Table des matières

1 Introduction **3**
1.1 Le projet 3
1.2 Pourquoi Linux? 3

2 La carte CanPCI **5**
2.1 Présentation de la carte CanPCI 5
2.2 Améliorations, travail demandé 6

3 La carte de tests **7**
3.1 Présentation 7
3.2 Schematique 8
3.3 Programmation de l'altera 9

4 Les drivers sous linux **11**
4.1 Introduction 11
4.2 Généralités, notion de driver 11
4.3 Rappel sur le fonctionnement d'un driver sous linux 12

5 Le PCI **15**
5.1 Généralités 15
5.1.1 Architecture du bus 16
5.1.2 Les cycles de bus 16
5.1.3 L'espace de configuration 17
5.2 PCI vs. ISA 17
5.3 Le support du noyau linux 2.2 17

6 Le driver canpci **19**
6.1 Organisation générale 19
6.2 Details 20
6.2.1 initialisation 20
6.2.2 les IOCTL 22

7 L'interfacage entre java et le code natif : JNI **23**

7.1 Qu'est-ce que JNI? 23

7.2 Proprietes interessantes de JNI. 23

7.3 Exemple d'interfacage avec le driver CanPCI. 24

7.3.1 Définition de la classe gérant le driver. 24

7.3.2 Compilation et création de l'en-tete. 26

7.3.3 Création de la bibliothèque. 28

8 Le programme java de test **31**

8.1 Présentation des swings 31

8.2 Programme de test 33

9 Conclusion **35**

10 webographie **37**

10.1 Linux 37

10.2 Java 37

10.2.1 Swing 37

10.2.2 JNI 37

A code du driver CanPCI pour linux **39**

A.1 canpcidrv.h 40

A.2 canpcidrv.c 40

B java et JNI **49**

B.1 librairie, CanPCI.c 51

B.2 l'objet canpci : JCanPCI.java 53

C code JAVA de l'interface graphique **57**

D le Driver-HOWTO **75**

Chapitre 1

Introduction

1.1 Le projet

Le projet est axé autour de la carte canpci. Celle-ci a été réalisé à l'IXL pour pouvoir tester les convertisseurs analogiques/numériques. Le but du projet était de réaliser une chaine complète permettant de tester les cartes sous Linux. Cela vas du générateur de pattern au driver et à l'interface Java.

1.2 Pourquoi Linux ?

Linux est actuellement à la mode. Ceci est du aux particularitées de Linux. C'est un système d'exploitation fiable, rapide et stable. De plus, son orientation réseau ne peut que le servir dans une période de tout connecté. La quantité de logiciel disponibles avec leurs sources, les nombreux ports sur différentes architectures, les possibilités d'interopérabilité avec d'autre systèmes font de Linux une machine interessante tant pour un serveur que pour un utilisateur averti.

L'utilisation de cette carte sous Linux permet d'envisager de nouvelles applications à la carte canpci : outre le test de carte, on peut imaginer mettre la machine en reseau avec un programme permettant de lire à distance les aquisitions faites. Ceci servirait pour des TPs par exemple.

Chapitre 2

La carte CanPCI

2.1 Présentation de la carte CanPCI

La carte CanPCI a été dévelopée par le laboratoire IXL. Elle permet le test de convertisseurs analogiques-numériques. Dans ses spécifications, cette carte est capable de faire l'acquisition de données sur 24 bits, à une vitesse de 50 MHz. Les données sont stockées dans une mémoire de 1 Mo x 24 bits. Les données peuvent alors être lues et analysées à partir du PC. La carte est connectée au PC par un connecteur de type PCI. La taille de l'acquisition est paramétrable logiciellement de 32 à 1048576 bits.

La dialogue entre le PC et la carte se fait par l'intermédiaire de registres de contrôle sur la carte PCI, registres accessibles dans l'espace des I/O. L'adresse de ces registres est attribuée par le BIOS PCI au démarrage du PC.

Les registres sont les suivants :

– Registre 1 : registre de contrôle,

  Le registre de contrôle a le format suivant :

31	30	29	...	7	6	5	4	3	2	1	0
xx	FRONT	ON/OFF	xx	CE7	CE6	CE5	CE4	CE3	CE2	CE1	CE0
xx	r/w	r/w	xx	w	w	w	w	w	w	w	w

– Registre 2 : registre de chargement,

31	..	20	19	...	1	0
xx	xx	xx	C19	...	C1	C0
xx	xx	xx	w	w	w	w

– Registre 3 et 4 : ils sont équivalents :

31	..	24	23	...	1	0
xx	xx	xx	D23	...	D1	D0
xx	xx	xx	w	w	w	w

2.2 Améliorations, travail demandé

Les drivers pour Windows 9x et MacOS ont déja été écrits. Il a fallu les porter sous linux. La notion de drivers sous cet OS étant plus contraignante, le driver a été complêtement refait. Pour vérifier le fonctionnement de la carte CanPCI, nous avons également réalisé une carte électronique générateur de patterns commandable avec la sortie de la carte. De plus, une interface en Java permet de commander la carte de test convivialement.

Chapitre 3

La carte de tests

3.1 Présentation

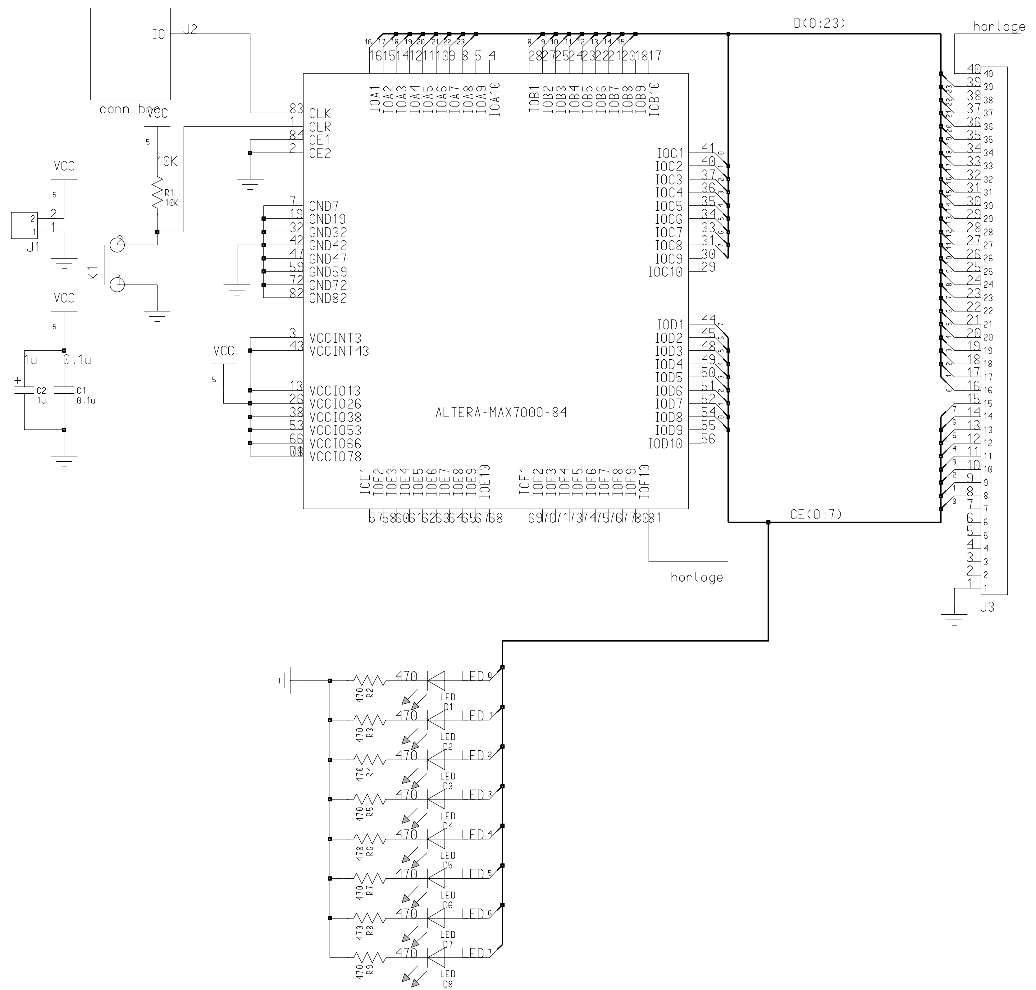
La carte de tests est un générateur de patterns permettant de vérifier le bon fonctionnement de la carte canpci. Celle-ci se compose de :

- deux bornes d'alimentation,
- une borne pour l'horloge,
- un connecteur vers la carte canpci,
- 8 leds rouges indiquant l'état de la sortie de la carte canpci,
- un altera reprogrammable pour générer les valeurs.

Pour vérifier le fonctionnement de la carte canpci, il suffit de tout brancher. La sortie est un compteur de 0 à 255 répliqué sur les trois octets. Les leds s'allument en fonction de la valeur de la sortie.

Pour le test actuel, l'altera n'est pas influencé par la valeur de son entrée.

3.2 Schematique



3.3 Programmation de l'altera

On définit le composant avec, en entrée, l'horloge, le reset et les 8 leds et en sortie, l'horloge pour l'aquisition et les 24 bits de test :

```
SUBDESIGN testCanPCI
(
  CLKIN, /RESET, LED[7..0]    : INPUT;
  CLKOUT, PATTERNOUT[23..0]   : OUTPUT;
)
```

On a besoin d'une variable temporaire pour pouvoir mémoriser la valeur actuelle.

```
VARIABLE
  PATTERN[23..0]   : DFF;

BEGIN
```

On relie l'horloge d'entrée à celle de sortie :

```
  CLKOUT = CLKIN;
```

On connecte les signaux d'horloge et de reset à la mémoire, puis on l'initialise à zéro :

```
  PATTERN[].CLK = CLKIN;
  PATTERN[].CLRN = /RESET;

  PATTERN[23..0] = 0;
```

On incremente le comptage de chaque octet:

```
  IF(PATTERN[7..0] < 255 ) THEN
    PATTERN[7..0] = PATTERN[7..0] + 1;
    PATTERN[15..8] = PATTERN[15..8] + 1;
    PATTERN[23..16] = PATTERN[23..16] + 1;

  ELSE
    PATTERN[23..0] = 0;
  END IF;
```

On envoit la mémoire vers la sortie :

```
  PATTERNOUT[23..0] = PATTERN[23..0];

END;
```

Chapitre 4

Les drivers sous linux

4.1 Introduction

L'écriture du driver pour la carte CanPCI nous a conduit à étudier en profondeur le fonctionnement des drivers sous linux, et avoir quelques notions du fonctionnement du noyau linux. Nous avons voulu mettre par écrit le résultat de nos recherche, et partager ces informations avec le reste de la communauté linux. Nous avons donc rédigé un "Driver-HOWTO" résumant le fonctionnement et les étapes de l'écriture d'un driver sous linux. Ce document sera disponible avec les autres HOWTO (le Driver-HOWTO est disponible en annexe).

Notons ci-après les points essentiels.

4.2 Généralités, notion de driver

Un driver est un programme particulier qui gère, contrôle, et dialogue avec un périphérique.

La première caractéristique importante du driver est d'offrir une interface standard entre les programmes et le matériel. Pour un type de matériel donné, une interface est définie ; tous les drivers de ce type de materiel implémenteront cette interface. On peut définir, par exemple, pour une liaison série des fonctions standard :

- l'envoi d'un octet,
- la réception d'un octet,
- le parametrage de la liaison.

Sur un système donné, tous les drivers de liaison série implémenteront ces 3 fonctions de la même façon. Le driver contient en plus le code spécifique à la liaison série qu'il gère. Cette architecture permet de rendre compatibles les péripheriques sous réserve qu'un driver existe pour chaque périphérique particulier.

