu de manière à ce que l'application puisse lire les caractéristiques du périphérique puis effectuer le paramétrage et l'acquisition. La structure schématique de l'API est donnée par la figure ci-dessous.

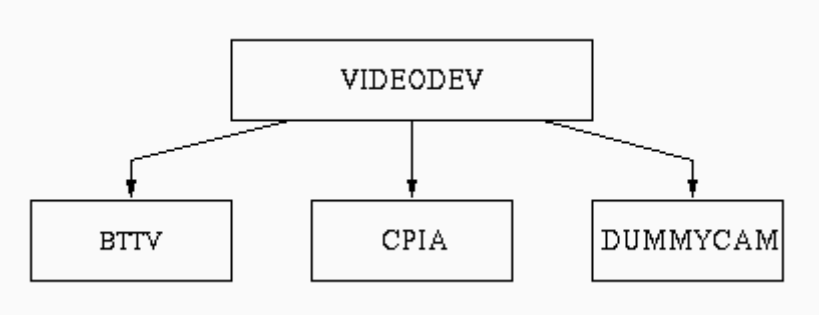

Figure 1: Structure de l'API Video4Linux

L'application utilise le module générique *videodev* qui correspond à l'entrée `/dev/video0` dans le système de fichiers Linux. Ce module *videodev* utilise ensuite les modules spécifiques associés aux périphériques réels. Il est bien entendu possible d'utiliser plusieurs périphériques qui correspondront alors aux entrées `/dev/video1`, `/dev/video2`, etc.

Il faut noter qu'une nouvelle API appelée V4L2 (pour *Video for Linux 2*) est toujours en cours de développement. Elle est supposée être plus évoluée que V4L et combler certaines lacunes. Cependant celle-ci ne semblait pas suffisamment mature pour l'intégration dans le noyau 2.6 et c'est donc toujours l'API V4L qui est disponible en standard.

Configuration du système pour l'utilisation de V4L

Les noyaux fournis avec les distributions classiques (Red Hat, Mandrake, Debian, etc.) sont paramétrés pour l'utilisation de V4L. En général il suffit de spécifier le type de périphérique associé dans le fichier `/etc/modules.conf` pour le noyau 2.4 (ou bien `/etc/modprobe.conf` pour le noyau 2.6). L'exemple ci-dessous est extraite des fichiers d'exemple du répertoire `Documentation/video4linux` livré avec l'arborescence des sources du noyau Linux.

```
# bttv
alias       char-major-81     videodev
alias       char-major-81-0   bttv
options     bttv              card=2 radio=1
options     tuner             debug=1
```

Cette exemple correspond à l'utilisation d'un périphérique de capture à base de contrôleur Bt8x8 qui est le type le plus répandu pour les cartes de capture grand public. Dans le cas de périphériques de captures USB comme les *webcams*, cette configuration ne sera pas toujours nécessaire et la connexion d'une caméra supportée par le noyau Linux conduira automatiques à la disponibilité du point d'entrée `/dev/video0`.

Si l'on veut utiliser un noyau adapté, il est nécessaire de valider quelques options au niveau de la configuration de ce noyau. La configuration est effectuée au travers de la procédure classique `make xconfig` pour 2.4 et `make gconfig` pour 2.6.

Dans le cas du noyau 2.4, il est faut tout d'abord valider l'option *Video for Linux* dans la rubrique *Multimedia devices* du menu principal. Ce point correspond à la compilation du module générique *videodev*.

Figure 2: Validation du support V4L pour le noyau 2.4

En cliquant sur le bouton *Video for Linux* de l'écran précédent, on peut sélectionner les périphériques matériels utilisables. Dans notre exemple nous validons le support BT8x8 et CPIA. Attention car l'accès à certains périphériques V4L peut être conditionné par la validation d'autres options du noyau (exemple: le bus I2C pour le BT8x8).

