Plataformas de Tiempo Real: Dispositivos y Drivers

UNIVERSIDAD DE CANTABRIA

Tema 6. Buses

- El bus PCI
- Programación del bus PCI
- Programación de la tarjeta PCI-9111DG

Introducción al bus PCI

UNIVERSIDAD DE CANTABRIA

El bus PCI (*Peripheral Component Interconnect*) es uno de los más habituales en las arquitecturas PC, así como en Alpha, PowerPC, SPARC64 e IA-64

Aparece como solución alternativa a buses como ISA o EISA

- para periféricos más rápidos
- pero también como solución conceptual a dispositivos autodetectables y de fácil configuración

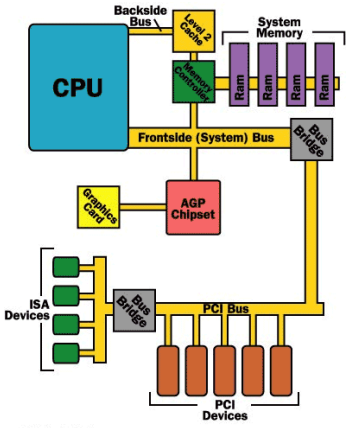
No se utiliza para tarjetas gráficas que disponen de otro tipo de buses como por ejemplo AGP o PCI-Express

Introducción al bus PCI (cont.)

UNIVERSIDAD DE CANTABRIA

Principales características:

- desacoplo entre el procesador y el bus de expansión mediante un puente (*bridge*)
- bus estándar de 32 bits con una transferencia máxima de 133 MB/s
- extensión a 64 bits con transferencias máximas de 266 MB/s
- soporte para sistemas multiprocesadores

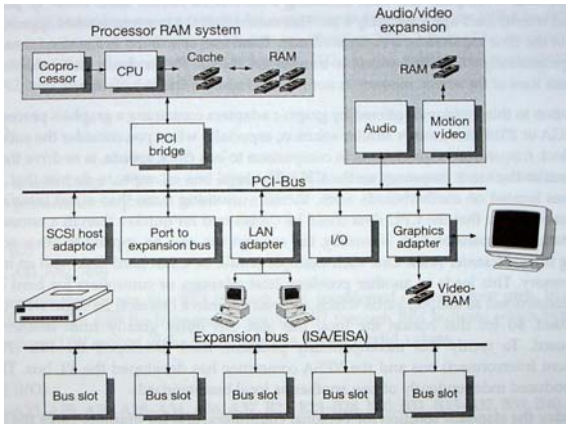


Introducción al bus PCI (cont.)

- transferencias a ráfagas (*burst*) de cualquier longitud
- soporta alimentaciones de 5 V. y 3.3 V.
- capacidad de tener varios *masters*
- frecuencias de operación de hasta 33 MHz
- multiplexado de las líneas de datos y direcciones
- configuración mediante software y registros
- especificación independiente del procesador

Todos los dispositivos se conectan a un bus único y el *bridge* de conexión con la RAM es transparente para el usuario

Estructura del bus PCI

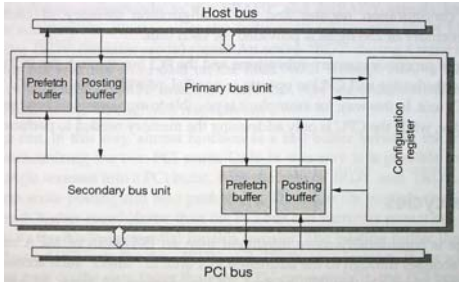


[1]

Estructura del bus PCI (cont.)

En la estructura normal de un PC, el bridge de conexión con el bus local de la CPU se conoce como *Northbridge* (bus primario)

El bridge que conecta el bus PCI con el resto de las unidades se conoce como *Southbridge* (bus secundario)



[1]

El ciclo del bus PCI

Los ciclos de bus requieren dos o tres ciclos de reloj por cada transferencia simple

- envío de la dirección en el primero
- escritura de datos en el segundo
- lectura de datos en el tercero

Así, el ritmo de transferencia máximo para 32 bits a 33 MHz es de 66 MB/s en escritura y 44 MB/s en lectura

Sin embargo, en el modo a ráfagas la dirección sólo se transfiere una vez por lo que el máximo ritmo de transferencia se duplica

- los accesos a direcciones consecutivas los transforma en un único acceso a ráfagas

El ciclo del bus PCI (cont.)

El PCI reconoce 12 tipos de acceso al bus (diferenciados por 4 líneas de control):

- secuencia INTA (0000)
  - direccionamiento del controlador de interrupciones
  - no hay dirección explícita
  - en la fase de datos se transfiere el vector de interrupción por las líneas de datos (ADx)
- ciclo especial (0001)
  - transferencia de datos a todas las unidades PCI conectadas
  - p.e., sobre el estado del procesador
  - las líneas de datos menos significativas (AD15-AD0) codifican la información: 0000h (procesador apagado), 0001h (procesador parado), 0002h (código x86), 0003h-ffffh (reservados)
  - el código específico x86 va en las líneas de datos AD31-AD16

El ciclo del bus PCI (cont.)

- acceso de lectura I/O (0010)
  - acceso de lectura a una unidad en el área de I/O
- acceso de escritura I/O (0011)
  - acceso de escritura a una unidad en el área de I/O
- acceso de lectura de memoria (0110)
  - acceso de lectura a una unidad en el área de memoria
- acceso de escritura de memoria (0111)
  - acceso de escritura a una unidad en el área de memoria
- acceso de lectura de configuración (1010)
  - acceso al área de configuración
  - las líneas AD7-AD2 en direccionamiento indican la palabra de 32 bits a leer del área de configuración
  - AD10-AD8 seleccionan la unidad de un dispositivo multifunción

El ciclo del bus PCI (cont.)

- acceso de escritura de configuración (1011)
  - como el anterior pero para escribir
- acceso de lectura de memoria de línea (1110)
  - indica al dispositivo que se quieren leer más de dos bloques de 32 bits
  - el acceso consta de una fase de direccionamiento y varias de datos
- acceso de escritura de memoria con invalidación (1111)
  - es como el caso anterior pero para escritura
  - indica que se quiere escribir al menos una línea completa de caché
- acceso de lectura múltiple de memoria (1100)
  - es una extensión de la lectura de memoria de línea, e indica al dispositivo que se quiere leer más de una línea de caché, o el bloque de datos correspondiente si no hay caché

El ciclo del bus PCI (cont.)

- ciclo de direccionamiento dual (1101)
  - se usa para transferir una dirección de 64 bits a una unidad PCI de 32 bits
  - en el primer ciclo se transfieren los 32 bits menos significativos y en el segundo los 32 más significativos
  - permite accesos de dispositivos o CPUs con menos de 64 bits de datos o direcciones

En un ciclo de lectura se introduce un ciclo ***dummy*** en el cambio de la dirección del bus ADx entre las direcciones y los datos (en un ciclo de escritura no es necesario)

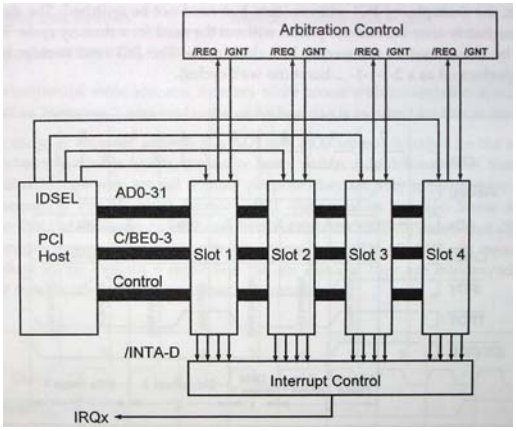
Arbitrio del bus

El arbitrio del bus se realiza en base a dos señales

- REQ: petición de uso del bus por el dispositivo
- GNT: concesión de uso del bus

Una vez concedido el uso del bus se puede comenzar la transferencia dentro de los siguientes 16 ciclos de reloj

Arbitrio del bus (cont.)