La deuxième fonction du driver est liée au fonctionnement du noyau. Le noyau du système d'exploitation bénéficie de privilèges particuliers par rapport aux autres programmes. Le noyau a, entre autre tâche, le rôle de gérer et de partager les ressources entres les différents programmes qui tournent. Pour cela, le noyau bénéficie du privilège le plus élevé (grâce par exemple au mode protégé sur les 80x86). Le noyau est donc le seul à avoir accès au matériel par le biais des I/O.

Un driver étant un programme du noyau, il a entièrement accès aux entrées-sorties. Le driver permet ainsi aux programmes utilisateur d'accéder indirectement au matériel sans avoir de privilège particulier.

4.3 Rappel sur le fonctionnement d'un driver sous linux

Du point de vue de l'utilisateur, un driver apparaît comme un fichier dans le répertoire /dev. L'utilisateur peut donc y effectuer les opérations courantes d'ouverture du fichier, de lecture et d'écriture, et de fermeture du fichier. Ce fichier a des propriétés indiquant qu'il s'agit d'un fichier particulier (un driver). L'utilisateur dispose alors d'une opération supplémentaire sur ce fichier : les IOCTL (contrôle des entrées/sorties). Par exemple sous linux le driver de liaison série est accessible par le fichier /dev/ttyS0. L'écriture sur ce fichier d'un octet permet d'émettre cet octet par la liaison serie et la lecture d'un octet, la réception de cet octet. L'opération d'IOCTL sur ce fichier permet de régler les paramètres de la transmission série (vitesse de transmission, parité, ...)

Le listage des fichiers du répertoire /dev donne, par exemple :

```
#bash: ls -al ttyS?
crw----rw-   1 root     tty        4,  64 Jun 19 12:41 ttyS0
crw----rw-   1 root     tty        4,  65 May  5  1998 ttyS1
crw-------   1 root     tty        4,  66 May  5  1998 ttyS2
crw-------   1 root     tty        4,  67 May  5  1998 ttyS3
```

Le premier caractère (c) indique qu'il s'agit d'un driver de type caractère. La commande ls donne en plus une information importante : le numéro de

major et minor associés à ces fichiers (ici le major est 4 et le minor va de 64 a 67).

Ce numéro permet au noyau de faire le lien entre le fichier de /dev et le code du noyau associé au fichier. Plus précisement, à chaque opération élémentaire possible sur un fichier est associé une fonction qui réalise cette opération.

On dispose ainsi d'une interface simple entre les programmes utilisateurs et le materiel du PC.

Chapitre 5

Le PCI

5.1 Généralités

Le bus PCI s'est imposé comme bus standard il y a quelques années. Le bus ISA vieillissant, il a fallu trouver son successeur. EISA et VLB ont été écartés au profit du PCI. Ce-dernier présente de nombreux avantages pour les architectures modernes. Contrairement à ses deux concurents, il n'a pas cherché la compatibilité avec l'ancien standard et possede donc des particularités intéressantes:

- couplage du processeur avec le bus d'extension par un pont,
- bus standard de 32 bits,
- extension à 64 bits,
- support du multi-processeurs,
- transferts en mode burst de longueur indéfinie,
- support d'alimentation à 5V et 3,3V,
- vitesse variable de 0 à 33MHz
- possibilité d'avoir plusieurs maîtres,
- multiplexage de données et des adresses pour une réduction du brochage,
- support de l'ISA/EISA/MCA,
- configuration par registres programmables,
- spécifications indépendantes du processeur.

5.1.1 Architecture du bus

Le bus PCI repose sur la notion de pont (bridge). Ils permettent de morceler un bus en plusieurs sous-bus. Il existe trois type d'unité de bus PCI.

La première est le chipset interfacant le bus PCI avec celui du CPU. Cette interface est invisible par l'utilisateur.

Ensuite, il y a les extensions (carte son, vidéo, contrôleur de disque...). Celles-ci sont souvent volumineuses et les unités d'interfaçage du PCI sont contenues sur les cartes d'extensions. Elles doivent être configurées au moment du boot.

Le dernier type d'unité est les extensions de bus (généralement ISA ou PCI). Elles sont considérées comme des unités du PCI. Le bus PCI le plus proche du processeur est le bus principal. Ceux situés plus loin sont dits secondaires.

5.1.2 Les cycles de bus

Le contrôleur de bus PCI est de loin plus intelligent que ceux ISA/EISA ou VLB. Il permet différents cycles de bus :

- INTA sequence : addresse un contrôleur d'interruption (PIC),
- Special cycle : indique les cycles spéciaux du CPU comme la mise en veille ou en arret,
- I/O read/write : lit/écrit les données dans l'unité PCI de la zone IO adressée,
- memory read/write access : lit/écrit les données de l'unité PCI de la zone mémoire adressée.
- Configuration read/write access : permet de configurer l'unité PCI,
- Memory multiple access : indique que la lecture se fera sur plusieurs blocks de données et permet une accélération du processus (c'est une amélioration du line memory read access),
- Dual addressing cycle : permet d'adresser 64 bits de donnée/adresse à une unité ne supportant que le 32 bits,
- line memory read access : indique que la lecture se fera sur un bloc de mémoire plus grand que 32/64 bits.
- memory write access with invalidation : écrit un bloc de données à une unité PCI sans phase de write-back dans le cache.

5.1.3 L'espace de configuration

C'est un espace de 256 octets dont 64 sont standards pour pouvoir configurer l'unité PCI. Il contient, entre autre, l'adresse des registres I/O, l'adresse de la mémoire de la carte...

5.2 PCI vs. ISA

	ISA	PCI
taille des données	16 ou 8 bits	32 ou 64 bits
vitesse	faible, en général 8 MHz	rapide de 0 a 33 MHz, 66 MHz pour le PCI 2.1
transfert en pointe	16 Mo/s	266 Mo/s

5.3 Le support du noyau linux 2.2

Le noyau linux offre aux modules noyau et aux programmes utilisateurs un ensemble de fonction permettant de gérer et d'accéder aux cartes PCI. Voir le fichier /usr/include/linux/pci.h

Chapitre 6

Le driver canpci

6.1 Organisation générale

Le driver pour la carte PCI reprend les idées vues aux chapitres précédents : il s'agit d'un driver de caractères sous forme de module, accedant aux fonctions PCI pour dialoguer avec la carte CanPCI. La communication avec la carte CanPCI se fait principalement par IOCTL sur le driver, les fonctions Read et Write ne suffisant pas.

Les IOCTLs implémentés correspondent naturellement aux fonctions de bases implémentées en hard sur la carte CanPCI, à savoir :

- ecriture sur les leds de sortie (IOCTL_WRITE_OUTPUT),
- ecriture de la taille d'acquisition (IOCTL_WRITE_LOAD),
- ecriture du front d'acquisition (IOCTL_WRITE_FRONT),
- lecture du front d'acquisiton (IOCTL_READ_FRONT),
- ecriture de la commande d'acquisition (IOTCL_WRITE_GO),
- lecture du bit de fin d'acquisition (IOCTL_READ_GO),

De plus, la fonction d'écriture sur les leds de la carte de test a été aussi implémenté sur la fonction Write du driver.

6.2 Details

6.2.1 initialisation

Lors de l'initialisation du module, les opérations suivantes sont effectuées :

- enregistrement du module auprès du noyau en appelant la fonction

  ```
  register_chrdev()
  ```

- recherche d'un bios PCI. On utilise pour cela la fonction

  ```
  pcibios_present()
  ```

  Cette fonction renvoit la valeur "vraie" si le bios PCI a été trouvé. On peut alors accéder aux fonctions PCI du noyau.

- recherche de la carte PCI, connaissant le Vendor ID et le device ID a l'aide de la fonction

  ```
  pci_find_device ()
  ```

  Cette fonction renvoit une structure de type pci_dev et contenant diverses informations sur la carte PCI trouvée :

- recherche des 5 adresses de l'espace IO pour accéder plus tard à la carte PCI. Ces informations sont contenues dans la structure dev_pci de la fonction précédente

- réservation des ces 5 adresses dans l'espace d'adressage du module auprès du noyau

Ces 5 adresses sont sauvegardées dans une variable globale, et pouront être utilisées par les autres fonctions du module. Les fonctions utilisées ici sont des fonctions du noyau linux.

La fonction d'initialistation au complet :

```
int init_module () {
  u16 pci_command, new_command;

  // register device driver
  major = register_chrdev(MAJOR_NUM, DEVICE_NAME, &fops);
  if (major<0) {
```

```
    printk ("[ERROR] Registring failed.\n");
    return major;
  }
  printk ("CANPCI device registered.\n");

  // look for pci card

  if (!pcibios_present()) {
    printk ("Not PCI bios present.\n");
    return -1;
  }

  dev=pci_find_device (VENDOR_ID_CANPCI, DEVICE_ID_CANPCI, dev);

  if (dev == NULL) {
    printk("No CANPCI card found.\n");
    return -1;
  }

 pci_read_config_word(dev, PCI_COMMAND, &pci_command);
  new_command = pci_command | PCI_COMMAND_IO;
  if (pci_command != new_command) {
    printk(KERN_INFO "  The PCI BIOS has not enabled this"
           " CANPCI!  Updating PCI command %4.4x->%4.4x.\n",
           pci_command, new_command);
    pci_write_config_word(dev, PCI_COMMAND, new_command);
  }

  canpci_ioaddr[0] = dev->base_address[0] & PCI_BASE_ADDRESS_IO_MASK;
  canpci_ioaddr[1] = dev->base_address[1] & PCI_BASE_ADDRESS_IO_MASK;
  canpci_ioaddr[2] = dev->base_address[2] & PCI_BASE_ADDRESS_IO_MASK;
  canpci_ioaddr[3] = dev->base_address[3] & PCI_BASE_ADDRESS_IO_MASK;
  canpci_ioaddr[4] = dev->base_address[4] & PCI_BASE_ADDRESS_IO_MASK;
  request_region(canpci_ioaddr[1], 4, DEVICE_NAME);
  request_region(canpci_ioaddr[2], 4, DEVICE_NAME);
  request_region(canpci_ioaddr[3], 4, DEVICE_NAME);
  request_region(canpci_ioaddr[4], 4, DEVICE_NAME);
  return SUCCESS;
}
```

6.2.2 les IOCTL

Lors d'un appel aux fonctions IOCTL par l'utilisateur sur le driver, la fonction device_ioctl du module est appelée. Cette fonction reçoit en paramètre le numéro de l'IOCTL demandé par l'utilisateur. Il suffit alors de réagir en conséquence : lire ou écrire sur la carte PCI via l'espace des IO. Remarque sur l'écriture sur le port : il s'agit en général de modifier un ou plusieurs bits du registre. Il faut alors d'abord lire le registre, effectuer un masquage logique des bits à modifier, et écrire cette valeur.

Exemple :

```
static int device_ioctl (struct inode *inode, struct file *file,
                         unsigned int ioctlnum, unsigned long ioctlparam)
{
  unsigned long reg;

  switch (ioctlnum) {
  case IOCTL_WRITE_OUTPUT:

    reg = inl(canpci_ioaddr[1]);   // lecture de la valeur actuelle
    reg &= 0xFFFFFF00;    // 8 bits de poids faible a zero
    reg |= (ioctlparam & 0xFF);    // positionnement des bits
   // selon valeur demandée par l'utilisateur
    outl(reg,canpci_ioaddr[1]);    // envoie de la valeur vers le registre
     // de la carte PCI
    break;

...

  }
}
```

Chapitre 7

L'interfacage entre java et le code natif : JNI

7.1 Qu'est-ce que JNI ?

JNI est l'abréviation de Java Native Interface, autrement dit, l'interfaçage de java avec du code natif (généralement du C). Cette interface est utile quand on veut utiliser du code natif dans un programme Java. Par example, pour accéder à une bibliothèque en C qui gère un driver. JNI impose un certain nombre de règles dans l'écriture du code :

- définition et codage du code java avec les fonctions externes et le chargement des bibliothèque natives,
- compilation du code java,
- création de l'en-tête de la bibliothèque à partir du code java,
- codage de la bibliothèque native,
- compilation de la bibliothèque.