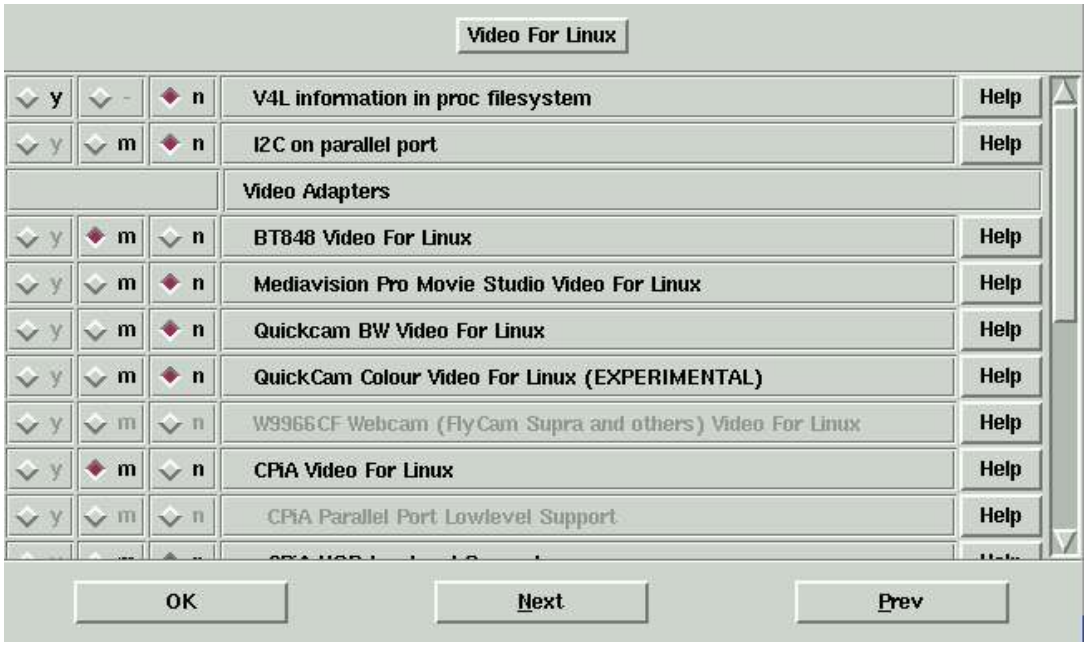

Figure 3: Sélection des périphériques V4L pour le noyau 2.4

Dans le cas du noyau 2.6, la configuration est similaire sauf que l'on peut la visualiser sur un écran unique.

