Driver bus OPP sous Linux

Andrei Bejenari
andrei.bejenari@inrialpes.fr

16 septembre 2005

1 Introduction

Open Parallel Platform™ (OPP) est un bus parallèle développé par Wany Robotics. Le bus permet d'envoyer des données à plusieurs récepteurs simultanément et fonctionne sans maître du système. Pour la description bas niveau du bus voir la notice électronique [1]. Dans le cas du robot Pekee le bus OPP permet la communication entre le microcontrôleur du robot et la carte PC embarqué. Le contrôleur gère tout l'équipement du robot (les capteurs, les moteurs, ...). Ansi, en envoyant des trames à travers le bus OPP, le PC peut interroger les capteurs, commander les moteurs, etc.

La liaison entre le processeur STPC et le bus OPP se fait par l'intermédiaire d'une couche d'interfaçage présente sur la carte PC embarqué. Les trames arrivant du bus OPP sont stoquées dans les FIFOs de la couche. Le processeur n'a qu'à lire certains ports pour récupérer le contenu des FIFOs. L'envoi de données vers le bus OPP se fait aussi à travers la couche d'interfaçage. L'écriture sur des ports agit sur les signaux de commande de l'interface permettant le contrôle de transmission. La couche d'interfaçage est décrite dans [2].

Le driver implémente les fonctions nécessaires pour envoyer et recevoir les données sur le bus OPP, ainsi que pour le contrôler. Le driver est thread-safe.

La section suivante décrit les procédures de compilation et de chargement du module. Ensuite, on présente l'interface de programmation. La dernière section explique la structure interne du module.

2 Installation

2.1 Compilation

Les codes sources du driver se trouvent dans le répertoire `src/drivers/` du projet. Dans le fichier `Makefile` il faut indiquer le chemin vers le répertoire du projet dans la variable `TOPDIR` et le chemin vers les en-têtes du noyau dans la variable `KERNELDIR`. Ensuite, il suffit de lancer `make`. Pour mettre les drivers dans `bin/` du projet il faut exécuter `make install`.

2.2 Chargement du module

Le driver OPP se trouve dans `/lib/modules/`*`release`*`/kernel/drivers/misc` sur le PC du Pekee. Le driver admet un paramètre `opp_major` qui indique le majeur du périphérique. Pour charger le module il faut exécuter :

```
# insmod opp [opp_major=NUM]
```

Ensuite, il faut créer un nœud vers ce périphérique dans `/dev`. On retrouve son majeur avec la commande suivante :

```
# grep opp /proc/devices
NUM opp
```

Et on crée le nœud :

```
# mknod -m 666 /dev/opp c NUM 0
```

Toute cette procédure est automatisée dans un script init.d `opp` qu'on trouve dans le répertoire `init.d/` du projet.

3 API

Ce driver est un driver de type caractère. On y accède à travers son nœud `/dev/opp`. Pour recevoir/envoyer des données, il suffit de lire/écrire des trames sur ce périphérique. Le driver utilise le même format de trame pour la lecture et l'écriture, celui-ci est décrit dans [3]. Le champ *Data Size*

est utilisé par le driver pour connaître la taille de la trame, donc le buffer passé en paramètre doit contenir au moins deux mots de 16 bits. Le checksum est géré par le driver et ne figure pas dans les trames lues. Pour communiquer sur le bus le driver offre deux interfaces. Les appels-système read() et write() permettent de recevoir et d'envoyer des trames sur le bus OPP. De même, on peut utiliser les requêtes ioctl qui offrent de plus des fonctions de contrôle de la carte.

3.1 read() et write()

Un appel read() retourne une trame dans le buffer passé en second paramètre. S'il n'y a pas de trame disponible et l'attribut O_NONBLOCK n'est pas fixé, le processus appelant va être endormi. Un appel write() écrit sur le bus une trame passée en second paramètre. Pour le deux fonction le format de la trame est la suite de champs suivants avec les tailles respectives :

Champ	Function Key	Data Size	Datas
Taille (octets)	2	2	$2 \times$ Data Size

Cette structure, OPPFRAME, est définie dans le fichier en-tête `opp/OppStructDef.h` qu'on trouve dans `include/` du projet. Dans les appels de read() et write() on peut passer simplement le pointeur vers cette structure.