[1]

DMA y master de bus

Los dispositivos PCI tiene la posibilidad de convertirse en masters del bus por lo que no necesitan ningún controlador de DMA

- **las transferencias directas a memoria las hace un master de bus usando el modo *burst***

Interrupciones

Normalmente cada unidad conectada al PCI dispone de una línea de interrupción INTA

- **las unidades multifunción pueden tener además otras tres líneas de interrupción INTB, INTC e INTD**
- **las interrupciones son disparadas por nivel (activo bajo)**

La petición de interrupciones se canaliza a través de las mismas líneas INTx del bus ISA, con varias posibilidades de mapeado de interrupciones PCI a las del procesador

Las unidades PCI suelen tener asociada la misma IRQ en el PC, lo que nos lleva a la situación de compartir interrupciones

- **Linux ofrece esta posibilidad**

Modelo de programación

Cada periférico PCI viene caracterizador por:

- un número de *bus* (hasta 256 buses)
- un número de *dispositivo* (hasta 32 dispositivos)
- un número de *función* (hasta 8 funciones)

Además, puede haber varios dominios PCI, cada uno con hasta 256 buses

Cuando hay varios buses se suelen conectar mediante un *bridge*, dando una estructura en árbol

- suele haber también un puente a otros buses (ISA)

Modelo de programación (cont.)

Los periféricos PCI atienden a tres tipos de direccionamiento

- posiciones de memoria (32 o 64 bits)
- puertos de entrada/salida (32 bits)
- registros de configuración

Los dos primeros están compartidos por todos los periféricos de un mismo bus

- se accede con las funciones de memoria y entrada/salida
- se configuran para que cada periférico use direcciones distintas

Modelo de programación (cont.)

Los registros de configuración usan direccionamiento "geográfico", dirigido específicamente al periférico

- se accede con funciones especiales
- 256 bytes para cada función y aquí se puede leer la configuración de las direcciones de memoria e I/O

Al arrancar, el dispositivo no tiene configurada la memoria ni los puertos de I/O

- sólo responde a transacciones de configuración

El *firmware* del sistema configura cada periférico poniéndolo en direcciones de memoria e I/O distintas

- ahora ya responde a accesos de memoria o I/O

Modelo de programación (cont.)

La especificación PCI soporta direccionamientos de 32 y 64 bits al espacio de I/O

- esto sólo se aplica al acceso a los dispositivos
- en una arquitectura x86 no se puede alcanzar una dirección mayor de 64K porque el procesador sólo tiene 16 bits de direcciones para I/O
- los puertos en un PC con bus PCI están por debajo de estos 64K

El bus PCI tiene dos registros de 32 bits en el espacio de I/O

- el registro de configuración de direcciones en la dirección 0cf8h
- el registro de configuración de los datos en la dirección 0cfch

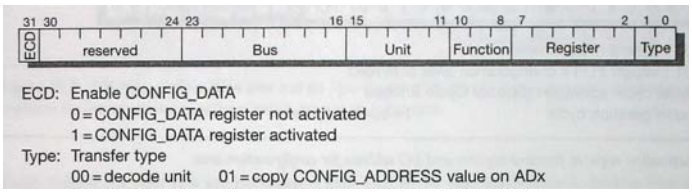
Modelo de programación (cont.)

Para leer o escribir una palabra de 32 bits en el área de configuración de un dispositivo PCI primero hay que transferir la dirección al registro de configuración de direcciones

- después se puede leer o escribir el valor del registro de datos en la dirección indicada en el registro de configuración de direcciones

Normalmente los sistemas operativos disponen de funciones especiales para realizar estos accesos

Registro de configuración de direcciones



[1]

El bit ECD permite seleccionar si el ciclo de acceso al bus es normal o al área de configuración

Registro de configuración de direcciones (cont.)

El resto selecciona:

- el bus (hasta 256)
- el dispositivo (hasta 32)
- la función (hasta 8)
- el registro (una de las 64 palabras de 32 bits del área de configuración)
- el tipo de transferencia
  - con valor 01 se copia el contenido del registro directamente en las líneas de direcciones del bus
  - el acceso al área de configuración mediante este registro se conoce como mecanismo de configuración #1

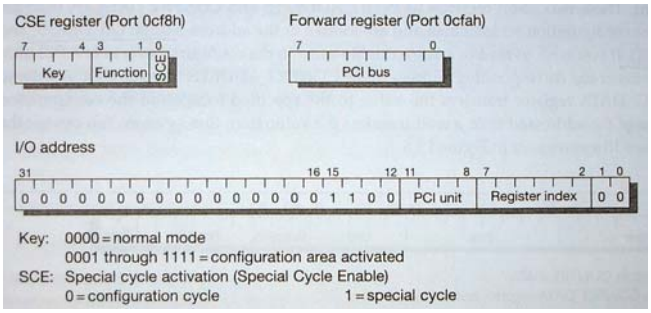
Registro de configuración de direcciones (cont.)

En los PCs existe también el mecanismo de configuración #2

- el área de configuración se mapea en los 4K del espacio de I/O en el rango c000h-cfffh

Esto se lleva a cabo mediante la programación del registro de activación CSE (*Configuration Space Enable*)

Registro de activación



[1]

Registro de activación (cont.)

Un valor distinto de 0 en *key* activa el mapeado del área de configuración

- **todos los accesos en el rango c000h-ffffh inician un ciclo de configuración**

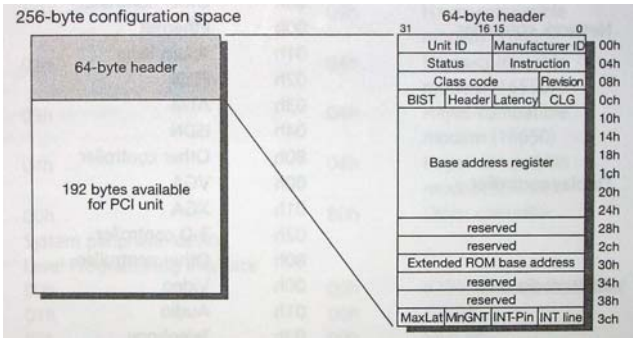
El valor del registro *Forward* indica el número de bus por el que hay que hacer el ciclo

Cualquier acceso dentro del rango de 4K de I/O especifica:

- **AD11-AD8: el dispositivo PCI (la función se toma del CSE)**
- **AD7-AD2: llevan el offset a palabras de 32 bits (índice del registro)**
- **AD31-AD12: se hacen igual a 0000ch**

Espacio de direcciones de configuración

Cada dispositivo PCI (o cada función) dispone de un área de 256 bytes de configuración (ordenados en little-endian)



[1]

Espacio de direcciones de configuración (cont.)