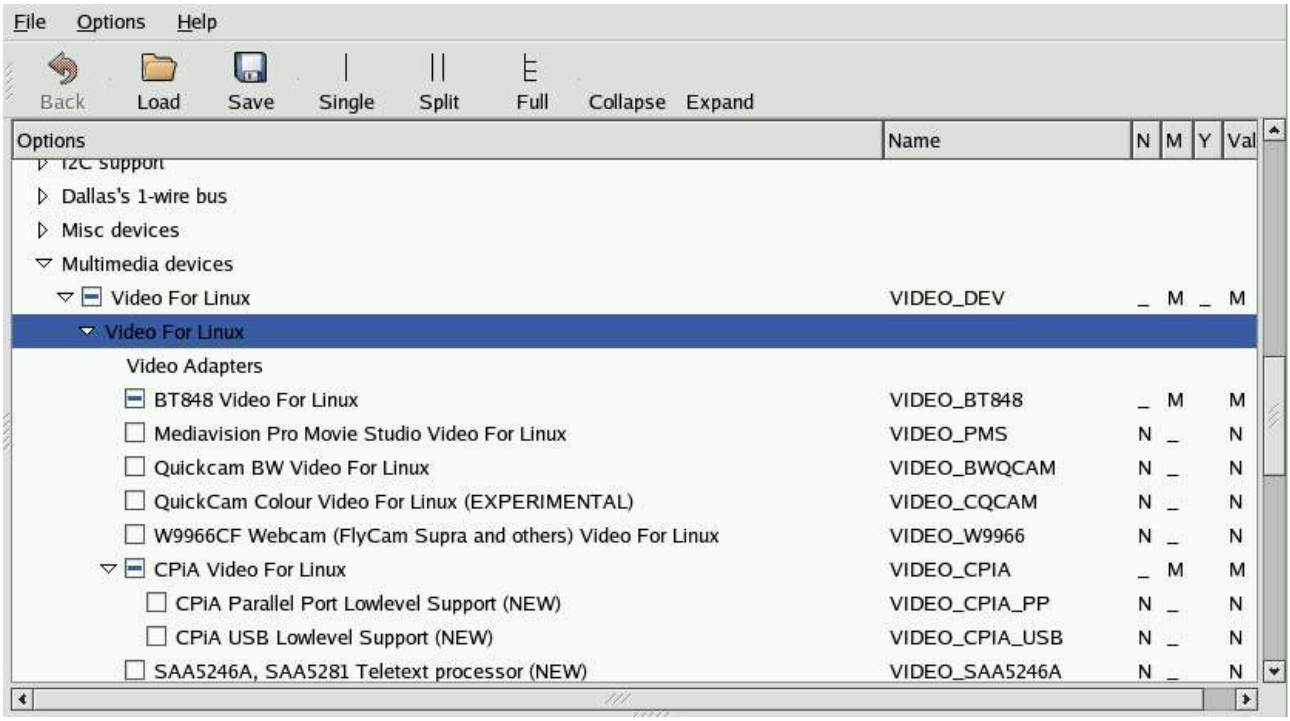

Figure 4: Validation du support V4L et des périphériques pour le noyau 2.6

Test d'un périphérique V4L

Lorsqu'un noyau compatible V4L est installé, on peut alors effectuer un test en utilisant un périphérique V4L. En effet, il existe de nombreuses applications permettant d'exploiter cette interface. Le but de cet article n'est pas de les recenser et d'autres publications sont beaucoup plus à même de fournir les meilleurs conseils pour regarder la TV sur votre PC Linux ou bien enregistrer des séquence vidéos avec la meilleure qualité possible. Dans le cas présent nous utiliserons la configuration suivante:

- Caméra couleur RGB connectée à une carte d'acquisition PCI à base de BT-878 (pilote *bttv*)
- Test avec l'application d'enregistrement *transcode*

Détection du périphérique

Si le support est configuré en module (et non en statique) dans le noyau, la carte est détectée lors de son utilisation par un programme d'acquisition grâce à la commande `dmesg`.

```
Linux video capture interface: v1.00
i2c-core.o: i2c core module
i2c-algo-bit.o: i2c bit algorithm module
bttv: driver version 0.7.96 loaded
bttv: using 4 buffers with 2080k (8320k total) for capture
bttv: Host bridge is Intel Corp. 82815 815 Chipset Host Bridge and Memory
Controller Hub
bttv: Bt8xx card found (0).
...
bttv0: registered device video0
bttv0: registered device vbi0
bttv0: PLL: 28636363 => 35468950 ... ok
```

Le programme Transcode

L'application *transcode* est fréquemment utilisée dans les systèmes d'enregistrement audio et vidéo sous Linux. C'est un programme extrêmement modulaire et extensible par *plugins*. Par défaut, transcode permet d'enregistrer et de convertir des données vers de nombreux formats audio et vidéo (MP3, MPEG1, MPEG2, MPEG4, JPEG, etc.). La compilation et l'installation ne présentent pas de difficulté puisque *transcode* utilise *autoconf/automake*. Il suffit donc d'utiliser la séquence:

```
$ ./configure
$ make
...
# make install
```

Lorsque le programme est installé, on peut faire une acquisition en utilisant la ligne de commande suivante, qui produira une liste de fichiers JPEG:

```
$ transcode -i /dev/video0 -g 320x240  -k -y jpg -o ./img_
transcode v0.6.12 (C) 2001-2003 Thomas Oestreich, 2003-2004 T. Bitterberg
[transcode] (probe) suggested AV correction -D 0 (0 ms) | AV 0 ms | 0 ms
[transcode] auto-probing source /dev/video0 (ok)
[transcode] V: import format    | unknown V4L (V=v4l|A=null)
[transcode] V: import frame     | 320x240  1.33:1
[transcode] V: bits/pixel       | 0.938
```

```
...
```

Cette commande crée une liste de fichiers JPEG nommés `img_000000.jpg, img_000001.jpg`, etc. La figure suivante correspond à la première image enregistrée.

