Periféricos Interfaces y Buses

UNIVERSIDAD DE CANTABRIA

I. Arquitectura de E/S

II. Programación de E/S

III. **Interfaces de E/S de datos**

Interfaces de comunicaciones serie (RS-232, USB, Firewire). Interfaz paralelo. Modelo de programación de dispositivos con las interfaces descritas.

IV. Dispositivos de E/S de datos

V. Buses

VI. Controladores e interfaces de dispositivos de almacenamiento

VII. Sistemas de almacenamiento

Interfaces de E/S de datos

UNIVERSIDAD DE CANTABRIA

Bloque II

- **Bus USB**
- **Programación del bus USB**
- **Firewire**

Introducción al bus USB

El USB (*Universal Serial Bus*) es una red para la conexión de periféricos

- no tiene una topología de bus al uso con arbitrio y estructura de maestro/esclavo, o con resolución de colisiones en un bus al que cualquiera puede acceder en cualquier instante (CSMA)
- se parece más a una topología de paso de testigo en estrella
  - tiene una arquitectura en árbol en el que las hojas representan a los periféricos
  - necesita usar *hubs* para la conexión de los periféricos
  - sólamente hay un *host* por bus para controlarlo
  - el *host* es responsable de realizar todas las transacciones y planificar el ancho de banda
  - esta topología permite desconectar a un dispositivo si falla sin afectar a los demás

Introducción al bus USB (cont.)

Algunas características del bus USB:

- el USB 1.1 dispone de dos especificaciones de la interfaz del controlador
  - UHCI - *Universal Host Controller Interface* desarrollada por Intel
  - OHCI - *Open Host Controller Interface* desarrollada por Compaq, Microsoft y National Semiconductors
- la especificación del USB 2.0 es única y ha sido desarrollada por Intel, Compaq, NEC, Lucent y Microsoft
  - EHCI - *Enhanced Host Controller Interface*
- velocidades de comunicación
  - 480 Mbps (*High Speed*) versión 2.0
  - 12 Mbps (*Full Speed*) versión 1.1
  - 1.5 Mbps (*Low Speed*) versión 1.1

Introducción al bus USB (cont.)

- admite hasta 127 dispositivos por controlador
  - si se necesitan más se pueden añadir más controladores
- longitudes de cable de hasta 5 metros
- usa 4 cables apantallados
  - alimentación (5 V) y tierra
  - un par trenzado por el que va la señal diferencial con los datos
- usa una codificación NRZI (*Non Return to Zero Invert*) en el envío, con un campo que sincroniza el *host* con los relojes de recepción
  - en NRZI un 1 se identifica por el mantenimiento de la señal y un 0 por el cambio de nivel
  - una secuencia de ceros produce un cambio de nivel en cada tiempo de bit y una secuencia de unos un nivel constante

Introducción al bus USB (cont.)

- soporta la carga y descarga dinámica de drivers
  - el *host* detecta la presencia del dispositivo
  - la carga del driver apropiado se hace mediante la identificación de los parámetros PID/VID (*Product* ID/*Vendor* ID)
- dispone de varios modos de transferencia incluyendo uno con ancho de banda garantizado
- dispone de varios tipos de conectores, para los tipos A y B

Pin	Cable	Función
1	Rojo	V_{BUS} (5 V)
2	Blanco	D-
3	Verde	D+
4	Negro	GND

Introducción al bus USB (cont.)

Características eléctricas para *Low Speed* y *Full Speed*

- en transmisión
  - un 1 diferencial se consigue con D+ por encima de 2.8 V y D- por debajo de 0.3 V
  - un 0 diferencial se consigue con D- por encima de 2.8 V y D+ por debajo de 0.3 V
- en recepción
  - un 1 diferencial es D+ 200 mV mayor que D-
  - un 0 diferencial es D+ 200 mV menor que D-
- un dispositivo USB puede especificar el consumo de potencia en unidades de 2 mA

Introducción al bus USB (cont.)

Suspensión

- un dispositivo entra en modo de suspensión si no tiene actividad en el bus por un tiempo de 3 ms
- la corriente que consume en este caso está limitada a 500 μA por unidad de consumo

Protocolo USB

A diferencia de interfaces como la RS-232 que sólo envía bytes, el USB dispone de protocolos con varios niveles

Todas las transacciones de datos las inicia el *host*

La mayoría de las transacciones USB constan de

- un paquete de testigo
  - enviado por el *host* para describir si se trata de una lectura o una escritura y cuál es el dispositivo con el que se va a comunicar
- un paquete de datos opcional
- paquete de estado
  - para reconocer la transacción (si los datos o el testigo se han recibido correctamente, o reportar el error)

Protocolo USB (cont.)