El área de configuración se compone de:

- **una cabecera fija de 64 bytes al comienzo con una estructura predefinida**
- **192 bytes cuyo uso depende del dispositivo**

La cabecera se divide en dos secciones:

- **los primeros 16 bytes (00h-0fh) son los mismos para todos los dispositivos**
- **la distribución de los otros 48 bytes puede variar dependiendo del dispositivo**
  - **la distribución se diferencia por el valor de la cabecera en 0eh**
  - **el bit más significativo (bit 7) indica si el dispositivo es multifunción (a 1) o simple (a 0)**

Espacio de direcciones de configuración (cont.)

La especificación PCI sólo requiere que estén disponibles los valores de:

- **`Unit ID`**: valores válidos entre 0000h-fffeh; el valor ffffh indica que el dispositivo no está instalado; lo asigna cada fabricante
- **`Manufacturer ID`**: lo asigna el comité de estandarización
- **`Status`**: registro de estado
- **`Instruction`**: registro de comandos

Espacio de direcciones de configuración (cont.)

El código de clase (**`Class Code`**) indica el tipo de unidad PCI de acuerdo con los valores de una tabla; el código se divide en tres campos de 1 byte:

- el byte más significativo (offset 0bh) muestra el código de clase básico
  - controlador de memoria, controlador de disco, etc.
- el byte medio (offset 0ah) muestra el código de subclase
  - RAM, Flash, otros; IDE, RAID, SCSI, Diskette, otros
- el byte menos significativo (offset 09h) muestra la interfaz de programación
  - para muchos dispositivos su valor es 00h

Espacio de direcciones de configuración (cont.)

La entrada **`CLS`** (***cache line size***) define el tamaño de la cache del sistema en unidades de 32 bits

La entrada de latencia (**`Latency`**) más 8 ciclos de reloj expresa la longitud del ciclo PCI

El bit más signiticativo de la entrada **`BIST`** (***Built-In Self-Test***) indica si la unidad puede hacer un test de sí misma

La entrada **`INT Line`** indica la interrupción que se va a usar (valores de 0 a15)

Espacio de direcciones de configuración (cont.)

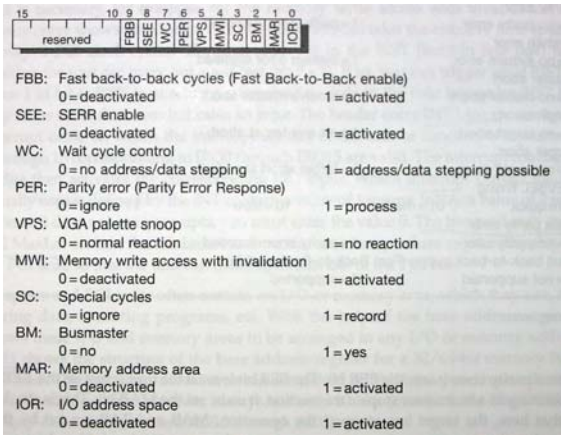
La entrada `INT-Pin` indica la línea de interrupción del PCI que se usa

- **INTA (1), INTB (2),...**
- **si no se usan interrupciones el valor es 0**

Las entradas `MinGNT` y `MaxLat` son de sólo lectura y contienen las latencias mínima y máxima requeridas por el fabricante del dispositivo

- **el objetivo es optimizar el uso del bus**

Registro de comandos



FBB:	Fast back-to-back cycles (Fast Back-to-Back enable)	
	0 = deactivated	1 = activated
SEE:	SERR enable	
	0 = deactivated	1 = activated
WC:	Wait cycle control	
	0 = no address/data stepping	1 = address/data stepping possible
PER:	Parity error (Parity Error Response)	
	0 = ignore	1 = process
VPS:	VGA palette snoop	
	0 = normal reaction	1 = no reaction
MWI:	Memory write access with invalidation	
	0 = deactivated	1 = activated
SC:	Special cycles	
	0 = ignore	1 = record
BM:	Busmaster	
	0 = no	1 = yes
MAR:	Memory address area	
	0 = deactivated	1 = activated
IOR:	I/O address space	
	0 = deactivated	1 = activated

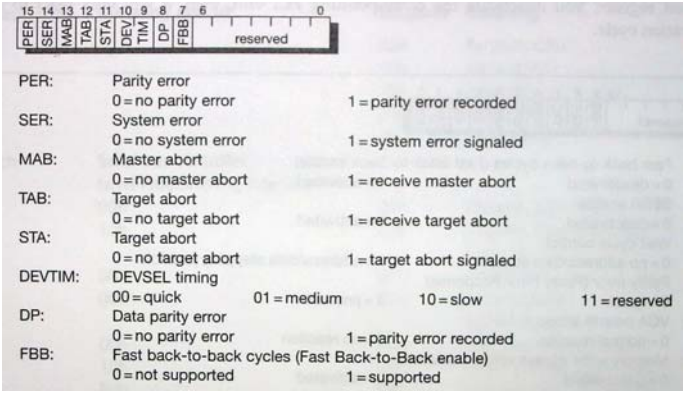
[1]

Registro de comandos (cont.)

Si se pone a valor 0000h se desactiva el dispositivo

- **FBB: reduce los ciclos muertos entre dos transacciones del bus**
- **SEE: activa la señal $\overline{\text{SERR}}$ del bus**
- **PER: activa la detección de errores de paridad**
- **WC: activación en pasos de las señales de direcciones/datos**
- **VPS: para un dispositivo VGA ignora los accesos a la paleta**
- **MWI: habilita la escritura con invalidación**
- **SC: habilita el reconocimiento de ciclos especiales**
- **BM: el dispositivo opera como *master* del bus**
- **MAR: el dispositivo responde a los accesos a memoria**
- **IOR: el dispositivo responde a los accesos de I/O**

Registro de estado



[1]

Registro de estado (cont.)

- PER: se activa si hay error de paridad
- SER: se activa si hay activación de la señal SERR
- MAB, TAB: las activa un dispositivo que opera como *master* del bus cuando se aborta una transacción
- STA: la activa un dispositivo *target* cuando aborta una transacción
- DEVTIM: los 2 bits llevan las características de tiempo de la señal DEVSEL
- DP: en un dispositivo que opera como *master* del bus se activa si hay error de paridad en los datos
- FBB: indica si un dispositivo *target* soporta la reducción de ciclos muertos

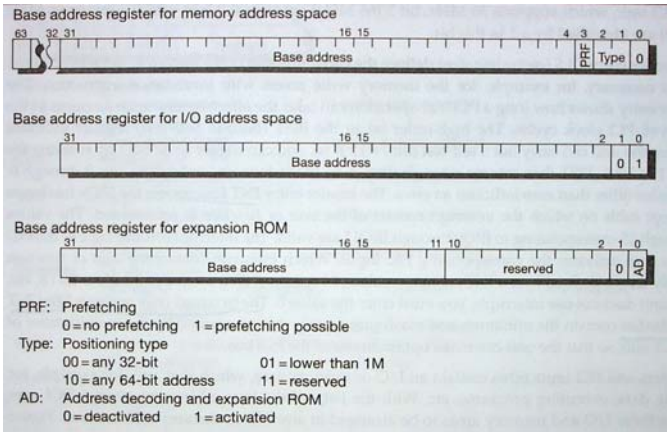
Registro de estado (cont.)

Los bit 6 a 0 están reservados y no se usaban hasta la versión 2.0

Desde la versión 2.2

- el bit 4 activa el acceso a un registro de nuevas capacidades
  - offset 34h
  - el byte 0 lleva la información (AGP, control de potencia, etc)
- el bit 5 identifica las unidades de 33MHz (a 0) y de 66MHz (a 1)

Registro de direcciones base



[1]

Registro de direcciones base (cont.)