Figure 5: Image JPEG capturée par transcode

Programmation de l'API V4L

Dans ce paragraphe nous allons décrire la syntaxe de l'API autour d'une application minimale permettant de capturer des images depuis un périphériques V4L. La documentation concernant l'API V4L n'est pas toujours très pédagogique mais elle est cependant décrite dans le fichier `Documentation/video4linux/API.html` des sources du noyau Linux. Quelques documents annexes sont disponibles sur Internet et cités dans la bibliographie.

Le programme décrit est volontairement très simple et n'a pour but que de démontrer le fonctionnement de l'API. Les sources du programme sont localisées dans le répertoire `test` de l'archive `dummy_camera-0.2` disponible sur l'espace de télé-chargement de cet article. Pour compiler le programme, il suffit de taper `make` dans le répertoire des sources.

Ouverture du périphérique

Comme pour tous les pilotes Linux, le périphérique est accessible par un fichier spécial, ici `/dev/video0`.

```
#define VIDEO_DEV "/dev/video0"

int fdv;

fdv = open (VIDEO_DEV, O_RDWR);
if (fdv <= 0) {
  perror (VIDEO_DEV);
  exit (1);
}
```

Lecture des capacités (ou capabilities) du périphérique

Cette partie permet de déterminer les paramètres du périphérique comme son nom, son type, le nombre de canaux disponibles, les dimensions d'image minimales et maximales supportées. Concernant les canaux, une carte pourra parfois comporter une entrée TV et une entrée vidéo composite ce qui fait deux canaux.

```
struct video_capability vcap;

if (ioctl (fdv, VIDIOCGCAP, &vcap) < 0) {
  perror ("VIDIOCGCAP");
  exit (1);
}

printf ("Video Capture Device Name : %s\n", vcap.name);
```

Lecture de la liste des canaux du périphérique

Cette portion de code permet d'afficher les noms et caractéristiques des canaux (utilisation de tuner ou non, etc.).

```
struct video_channel vc;

for (i = 0 ; i < vcap.channels ; i++){
  vc.channel = i;
  if(ioctl (fdv, VIDIOCGCHAN, &vc) < 0) {
    perror ("VIDIOCGCHAN");
    exit (1);
  }

  printf("Video Source (%d) Name : %s\n", i, vc.name);
}
```

Nous pouvons ensuite sélectionner le canal d'enregistrement ainsi que le standard de la source vidéo (PAL, SECAM ou NTSC). Dans notre cas, le numéro du canal est spécifié par l'option `-c`.

```
vc.channel = channel;
vc.norm = VIDEO_MODE_PAL;

if (ioctl(fdv, VIDIOCSCHAN, &vc) < 0){
  perror ("VIDIOCSCHAN");
  exit (1);
}
```

Attachement de la mémoire vidéo (mmap)

Ce point est probablement le plus caractéristiques et le plus important de l'API V4L. Vu que la taille des données transférées entre l'application et le pilote est importante (taille d'une image vidéo), on peut difficilement envisager que le transfert se fasse par un appel système *read*. On utilise donc l'appel système *mmap* qui permet d'attacher la mémoire du périphérique à un pointeur situé dans l'espace utilisateur. Lorsque l'attachement est réalisé, les données sont lues par l'application comme de la mémoire locale. Il est bien sûr nécessaire de définir une entrée *mmap* dans le pilote, ce qui sera décrit au paragraphe suivant. Coté application, le code est décrit ci-après. Le deuxième paramètre

d'appel correspond à la taille de la mémoire attachée, soit largeur fois hauteur par 3 puisque nous coderons l'image en RGB 24 bits (1 octet par couleur).

```
  unsigned char *framebuf;

  framebuf = (unsigned char*)mmap(0, width * height * 3, PROT_READ | PROT_WRITE,
MAP_SHARED, fdv, 0);

  if ((unsigned char *)-1 == (unsigned char *)framebuf) {
    perror ("mmap");
    return(-1);
  }
```

Capture des images

La capture est une boucle basée sur deux appels *ioctl*. Nous capturons 10 images en augmentant la luminosité de l'image à chaque tour de boucle.