Une fois terminé, on peut lancer le code java. Celui-ci chargera la bibliothèque dynamique et l'éxécution du code natif sera invisible à l'utilisateur.

7.2 Proprietes interessantes de JNI.

JNI permet de garder une pseudo-portabilité malgré la liaison au code natif. En effet, celui-ci etant charge dynamiquement, on peut utiliser le même code java sur des OS différents, pouvu que la bibliothèque soit portée sur ces

différents OS. En effet, le P-code[1] java reste le même d'un OS à l'autre, mais la machine virtuelle est spécifique au système. Ainsi, si on charge la bibliothèque XXX, une machine virtuelle sous UNIX chargera XXX.so, par contre, celle sous Windows, chargera XXX.dll.

JNI permet également de créer des objets java à partir du code natif.

7.3 Exemple d'interfacage avec le driver CanPCI.

7.3.1 Définition de la classe gérant le driver.

```
class JCanPCI
{
```

On définit ici deux constantes pour manipuler plus facilement la sélection des fronts :

```
  public static final int RisingEdge = 1;
  public static final int FallingEdge = 0;
```

Puis on définit les fonctions utilisées par le driver

- Initialisation de la carte :

  ```
  protected native boolean Open();
  ```

- Fin d'utilisation de la carte :

  ```
  protected native void    Close();
  ```

- Sélection du front :

  ```
  public    native void    SetEdge( int montant );
  ```

- Valeur du front actuel :

  ```
  public    native int     GetEdge();
  ```

- Sélection de la taille de l'acquisition :

  ```
  public    native void    Size( long size );
  ```

1. Pseudo code : c'est le code interprété par la machine virtuelle java.

- Lancement de l'acquisition :

```
public    native void    Start();
```

- Test sur l'état de la carte (vrai si l'acquisition est en cours) :

```
public    native boolean IsWorking();
```

- Lecture d'une valeur de l'aquisition :

```
public    native int     Read();
```

- Ecriture d'un octet sur le port de sortie :

```
public    native void    Write( char output );
```

La lecture de la bibliothèque dynamique doit se faire après la création de l'objet, mais avant l'appel au constructeur, puisque celui-ci se charge de l'initialisation de la carte, et a donc besoin de faire des appels au code natif.

Si une erreur survient lors du chargement, on sort du programme. Cette méthode brutale est nécessaire car sinon, on ne peut empêcher la création de l'objet. Or ce-dernier a besoin de la bibliothèque pour fonctionner. On est donc obliger de prendre des mesures radicales.

```
static {
  try {
    System.loadLibrary("CanPCI");
  }
  catch(UnsatisfiedLinkError excep) {
    System.err.println("CanPCI library not found!");
    System.exit(0);
  }
}
```

Enfin, vient le constructeur. Celui-ci initialise la carte grâce à la fonction native Open. Si celle-ci renvoit la valeur fausse, l'initialisation n'a pu être faite (le driver n'est pas chargé ou déja utilisé) et on est donc obligé de prendre les même mesures que ci-dessus.

```
public JCanPCI() {
  try {
    if( Open() == false ) {
```

```
        System.err.println("Driver not found or already in use!");
        System.exit(0);
      }
    }
    catch(UnsatisfiedLinkError excep) {
      System.err.println(excep);
      System.exit(0);
    }
  }
```

Pour pouvoir tester la classe JCanPCI, nous avons écrit un petit programme de test illustrant l'utilisation qui peut être faite de cette classe.

Il :

- cré un objet JCanPCI,
- sélectionne le front montant,
- indique une taille de 32 aquisitions,
- lance l'aquisition,
- boucle jusqu'à sa fin,
- écrit les valeurs lues.

```
  public static void main( String[] args ) {
    JCanPCI CanPCITest = new JCanPCI();
    CanPCITest.SetEdge( RisingEdge );
    CanPCITest.Size( 32 );
    CanPCITest.Start();
    while( CanPCITest.IsWorking() == true )
      ;
    for( int i=0; i<16; i++ ) {
      System.out.println(CanPCITest.Read());
    }
  }
}
```

7.3.2 Compilation et création de l'en-tete.

Une fois la classe JCanPCI compilée, on génère l'en-tête grâce a "javah -jni JCanPCI". Après mise en forme, l'en-tête est :

```
  /* DO NOT EDIT THIS FILE - it is machine generated */
```

```
#include <jni.h>
/* Header for class JCanPCI */

#ifndef _Included_CanPCI
#define _Included_CanPCI

typedef unsigned char  byte;
typedef unsigned short word;
typedef unsigned long  dword;
typedef int            bool;

#define TRUE  1
#define FALSE 0
#define true  1
#define false 0

#ifdef __cplusplus
extern "C" {
#endif

JNIEXPORT jboolean JNICALL Java_JCanPCI_Open      (JNIEnv *,
                                                    jobject);
JNIEXPORT void     JNICALL Java_JCanPCI_Close     (JNIEnv *,
                                                    jobject);
JNIEXPORT void     JNICALL Java_JCanPCI_SetEdge   (JNIEnv *,
                                                    jobject,
                                                    jint);
JNIEXPORT jint     JNICALL Java_JCanPCI_GetEdge   (JNIEnv *,
                                                    jobject);
JNIEXPORT void     JNICALL Java_JCanPCI_Size      (JNIEnv *,
                                                    jobject,
                                                    jlong);
JNIEXPORT void     JNICALL Java_JCanPCI_Start     (JNIEnv *,
                                                    jobject);
JNIEXPORT jboolean JNICALL Java_JCanPCI_IsWorking (JNIEnv *,
                                                    jobject);
JNIEXPORT jint     JNICALL Java_JCanPCI_Read      (JNIEnv *,
                                                    jobject);
JNIEXPORT void     JNICALL Java_JCanPCI_Write     (JNIEnv *,
                                                    jobject,
                                                    jchar);
```

```
#ifdef __cplusplus
}
#endif

#endif
```

La définition des booléens et des types est ajoutée pour des raisons de portages inter-OS.

7.3.3 Création de la bibliothèque.

Une fois les declarations faites, on doit encore les implémenter :

```
#include "CanPCI.h"

#ifdef CANPCI_FOR_LINUX

/* Include du device driver. */
#include "canpcidrv.h"

/* Includes standards. */
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

/* descripteur fichier du device driver. */
int fd = -1;

/*
 * Fonction d'ouverture du fichier du device driver.
 */
JNIEXPORT jboolean JNICALL Java_JCanPCI_Open (JNIEnv * env, jobject obj) {
  if ((fd = open ("/dev/canpci",O_RDWR)) <0) {
    return FALSE;
  }
  return TRUE;
```

```
}

/*
 * Fonction de fermeture du fichier du device driver.
 */
JNIEXPORT void JNICALL Java_JCanPCI_Close (JNIEnv * env, jobject obj) {
  close(fd);
}

/*
 * Fonction de selection du front d'aquisition.
 */
JNIEXPORT void JNICALL Java_JCanPCI_SetEdge (JNIEnv * env, jobject obj, jint edge)
  ioctl (fd, IOCTL_WRITE_FRONT, edge);
}

/*
 * Fonction de lecture du front d'aquisition.
 */
JNIEXPORT jint JNICALL Java_JCanPCI_GetEdge (JNIEnv * env, jobject obj) {
  byte c;

  ioctl (fd, IOCTL_READ_FRONT, &c);
  return c;
}

/*
 * Selection de la taille d'aquisition
 */
JNIEXPORT void JNICALL Java_JCanPCI_Size (JNIEnv * env, jobject obj, jlong size)
  ioctl (fd, IOCTL_WRITE_LOAD, size);
}

/*
 * Mise en route de l'aquisition
 */
JNIEXPORT void JNICALL Java_JCanPCI_Start (JNIEnv * env, jobject obj) {
  ioctl (fd, IOCTL_WRITE_GO, 1);
}

/*
```

```
 * Teste si une aquisition est en cours
 */
JNIEXPORT jboolean JNICALL Java_JCanPCI_IsWorking (JNIEnv * env, jobject ob
  byte c;

  ioctl (fd, IOCTL_READ_GO, &c);
  return c;
}

/*
 * Lecture d'une aquisition de la carte
 */
JNIEXPORT jint JNICALL Java_JCanPCI_Read (JNIEnv * env, jobject obj) {
  dword l;

  read (fd, &l, 4);
  return l;
}

/*
 * Ecriture sur le port de sortie de la carte
 */
JNIEXPORT void JNICALL Java_JCanPCI_Write (JNIEnv * env, jobject obj, jchar
  write (fd, &output, 1);
}

#endif
```

Chapitre 8

Le programme java de test

Le driver linux pour la carte CanPCI a été écrit. Il permet de contrôler la carte en effectuant des opérations élémentaires sur le “fichier” /dev/canpci (lecture, ecriture et IOCTL).

Il a été ensuite nécessaire de développer un programme de test de la carte CanPCI. Ce programme doit en fait piloter la carte de test à base d'altera. Le but est de vérifier le bon fonctionnement de l'ensemble des fonctionnalités de la carte CanPCI.

Ce programme doit aussi offrir une interface graphique agréable pour piloter simplement et intuitivement la carte CanPCI.

Enfin, il ́tait souhaitable que ce programme soit developpé sous les differents systèmes d'exploitation pour lesquels ont été realisé des drivers pour la carte CanPCI.

Ces différentes contraintes ont naturellement conduit a choisir le langage JAVA pour développer cette application de test.

Java permet de réaliser des programmes qui tournent sous differents systemes d'exploitation. Le code source (un fichier .java) est compilé en un fichier .class, qui peut s'éxecuter sous toutes les architectures disposant d'une machine virtuelle JAVA (le JRE : Java Runtime Environment)

De plus, ce langage offre des objets permettant de réaliser de puissantes interfaces graphiques très simplememnt.

8.1 Présentation des swings

La principale difficulté lors de la réalisation de l'application de test a été le développement de l'interface graphique (GUI en anglais pour Graphic User Interface).

Dans sa version récente (Java 2) Java met à disposition du développeur un

ensemble de classes Java permettant de réaliser tres aisément des interfaces agréables et puissantes. Elles proposent l'ensemble des élements standards utils à une interface complête :

- éléments de fenêtre,
- de menus, de containers,
- de boutons, liste,
- cases a cocher,
- images,
- zone de texte...

Ces classes sont regroupées dans le package swing (que l'on doit maintenant préférer a l'ancien package awt (abstract window toolkit)).

L'utilisation de ces classes est relativement simple. Parmis les classes disponibles dans le package swing, il existe un ensemble de classes java representants des objets graphiques de l'interface (bouton, combo box, list, cadre, elements de menu, ...). Chacun de ces objets possède un ensemble d'attributs definissant ses caracteristiques et son comportement.

Par exemple,

```
JButton button = new JButton("I'm a Swing button!");
```

crée un bouton avec comme label "I'm a Swing button!".

La programmation de l'interaction de l'utilisateur avec l'interface graphique se fait de facon événementielle : à chaque interaction de l'utilisateur un événement est généré. Un deuxième ensemble de classes (les listeners) sont chargés d'écouter ces événements et d'appeler des méthodes de classes particulières.

Par exemple : bouton

```
button.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    numClicks++;
  }
};
```

rattache au bouton un listener qui incrémente la variable "numClicks" lorsque l'utilisateur clique sur le bouton.