Listing 1 – Exemples de read() et write().

```
#include <opp/OppDef.h>

OPPFRAME frame;
int opp_fd = open('/dev/opp', O_RDWR);
5
frame.wFunctionKey = FK_SET_SOUND;
frame.wDataSize = 3;
frame.Datas[0] = 1;                          // wSignature
frame.Datas[1] = FK_SOUND_TYPE_BUZZER;       // wType
10 frame.Datas[2] = 500;                     // wFrequency

write(opp_fd, &frame, (frame.wDataSize+2)*sizeof(WORD));
/* Pekee beeps */

15 /* ... */

frame.wFunctionKey = FK_GET_MOVE;
frame.wDataSize = 2;
frame.Datas[0] = 1;                          // wSignature
20 frame.Datas[1] = FK_GETMOVE_TYPE_POSITION; // wType

write(opp_fd, &frame, (frame.wDataSize+2)*sizeof(WORD));
read(opp_fd, &frame, sizeof(OPPFRAME));
/* frame contains Pekee's position */
```

3.2 ioctl()

Le requêtes ioctl sont définies dans le fichier `opp/opp.h`. On présente ici les fonctionnalités de chacune.

3.2.1 OPP_IOCRESETFIFO

Cette commande permet de vider le buffer de la couche d'interfaçage.

Listing 2 – Exemple de OPP_IOCRESETFIFO

```
#include <opp/opp.h>

int opp_fd = open(OPP_DEVICE, O_RDWR);

ioctl (opp_fd, OPP_IOCRESETFIFO);
```

3.2.2 OPP_IOCPOWEROFF

Cette requête arrête la carte en coupant son alimentation. Il vaut mieux faire au moins un `sync` avant d'envoyer cette commande.

Listing 3 – Exemple de OPP_IOCPOWEROFF

```
ioctl (opp_fd, OPP_IOCPOWEROFF);
```

3.2.3 OPP_IOCBATTFULL

Cette requête retourne dans un octet la valeur 1 si la batterie de la carte PC embarqué est chargée et 0 sinon. Le troisième paramètre de ioctl () est un pointeur vers un octet.

Listing 4 – Exemple de OPP_IOCBATTFULL

```
char is_full ;

ioctl (opp_fd, OPP_IOCBATTFULL, &is_full);
```

3.2.4 OPP_IOCSLOTNUM

Cette requête renvoie le numéro du slot dans lequel la carte est insérée (position physique sur le bus OPP). La numérotation commence à 0 (la tête de Pekee) et se poursuit jusqu'à 4 (la queue de Pekee).

Listing 5 – Exemple de OPP_IOCSLOTNUM

```
unsigned char slot_no;

ioctl (opp_fd, OPP_IOCSLOTNUM, &slot_no);
```

3.2.5 OPP_IOCSOFTVER

Cette requête retourne le version du soft du EPLD (logique programmable).

Listing 6 – Exemple de OPP_IOCSOFTVER

```
unsigned char soft_ver;

ioctl (opp_fd, OPP_IOCSOFTVER, &soft_ver);
```

3.2.6 OPP_IOCRECVFRAME

Cette commande est identique à un appel read(). L'adresse de la trame est passée en troisème paramètre de ioctl ().

Listing 7 – Exemple de OPP_IOCRECVFRAME

```
#include <opp/OppStructDef.h>

OPPFRAME frame;

/* ... */

ioctl (opp_fd, OPP_IOCRECVFRAME, &frame);
```

3.2.7 OPP_IOCSENDFRAME

Cette commande est identique à un appel write(). L'adresse de la trame est passée en troisème paramètre de ioctl ().

Listing 8 – Exemple de OPP_IOCSENDFRAME

```
OPPFRAME frame;

/* ... */

ioctl (opp_fd, OPP_IOCSENDFRAME, &frame);
```

4 Internals

Le code du driver se compose de quatre fichiers : `Configuration_OPP.h`, `opp.c`, `opp_p.h` et `opp.h`. Le dernier défini l'interface du driver. opp_p.h est privé au module et permet de configurer certain paramètres du driver. `Configuration_OPP.h` contient les macros permettant d'accèder à la couche d'interfaçage OPP.