1. L'appel *VIDIOCMCAPTURE* demande la capture d'une image au périphérique. Nous précisons en paramètre la taille de l'image et le format de codage (soit ici *RGB24*)
2. L'appel *VIDIOCSYNC* attend la fin effective de la capture. A la sortie de cet appel, l'image est disponible dans le tampon. Dans notre exemple, nous stockons ensuite l'image dans un fichier au format PPM (Portable PixMap).

```
    struct video_mmap mm;

    /* Set brightness */
    bright_set (fdv, b);
    b += 10;

    /* Set frame size and format */
    mm.frame  = 0;
    mm.height = height;
    mm.width  = width;
    mm.format = VIDEO_PALETTE_RGB24;

    /* Get frame */
    if (ioctl(fdv, VIDIOCMCAPTURE, &mm) < 0) {
      perror ("VIDIOCMCAPTURE");
      exit (1);
    }

    /* Wait frame to be completed */
    if (ioctl(fdv, VIDIOCSYNC, &mm.frame) < 0) {
      perror ("VIDIOCSYNC");
      exit (1);
    }

    /* Save to PPM file */
    if (swap)
      rgb_swap (framebuf, width * height);
    sprintf (buf, "img_%02d.ppm", i);
    write_ppm (buf, framebuf, width, height);
```

La fonction de modification de luminosité (ou *brightness*) est donnée ci-dessous.

```
/* Set brightness from 0 to 100 */
int bright_set (int fd, unsigned int newbright)
{
  struct video_picture vpic;

  if (ioctl (fd, VIDIOCGPICT, &vpic) < 0) {
    perror ("grab_bright_set");
    return -1;
  }

  vpic.brightness = newbright * 65536 / 100;

  return ioctl (fd, VIDIOCSPICT, &vpic);
}
```

Fin du programme

Le programme se termine par la fermeture du fichier spécial associé au périphérique.

```
  close (fdv);
```

Résultat de l'exécution programme

Le programme est exécuté avec la ligne de commande qui suit:

```
$ capture -w 320 -h 240 -s -c 1
```

Le résultat de l'exécution correspond à 10 fichiers PPM. Un exemple d'image est donné par la figure suivante:

Figure 6: Image capturée par le programme de test

Ecriture d'un pilote de périphérique V4L (dummy_camera)

L'écriture d'un périphériques V4L suit les règles d'écriture d'un pilote de périphérique Linux en mode caractère. L'exemple présenté ici est volontairement simple puisqu'il est destiné à retourner

une mire (image fixe) constituée de bandes verticales de couleur. Ce pilote est cependant suffisamment évolué pour fonctionner avec le programme de capture décrit précédemment ou même le programme *transcode*. La base de ce pilote est la version écrite par Kevin Boone (kb@kevinboone.com). Nous avons ajouté à la version de Kevin le support *mmap* ainsi que le support pour intégration au noyau 2.6. La version 2.6 utilise un *Makefile* décrit par Joachim Nilsson (joachim.nilsson@member.fsf.org) dans son *module2.6-howto*.

La principale différence entre les versions 2.4 et 2.6 se situe au niveau de traitement de *mmap*, les macro-instructions à utiliser étant légèrement différentes. D'autres exemples plus complexes de pilotes V4L sont disponibles dans les sources du noyau Linux au niveau du répertoire `drivers/media/video`.