La création d'une interface graphique en java se déroule de la facon suivante :

- instantiation des objets du GUI,

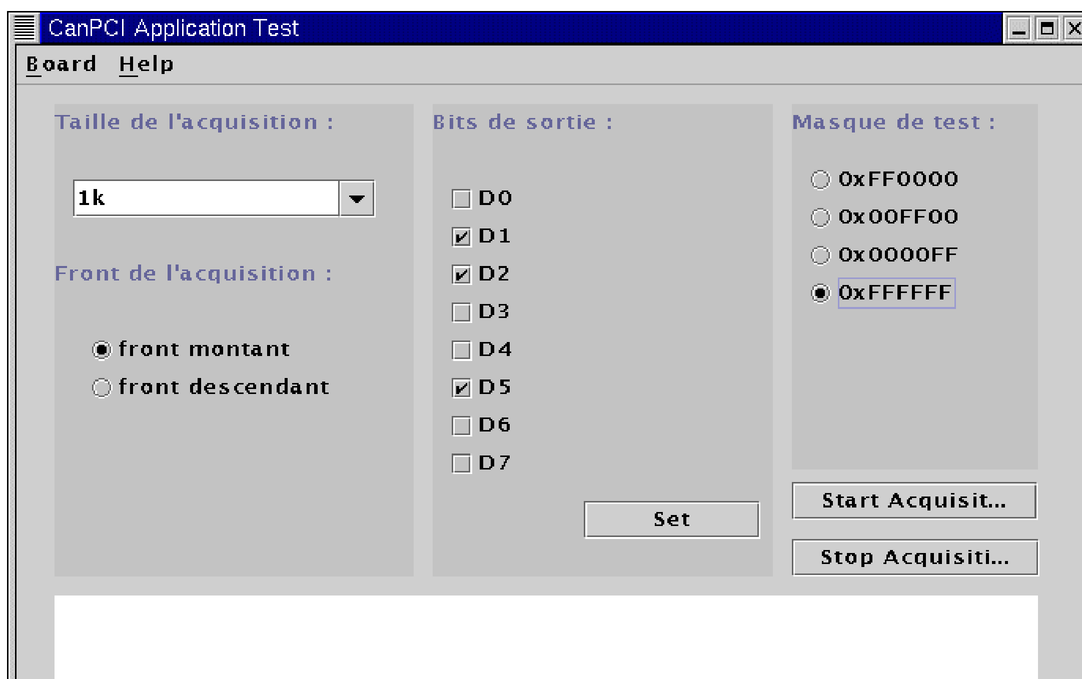
- choix des attributs de ces objets,
- rattachement à un container,
- affectation d'événement à des methodes callback.

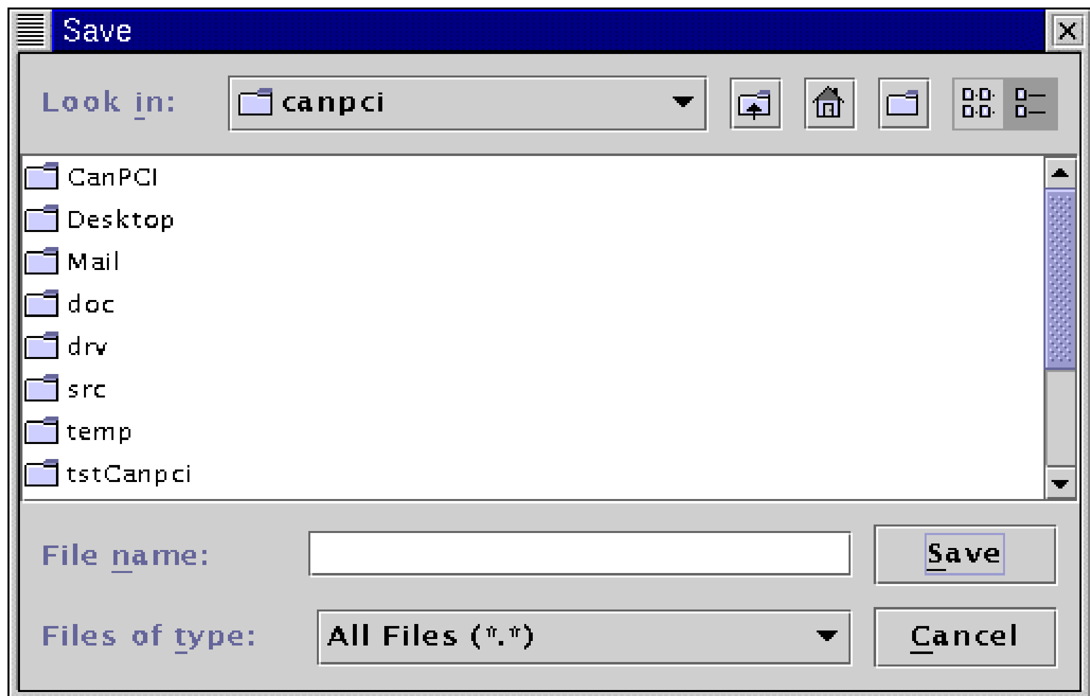
8.2 Programme de test

L'écriture du programme de test consiste donc à écrire le programme réalisant l'interface graphique et à rattacher aux differents élements graphiques des événements et des méthodes Java.

Les fonctionnalités integrées au programme sont les suivantes :

- choix de la taille d'acquisition avec une combo-list,
- choix du front d'acquisition avec des boutons radio,
- choix du masque de test avec des boutons radio,
- lancement et arrêt de l'acquisition avec des boutons,
- affichage du status du programme dans une zone de texte.





Le dialogue avec la carte a partir du programme Java se fait comme décrit dans les chapitres précédent. On appelle tout simplement des méthodes de la classe CanPCI qui est une classe JNI :

- instanciation de l'objet canpci :

```
JCanPCI canpci = new JCanPCI();
```

- lecture des données sur la carte CanPCI :

```
\scalebox{0.7}{\includegraphics{schem.eps}}
  for (int i=0; i<size; i++) {
            buffer[i] = canpci.Read();
  }
```

On réalise ainsi très aisement une interface graphique conviviale, et surtout portable, pour peu qu'on adapte la bibliothèque JNI à l'OS de destination

Chapitre 9

Conclusion

Une architecture complete a été réalisée : le driver pilotant la carte, ainsi qu'un programme de test utilisant ce driver via une bibliothèque ont été développés et sont operationnels. Il est interessant de faire la difference entre le code portable et le code natif écrit. Il est encore plus interessant de noter que l'architecture est telle que même le code natif possede une interface standard de sorte que la réecriture du code natif pour un autre OS se fait aisement.

On peut schématiser l'architecture par le schéma suivant : Java ¡–JNI –¿bibliotheque JNI (natif, Interface standard) ¡–¿Driver (natif mais interface standard)

Ce projet a été l'occasion pour nous de découvrir un peu plus le monde de Linux et d'étudier de près une partie du fonctionnement du noyau de ce système d'exploitation. Il faut noter que l'on a pu découvrir le fonctionnement de Linux grâce à l'architecture ouverte de Linux : les sources du noyau sont disponibles et reutilisables gratuitement (sous licence GPL). Cette philosophie conduit les utilisateurs de Linux à partager le plus possible leurs connaissances et leurs réalisations sous Linux, de sorte qu'il est très aisé de trouver de la documentation technique de qualitée.

Notons aussi que Linux est en constante évolution, dans la mesure où toute la communauté Linux participe au développement et à l'amélioration de ce système d'exploitation. Le driver a été écrit pour la version 2.2 du noyau, version actuellement diffusée. Mais des préversion du noyau 2.4 sont déjà disponibles sur internet, et quelques améliorations notables ont été apporté à ce nouveau noyau. Par exemple, le liens entre fichier de /dev et module du noyau se faisait avec le numero de major/minor dans le noyau 2.2 et se fera désormais avec l'utilisation d'un filesystem à part entiere dans le noyau 2.4. Il faudrait donc envisager un passage du driver CanPCI à une version compatible 2.4.

Chapitre 10

webographie

10.1 Linux

On peut trouver sur Linux Documentation Project (www.linuxdoc.org) une grosse partie de la documentation Linux : les HOWTOs, des livres... Deux guides ont retenu notre attention. Il s'agit de "The Linux Kernel Hackers' Guide" expliquant de manière assez schématique le fonctionnement du noyeau Linux. L'autre est l'ancien "Device Driver Guide" renommé maintenant "The Linux Kernel Module Programming Guide". Il explique comment écrire des modules sous Linux et a été notre référence pour l'ecriture du driver.

10.2 Java

10.2.1 Swing

http://java.sun.com/docs/books/tutorial/uiswing/start/swingTour.html

10.2.2 JNI

Spécifiactions de JNI:
http://java.sun.com/j2se/1.3/docs/guide/jni/spec/jniTOC.doc.html
Example d'utilisation de JNI:
http://developer.java.sun.com/developer/onlineTraining/ Programming/JDCBook/jniexamp.html

Annexe A

code du driver CanPCI pour linux

A.1 canpcidrv.h

```
#ifndef __CANPCI_DRIVER__
#define __CANPCI_DRIVER__

#define DEVICE_NAME "canpci"

#define VENDOR_ID_CANPCI 0x10e8
#define DEVICE_ID_CANPCI 0x8110

#define MAJOR_NUM 0xB0

#define IOCTL_WRITE_OUTPUT _IOW( MAJOR_NUM, 1, unsigned char )
#define IOCTL_WRITE_LOAD   _IOW( MAJOR_NUM, 2, unsigned long )
#define IOCTL_WRITE_FRONT  _IOW( MAJOR_NUM, 3, unsigned char )
#define IOCTL_READ_FRONT   _IOR( MAJOR_NUM, 4, unsigned char * )
#define IOCTL_WRITE_GO     _IOW( MAJOR_NUM, 5, unsigned char )
#define IOCTL_READ_GO      _IOR( MAJOR_NUM, 6, unsigned char * )

#endif
```