Este registro permite manejar las áreas de memoria y de I/O que un dispositivo PCI puede usar para almacenar datos o programas

Las direcciones de memoria pueden expresarse con 32 ó 64 bits dependiendo de la implementación

Las direcciones de I/O siempre se expresan con 32 bits (en la arquitectura x86 sólo se usan las 16 menos significativas)

El bit 0 diferencia el tipo de dirección base:

- **de memoria a valor 0**
- **de I/O a valor 1**

Registro de direcciones base (cont.)

Dependiendo del tamaño de la dirección se pueden usar tres o seis direcciones base de acuerdo con los 24 bytes reservados en la cabecera

Direcciones de memoria:

- **los 4 bits menos significativos no se pueden cambiar**
- **se pueden mapear direcciones de 16 bytes, 32 bytes, 64 bytes, ..., 1K, 2K, etc.**
- **el bit PRF a 1 permite el *prefetching* (leer datos por adelantado)**
- **el tipo de acceso**
  - **00 indica que cualquier registro de 32 bits se puede mapear en un área cuyo direcccionamiento se hace con 32 bits**
  - **10 indica lo mismo pero para 64 bits**

Registro de direcciones base (cont.)

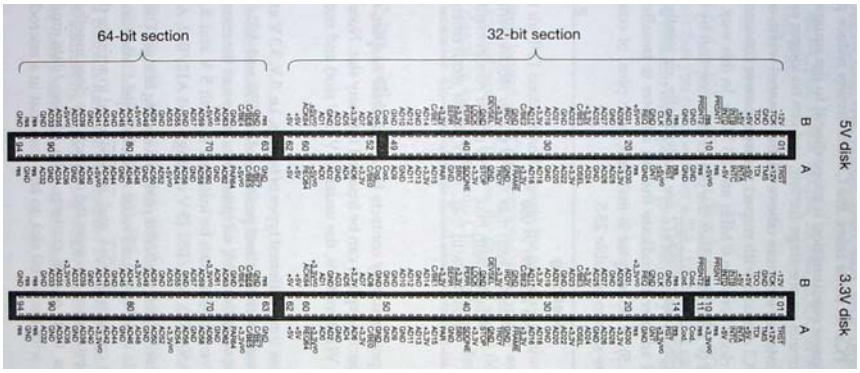
Direcciones de I/O:

- sólo los 2 bits menos significativos no se pueden cambiar
- se pueden mapear direcciones de 2 bytes, 4 bytes, etc.

El registro de expansión de la ROM permite mapear ésta en cualquier dirección en el rango de 32 bits

- funciona como el mapeado de memoria de 32 bits
- activando el bit AD se activa el uso de este registro

Slots PCI



[1]

Plataformas de Tiempo Real: Dispositivos y Drivers

Tema 6. Buses

- El bus PCI
- Programación del bus PCI
- Programación de la tarjeta PCI-9111DG

Introducción a los drivers PCI

La estructura de un sistema PCI es un árbol en el que los buses se conectan mediante *bridges* hasta llegar al bus 0, que es el principal

- sistemas como el CardBus también se conectan al sistema PCI mediante *bridges*

Los drivers de Linux no necesitan utilizar directamente las direcciones de 16 bits definidas en el hardware del x86

- en su lugar utilizan la estructura de datos `struct pci_dev` para actuar sobre los dispositivos
- la información de esta estructura está mayormente oculta, pero parte de la misma se puede obtener con el comando `lspci`, y la información contenida en `/proc/pci` o `/proc/bus/pci`

Introducción a los drivers PCI (cont.)

Cuando se muestran las direcciones hardware de 16 bits, se muestran como:

- número de bus (8 bits), número de dispositivo (5 bits) y número de función (3 bits)
  - p.e. `less /proc/bus/pci/devices`
- en ocasiones se muestra también el dominio
  - p.e. con `lspci`

Accesos a memoria:

- el acceso a las regiones de memoria y de entrada/salida se hace con las funciones usuales, como `inb`, `outb`, etc.
- el acceso a los registros de configuración se realiza con funciones especiales

Introducción a los drivers PCI (cont.)

El espacio de direcciones de I/O usa 32 bits por lo que puede acceder a 4 GB de puertos de I/O

El direccionamiento de memoria puede usar 32 ó 64 bits

Cada tarjeta tiene cuatro líneas de interrupción

- se mapean por hardware a las líneas de interrupción del procesador fuera del bus PCI
- como las interrupciones en el PC son limitadas las líneas de interrupción se deben poder compartir

Arranque del sistema

En el arranque del sistema se produce la configuración de los dispositivos asignándoles las regiones de memoria que van a usar

- también se puede producir la configuración en dispositivos que se conectan en caliente

Después del arranque la información se guarda en la estructura de ficheros

En el fichero `/proc/bus/pci/devices` se encuentra la información del dispositivo en formato de texto

En `/proc/bus/pci/*/*` se encuentran los ficheros binarios que llevan la información de los registros de configuración de cada dispositivo

Arranque del sistema (cont.)

La información de los directorios de cada dispositivo PCI se encuentra en el directorio `/sys/bus/pci/devices`

Cada directorio de dispositivo (p.e. 0000:00:1d.0) contiene los siguientes ficheros:

- `config` contiene la información de configuración
- `vendor`, `device`, `subsystem_device`, `subsystem_vendor`, y `class` son valores específicos del dispositivo PCI
- `irq` muestra la interrupción asignada a un dispositivo
- `resource` muestra la memoria asignada al dispositivo

Toda esta información se puede obtener con `lspci` y sus opciones (p.e., -x, -vv)

Configuración e inicialización

Normalmente hay tres registros en las direcciones de configuración que identifican un dispositivo:

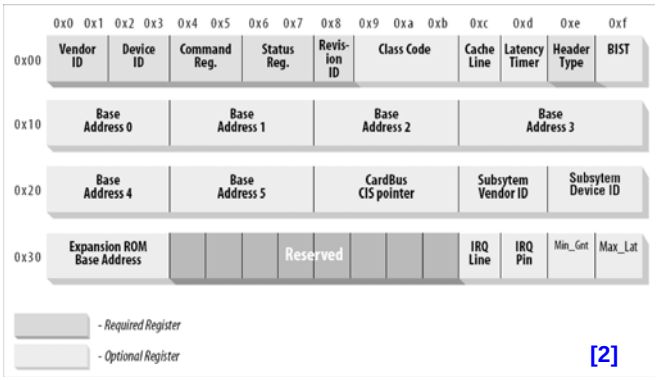
- Vendor ID
- Device ID
- Class

Puede haber además otros dos registros que añaden más información sobre la identificación del dispositivo:

- Subsystem Vendor ID
- Subsystem Device ID

Cuando se manejan los registros de configuración es preciso tener cuidado con el orden de los bytes

Configuración e inicialización (cont.)



[2]

Configuración e inicialización (cont.)

Los drivers disponen del tipo de dato `struct pci_device_id` con la información sobre la identificación del dispositivo (en `<linux/mod_devicetable.h>`)

```
__u32 vendor;
__u32 device;
__u32 subvendor;
__u32 subdevice;
```

- si el driver puede soportar cualquier fabricante o dispositivo puede tomar el valor `PCI_ANY_ID`

```
__u32 class;
__u32 class_mask;
```

- si el driver puede soportar cualquier clase se debe poner a 0

```
kernel_ulong_t driver_data
```

- no es necesario, pero puede soportar cualquier información que el driver utilice para diferenciar el dispositivo

Configuración e inicialización (cont.)