Les structures de définition du pilote

Le pilote est de type caractère, il contient donc une structure *file_operations* classique qui répertorie les différents points d'entrée. Noter qu'il n'y a pas de point d'entrée *camera_write*, car il est peu probable que l'on écrive sur la caméra! La configuration du périphérique s'effectue au travers d'appels *ioctl*.

```
static struct file_operations camera_fops = {
      owner:      THIS_MODULE,
      open:       camera_open,
      release:    camera_close,
      read:       camera_read,
      mmap:       camera_mmap,
      ioctl:      camera_ioctl,
      llseek:     no_llseek,
};
```

Vu que nous avons affaire à un pilote V4L, il est nécessaire d'initialiser une structure *video_device* qui définit les caractéristiques du périphérique.

```
static struct video_device camera = {
      owner:      THIS_MODULE,
      name:       "Dummy camera",
      type:       VID_TYPE_CAPTURE,
      hardware:   VID_HARDWARE_CAMERA,
      fops:       &camera_fops,
      minor:      -1,
};
```

Le chargement du module

Le chargement et l'initialisation s'effectuent par l'entrée classique *init_module*. Dans cette fonction nous effectuons l'allocation dynamique de la mémoire nécessaire au stockage de l'image fixe. Cette adresse mémoire sera ensuite utilisée par l'entrée *mmap*. Pour simplifier nous utilisons pour cela l'appel `kmalloc`, ce qui limite la taille de la mémoire allouée à 128 Ko, d'où la limitation de la taille de l'image à 128x128 pixels et non 320x240 ou plus. Dans cette fonction, nous ajoutons également le périphérique à la liste des périphériques V4L.

```
int init_module()
{
  struct page *page;

  my_image = kmalloc (PIC_BUF_SIZE,GFP_KERNEL);
```

```
  if (!my_image) {
    printk (KERN_INFO "failed kmalloc\n");
    return -EINVAL;
  }

  /* now we've got PIC_BUF_SIZE bytes of kernel memory, but it can still be
     swapped out. We need to stop the VM system from removing our
     pages from main memory. To do this we just need to set the PG_reserved
     bit on each page, via mem_map_reserve() macro. */

  for (page = virt_to_page(my_image); page < virt_to_page(my_image +
PIC_BUF_SIZE); page++) {
    mem_map_reserve(page);
  }

  if(video_register_device(&camera, VFL_TYPE_GRABBER, 0) < 0)
    return -EINVAL;

  return 0;
}
```

Noter que dans le cas du noyau 2.6, l'appel `mem_map_reserve` est remplacé par `SetPageReserved`.

Le dé-chargement du module

Le chargement et l'initialisation s'effectuent par l'entrée classique *cleanup_module*. Noter que dans le cas du noyau 2.6, l'appel `mem_map_unreserve` est remplacé par `ClearPageReserved`. Dans cette fonction nous effectuons également la libération de la mémoire allouée et nous supprimons le périphérique de la liste des périphériques V4L.

```
void cleanup_module()
{
  struct page *page;

  video_unregister_device(&camera);

  for (page = virt_to_page(my_image); page < virt_to_page(my_image +
PIC_BUF_SIZE); page++) {
    mem_map_unreserve(page);
  }

  kfree(my_image);
}
```

L'ouverture du périphérique (open)

Dans notre cas, l'ouverture du périphérique se limite à l'initialisation de la mémoire image par la fonction *update_pictur*e qui construit la mire. Dans le cas du noyau 2.4, nous appelons également la macro *MOD_INC_USE_COUNT* qui n'existe plus en 2.6.

```
static int camera_open(struct inode *inode, struct file *file)
{
  if(usage > 0)
    return -EBUSY;
```

```
  usage++;

  MOD_INC_USE_COUNT;

  update_picture ();

  return 0;
}
```

La fermeture du périphérique (close)

Dans le cas du noyau 2.4, nous appelons la macro *MOD_DEC_USE_COUNT* qui n'existe plus en 2.6.

```
static int camera_close(struct inode *inode, struct file *file)
{
  usage--;
  MOD_DEC_USE_COUNT;

  return 0;
}
```

La lecture du périphérique (read)

La fonction de lecture retourne simplement le contenu de l'image en utilisant la fonction *copy_to_user*. En pratique cette entrée est peu utilisée et un programme de capture utilisera *mmap* pour lire directement le contenu de la mémoire vidéo. Pour simuler le nombre d'images par secondes (FPS) d'un périphérique réel, nous utilisons une fonction d'attente *interruptible_sleep_on_timeout*.

```
static int camera_read(struct file *file, char *buf, size_t count, loff_t *ppos)
{
  init_waitqueue_head(&wait);
  interruptible_sleep_on_timeout (&wait, HZ / FPS);

  if (copy_to_user((void*)buf, (void*)my_image, WIDTH*HEIGHT*3))
    return -EFAULT;

  return WIDTH * HEIGHT * 3;
}
```