A.2 canpcidrv.c

```
/***********************************
 * TODO / A FAIRE
 * voir le kfree du cleanup_module
 * device_read : i+=4
 ***********************************/


#include <linux/version.h>
#if LINUX_VERSION_CODE < 0x020200
#err Source code only available for linux kernel version 2.2.0 or higher
#endif
#include <linux/module.h>
#include <linux/errno.h>
#include <linux/kernel.h>
```

```
#include <linux/pci.h>
#include <linux/ioport.h>
#include <asm/uaccess.h>
#include <asm/io.h>

#include "canpcidrv.h"

#define SUCCESS 0

static int device_opened = 0;
int major;


unsigned long canpci_ioaddr[5];
struct pci_dev *dev = NULL;

/************************************************
 * Device Open
 ************************************************/

static int device_open(struct inode *inode, struct file *file) {

  if (device_opened)
    return -EBUSY;

  device_opened++;

#ifdef DEBUG
  printk("[DEBUG] device opened\n");
#endif

  MOD_INC_USE_COUNT;

  return SUCCESS;
}


/************************************************
 * Device Release
 ************************************************/
```

```
static int device_release(struct inode *node, struct file *file) {

  device_opened--;
  MOD_DEC_USE_COUNT;

#ifdef DEBUG
  printk ("[DEBUG] device released\n");
#endif

  return SUCCESS;
}



/************************************************
 * Device Read
 ************************************************/

static ssize_t device_read(struct file *file,
   char *buffer,    /* The buffer to fill with data */
   size_t length,   /* The length of the buffer */
   loff_t *offset)  /* Our offset in the file */
{

  int i;
  unsigned long val;

#ifdef DEBUG
  printk ("[DEBUG] device : try to read %d bytes\n", length);
#endif

  for (i=0; i<(length & 0xFFFFFFFC); i+=4) {
     val = (inl(canpci_ioaddr[3])) & 0xFFFFFF;
     copy_to_user ((void *) buffer++,(void *) &val, 4);
  }

  return i;  // nothing read
}
```

```
/*********************************************
 * Device Write
 *********************************************/

static ssize_t device_write(struct file *file,
    const char *buffer,    /* The buffer */
    size_t length,   /* The length of the buffer */
    loff_t *offset)  /* Our offset in the file */
{

  unsigned char val;
  unsigned long reg;

#ifdef DEBUG
  printk ("[DEBUG] device : try to write %d bytes\n", length);
#endif

  copy_from_user ((void *) (&val), (void *)buffer, 1);

  reg = inl(canpci_ioaddr[1]);
  reg &= 0xFFFFFF00;
  reg |=val;
  outl(reg,canpci_ioaddr[1]);

  return 1; //todo
}

/*********************************************
 * Device IOCTL
 *********************************************/

static int device_ioctl (struct inode *inode, struct file *file,
 unsigned int ioctlnum, unsigned long ioctlparam)
{
  unsigned long reg;

  switch (ioctlnum) {

  case IOCTL_WRITE_OUTPUT:
#ifdef DEBUG
    printk("IOCTL_WRITE_OUTPUT\n");
```

```
#endif
    reg = inl(canpci_ioaddr[1]);
    reg &= 0xFFFFFF00;
    reg |= (ioctlparam & 0xFF);
    outl(reg,canpci_ioaddr[1]);
    break;

  case IOCTL_WRITE_LOAD:
#ifdef DEBUG
    printk("IOCTL_WRITE_LOAD %d\n", ioctlparam);
#endif
    reg = 0x100000 - ioctlparam;
    outl(reg,canpci_ioaddr[2]);
    break;

  case IOCTL_WRITE_FRONT:
#ifdef DEBUG
    printk("IOCTL_WRITE_FRONT %d\n", ioctlparam);
#endif
    reg = inl(canpci_ioaddr[1]);
    if( ioctlparam )
      reg |= 0x40000000;
    else
      reg &= 0xBFFFFFFF;
    outl(reg,canpci_ioaddr[1]);
    break;

  case IOCTL_READ_FRONT:
#ifdef DEBUG
    printk("IOCTL_READ_FRONT\n");
#endif
    reg = inl(canpci_ioaddr[1]);
    *((char *)ioctlparam) = ((reg & 0x40000000) == 0x40000000);
    break;

  case IOCTL_WRITE_GO:
#ifdef DEBUG
    printk("IOCTL_WRITE_GO %ld\n", ioctlparam);
#endif
    reg = inl(canpci_ioaddr[1]);
    if( ioctlparam )
```

```
      reg |= 0x20000000;
    else
      reg &= 0xDFFFFFFF;
    outl(reg,canpci_ioaddr[1]);
    break;

  case IOCTL_READ_GO:
    reg = inl(canpci_ioaddr[1]);
    *((char *)ioctlparam) = ((reg & 0x20000000) == 0x20000000);
#ifdef DEBUG
    printk("IOCTL_READ_GO %d\n", reg & 0x20000000);
#endif

    break;

  }

  return 0;
}


/*************************************************
 * MODULE DECLARATION
 *************************************************/

struct file_operations fops = {
  NULL, // seek
  device_read,
  device_write,
  NULL, // read dir
  NULL, // select
  device_ioctl, // ioctl
  NULL, // mmap
  device_open,
  NULL,//flush
  device_release
};

int init_module () {

  u16 pci_command, new_command;
```

```
// register device driver
major = register_chrdev(MAJOR_NUM, DEVICE_NAME, &fops);
if (major<0) {
  printk ("[ERROR] Registring failed.\n");
  return major;
}
printk ("CANPCI device registered.\n");

// look for pci card

if (!pcibios_present()) {
  printk ("Not PCI bios present.\n");
  return -1;
}

dev=pci_find_device (VENDOR_ID_CANPCI, DEVICE_ID_CANPCI, dev);

if (dev == NULL) {
  printk("No CANPCI card found.\n");
  return -1;
}

pci_read_config_word(dev, PCI_COMMAND, &pci_command);
new_command = pci_command | PCI_COMMAND_IO;
if (pci_command != new_command) {
  printk(KERN_INFO "  The PCI BIOS has not enabled this"
 " CANPCI!  Updating PCI command %4.4x->%4.4x.\n",
 pci_command, new_command);
  pci_write_config_word(dev, PCI_COMMAND, new_command);
}

canpci_ioaddr[0] = dev->base_address[0] & PCI_BASE_ADDRESS_IO_MASK;
canpci_ioaddr[1] = dev->base_address[1] & PCI_BASE_ADDRESS_IO_MASK;
canpci_ioaddr[2] = dev->base_address[2] & PCI_BASE_ADDRESS_IO_MASK;
canpci_ioaddr[3] = dev->base_address[3] & PCI_BASE_ADDRESS_IO_MASK;
canpci_ioaddr[4] = dev->base_address[4] & PCI_BASE_ADDRESS_IO_MASK;
request_region(canpci_ioaddr[1], 4, DEVICE_NAME);
request_region(canpci_ioaddr[2], 4, DEVICE_NAME);
request_region(canpci_ioaddr[3], 4, DEVICE_NAME);
request_region(canpci_ioaddr[4], 4, DEVICE_NAME);
```

```
  /*
    outl(0xFFFFF0,canpci_ioaddr[2]);

    registre = inl(canpci_ioaddr[1]);
    printk("input : %lx\n",registre);
    registre |= 0x60000000;
    outl(registre,canpci_ioaddr[1]);
    registre = inl(canpci_ioaddr[1]);
    printk("input : %lx\n",registre);

    registre = inl(canpci_ioaddr[1]);
    registre &= 0x20000000;
    while(registre) {
      registre = inl(canpci_ioaddr[1]);
      registre &= 0x20000000;
    }

    for( i=0; i<16; i++ ) {
      registre = inl(canpci_ioaddr[3]);
      registre &= 0x00FFFFFF;
      printk("saisie: %06lx\n",registre);
    }
  */

  return SUCCESS;
}

void cleanup_module ()
{
  int ret;

  ret = unregister_chrdev(MAJOR_NUM, DEVICE_NAME);
  if (ret<0) {
    printk ("[ERROR] unregistring failed\n");
    return;
  } else
    printk ("module unregistered\n");

  release_region(canpci_ioaddr[1], 4);
  release_region(canpci_ioaddr[2], 4);
```

```
  release_region(canpci_ioaddr[3], 4);
  release_region(canpci_ioaddr[4], 4);
  /*freek(dev);*/
}
```

Annexe B

java et JNI

section librairie, CanPCI.h

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class JCanPCI */

#ifndef _Included_CanPCI
#define _Included_CanPCI

typedef unsigned char  byte;
typedef unsigned short word;
typedef unsigned long  dword;
typedef int            bool;

#define TRUE  1
#define FALSE 0
#define true  1
#define false 0

#ifdef __cplusplus
extern "C" {
#endif

JNIEXPORT jboolean JNICALL Java_JCanPCI_Open      (JNIEnv *, jobject);
JNIEXPORT void     JNICALL Java_JCanPCI_Close     (JNIEnv *, jobject);
JNIEXPORT void     JNICALL Java_JCanPCI_SetEdge   (JNIEnv *, jobject, jint);
JNIEXPORT jint     JNICALL Java_JCanPCI_GetEdge   (JNIEnv *, jobject);
JNIEXPORT void     JNICALL Java_JCanPCI_Size      (JNIEnv *, jobject, jlong);
JNIEXPORT void     JNICALL Java_JCanPCI_Start     (JNIEnv *, jobject);
JNIEXPORT jboolean JNICALL Java_JCanPCI_IsWorking (JNIEnv *, jobject);
JNIEXPORT jint     JNICALL Java_JCanPCI_Read      (JNIEnv *, jobject);
JNIEXPORT void     JNICALL Java_JCanPCI_Write     (JNIEnv *, jobject, jchar);

#ifdef __cplusplus
}
#endif

#endif
```

B.1 librairie, CanPCI.c

```
#include "CanPCI.h"

#ifdef CANPCI_FOR_DOS

/* TODO : the DOS driver */

unsigned long registre1;
unsigned long registre2;
unsigned long registre3;

#endif

/*****************************************************************************
 *                                                                           *
 *                      LIBRAIRIE CANPCI POUR LINUX                          *
 *                                                                           *
 *****************************************************************************/
#ifdef CANPCI_FOR_LINUX

/* Include du device driver. */
#include "canpcidrv.h"

/* Includes standards. */
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

/* descripteur fichier du device driver. */
int fd = -1;

/*
 * Fonction d'ouverture du fichier du device driver.
 */
JNIEXPORT jboolean JNICALL Java_JCanPCI_Open (JNIEnv * env, jobject obj) {
```

```
  if ((fd = open ("/dev/canpci",O_RDWR)) <0) {
      return FALSE;
  }
  return TRUE;
}

/*
 * Fonction de fermeture du fichier du device driver.
 */
JNIEXPORT void JNICALL Java_JCanPCI_Close (JNIEnv * env, jobject obj) {
  close(fd);
}

/*
 * Fonction de selection du front d'aquisition.
 */
JNIEXPORT void JNICALL Java_JCanPCI_SetEdge (JNIEnv * env, jobject obj, jint
  ioctl (fd, IOCTL_WRITE_FRONT, edge);
}

/*
 * Fonction de lecture du front d'aquisition.
 */
JNIEXPORT jint JNICALL Java_JCanPCI_GetEdge (JNIEnv * env, jobject obj) {
  byte c;

  ioctl (fd, IOCTL_READ_FRONT, &c);
  return c;
}

/*
 * Selection de la taille d'aquisition
 */
JNIEXPORT void JNICALL Java_JCanPCI_Size (JNIEnv * env, jobject obj, jlong si
  ioctl (fd, IOCTL_WRITE_LOAD, size);
}

/*
 * Mise en route de l'aquisition
 */
JNIEXPORT void JNICALL Java_JCanPCI_Start (JNIEnv * env, jobject obj) {
```

```
  ioctl (fd, IOCTL_WRITE_GO, 1);
}

/*
 * Teste si une aquisition est en cours
 */
JNIEXPORT jboolean JNICALL Java_JCanPCI_IsWorking (JNIEnv * env, jobject obj) {
  byte c;

  ioctl (fd, IOCTL_READ_GO, &c);
  return c;
}

/*
 * Lecture d'une aquisition de la carte
 */
JNIEXPORT jint JNICALL Java_JCanPCI_Read (JNIEnv * env, jobject obj) {
  dword l;

  read (fd, &l, 4);
  return l;
}

/*
 * Ecriture sur le port de sortie de la carte
 */
JNIEXPORT void JNICALL Java_JCanPCI_Write (JNIEnv * env, jobject obj, jchar output)
  write (fd, &output, 1);
}

#endif
```

B.2 l'objet canpci : JCanPCI.java

```
class JCanPCI
{
```

```
    public static final int RisingEdge = 1;
    public static final int FallingEdge = 0;

    protected native boolean Open();
    protected native void    Close();
    public    native void    SetEdge( int montant );
    public    native int     GetEdge();
    public    native void    Size( long size );
    public    native void    Start();
    public    native boolean IsWorking();
    public    native int     Read();
    public    native void    Write( char output );

    // Lecture de la librairie native
    static {
try {
    System.loadLibrary("CanPCI");
}
catch(UnsatisfiedLinkError excep) {
    System.err.println("CanPCI library not found!");
    System.exit(0);
}
    }

    public JCanPCI() {
try {
    if( Open() == false ) {
System.err.println("Driver not found or already in use!");
System.exit(0);
    }
}
catch(UnsatisfiedLinkError excep) {
    System.err.println(excep);
    System.exit(0);
}
    }

    public static void main( String[] args ) {
JCanPCI CanPCITest = new JCanPCI();
CanPCITest.SetEdge( RisingEdge );
```