4.1 Chargement et retrait du module

Au chargement—le noyau invoque opp_init (), le module alloue les ports définis dans `Configuration_OPP.h` et enregistre un périphérique de type caractère. La structure du listing 9 contient les pointeurs vers les fonctions implémentées dans le module.

Listing 9 – Appels-système implémentés dans le module.

```
static struct file_operations opp_fops =
{
    owner:      THIS_MODULE,
    open:       opp_open,
    release:    opp_close,
    read:       opp_read,
    write:      opp_write,
    ioctl:      opp_ioctl
};
```

On passe opp_fops en troisiéme paramètre de l'appel register_chrdev (). Ainsi, le noyau établit la correspondance entre les appels-système et les fonctions du module. Par exemple, quand une application va

appeler open() ou write(), le noyau va invoquer opp_open() et opp_write() respectivement. Les mutexes rlock et wlock sont utilisés pour assurer la lecture et l'écriture d'une trame en entier par un seul processus. Les deux variables sont de type sémaphore. La fonction init_MUTEX() initialise un sémaphore à 1, i.e. une exclusion mutuelle.

Pour décharger le module, le noyau appele opp_exit() et libère les ressources occupées par celui-ci.

4.2 Overture et fermeture du périphérique

La variable opp_open_count compte le nombre d'ouvertures du périphérique. Quand celle-ci est nulle (première ouverture), le driver alloue l'interruption numéro 7. Il s'agit de l'interruption, décrite dans [2], active à chaque réception d'un en-tête valide. Elle est reliée directement à la ligne 7 du contrôleur d'interruption i8259. Cette interruption permet de faire un read() (et le ioctl() respectif) bloquant. Quand aucun processus n'utilise plus ce périphérique l'interruption est libérée.

4.3 Lecture et écriture

L'envoi et la réception de données se fait à l'aide des fonctions opp_send_frame() et opp_receive_frame(). Elles implémentent le protocole de communication décrit dans [2]. La gestion de mutex autour de celles-ci et du flag O_NONBLOCK se fait dans les fonctions opp_do_send() et opp_do_receive().

La lecture d'un mot se fait par la fonction opp_receive_frame(). Avant d'appeler la macro FONCTION_OPP_RD_FIFO() pour récupérer le mot suivant, on doit vérifier que la fifo est prête (sinon on lit une valeur erronée). Pour cela on regarde l'état du flag OPP_b_Empty_Flag. Si la fifo n'est pas prête au bout de OPP_RECEIVE_TIMEOUT secondes, la fonction renvoie un timeout. Une trame est lue mot par mot dans opp_send_frame() dans un tampon alloué par celle-ci. Quand la trame a été lue en entier, on vérifie sa checksum. Avant de commencer la lecture, opp_do_receive() vérifie l'état de O_NONBLOCK. Si le descripteur de fichier est bloquant et il n'y a pas de données disponibles dans la fifo le processus sera endormi. L'attente se fait sur la queue opp_wait_frame. Son exécution continuera á la réception d'un en-tête valide, cela est géré par la routine de traitement d'interruption opp_irq_handler().

Pour l'écriture on n'a pas d'interruption qui pourrait nous indiquer quand le bus est libre. Si le bus est occupé et le descripteur de fichier est non bloquant la fonction opp_do_send() revient immédiatement. Sinon la fonction opp_send_frame() est appelée. Cette fonction tente de déposer une requête (prendre possession du bus) pendant OPP_SEND_TIMEOUT secondes. Ensuite si la carte devient propriétaire du bus, la fonction envoie la trame avec son checksum. Dans le cas contraire, la fonction renvoie un timeout.