L'attachement de la mémoire d'image du périphérique (mmap)

Pour des raisons de concision, nous avons volontairement omis le début de la fonction et le lecteur intéressé se reportera au code source disponible (voir la partie bibliographie). La principale tâche de cette fonction est de « mapper » les différentes pages de la mémoire d'image allouée sur la mémoire du processus. Pour cela on utilise les fonctions *virt_to_phys* et *remap_page_range*, la dernière fonction étant remplacée par *remap_pfn_range* pour le noyau 2.6.

```
static int camera_mmap(struct file *file, struct vm_area_struct *vma)
{
  ...
  /* start off at the start of the buffer */
  pos=(unsigned long) my_image;
```

```
  /* loop through all the physical pages in the buffer
     Remember this won't work for vmalloc()d memory ! */

  while (size > 0) {
    /* remap a single physical page to the process's vma */
    page = virt_to_phys((void *)pos);

    /*
     * Fourth argument is the protection of the map. you might
     * want to use vma->vm_page_prot instead.
     */
    if (remap_page_range(start, page, PAGE_SIZE, PAGE_SHARED))
      return -EAGAIN;

    start += PAGE_SIZE;
    pos += PAGE_SIZE;
    size -= PAGE_SIZE;
  }
 ...
}
```

Le paramétrage du périphérique (ioctl)

Dans le cas d'un pilote V4L, il y a systématiquement une entrée *ioctl* car celle-ci correspond aux différentes fonctions de paramétrage de l'API V4L (voir la partie *Programmation de l'API V4L* plus haut dans l'article). Comme précédemment, nous présentons ci-dessous un extrait de la fonction.

L'appel *VIDIOCGCAP* permet de récupérer les caractéristiques du périphérique. Dans notre cas nous retournons une taille fixe (128 sur 128 pixels) et un seul canal de capture. Le périphérique est nommé *Dummy camera*.

```
static int camera_ioctl(struct inode *inode, struct file *file, unsigned int
cmd, unsigned long arg)
{
  ...
  switch(cmd) {

    case VIDIOCGCAP:
    {
      struct video_capability v;

      v.type = VID_TYPE_CAPTURE;
      v.channels = 1;
      v.audios = 0;
      v.maxwidth = width;
      v.minwidth = height;
      v.maxheight = height;
      v.minheight = width;
      strcpy(v.name, "Dummy camera");

      if(copy_to_user((void*)arg, &v, sizeof(v)))
         return -EFAULT;
      else
         return 0;
    }
```

L'appel *VIDIOCGCHAN* permet de récupérer les caractéristiques du canal. Dans notre cas il n'y a qu'un seul canal nommé *Camera*.

```
    case VIDIOCGCHAN:
    {
      struct video_channel v;

      if(copy_from_user(&v, (void*)arg, sizeof(v)))
      return -EFAULT;

      v.flags = 0;
      v.tuners = 0;
      v.type = VIDEO_TYPE_CAMERA;
      v.norm = VIDEO_MODE_AUTO;
      v.channel = 0;
      strcpy(v.name, "Camera");

      if(copy_to_user((void*)arg, &v, sizeof(v)))
      return -EFAULT;
      else
      return 0;
    }
```

Dans le cas d'un périphérique réel, l'appel *VIDIOCSYNC* permet d'attendre la disponibilité effective dans données dans le tampon de capture. Dans notre cas, il n'est présent que pour compatibilité et se limite donc à un *return 0*.

```
  case VIDIOCSYNC:
    return 0;
```