```
CanPCITest.Size( 32 );
CanPCITest.Start();
while( CanPCITest.IsWorking() == true )
    ;
for( int i=0; i<16; i++ ) {
    System.out.println(CanPCITest.Read());
}
    }
}
```

Annexe C

code JAVA de l'interface graphique

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;

import javax.swing.*;
import javax.swing.filechooser.*;


public class JMainFrame extends javax.swing.JFrame
{

    javax.swing.JPanel JMainPanel = new javax.swing.JPanel();
    javax.swing.JButton JStartButton = new javax.swing.JButton();
    javax.swing.JPanel JOutputPanel = new javax.swing.JPanel();
    javax.swing.JLabel JOutputLabel = new javax.swing.JLabel();
    javax.swing.JCheckBox JCheckBox1 = new javax.swing.JCheckBox();
    javax.swing.JCheckBox JCheckBox2 = new javax.swing.JCheckBox();
    javax.swing.JCheckBox JCheckBox3 = new javax.swing.JCheckBox();
    javax.swing.JCheckBox JCheckBox4 = new javax.swing.JCheckBox();
    javax.swing.JCheckBox JCheckBox5 = new javax.swing.JCheckBox();
    javax.swing.JCheckBox JCheckBox6 = new javax.swing.JCheckBox();
    javax.swing.JCheckBox JCheckBox7 = new javax.swing.JCheckBox();
    javax.swing.JCheckBox JCheckBox8 = new javax.swing.JCheckBox();
    javax.swing.JButton JOutputButton = new javax.swing.JButton();
    javax.swing.JTextArea JStatusText = new javax.swing.JTextArea();
    javax.swing.JPanel JAcqPanel = new javax.swing.JPanel();
    javax.swing.JLabel JAcqSizeLabel = new javax.swing.JLabel();
    javax.swing.JComboBox JComboBox = new javax.swing.JComboBox();
    javax.swing.JLabel JLabel1 = new javax.swing.JLabel();
    javax.swing.JRadioButton JRisingEdgeRadio =
new javax.swing.JRadioButton();
    javax.swing.JRadioButton JFallingEdgeRadio =
new javax.swing.JRadioButton();
    javax.swing.JButton JStopButton = new javax.swing.JButton();
    javax.swing.JPanel JMaskPanel = new javax.swing.JPanel();
    javax.swing.JLabel JLabel2 = new javax.swing.JLabel();
    javax.swing.JRadioButton JMSBRadio = new javax.swing.JRadioButton();
    javax.swing.JRadioButton JmSBRadio = new javax.swing.JRadioButton();
    javax.swing.JRadioButton JLSBRadio = new javax.swing.JRadioButton();
    javax.swing.JRadioButton JAllRadio = new javax.swing.JRadioButton();
```

```
    javax.swing.JMenuBar JMenuBar1 = new javax.swing.JMenuBar();
    javax.swing.JMenu fileMenu = new javax.swing.JMenu();
    javax.swing.JMenuItem JMenuItem1 = new javax.swing.JMenuItem();
    javax.swing.JMenuItem JMenuItem2 = new javax.swing.JMenuItem();
    javax.swing.JSeparator JSeparator1 = new javax.swing.JSeparator();
    javax.swing.JMenuItem exitItem = new javax.swing.JMenuItem();
    javax.swing.JMenu helpMenu = new javax.swing.JMenu();
    javax.swing.JMenuItem aboutItem = new javax.swing.JMenuItem();

    javax.swing.ButtonGroup edgeGroup = new javax.swing.ButtonGroup ();
    javax.swing.ButtonGroup maskGroup = new javax.swing.ButtonGroup ();
    javax.swing.JScrollPane scrollpane = new javax.swing.JScrollPane ();

    // Used by addNotify
    boolean frameSizeAdjusted = false;

    JCanPCI canpci = new JCanPCI();
    WaitAcquisition wa = new WaitAcquisition();

    boolean acqAbordRequest = false;
    long size = 0;
    int buffer[] = new int[1024*1024];

    static final String APP_TITLE = "CanPCI Application Test";

    public JMainFrame()
    {
setJMenuBar(JMenuBar1);
setTitle(APP_TITLE);
setDefaultCloseOperation(javax.swing.JFrame.HIDE_ON_CLOSE);
getContentPane().setLayout(new BorderLayout(0,0));
setSize(677,519);
setVisible(false);
JMainPanel.setLayout(null);
getContentPane().add(BorderLayout.CENTER,JMainPanel);
JMainPanel.setBounds(0,0,677,519);

// start acquisition button
JStartButton.setText("Start Acquisition");
JStartButton.setActionCommand("Start Acquisition");
JMainPanel.add(JStartButton);
```

```
JStartButton.setBounds(492,252,156,24);

// output panel
JOutputPanel.setLayout(null);
JMainPanel.add(JOutputPanel);
JOutputPanel.setBackground(java.awt.Color.lightGray);
JOutputPanel.setBounds(264,12,216,300);
JOutputLabel.setText("Bits de sortie :");
JOutputPanel.add(JOutputLabel);
JOutputLabel.setBounds(0,0,216,24);

// check box
JCheckBox1.setText("D0");
JCheckBox1.setActionCommand("D0");
JOutputPanel.add(JCheckBox1);
JCheckBox1.setBounds(12,48,54,24);
JCheckBox2.setText("D1");
JCheckBox2.setActionCommand("D1");
JOutputPanel.add(JCheckBox2);
JCheckBox2.setBounds(12,72,54,24);
JCheckBox3.setText("D2");
JCheckBox3.setActionCommand("D2");
JOutputPanel.add(JCheckBox3);
JCheckBox3.setBounds(12,96,54,24);
JCheckBox4.setText("D3");
JCheckBox4.setActionCommand("D3");
JOutputPanel.add(JCheckBox4);
JCheckBox4.setBounds(12,120,54,24);
JCheckBox5.setText("D4");
JCheckBox5.setActionCommand("D4");
JOutputPanel.add(JCheckBox5);
JCheckBox5.setBounds(12,144,54,24);
JCheckBox6.setText("D5");
JCheckBox6.setActionCommand("D5");
JOutputPanel.add(JCheckBox6);
JCheckBox6.setBounds(12,168,54,24);
JCheckBox7.setText("D6");
JCheckBox7.setActionCommand("D6");
JOutputPanel.add(JCheckBox7);
JCheckBox7.setBounds(12,192,54,25);
JCheckBox8.setText("D7");
```

```
JCheckBox8.setActionCommand("D7");
JOutputPanel.add(JCheckBox8);
JCheckBox8.setBounds(12,216,54,24);

JCheckBox1.setBackground(java.awt.Color.lightGray);
JCheckBox2.setBackground(java.awt.Color.lightGray);
JCheckBox3.setBackground(java.awt.Color.lightGray);
JCheckBox4.setBackground(java.awt.Color.lightGray);
JCheckBox5.setBackground(java.awt.Color.lightGray);
JCheckBox6.setBackground(java.awt.Color.lightGray);
JCheckBox7.setBackground(java.awt.Color.lightGray);
JCheckBox8.setBackground(java.awt.Color.lightGray);

// set button
JOutputButton.setText("Set");
JOutputButton.setActionCommand("Set");
JOutputPanel.add(JOutputButton);
JOutputButton.setBounds(96,252,112,24);

// status text
JStatusText.setEditable(false);
JMainPanel.add(JStatusText);
JStatusText.setBounds(24,324,624,432);

// acquisition panel
JAcqPanel.setLayout(null);
JMainPanel.add(JAcqPanel);
JAcqPanel.setBackground(java.awt.Color.lightGray);
JAcqPanel.setBounds(24,12,228,300);

// acquisition label
JAcqSizeLabel.setText("Taille de l\'acquisition :");
JAcqPanel.add(JAcqSizeLabel);
JAcqSizeLabel.setBounds(0,0,228,24);

// combo box
JComboBox.setToolTipText("taille de l\'acquisition en octets");
JComboBox.setEditable(true);
JAcqPanel.add(JComboBox);
JComboBox.setBounds(12,48,192,24);
```

```
// default entry
JComboBox.addItem ("1k");
JComboBox.addItem ("16k");
JComboBox.addItem ("128k");
JComboBox.addItem ("512k");
JComboBox.addItem ("1M");


// acquisition label
JLabel1.setText("Front de l\'acquisition :");
JAcqPanel.add(JLabel1);
JLabel1.setBounds(0,96,216,24);

// edge radio
JRisingEdgeRadio.setText("front montant");
JRisingEdgeRadio.setActionCommand("front montant");
JRisingEdgeRadio.setBackground(java.awt.Color.lightGray);
JAcqPanel.add(JRisingEdgeRadio);
JRisingEdgeRadio.setBounds(24,144,192,24);
JFallingEdgeRadio.setText("front descendant");
JFallingEdgeRadio.setActionCommand("front descendant");
JFallingEdgeRadio.setBackground(java.awt.Color.lightGray);
JAcqPanel.add(JFallingEdgeRadio);
JFallingEdgeRadio.setBounds(24,168,192,24);

JRisingEdgeRadio.setSelected(true);

edgeGroup.add(JRisingEdgeRadio);
edgeGroup.add(JFallingEdgeRadio);

// stop acquisition button
JStopButton.setText("Stop Acquisition");
JStopButton.setActionCommand("Stop Acquisition");
JMainPanel.add(JStopButton);
JStopButton.setBounds(492,288,157,24);

// mask panel
JMaskPanel.setLayout(null);
JMainPanel.add(JMaskPanel);
JMaskPanel.setBackground(java.awt.Color.lightGray);
JMaskPanel.setBounds(492,12,156,232);
```

```
// mask label
JLabel2.setText("Masque de test :");
JMaskPanel.add(JLabel2);
JLabel2.setBounds(0,0,156,24);

// mask radio
JMSBRadio.setText("0xFF0000");
JMaskPanel.add(JMSBRadio);
JMSBRadio.setBounds(12,36,144,24);
JmSBRadio.setText("0x00FF00");
JMaskPanel.add(JmSBRadio);
JmSBRadio.setBounds(12,60,144,24);
JLSBRadio.setText("0x0000FF");
JMaskPanel.add(JLSBRadio);
JLSBRadio.setBounds(12,84,144,24);
JAllRadio.setText("0xFFFFFF");
JMaskPanel.add(JAllRadio);
JAllRadio.setBounds(12,108,144,24);

JMSBRadio.setSelected(true);

JMSBRadio.setBackground(java.awt.Color.lightGray);
JmSBRadio.setBackground(java.awt.Color.lightGray);
JLSBRadio.setBackground(java.awt.Color.lightGray);
JAllRadio.setBackground(java.awt.Color.lightGray);

maskGroup.add(JMSBRadio);
maskGroup.add(JmSBRadio);
maskGroup.add(JLSBRadio);
maskGroup.add(JAllRadio);

// menu bar
fileMenu.setText("Board");
fileMenu.setActionCommand("File");
fileMenu.setMnemonic((int)'B');
JMenuBar1.add(fileMenu);
JMenuItem1.setText("Find Board");
JMenuItem1.setActionCommand("Find Board");
JMenuItem1.setMnemonic((int)'F');
fileMenu.add(JMenuItem1);
```

```
JMenuItem2.setText("Save Data");
JMenuItem2.setMnemonic((int)'S');
fileMenu.add(JMenuItem2);
fileMenu.add(JSeparator1);
exitItem.setText("Exit");
exitItem.setActionCommand("Exit");
exitItem.setMnemonic((int)'X');
fileMenu.add(exitItem);
helpMenu.setText("Help");
helpMenu.setActionCommand("Help");
helpMenu.setMnemonic((int)'H');
JMenuBar1.add(helpMenu);
aboutItem.setHorizontalTextPosition(
javax.swing.SwingConstants.RIGHT);
aboutItem.setText("About...");
aboutItem.setActionCommand("About...");
aboutItem.setMnemonic((int)'A');
helpMenu.add(aboutItem);


// listeners

SymWindow aSymWindow = new SymWindow();
this.addWindowListener(aSymWindow);
SymAction lSymAction = new SymAction();
exitItem.addActionListener(lSymAction);
aboutItem.addActionListener(lSymAction);
JStartButton.addActionListener(lSymAction);
// SymItem lSymItem = new SymItem();
JOutputButton.addActionListener(lSymAction);
JStopButton.addActionListener(lSymAction);
JMenuItem1.addActionListener(lSymAction);
/*
JMSBRadio.addItemListener(lSymItem);
JmSBRadio.addItemListener(lSymItem);
JLSBRadio.addItemListener(lSymItem);
JAllRadio.addItemListener(lSymItem);
*/
JMenuItem2.addActionListener(lSymAction);

    }
```

```
    public JMainFrame(String sTitle)
    {
this();
setTitle(sTitle);
    }

    static public void main(String args[])
    {
        try {
    // UIManager.setLookAndFeel(
UIManager.getSystemLookAndFeelClassName());
}
catch (Exception e) {
}

try {
    (new JMainFrame()).setVisible(true);
}
catch (Throwable t) {
    t.printStackTrace();
    System.exit(1);
}
    }

    public void addNotify()
    {
// Record the size of the window prior
// to calling parents addNotify.
Dimension size = getSize();

super.addNotify();

if (frameSizeAdjusted)
    return;
frameSizeAdjusted = true;

// Adjust size of frame according to the insets and menu bar
javax.swing.JMenuBar menuBar = getRootPane().getJMenuBar();
int menuBarHeight = 0;
if (menuBar != null)
```

```
    menuBarHeight = menuBar.getPreferredSize().height;
Insets insets = getInsets();
setSize(insets.left + insets.right + size.width,
insets.top + insets.bottom + size.height
+ menuBarHeight);
    }



    void log (String s) {
JStatusText.append("[" +JStatusText.getLineCount() + "]"
    + s + "\n");
    }

    void clear_log () {
JStatusText.setText("");
    }


    void check_data () {

long mask = 0;

// get mask
if (JMSBRadio.isSelected())
    mask = 0xFF0000;
else if (JmSBRadio.isSelected())
    mask = 0x00FF00;
else if (JLSBRadio.isSelected())
    mask = 0x0000FF;
else mask = 0xFFFFFF;

// read data
for (int i=0; i<size; i++) {
    buffer[i] = canpci.Read();
}

// check data
log("TODO : change the test once the new test card is ready");
for (int i=0; i<(size-1); i++) {
    if ( ((buffer[i+1]-buffer[i]) & mask) !=
```

```
(0x00010101 & mask) ) {
      if (!( ((buffer[i] & mask) == (0x00FFFFFF & mask))
&& ((buffer[i+1] & mask) == 0) )) {
log("Check failed at line : "+i);
return;
      }
    }
}
log("Check OK");
    }

    void exitApplication()
    {
try {
    // Beep
    Toolkit.getDefaultToolkit().beep();
    // Show a confirmation dialog
    int reply = JOptionPane.showConfirmDialog(this,
"Do you really want to exit?",
"CanPCI Test Application - Exit" ,
JOptionPane.YES_NO_OPTION,
JOptionPane.QUESTION_MESSAGE);
    // If the confirmation was affirmative, handle exiting.
    if (reply == JOptionPane.YES_OPTION)
{
    this.setVisible(false);    // hide the Frame
    this.dispose();
// free the system resources
    System.exit(0);            // close the application
}
} catch (Exception e) {
    this.setVisible(true);            // show the Frame (bug ?)
}
    }


    class WaitAcquisition extends Thread {
public void run() {
    acqAbordRequest = false;
    while (canpci.IsWorking() & !acqAbordRequest);
```

```
    if (acqAbordRequest)
log ("Acquisition aborded");
    else {
// end of acquisition
log ("Acquisition ended");
check_data();
    }
}
    }


    class SymWindow extends java.awt.event.WindowAdapter
    {
public void windowClosing(java.awt.event.WindowEvent event)
{
    Object object = event.getSource();
    if (object == JMainFrame.this)
JMainFrame_windowClosing(event);
}
    }

    void JMainFrame_windowClosing(java.awt.event.WindowEvent event)
    {
// to do: code goes here.
JMainFrame_windowClosing_Interaction1(event);
    }

    void JMainFrame_windowClosing_Interaction1(
java.awt.event.WindowEvent event) {
try {
    this.exitApplication();
} catch (Exception e) {
}
    }

    class SymAction implements java.awt.event.ActionListener
    {
public void actionPerformed(java.awt.event.ActionEvent event)
{
    Object object = event.getSource();
    if (object == exitItem)
```

```
exitItem_actionPerformed(event);
    else if (object == aboutItem)
aboutItem_actionPerformed(event);
    else if (object == JStartButton)
JStartButton_actionPerformed(event);
    else if (object == JOutputButton)
JOutputButton_actionPerformed(event);
    else if (object == JStopButton)
JStopButton_actionPerformed(event);
    else if (object == JMenuItem1)
JMenuItem1_actionPerformed(event);
    else if (object == JMenuItem2)
JMenuItem2_actionPerformed(event);

}
    }

    void exitItem_actionPerformed(java.awt.event.ActionEvent event)
    {
// to do: code goes here.

exitItem_actionPerformed_Interaction1(event);
    }

    void exitItem_actionPerformed_Interaction1(
java.awt.event.ActionEvent event) {
try {
    this.exitApplication();
} catch (Exception e) {
}
    }

    void aboutItem_actionPerformed(java.awt.event.ActionEvent event)
    {
// to do: code goes here.

aboutItem_actionPerformed_Interaction1(event);
    }

    void aboutItem_actionPerformed_Interaction1(
java.awt.event.ActionEvent event) {
```

```
try {
    // JAboutDialog Create with owner and show as modal
    {
JAboutDialog JAboutDialog1 = new JAboutDialog(this);
JAboutDialog1.setModal(true);
JAboutDialog1.show();
    }
} catch (Exception e) {
}
    }

    void JStartButton_actionPerformed(java.awt.event.ActionEvent event)
    {
clear_log();

if (wa.isAlive()) {
    log ("Acquisition en cours ...");
} else {

    // set edge
    if (JRisingEdgeRadio.isSelected()) {
log ("Setting Rising Edge");
canpci.SetEdge(canpci.RisingEdge);
    } else {
log ("Setting Falling Edge");
canpci.SetEdge(canpci.FallingEdge);
    }

    // set size
    int mult = 1;
    StringBuffer st = new StringBuffer (
(String) JComboBox.getSelectedItem());
    char c = st.charAt (st.length() - 1);
    if (c == 'k' || c == 'K') {
mult = 1024;
st.setCharAt (st.length() - 1, ' ');
st = new StringBuffer (st.toString().substring (0,
st.length() - 1));
    }
    if (c == 'm' || c == 'M') {
mult = 1024*1024;
```

```
st = new StringBuffer (st.toString().substring (0,
st.length() - 1));
    }

    size = -1; //todo : pas terrible

    try {
size =  (new Integer (st.toString())).longValue();
    } catch (java.lang.NumberFormatException e) {
log ("Number Format Exception : \"" + st
+ "\" is not a valid size");
    }

    if (size != -1) {
size *= mult;
log ("Setting size : " + size);

canpci.Size(size);
canpci.Start();
log ("Starting acquisition  ...");

// thread d'attente de la fin d'acquisition
wa = new WaitAcquisition();
wa.start();

    }
}
    }


    /*
    class SymItem implements java.awt.event.ItemListener
{
    public void itemStateChanged(java.awt.event.ItemEvent event)
{
    Object object = event.getSource();
    if (object == JMSBRadio)
JMSBRadio_itemStateChanged(event);
    else if (object == JmSBRadio)
JmSBRadio_itemStateChanged(event);
    else if (object == JLSBRadio)
```

```
JLSBRadio_itemStateChanged(event);
    else if (object == JAllRadio)
JAllRadio_itemStateChanged(event);
}
    }

    */

    void JOutputButton_actionPerformed(java.awt.event.ActionEvent event)
    {

int val = 0;
if (JCheckBox1.isSelected()) val+=1;
if (JCheckBox2.isSelected()) val+=2;
if (JCheckBox3.isSelected()) val+=4;
if (JCheckBox4.isSelected()) val+=8;
if (JCheckBox5.isSelected()) val+=16;
if (JCheckBox6.isSelected()) val+=32;
if (JCheckBox7.isSelected()) val+=64;
if (JCheckBox8.isSelected()) val+=128;

log ("Output : " + val);
canpci.Write ((char)val);


    }

    void JStopButton_actionPerformed(java.awt.event.ActionEvent event)
    {
// to do: code goes here.
if (!wa.isAlive())
    log ("No acquisition to stop");
else
    acqAbordRequest = true;

    }

    void JMenuItem1_actionPerformed(java.awt.event.ActionEvent event)
    {
// to do: code goes here.
// JStatusText.append("Not board found\n");
```

```
log ("Find Board : Not Yet implemented");
    }


    void JMenuItem2_actionPerformed(java.awt.event.ActionEvent event)
    {
FileOutputStream fout;
DataOutputStream dout;

JFileChooser     fc = new JFileChooser();

int returnedVal = fc.showSaveDialog(JMainFrame.this);
if (returnedVal == JFileChooser.APPROVE_OPTION) {
    try {
fout = new FileOutputStream(fc.getSelectedFile());
dout = new DataOutputStream(fout);
dout.writeLong(size);
for (int i=0; i<size; i++) {
    dout.writeInt(buffer[i]);
}
dout.close();
fout.close();
log(fc.getSelectedFile().getName()+" writen.");
    }
    catch (Exception excep) {
System.err.println(excep);
System.exit(0);
    }
}
    }
}
```