Hay dos macros que permiten crear e inicializar la `struct pci_device_id` con los campos especificados

```
PCI_DEVICE(vendor,device)
PCI_DEVICE_CLASS(device_class,device_class_mask)
```

Si se quiere arrancar un dispositivo en caliente y cargar el módulo adecuado se debe exportar esta estructura al espacio de usuario

- se dispone de la macro

```
MODULE_DEVICE_TABLE(pci,list)
```

- `list` es una estructura de estructuras `pci_device_id` con la última rellena con ceros

Acceso al espacio de configuración

La API para el acceso al bus PCI está en `<linux/pci.h>`

Para el acceso al espacio de configuración, Linux dispone de un conjunto de funciones que permite transferencias de 8, 16 ó 32 bits

```
int pci_read_config_byte
    (struct pci_dev *dev, int where, u8 *val)
int pci_read_config_word
    (struct pci_dev *dev, int where, u16 *val)
int pci_read_config_dword
    (struct pci_dev *dev, int where, u32 *val)
```

- lee uno, dos o cuatro bytes del espacio de configuración representado en la `struct pci_dev`
- la función devuelve un código de error si va mal
- no hay que preocuparse del orden de los bytes

Acceso al espacio de configuración (cont.)

```
int pci_write_config_byte
    (struct pci_dev *dev, int where, u8 val)
int pci_write_config_word
    (struct pci_dev *dev, int where, u16 val)
int pci_write_config_dword
    (struct pci_dev *dev, int where, u32 val)
```

- igual que las anteriores pero para escritura

Hay constantes que definen el valor `where` representando los diferentes registros del espacio de configuración

- se encuentran en `<linux/pci_regs.h>`

Acceso al espacio de configuración (cont.)

En el caso de que el driver no tenga acceso a la `struct pci_dev` se puede usar otro conjunto de funciones:

```
int pci_bus_read_config_byte
    (struct pci_bus *bus,
     unsigned int devfn, int where, u8 *val)
int pci_bus_read_config_word
    (struct pci_bus *bus,
     unsigned int devfn, int where, u16 *val)
int pci_bus_read_config_dword
    (struct pci_bus *bus,
     unsigned int devfn, int where, u32 *val)
```

Acceso al espacio de configuración (cont.)

```
int pci_bus_write_config_byte
    (struct pci_bus *bus,
     unsigned int devfn, int where, u8 val)
int pci_bus_write_config_word
    (struct pci_bus *bus,
     unsigned int devfn, int where, u16 val)
int pci_bus_write_config_dword
    (struct pci_bus *bus,
     unsigned int devfn, int where, u32 val)
```


Registro de un driver PCI

Para registrar un driver PCI se debe crear una variable del tipo `struct pci_driver`

- contiene un conjunto de funciones (*callbacks*) y variables que el driver pone a disposición del núcleo PCI

Los principales campos de esta estructura son:

```
const char *name;
```
- nombre del driver, que debe ser único en el kernel
- normalmente es el mismo que el del módulo
- aparecerá en el directorio `/sys/bus/pci/drivers`

```
const struct pci_device_id *id_table;
```
- puntero a la tabla de dispositivos que soporta

Registro de un driver PCI (cont.)

```
int (*probe) (struct pci_dev *dev,
              const struct pci_device_id *id)
```
- función de prueba del driver
- la llama el núcleo PCI cuando tiene una `struct pci_dev` que cree que el driver quiere controlar
- si el driver estaba esperando esta estructura, la debe inicializar y retornar un 0
- si no la esperaba o sucede un error, devuelve un valor negativo

```
void (*remove) (struct pci_dev *dev)
```
- el núcleo PCI llama a esta función cuando quiere eliminar la `struct pci_dev` o cuando el kernel descarga el driver

Registro de un driver PCI (cont.)

```
int (*suspend) (struct pci_dev *dev, u32 state)
```
- el núcleo PCI llama a esta función cuando se quiere suspender la `struct pci_dev`
- el estado de suspendido se pasa en `state`
- esta función es opcional

```
int (*resume) (struct pci_dev *dev)
```
- el núcleo PCI llama a esta función cuando se quiere restablecer la `struct pci_dev`
- siempre se llama después de `suspend`, y también es opcional

Registro de un driver PCI (cont.)

Para la creación de la `struct pci_driver` sólo se necesita inicializar cuatro campos

```
static struct pci_driver un_driver = {
    .name = "nombre_driver";
    .id_table = ids;
    .probe = probe;
    .remove = remove;
}
```

Para registrar el driver PCI se hace con la función

```
pci_register_driver(&un_driver)
```
- devuelve un 0 si va bien y un valor negativo en caso contrario

Registro de un driver PCI (cont.)

Para eliminar un driver PCI se hace con la función

```
pci_unregister_driver(&un_driver)
```
- se debe llamar cuando el driver es desinstalado
- asegura que se hace una llamada a `remove` antes de que la función retorne

Habilitación de un dispositivo PCI

En la función `probe` del driver y antes de acceder a los recursos del dispositivo se debe llamar a la función de habilitación del dispositivo

```
int pci_enable_device(struct pci_dev *dev)
```

- esta función activa el dispositivo y en algunos casos asigna su línea de interrupción y regiones de I/O

Alternativa a la detección de un dispositivo PCI

Si se necesita encontrar un dispositivo se pueden usar la siguiente función:

```
struct pci_dev *pci_get_device
  (unsigned int vendor, unsigned int device,
   struct pci_dev *from)
```

- escanea la lista de dispositivos PCI presentes en el sistema y trata de encontrar el que coincide con `vendor` y `device`
- devuelve una `struct pci_dev` e incrementa un contador interno que se debe decrementar con `pci_dev_put()` cuando se deje de usar el dispositivo
- `from` permite hacer una búsqueda desde el dispositivo indicado; para encontrar el primero debe ser `NULL`

Alternativa a la detección de un dispositivo PCI (cont.)

Ejemplo de uso:

```
struct pci_dev *dev;
dev=pci_get_device(0x8086,0x1031,NULL);
if (dev) {
   /*uso el dispositivo*/
   pci_dev_put(dev);
}
```

Alternativa a la detección de un dispositivo PCI (cont.)

Otras funciones similares son:

```
struct pci_dev *pci_get_subsys
  (unsigned int vendor, unsigned int device,
   unsigned int ss_vendor, unsigned int ss_device,
   struct pci_dev *from)
```

- permite especificar más parámetros en la búsqueda

```
struct pci_dev *pci_get_slot
  (struct pci_bus *bus, unsigned int devfn)
```

- hace la búsqueda sobre la estructura **bus** y el número de función

Acceso a la memoria

Un dispositivo PCI puede implementar hasta 6 regiones de memoria

- pueden ser direcciones de memoria convencional o memoria de I/O
- la mayoría de dispositivos implementan sus registros en memoria convencional
- la diferencia fundamental en el uso de una u otra memoria es si se utiliza la caché
  - en la memoria de I/O no se utiliza la caché
  - en la memoria convencional sí se utiliza (se marca como ***memory-is-prefetchable*** en el registro de configuración)
  - si un dispositivo mapea sus registros de control en memoria convencional tampoco debería usar la caché

Acceso a la memoria (cont.)