Références

[1] *Bus OPP électronique.*
`P003A-V1-NOT9-A- Bus OPP electronique_FR.PDF`

[2] *Carte de développement OPP.*
`OPP Prototyping Board User Manual.pdf` ou `P016B-V1-EMP-A_FR.PDF`

[3] *Bible des Function Keys.*
`Bible des FunctionsKey.pdf` ou `FunctionsKey Bible.pdf`

[4] Alessandro Rubini and Jonathan Corbet. *Linux Device Drivers.* O'Reilly, 2nd Edition, 2001.
Consultable sur le web : `http://www.xml.com/ldd/chapter/book/`

Driver V4L pour STPC Video Input Port

Andrei Bejenari
andrei.bejenari@inrialpes.fr

16 septembre 2005

1 Introduction

L'acquisition d'images de la caméra du Pekee se fait à travers deux périphériques. Cette architecture est schématisée sur la figure 1. La caméra de Pekee est un capteur CMOS C-Cam8 (*voir* [1]) qui renvoie

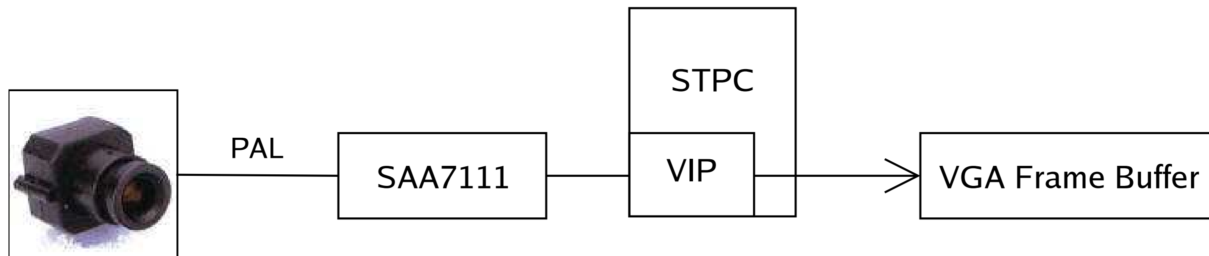

Fig. 1 – Flux vidéo.

un signal analogique. Elle est connectée au processeur vidéo SAA7111 (*voir* [2]) qui convertit son signal en un signal numérique. Finalement, Video Input Port du STPC (*voir* [3] et [4]) écrit l'image dans la RAM à une adresse spécifiée (mais qui ne peut être que dans le VGA Frame Buffer).

Le driver est compatible avec l'interface Video for Linux. Celle-ci permet d'utiliser de manière uniforme les caméras et les tuners. Ainsi une application peut fonctionner avec des différentes caméras sans aucun changement dans le code.

La section suivante décrit les procédures de compilation et de chargement du module. Ensuite, on présente l'interface de programmation. La dernière section explique la structure interne du driver.

2 Installation

2.1 Compilation

Les codes sources du driver se trouvent dans le répertoire `src/drivers/` du projet. Le driver utilise le kit de développement STPC[1]. Le fichier `MakeKrlLib` sert à compiler une librarie contenant les fonctions utilisées par le driver. Il faut le recopier (ou mieux juste créer un lien symbolique) dans le répertoire `lib/` du kit. Attention : après la décompression du kit téléchargé ses fichiers sources sont en format DOS, donc avant de les compiler il faut faire la conversion en format UNIX (par exemple, en faisant :

```
for F in `find -name '*.[ch]' -print`; do dos2unix $F; done
```

dans la racine du kit). Avant de lancer `make` il faut configurer dans `Makefile` et `MakeKrlLib` les variables suivantes : `TOPDIR`, `KERNELDIR` et `GDKDIR` qui désignent les chemins vers le répertoire du projet, les en-têtes du noyau et la racine du kit respectivement. Ensuite, on lance `make`. Pour mettre les drivers dans `bin/` du projet il faut exécuter `make install`.

2.2 Chargement du module

Le driver OPP se trouve dans `/lib/modules/`*`release`*`/kernel/drivers/misc` sur le PC du Pekee. Avant de charger le module il faut d'abord charger la couche V4L `videodev`. Ensuite on charge `stpcvip.o`. Cette procédure est automatisée dans un script init.d `stpcvip` qu'on trouve dans le répertoire `init.d/` du projet.

[1] STPC Development Kit peut être télécharger sur le site de ST : `http://www.stmcu.com`

3 API

Ce driver implémente l'interface Video for Linux. Celle-ci est décrite dans [5] qu'on trouve dans l'arborescence source Linux. V4l définit principalement les requêtes ioclt permettant d'obtenir et de fixer les paramètres de la caméra et de l'image, ainsi que de faire la capture de celles-ci.