Annexe D

le Driver-HOWTO

```
Linux Driver writing HOWTO

Xavier Ordoquy,
Christophe Maillot

Version 0.1, May 23rd, 2000



    This document describe what are device drivers, how to write ,
    how to use the PCI and port from other OS to linux.


=================================================================
I) Introduction.
=================================================================

I.1) License
-----------------------------------------------------------------
 FPL

I.2) Restriction on the linux version.
-----------------------------------------------------------------
     This document has been written for the 2.2 kernel.

=================================================================
II) What are device driver under Linux ?
=================================================================

II.1) Drivers'overview.
-----------------------------------------------------------------


II.1.1) What's a driver ?

          We could answer that a driver is a chunk of software that drives
some hardware (or some software). It is part of a system architecture.
It enables to hide hardware's specifics from the
system : For example, your network device can change without changing
anything in the system except the associated driver.
```

The different drivers of a device implement standard functions that the system uses. It alows to change driver without changing the kernel.

Therefore we have something like that:

```
user's programme -----------> driver ------------> hardware
                   standard           specific
  interface              code
```

II.2) The linux kernel and drivers.

I'm not going to describe you the linux kernel into details, but only briefly the parts relating to drivers. We'll see some general things about linux in order to understand how they work.

II.2.1) What's the memory management ?

The memory management subsystem is one of the most important part of the kernel. The important feature for us is the memory protection. Each process has it's own memory space which is isolated from the other ones. This is due to the processor's Memory Management Unit (MMU) which enables different levels of protection (Higher level can access lowest but not the reverse and same levels are separate). User programmes run on the same level which has lower priority than the kernel one. Therefore the kernel can access any programme memory location.

II.2.2) How to access kernel functions then ?