Dans le cas d'un périphérique réel, l'appel *VIDIOCMCAPTURE* demande la capture d'une image au périphérique à partir des paramètres envoyés par l'application (dimensions de l'image, format, etc.). Dans notre cas, la fonction se contente d'afficher les paramètres et de simuler les nombre de trames par seconde avec une fonction d'attente.

```
   case VIDIOCMCAPTURE:
    {
      struct video_mmap *vm = (struct video_mmap*)arg;

#ifdef DEBUG
      printk (PREFIX "VIDIOCMCAPTURE: frame= %d w= %d h= %d format= %d\n", vm-
>frame, vm->width, vm->height, vm->format);
#endif

      /* Just wait to simulate FPS */
      init_waitqueue_head(&wait);
      interruptible_sleep_on_timeout (&wait, HZ / FPS);

      return 0;
    }
```

Compilation et test du pilote

Les sources du pilotes sont situées dans les répertoires `2.4` et `2.6` de la distribution *dummy_camera-0.2*, disponible en télé-chargement (voir bibliographie). Le pilote est compilé en utilisant la commande `make`.

Avant d'utiliser le pilote il est nécessaire de charger les modules, soit les commandes suivantes dans le cas du noyau 2.4:

```
# modprobe videodev
# insmod dummycam.o
```

ou bien les commandes suivantes dans le cas du noyau 2.6:

```
# modprobe videodev
# insmod dummycam.ko
```

Comme avec le pilote *bttv*, nous effectuons un test de capture avec notre petit programme de test. La taille de l'image étant par défaut de 128 sur 128 pixels, l'appel du programme se réduit à:

```
$ ./capture
Video Capture Device Name : Dummy camera
Video Source (0) Name : Camera
image 0, brightness = 5
image 1, brightness = 15
image 2, brightness = 25
image 3, brightness = 35
image 4, brightness = 45
image 5, brightness = 55
image 6, brightness = 65
image 7, brightness = 75
image 8, brightness = 85
image 9, brightness = 95
```

Le résultat de l'exécution correspond à 10 fichiers PPM. Un exemple d'image est donné par la figure suivante:

Figure 7: Image de notre caméra virtuelle capturée par le programme de test

Nous pouvons également effectuer la capture avec le programme transcode. L'appel est identique à celui utilisé par *bttv* mis à part la taille de l'image et l'absence d'inversion des composantes rouges et bleues (option *-k*).

```
$ transcode  -i /dev/video0 -g 128x128 -y jpg -o ./dummycam_
transcode v0.6.12 (C) 2001-2003 Thomas Oestreich, 2003-2004 T. Bitterberg
[transcode] (probe) suggested AV correction -D 0 (0 ms) | AV 0 ms | 0 ms
```

```
[transcode] auto-probing source /dev/video0 (ok)
[transcode] V: import format    | unknown V4L (V=v4l|A=null)
[transcode] V: import frame     | 128x128  1.00:1
...
encoding frames [000000-000014],   3.00 fps, EMT: 0:00:00, ( 0| 0| 0)
...
```

On note que le nombre de trames par seconde est bien de 3, comme programmé dans le pilote. Au final on obtient des images JPEG similaires à la figure 7.

Bibliographie

- *Introduction to Video for Linux* à l'adresse http://www.glue.umd.edu/~ankurm/video4linux/introduction.html
- *Video4Linux Programming* à l'adresse http://www.kernelnewbies.org/documents/kdoc/videobook/v4lguide.html
- *Video4Linux Kernel API Reference* dans les sources du noyau Linux, fichier `Documentation/video4linux/API.html`
- Exemple d'utilisation de *mmap* pour le noyau 2.4 à l'adresse http://kernelnewbies.org/code/mmap
- Le même exemple adapté pour le noyau 2.6 sur http://pficheux.free.fr/articles/lmf/v4l/mmap_example_2.6.tgz
- Quelques informations concernant la construction de modules externes pour le noyau 2.6 sur http://vmlinux.org/jocke/linux/external-modules-2.6.shtml
- Version originale du pilote *dummy_camera-0.1* par Kevin Boone sur http://www.kevinboone.com/README_dummy-camera.html
- Codes sources du pilote *dummy_camera-0.2* sur http://pficheux.free.fr/articles/lmf/v4l/dummy_camera-0.2.tgz
- Le programme de capture vidéo *transcode* à l'adresse http://www.theorie.physik.unigoettingen.de/~ostreich/transcode