3.1 Ouverture du périphérique vidéo

Le module `videodev` enregistre un périphérique dont le majeur est 81 et chaque caméra obtient un mineur. Au chargement, tout module compatible v4l s'enregistre auprès de `videodev`. On peut accéder au driver en ouvrant le nœud vidéo :

```
int fd = open("/dev/video0", O_RDRW);
```

3.2 ioctl()

3.2.1 VIDIOCGCAP : Get capabilities

Cette requête retourne les capacités de la caméra. La structure retournée contient le nom, le type, le nombre de canaux, le nombre de canaux audio et les dimensions min et max.

Listing 1 – Exemple de VIDIOCGCAP

```
struct video_capability cap;
if ( ioctl (fd , VIDIOCGCAP, &cap) < 0)
    exit (1);

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

3.2.2 VIDIOCGCHAN : Get channel info (sources)

Cette requête retourne les information concernant un canal. Pour appeler cette fonction, il faut remplir le champ channel. Ici on n'a qu'un seul canal (numéro 0).

Listing 2 – Exemple de VIDIOCGCHAN

```
struct video_channel vc;
vc.channel = 0;
if ( ioctl (fd , VIDIOCGCHAN, &vc) < 0)
    exit (1);

printf ("Channel 0 name : %s\n", vc.name);
```

3.2.3 VIDIOCSCHAN : Set channel

Cette fonction change le canal. Une caméra n'en a qu'un seul.

Listing 3 – Exemple de VIDIOCSCHAN

```
int vc = 0;

if ( ioctl (fd , VIDIOCSCHAN, &vc) < 0)
    exit (1);
```

3.2.4 VIDIOCGMBUF : Memory map buffer info

Cette requête retourne les informations concernant les buffers de la caméra. Un pointeur sur cette mémoire peut être obtenu en utilisant la fonction mmap().

Listing 4 – Exemple de VIDIOCGMBUF

```
struct video_mbuf mbuf;
unsigned char *buf1, *buf2, *base;

if ( ioctl (fd , VIDIOCGMBUF, &mbuf) < 0)
    exit (1);

printf ("Number of frame buffers : %d\n", mbuf.frames);

base = (unsigned char *) mmap(0, mbuf.size, PROT_READ | PROT_WRITE, MAP_SHARED, vidfd, 0);
buf1 = base + mbuf.offsets [0];
buf2 = base + mbuf.offsets [1];
```

3.2.5 VIDIOCMCAPTURE : Grab frames

Cette fonction lance la capture dans un des buffers du driver. Le champ frame doit contenir le numéro du buffer à écrire. STPC VIP écrit une image en YUV422, donc le driver n'accepte que ce format.

Listing 5 – Exemple de VIDIOCMCAPTURE

```
struct video_mmap mm;

mm.height = 288;
mm.width = 360;
mm.format = VIDEO_PALETTE_YUV422;
mm.frame = 0;

if ( ioctl (fd , VIDIOCMCAPTURE, &mm) < 0)
    exit (1);
```

3.2.6 VIDIOCSYNC : Sync with mmap grabbing

Cette fonction attend la fin d'une capture lancée avec VIDIOCMCAPTURE. C'est un appel bloquant, le processus sera endormi jusqu'à la fin de la capture. Ensuite, ce buffer va contenir l'image valide jusqu'au prochain appel de VIDIOCMCAPTURE.

Listing 6 – Exemple de VIDIOCSYNC

```
int frame = 0;

if ( ioctl (fd , VIDIOCSYNC, &frame) < 0)
    exit (1);

printf ("Frame %d ready\n", frame);
```

3.3 Capture d'images

4 Internals

Références

[1] *C-Cam8 datasheet.*
`C-Cam8.pdf`

[2] *SAA7111 datasheet.*
`SAA7111.pdf`

[3] *STPC Video Input Port Writer's Guide.*
`vipwg01.pdf`

[4] *STPC Consumer Programming Manual.*
`c1pm.pdf`

[5] *Video4Linux Kernel API Reference.*
`Documentation/video4linux/API.html`

[6] Alessandro Rubini and Jonathan Corbet. *Linux Device Drivers.* O'Reilly, 2nd Edition, 2001.
Consultable sur le web : `http://www.xml.com/ldd/chapter/book/`