A call to the kernel functions is done via a gate (or a trap) which allow cross-level calls that would not be possible otherwise.

II.2.3) The input/output.

The MMU can also lock the input output ports. To do I/O you have to choose to :

- enable restricted acces to I/O ports to user programmes, which may be conflicting with the stability policy.
- load a driver in the kernel.

Despite exceptions (XFree and libsvga) the solution choosen is the driver (NOTE that the XFree86 v4.0 3D acceleration is included in the kernel).

```
II.2.4) The interruptions.

II.2.5) The /dev/* files in kernel 2.2 and major/minor
          In the /dev directory, you can see files that are linked to drivers
You can communicate with them just as if they were files.
  To understand how they work, one must know that linux uses a
virtual filesystem (VFS) that is an interface between the kernel
and the filesystem that is really used (whatever it is : ext2, FAT...)
To create the /dev files, one create an inode in the VFS with the
major and minor numbers.
  Those numbers are used to connect the file and the device driver.
Once loaded the driver registers itself with the major and minor
and the filesystem calls it when operations on the file are performed.
Notice that the file must be created before the driver is loaded.
The current policy is to use the major number for a type of device
( hd* for ide hard drives, cua* for serial ports, ...) and minor for
subsystem selection (hda/hda1/hda2/../hdb/../hdc/..). This is
not valuable for specific hardware for which one can find different
type within the same major. An organism is needed to register them,
but with the growing number of devices and the private device driver
make hard to keep the numbers unic. The solution is to shift to the
new dev filesystem (devfs) with uses names instead of numbers. It is
included in the new 2.4 kernels.

II.2.6) The new dev filesystem in kernel 2.4

II.2.6) Adding and removing modules.

II.2.7) Types of drivers.
          There are 3 differents type of drivers :
  - character device drivers : the most common drivers,

  - block device drivers : they are used for devices that are
    accessed by block : hard drives, floppy, cdrom, tapes...

  - network device drivers : they allow the kernel to access
    network cards.

  The most current type is the character one. The two others are
        special because used (only ?) by the kernel. Therefore if you whish
to write a driver it may be a character one.
```

II.2.8) Restrictions.

Everything can not be in the kernel or the driver. The usual policy is to put low-level communication in the kernel and higher-level computing in a shared library. There are many reasons for this.

- Because the driver is in kernel-space memory, it can access everything and any bug can broke the system. Therefore when the big part of the code is in a library the bug just crash the programme instead of the whole system.
- Because there is NO dynamic memory allocation within the kernel.
- Because it sounds nicer to update a library rather than a module.

==

III) The character device driver for kernels 2.2

==

III.1) A simple char driver

III.1.1) The code

```
        #include <linux/version.h>
#if LINUX_VERSION_CODE < 0x020200
#err Source code only available for linux kernel version 2.2.0 or higher
#endif
#include <linux/module.h>
#include <linux/kernel.h>

#define SUCCESS 0

int init_module () {
  printk ("test1 : module loaded.\n");
  return SUCCESS;
}

void cleanup_module () {
  printk ("test1 : module unloaded\n");
}
```

```
 You can compile it with something like
          gcc -Wall -O2 -D__KERNEL__ -DMODULE -DLINUX
           -Wall is to check all the warnings possible,
   -O2 ID NEEDED to compile kernel parts,
   -D__KERNEL__ defines the macro for kernel,
   -DMODULE defines that we are working on module
            (by oposition to built-in),
   -DLINUX defines that we work on linux (for multi-os drivers).

 and insert it with

          insmod testeur.o

 then watch the console. It should say : "test1: module loaded."

III.1.2) Basics
           a) The module loading
 The function called when the module is loaded is

     int init_module(void)

 It has to initialize the module.

   b) The module unloading
 The function called when the module is unloaded is

     void cleanup_module()

 It does clean everything before the module is removed.

III.1.3) Some function to be used in the kernel space

III.1.3) Adding arguments to the module

III.2) Using a device file
--------------------------------------------------------------------------------

III.2.1) The code


#include <linux/version.h>
```

```
#if LINUX_VERSION_CODE < 0x020200
#err Source code only available for linux kernel version 2.2.0 or higher
#endif
#include <linux/module.h>
#include <linux/errno.h>
#include <linux/kernel.h>
/*#include <linux/pci.h>*/
/*#include <linux/ioport.h>*/
#include <asm/uaccess.h>
/*#include <asm/io.h>*/

#include "test2.h"

#define SUCCESS 0

static int file_opened = 0;
int major;

char msg[2][20] = {"Hello world! \n", "good morning!\n"};
int selected = 0;

static int test2_open(struct inode *inode, struct file *file)
{
   if (file_opened)
     return -EBUSY;
   file_opened++;
   MOD_INC_USE_COUNT;
#ifdef DEBUG
   printk("test2 : test2 file opened.\n");
#endif
   return SUCCESS;
}

static int test2_release(struct inode *node, struct file *file)
{
   file_opened--;
   MOD_DEC_USE_COUNT;
#ifdef DEBUG
   printk ("test2 : test2 file closed.\n");
#endif
   return SUCCESS;
```

```
}

static ssize_t test2_read(struct file *file,
  char *buffer,    /* The buffer to fill with data */
  size_t length,   /* The length of the buffer */
  loff_t *offset)  /* Our offset in the file */
{
   int i,size;
#ifdef DEBUG
   printk ("test2 : test_read called.");
#endif
   size = (length%15)*15;
   for( i=0; i<size; i++ )
     put_user(msg[selected][i%15],buffer++);

   return i;
}

static ssize_t test2_write(struct file *file,
   const char *buffer,    /* The buffer */
   size_t length,   /* The length of the buffer */
   loff_t *offset)  /* Our offset in the file */
{
   int i;
#ifdef DEBUG
   printk ("test2 : test_write called, message written :");
#endif
   for( i=0; i<length; i++ )
     printk (buffer);
   return length;
}

static int test2_ioctl (struct inode *inode, struct file *file,
unsigned int ioctlnum, unsigned long ioctlparam)
{
  switch (ioctlnum) {
   case IOCTL_WRITE_CONFIG:
#ifdef DEBUG
     printk("IOCTL_WRITE_OUTPUT\n");
#endif
     if( ioctlparam == 0 ) {
```

```
selected = 0;
     } else {
selected = 1;
     }
     break;

   case IOCTL_READ_CONFIG:
#ifdef DEBUG
     printk("IOCTL_WRITE_OUTPUT\n");
#endif
     *((int *)ioctlparam) = selected;
     break;
  }

  return 0;
}


/************************************************
 * MODULE DECLARATION
 ************************************************/

struct file_operations fops = {
   open:     test2_open,
   release:  test2_release,
   read:     test2_read,
   write:    test2_write,
   ioctl:    test2_ioctl
};

int init_module () {

  // register device driver
  major = register_chrdev(MAJOR_NUM, DEVICE_NAME, &fops);
  if (major<0) {
    printk ("test2 : [ERROR] Registring failed.\n");
    return major;
  }
  printk ("test2 : test2 module registered.\n");

  return SUCCESS;
```

```
}

void cleanup_module ()
{
  int ret;

  ret = unregister_chrdev(MAJOR_NUM, DEVICE_NAME);
  if (ret<0) {
    printk ("test2 : [ERROR] unregistring failed\n");
    return;
  } else
    printk ("test2 : test2 module unregistered\n");
}


           mknod test c 0xB0 0

III.2.2) The basics of the /dev/* files
           The driver is connected to the device file with the major
 and minor numbers. We need to specify the operations which the
 driver can handle by register a file operation structure. This
 is the file_operation structure which can be found in fs.h. It
 descibes in an array of functions what the driver can do. The most
 important are :
   - read    : read a char array from the driver
   - write   : write a char array to the driver
   - ioctl   : makes misc functions
   - open    : opens the file
   - release : closes the file
 The operations were named from the user's point of view. Therefore
 read moves data from the driver to the user programme.

 Once the operations are described, we need to register them to
 connect them to the file. This is done by calling
 "module_register_chrdev" with the major, minor and the file_operation
 structure as arguments when the module is loaded in.
 At the time the module is unloaded, a call to
 "module_unregister_chrdev" removes the link between the file and the
 driver.

*****************************************************************************
```

```
****   Warning : in the following, I will name the functions that do file ****
**** operations by their name in the example. Keep in mind that you can   ****
**** call them with your own names as long as you register them with the  ****
**** right name.                                                          ****
*****************************************************************************
```

III.2.3) Opening and closing

To use the file, we have to be able to open and close it. When the user opens a file, a called to the function corresponding to the open field in the file operation structure is issued. In our example, it calls test_open. The function returns an error code which are defined in << asm/errno.h >>. We just check here if the file is already open and returns the error code file busy or return successful code.

Once the file is closed, the function corresponding to the release field is called. From the 2.2 version, this function returns an int which tells when a trouble occurs while closing the file.

You may have noticed the use of the macros MOD_INC_USE_COUNT and MOD_DEC_USE_COUNT. They allows the kernel to know when it can removes a module. When a user opens the file, MOD_INC_USE_COUNT increments the number of users. The kernel knows whether a module is used or not. This prevent crashs by removing used modules.

III.2.4) Reading and writing

Once the file has been opened, it is nice to do some operation over it: for exemple, reading and writing. One must notice that those operations are named form the user's point of view. In our example, reading is done by test_read and writing by test_write. Both returns the number of bytes they moved. Notice that the file descriptor is needed is case of modules having many files to handle (hda1/hda2/.../hda?).

One may wonder that there is not strcpy by put_user and write_user. This is because of the memory management (remember what I said about it ?). Since the buffer is in the user's space which is different from the driver's space, we need to let the kernel do some memory translation by calling those functions. They can find by themselves whenever that pointer to the data is 1, 2 or 4 bytes. Take a look at asm/uaccess.h

```
III.2.5) Input output control (ioctl)
          Reading and writing is nice, but what if the driver needs
 parameters to set up ? Ioctls are the solution. They are use in
 UNIX system to talk to the driver. The function called takes three
 parameters :
   - the file descriptor,
   - the ioctl number,
   - the parameter.
 The ioctl number is made by the _IO, _IOR, _IOW and _IORW macros
 which are defined in asm/ioctl.h. They made by the driver's major
 the ioctl command number and the type. Thus you can't pass a
         structure, you can use it by passing a pointer to it and use
         user_read, user_write macros. For more informations on ioctl numbers
 and there registry, be sure to read Documentation/ioct-number.txt in
 the kernel tree.

=============================================================================
IV) Driver for PCI Card for kernels 2.2
=============================================================================

  In the following, I will not give source code because the driver I wrote
was for a specific card that you won't have. Therefore I will take the
source of the NE2000 PCI driver (driver/net/ne2k-pci.c).

IV.1) Setting up the PCI card.

        Before any command, we need to be sure that one can speak to the
      PCI bus via the PCI bios. The pcibios_present() returns 0 when there
      is no PCI bios present (or PCI has been disabled). We also need to
      include the linux/pci.h to get all the functionnalities of PCI API.

          #include <linux/pci.h>

          if (!pcibios_present())
    return -ENODEV;

      Then we need to find the card with the vendor and card identity.

  struct pci_dev *pdev = NULL;
  pdev = pci_find_device(VENDOR_ID, DEVICE_ID, pdev);
```

We check that the card is activated. This is a fix for some brain-dammaged BIOS.

```
u16 pci_command, new_command;

pci_read_config_word(pdev, PCI_COMMAND, &pci_command);
new_command = pci_command | PCI_COMMAND_IO;
if (pci_command != new_command) {
  printk(KERN_INFO " Activating the Card !\n");
  pci_write_config_word(pdev, PCI_COMMAND, new_command);
}
```

After that, we can use the card as we'll see in the next section.

IV.2) Using the PCI card.

IV.2.1) Accessing the IO ports.