Para el acceso a las regiones de memoria (puertos de I/O o memoria) especificadas en el área de configuración se pueden utilizar las constantes

- PCI_BASE_ADDRESS_0 hasta PCI_BASE_ADDRESS_5
- si el dispositivo usa 64 bits de direcciones utiliza dos regiones consecutivas para especificar cada región
- un dispositivo puede ofrecer regiones de 32 y de 64 bits a la vez

Para el acceso a la información de las regiones de memoria de los dispositivos PCI, el kernel proporciona las siguientes funciones:

Acceso a la memoria (cont.)

```
unsigned long pci_resource_start
    (struct pci_dev *dev, int bar)
```
- devuelve la dirección de comienzo de la región de memoria indicada por `bar` (0-5)

```
unsigned long pci_resource_end
    (struct pci_dev *dev, int bar)
```
- devuelve la dirección final de la región de memoria `bar`

```
unsigned long pci_resource_flags
    (struct pci_dev *dev, int bar)
```
- devuelve los `flags` asociados a la región de memoria `bar`

Acceso a la memoria (cont.)

Los `flags` de la ultima función se definen en `<linux/ioport.h>`, los más importantes son:

`IORESOURCE_IO`
`IORESOURCE_MEM`
- indican si la región asociada es memoria o I/O
- sólo uno de los dos puede estar activo

`IORESOURCE_PREFETCH`
- indica si la región asociada usa la caché

`IORESOURCE_READONLY`
- indican si la región asociada está protegida contra escritura

Interrupciones PCI

En el arranque del sistema se asigna un único número de interrupción al dispositivo, que se guarda en el registro de configuración PCI_INTERRUPT_LINE (1 byte)
- la lectura de este registro permite saber la IRQ que se usa

Si el dispositivo no soporta interrupciones el registro de configuración PCI_INTERRUPT_PIN (1 byte) tiene valor 0
- normalmente no es necesario usarlo, si se sabe si el dispositivo soporta interrupciones

Abstracciones del hardware

En el bus PCI las únicas operaciones dependientes del hardware son las de lectura y escritura de los registros de configuración

- las demás operaciones se hacen directamente sobre la memoria o los puertos de I/O

La estructura para acceder a los registros de configuración es `struct pci_ops`, en `<linux/pci.h>`; tiene dos campos:

```
int (*read) (struct pci_bus *bus,
             unsigned int devfn,
             int where, int size, u32 *val)
int (*write) (struct pci_bus *bus,
             unsigned int devfn,
             int where, int size, u32 val)
```


Creación de drivers PCI

Para crear un driver PCI estos son los pasos "*extra*" a dar:

- crear una variable del tipo `struct pci_device_id` e inicializarla poniendo
  - con `PCI_DEVICE` o con `PCI_DEVICE_CLASS`
- exportar la lista de dispositivos manejados por el driver con la macro `MODULE_DEVICE_TABLE`
- registrar el driver con `pci_register_driver`
- habilitar en la función `probe` el dispositivo, con `pci_enable_device`
- obtener las direcciones de memoria usadas del espacio de configuración con `pci_read_config_dword`

Creación de drivers PCI (cont.)

- acceder a los espacios de memoria e I/O desde los puntos de entrada
  - ojo con las optimizaciones del compilador
- desinstalar el driver con `pci_unregister_driver`

El resto de la programación del driver depende del dispositivo concreto y se hace como hemos visto hasta ahora

Tema 6. Buses

- El bus PCI
- Programación del bus PCI
- Programación de la tarjeta PCI-9111DG

Tarjeta PCI-9111DG

Tarjeta de adquisición de datos con arquitectura del bus PCI de 32 bits fabricada por ADLINK Technologies INC.

Las principales características son:

- 16 canales analógicos de entrada de 12 bits
- frecuencias de muestreo de hasta 100 KHz
- ganancias programables
- una salida analógica de 12 bits
- 16 canales digitales de entrada y 16 de salida
- 3 contadores programables *up-down*
- tres modos de disparo para la conversión A/D

Tarjeta PCI-9111DG (cont.)

Especificaciones de las entradas analógicas:

- resolución de 12 bits
- ganancias programables: x1, x2, x4, x8, x16
- rangos de entrada: +-10 V, +-5 V, +-2.5 V, +-1.25 V, +-0.625 V
- tiempo de conversión 8.5 μs
- transferencia de datos: polling, interrupción con una muestra, e interrupción con la cola FIFO a la mitad
- transferencia de datos: 110 KHz máximo
- tamaño de la FIFO: 1024 muestras

Tarjeta PCI-9111DG (cont.)

Especificaciones de la salida analógica:

- **resolución de 12 bits**
- **tensión de salida (*jumper*)**
  - **unipolar: 0..10 V**
  - **bipolar: -10..+10 V**
- **tiempo de establecimiento 30 μs**

Nos centramos en la parte analógica de la tarjeta (para tener la información completa consultar el manual [5])

Registros de la PCI-9111DG

I/O Address	Write	Read
Base + 00h	DA value	AD FIFO value
Base + 02h	Digital Output	Digital Input
Base + 04h	Extended DO	Extended DI
Base + 06h	AD channel control	AD channel read back
Base + 08h	AD range control	AD range and AD status read back
Base + 0Ah	AD trigger mode	AD mode and interrupt setting read back
Base + 0Ch	Interrupt control	ISC2 & Trigger even read back*1)
Base + 0Eh	Software AD trigger	(Not used)
Base + 10h ~3Eh	Reserved	
Base + 40h	Timer 8254 Ch#0	
Base + 42h	Timer 8254 Ch#1	
Base + 44h	Timer 8254 Ch#2	
Base + 46h	Timer Control	Timer Status
Base + 48h	Clear H/W IRQ	(Not used)

[5]

Registros para A/D

A/D Data Register

- **BASE+0h, sólo lectura**
- **obtiene el dato de la FIFO y el canal que lo ha producido**

A/D Channel Control Register

- **BASE+6h, sólo escritura**
- **selecciona el canal a convertir (depende del modo de conversión)**

A/D Channel Read Back Register

- **BASE+6h, sólo lectura**
- **informa del canal seleccionado**

Registros para A/D (cont.)

A/D Input Signal Range Control Register
- BASE+8h, sólo escritura
- selecciona la ganancia del canal

A/D Range and Status Read Back Register
- BASE+8h, sólo lectura
- informa de la ganancia seleccionada y del estado de la FIFO

Software Trigger Register
- BASE+0eh, sólo escritura
- la escritura de cualquier valor ordena una conversión del A/D

Registros para A/D (cont.)

A/D Trigger Mode Control Register
- BASE+0ah, sólo escritura
- controla la fuente y el modo de disparo

Interrupt Control Register
- BASE+0ch, sólo escritura
- selecciona la fuente de generación de interrupciones

Hardware Interrupt Clear Register
- BASE+48h, sólo escritura
- se debe escribir cualquier valor después de manejar la interrupción para eliminar la fuente (disparada por nivel)

Registros para A/D (cont.)

A/D Mode & Interrupt Control Read Back Register
- BASE+0ah, sólo lectura
- informa del modo de disparo y control de interrupción programado

Registro para D/A

D/A Output Register

- **BASE+0h, sólo escritura**
- **transforma el dato de 12 bits escrito en una señal analógica unipolar o bipolar de acuerdo con la selección del *jumper* de la tarjeta**

Driver para la tarjeta PCI-9111DG

Una opción sencilla para construir un driver de la tarjeta que controle la parte analógica es

- **permitir que los puntos de entrada de lectura y escritura accedan directamente a los valores de conversión**
  - **la lectura devolverá un valor de la conversión A/D del canal seleccionado (esperará si es necesario a tener el dato)**
  - **la escritura establecerá el dato pasado como valor de salida del convertidor D/A (sólo un dato)**
  - **se pueden programar comandos para cambiar el canal de conversión A/D**
- **dejar el uso de estos valores y su posible almacenamiento a la aplicación**

Conversión A/D

Para realizar una conversión A/D se debe tener en cuenta la naturaleza de las señales a convertir y las características de la tarjeta de la que se dispone:

- **asegurarse de que el hardware está correctamente conectado**
  - **señales dentro de los rangos admitidos con las ganancias que se pueden programar**
  - **selección de los canales a usar**
- **decidir cómo se va a disparar la conversión y cómo se va a leer el valor convertido (polling o interrupción)**
  - **los datos convertidos se van guardando en la FIFO; a la máxima frecuencia 100 KHz puede guardar sobre 10 ms.**

En un driver se debe programar la configuración de la tarjeta en la instalación o en la apertura del dispositivo

Conversión A/D (cont.)

El modo de operación para la conversión puede ser por ejemplo:

- polling con el punto de entrada de lectura
  - disparar la conversión por software
  - esperar el tiempo de conversión (8.5 μs)
  - leer el dato convertido
- interrupción y punto de entrada lectura
  - disparar la conversión y esperar en un mecanismo de sincronización de eventos hasta que el manejador de intrerrupción le despierte; después de despertar devolver el dato
  - el manejador de interrupción lee el dato y despierta al punto de entrada de lectura

Conversión D/A

La escritura del convertidor D/A se hace desde el punto de entrada de escritura

- sólo tiene que colocar el dato con el formato adecuado en el registro de salida del convertidor

Ejemplo: muestreo y reconstrucción de señales analógicas

Vamos a realizar una aplicación que sea capaz de:

- muestrear una señal analógica y almacenarla en memoria
- reproducir la señal a partir del valor almacenado en la memoria

Para ello realizamos el driver sencillo para la tarjeta PCI-9111DG que permite leer un valor de la entrada analógica y escribir un valor en la salida analógica

Driver PCI-9111DG: includes

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/slab.h>
#include <asm/uaccess.h>
#include <asm/io.h>
#include <linux/ioport.h>
#include <linux/timer.h>
#include <linux/jiffies.h>
#include <linux/delay.h>
#include <linux/pci.h>
#include <linux/pci_regs.h>
#include "pci_9111.h"

MODULE_LICENSE("GPL");
```


Driver PCI-9111DG: constantes

```
// Identificacion de la tarjeta
#define VENDOR        0x144a
#define DEVICE        0x9111

// Offset de los registros de la tarjeta PCI-9111DG
#define AD_DR         0x00
#define AD_CCR        0x06
#define AD_CRBR       0x06
#define AD_ISRCR      0x08
#define AD_RSRBR      0x08
#define AD_STR        0x0e
#define AD_TMCR       0x0a
#define AD_ICR        0x0c
#define AD_HICR       0x48
#define AD_MICRBR     0xac
#define DA_OR         0x00
```


Driver PCI-9111DG: datos del dispositivo

```
// Datos del dispositivo
struct pci_9111_datos {
  struct cdev *cdev;          // Character device structure
  dev_t dev;                  // informacion con el numero mayor y menor
  struct pci_dev *pcidev;     // PCI device structure
  u16 base;                   // direccion base de la tarjeta
};

static struct pci_9111_datos datos;
```

Driver PCI-9111DG: instalación

```
static int modulo_instalacion(void) {
  int result;

  // ponemos los puntos de entrada
  pci_9111_fops.open=pci_9111_open;
  pci_9111_fops.release=pci_9111_release;
  pci_9111_fops.write=pci_9111_write;
  pci_9111_fops.read=pci_9111_read;

  // escanea el bus para encontrar el dispositivo
  datos.pcidev= pci_get_device(VENDOR,DEVICE,NULL);

  if (datos.pcidev==0) {
    printk(KERN_WARNING "pci> (init_module) fallo pci_get\n");
    result=-ENODEV;
    goto error_scan_reserva;
  }
```


Driver PCI-9111DG: instalación (cont.)

```
  // reserva dinamica del número mayor del módulo
  // numero menor = cero, numero de dispositivos =1
  result=alloc_chrdev_region(&datos.dev,0,1,"pci_9111");
  if (result < 0) {
    printk(KERN_WARNING "pci> (init_module) fallo con el major %d\n",
           MAJOR(datos.dev));
    goto error_scan_reserva;
  }

  // instalamos driver
  datos.cdev=cdev_alloc();
  cdev_init(datos.cdev, &pci_9111_fops);
  datos.cdev->owner = THIS_MODULE;
  result= cdev_add(datos.cdev, datos.dev, 1);
  if (result!=0) {
   printk(KERN_WARNING "pci> (init_module) Error %d al anadir",result);
    goto error_instalacion_dev;
  }
```


Driver PCI-9111DG: instalación (cont.)

```
// habilita dispositivo
  pci_enable_device(datos.pcidev);

  // inicializa datos del dispositivo
  datos.base = pci_resource_start(datos.pcidev,2);
  printk( KERN_INFO "pci> Region   : %x \n",datos.base);

  // inicializa registros del dispositivo

  // usa el canal 0
  outb(0x00,datos.base+AD_CCR);
  // ganancia 1
  outb(0x00,datos.base+AD_ISRCR);
  // modo de disparo por software y no auto-scan
  outb(0x00,datos.base+AD_TMCR);
  // limpia interrupcion
  outb(0x00,datos.base+AD_HICR);
```

Driver PCI-9111DG: instalación/ errores

```
  // Secuencia de reset FIFO
  outb(0x00,datos.base+AD_ICR);
  outb(0x04,datos.base+AD_ICR);
  outb(0x00,datos.base+AD_ICR);

  // todo correcto: mensaje y salimos
  printk( KERN_INFO "pci> (init_module) OK con major %d\n",
          MAJOR(datos.dev));
  return 0;

  // Errores

 error_instalacion_dev:
  cdev_del(datos.cdev);
  unregister_chrdev_region(datos.dev,1);

 error_scan_reserva:
  return result;
}
```


Driver PCI-9111DG: desinstalación

```
static void modulo_salida(void) {
  cdev_del(datos.cdev);
  unregister_chrdev_region(datos.dev,1);

  printk( KERN_INFO "pci> (cleanup_module) descargado OK\n");
}
```


Driver PCI-9111DG: open/release

```
int pci_9111_open(struct inode *inodep, struct file *filp) {
  int menor= MINOR(inodep->i_rdev);
  printk(KERN_INFO "pci> (open) menor= %d\n",menor);

  return 0;
}

int pci_9111_release(struct inode *inodep, struct file *filp) {
  int menor= MINOR(inodep->i_rdev);
  printk(KERN_INFO "pci> (release) menor= %d\n",menor);

  return 0;
}
```

Driver PCI-9111DG: read

```
ssize_t pci_9111_read (struct file *filp, char *buff,
                       size_t count, loff_t *offp)
{
  unsigned long not_copied;
  u16 dato;

  //printk(KERN_INFO "pci> (read) leido el canal\n");

  // disparo por software y espera
  outb(0x00,datos.base+AD_STR);
  ndelay(8500);

  // lectura y transformacion del dato
  dato=inw(datos.base+AD_DR);
  dato=(dato>>4)&0x0FFF;
```


Driver PCI-9111DG: read (cont.)

```
  // copia del dato al espacio de usuario

  not_copied=copy_to_user(buff,&dato,2);
  if (not_copied!=0) {
   printk(KERN_WARNING "pci> (read) AVISO, no se leyeron los datos\n");
    return(-EFAULT);
  }

  return 2;
}
```


Driver PCI-9111DG: write

```
ssize_t pci_9111_write (struct file *filp, const char *buff,
                        size_t count, loff_t *offp)
{
  unsigned long not_copied;
  u16 dato;

  //printk(KERN_INFO "pci> (write) escrito en D/A\n");

  not_copied=copy_from_user(&dato,buff,2);
  if (not_copied!=0) {
   printk(KERN_WARNING "veloc> (write) AVISO, no se escribio bien\n");
    return(-EFAULT);
  }

  outw(dato,datos.base+DA_OR);
  //udelay(30);
  return 2;
}
```

Driver PCI-9111DG: función auxiliar de muestra de la configuración

```
void muestra_config (void) {

  u8 d;
  u32 dw;
  int i,j;

  for (i=0;i<4;i++) {
    for (j=0;j<16;j++) {
      pci_read_config_byte(datos.pcidev,(i*16)+j,&d);
      printk("%4x",d);
    }
    printk("\n");
  }
```


Driver PCI-9111DG: función auxiliar de muestra de la configuración (cont.)

```
  pci_read_config_dword(datos.pcidev,PCI_BASE_ADDRESS_0,&dw);
  printk("Direccion 0: %x\n",dw);
  pci_read_config_dword(datos.pcidev,PCI_BASE_ADDRESS_1,&dw);
  printk("Direccion 1: %x\n",dw);
  pci_read_config_dword(datos.pcidev,PCI_BASE_ADDRESS_2,&dw);
  printk("Direccion 2: %x\n",dw);
  pci_read_config_dword(datos.pcidev,PCI_BASE_ADDRESS_3,&dw);
  printk("Direccion 3: %x\n",dw);
  pci_read_config_dword(datos.pcidev,PCI_BASE_ADDRESS_4,&dw);
  printk("Direccion 4: %x\n",dw);
  pci_read_config_dword(datos.pcidev,PCI_BASE_ADDRESS_5,&dw);
  printk("Direccion 5: %x\n",dw);
```


Driver PCI-9111DG: función auxiliar de muestra de la configuración (cont.)

```
 printk("Region 0 start: %x \n",pci_resource_start(datos.pcidev,0));
 printk("Region 0 end  : %x \n",pci_resource_end(datos.pcidev,0));
 printk("Region 1 start: %x \n",pci_resource_start(datos.pcidev,1));
 printk("Region 1 end  : %x \n",pci_resource_end(datos.pcidev,1));
 printk("Region 2 start: %x \n",pci_resource_start(datos.pcidev,2));
 printk("Region 2 end  : %x \n",pci_resource_end(datos.pcidev,2));
 printk("Region 3 start: %x \n",pci_resource_start(datos.pcidev,3));
 printk("Region 3 end  : %x \n",pci_resource_end(datos.pcidev,3));
 printk("Region 4 start: %x \n",pci_resource_start(datos.pcidev,4));
 printk("Region 4 end  : %x \n",pci_resource_end(datos.pcidev,4));
 printk("Region 5 start: %x \n",pci_resource_start(datos.pcidev,5));
 printk("Region 5 end  : %x \n",pci_resource_end(datos.pcidev,5));

}
```

Programa de prueba del driver PCI-9111DG

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <stdlib.h>

// Programa principal

int main() {

  short int dato;
  ssize_t leidos, escritos;
  static int fd;
```


Programa de prueba del driver PCI-9111DG (cont.)

```
  // uso del dispositivo

  fd=open("/dev/pci9111", O_RDWR);

  if (fd==-1) {
    perror("Error al abrir el fichero");
    exit(1);
  }
  printf("Fichero abierto\n");

  while (1) {
    leidos=read(fd,&dato,2);
    //conversión del dato de la tensión de entrada a la de salida
    dato=(dato+0x800)%0x1000;
    escritos=write(fd,&dato,2);
  }

  exit(1);
}
```


Instalación en caliente del driver PCI-9111

```
static struct pci_device_id lista_pci_9111[] = {
  {PCI_DEVICE(VENDOR,DEVICE)},
  {0,},
};


int probe (struct pci_dev *dev,const struct pci_device_id *id) {

  printk( KERN_INFO "pci> (probe) OK \n");

  if (pci_enable_device(dev)!=0) {
    printk(KERN_WARNING "pci> (probe) Error en enable!!\n");
    return -EBUSY;
  }

  // inicializa datos del dispositivo
  datos.pcidev=dev;
  datos.base = pci_resource_start(datos.pcidev,2);
  printk( KERN_INFO "pci> Region   : %x \n",datos.base);
```

Instalación en caliente del driver PCI-9111 (cont.)

```
  // inicializa registros del dispositivo

  // uso el canal 0
  outb(0x00,datos.base+AD_CCR);
  // ganancia 1
  outb(0x00,datos.base+AD_ISRCR);
  // modo de disparo por software y no auto-scan
  outb(0x00,datos.base+AD_TMCR);
  // limpia interrupcion
  outb(0x00,datos.base+AD_HICR);
  // Secuencia de reset FIFO
  outb(0x00,datos.base+AD_ICR);
  outb(0x04,datos.base+AD_ICR);
  outb(0x00,datos.base+AD_ICR);

  return 0;
}
```


Instalación en caliente del driver PCI-9111 (cont.)

```
void remove (struct pci_dev *dev) {
  printk( KERN_INFO "pci> (remove) OK \n");
}


static struct pci_driver pci_9111_driver = {
  .name="pci_9111",
  .id_table= lista_pci_9111,
  .probe=&probe,
  .remove=&remove,
};
```


Instalación en caliente del driver PCI-9111: registro del driver

En la instalación del módulo se debe declarar el driver en la estructura de gestión del PCI de Linux:

- **En la operación de instalación del módulo:**

```
  // registro de PCI
  result=pci_register_driver(&pci_9111_driver);
  if (result < 0) {
    printk(KERN_WARNING "pci> (init_module) fallo registro PCI\n");
    goto error_registro_pci;
  }
```

- **En la operación de desinstalación del módulo:**

```
  // registro de PCI
  pci_unregister_driver(&pci_9111_driver);
```

Bibliografía

[1] H.P. Messmer, "The Indispensable PC Hardware Book", 4th Ed., Addison-Wesley, 2002

[2] Jonathan Corbet, Greg Kroah-Hartman, Alessandro Rubini, "Linux Device Drivers", 3rd Ed., O'Reilly, 2005.

[3] P. J. Salzman, M. Burian, O. Pomerantz, "The Linux Kernel Module Programming Guide", Ver. 2.6.4, 18-5-2007: http://tldp.org/LDP/lkmpg/2.6/html/

[4] Scott Mueller, "Upgrading and Repairing PCs", 17th Ed., QUE, 2006

[5] User´s Guide PCI-9111DG/HR. Adlink Technologies Inc.