El modelo de transferencia de datos entre el *host* y un *endpoint* (dispositivo) se denomina *pipe* (tubería), y puede ser de dos tipos:

- *stream*: los datos no tienen una estructura definida
- mensaje: los datos tienen una estructura definida

A las tuberías en tiempo de configuración del dispositivo se les asocia:

- el ancho de banda
- el tipo de servicio
- las características del *endpoint*
  - direccionalidad
  - tamaño de los buffers

Protocolo USB (cont.)

Otras características del protocolo:

- Robustez incorporando mecanismos de detección y corrección de errores
- El bus USB soporta que un dispositivo se conecte o desconecte en cualquier instante
  - el software debe soportar estos cambios en caliente
- El bus asigna una dirección única a los dispositivos conectados (la asignación de direcciones es una actividad dinámica)

Protocolo USB (cont.)

Tipos de datos del protocolo:

- *Control*: usados para configurar un dispositivo en el momento en el que se conecta y para otras actividades específicas del dispositivo
- *Bulk*: usado para datos generados o consumidos relativamente grandes y a ráfagas
- *Interrupt*: para transmisión temporizada y fiable de datos
- *Isochronous*: utiliza un ancho de banda garantizado previamente negociado (también llamado *streaming real time transfers*)

Dispositivos USB

Los dispositivos se dividen en clases como *hubs*, interfaces humanas, impresoras, dispositivos de imagen, o dispositivos de almacenamiento

Sin embargo, hay una división fundamental en dos clases:

- *hubs*
  - permiten la conexión de otros dispositivos
- *functions*
  - dispositivo capaz de recibir o transmitir información de datos o de control
  - aquí están todos los dispositivos

Dispositivos USB (cont.)

Todos los dispositivos tienen una dirección en el bus, y puede soportar una o más tuberías con las que se comunican con el *host*

En particular todo dispositivo debe soportar una tubería en el *endpoint* 0 a través de la cual se puede controlar y que lleva los siguientes tipos de información asociados:

- *Standard*: información común a todos los dispositivos
  - identificador del fabricante, clase de dispositivo, capacidad de control de la energía
  - descripciones de dispositivo, configuración, interfaz USB y *endpoint*
- *Class*: información dependiente de la clase
- *USB Vendor*: información dependiente del fabricante

Interfaces de E/S de datos

Bloque II

- Bus USB
- Programación del bus USB
- Firewire

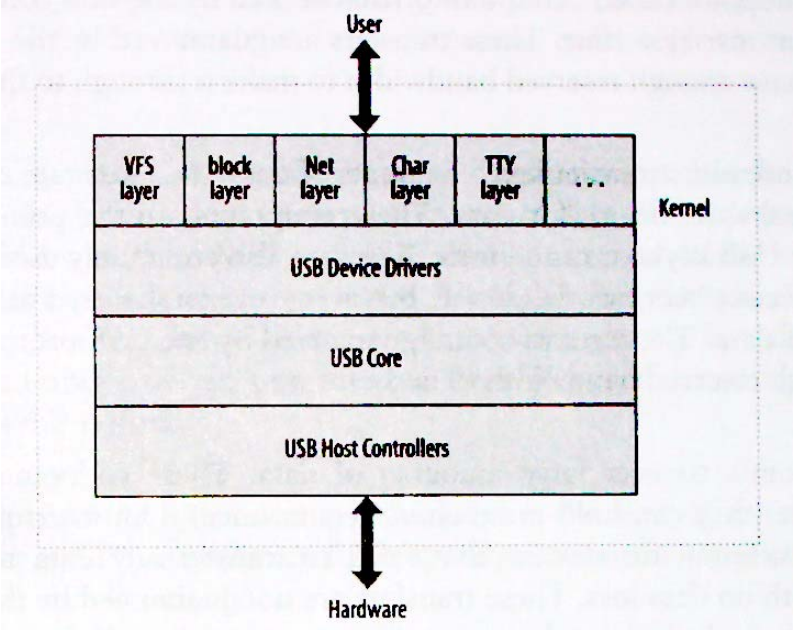
Drivers USB

El núcleo de Linux soporta dos tipos de drivers USB:

- drivers de *host*
  - control desde el punto de vista del *host* (computador)
- drivers de dispositivo
  - control desde el punto de vista del dispositivo
  - para algunos sistemas empotrados
  - para distinguirlos se llaman también *USB gadget drivers*

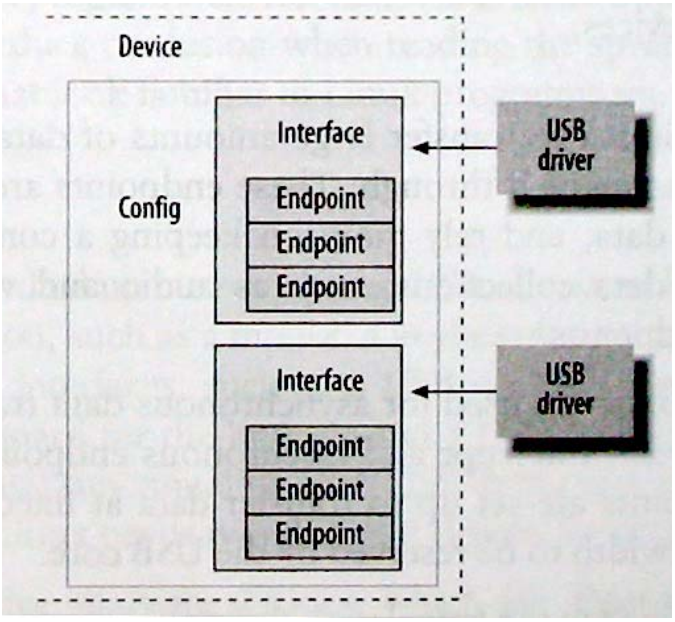
Linux soporta los drivers del primer grupo (*host*) mediante un subsistema llamado *USB Core* que maneja la mayor parte de la complejidad asociada al bus USB

Estructura de drivers USB



[2]

Estructura de dispositivos USB



[2]

Endpoints

Soporta la comunicación más básica en el bus USB:

- lleva datos en una sola dirección
  - *out endpoint*: del computador al dispositivo
  - *in endpoint*: del dispositivo al computador
- se puede pensar como una tubería unidireccional
- puede ser de uno de tipos de datos USB
  - *Control*
  - *Interrupt*
  - *Bulk*
  - *Isochronous*

Endpoints (cont.)

Normalmente *control* y *bulk* se usan para transferencias de datos asíncronas

En cambio *interrupt* e *isochronous* se usan para transferencias periódicas en las que puede haber una reserva del ancho de banda

El núcleo describe los *endpoints* con el tipo de dato `struct usb_host_endpoint`:

- esta estructura contiene el tipo de dato `struct usb_endpoint_descriptor` en el que se encuentra la información en el formato especificado para el dispositivo

Endpoints (cont.)

Los campos más representativos de esta última estructura (`struct usb_endpoint_descriptor`) son:

- **`bEndpointaddres`**
  - dirección USB del *endpoint*
  - se pueden aplicar las máscaras `USB_DIR_OUT` y `USB_DIR_IN` para determinar la dirección del *endpoint*
- **`bmAttributes`**
  - determina el tipo de endpoint respecto de los datos que maneja
  - se puede aplicar la máscara `USB_ENDPOINT_XFERTYPE_MASK` para saber el tipo
    - `USB_ENDPOINT_XFER_ISOC`,
    - `USB_ENDPOINT_XFER_BULK`, o
    - `USB_ENDPOINT_XFER_INT`

Endpoints (cont.)

- **`wMaxPacketSize`**
  - tamaño máximo en bytes que el *endpoint* puede manejar de una vez
  - si el dato que se envía al driver es mayor se divide en paquetes de este tamaño para su transmisión al dispositivo
- **`bInterval`**
  - para un *endpoint* del tipo *interrupt* representa el intervalo entre interrupciones en milisegundos

Interfaces USB

Los *endpoints* se agrupan en interfaces USB:

- manejan sólo un tipo de conexión lógica USB
  - ratón, teclado, audio, etc.
- algunos dispositivos pueden constar de varias interfaces USB
  - p.e., un altavoz puede tener dos interfaces USB: una de teclado para los botones, y otra de audio para el sonido
- representa una funcionalidad básica, de manera que cada driver controla una sola interfaz USB
  - para el ejemplo anterior Linux necesita dos drivers diferentes para un único dispositivo hardware

Interfaces USB (cont.)

Las interfaces USB pueden tener diferentes estados que pueden corresponder a elecciones concretas de algunos parámetros:

- el estado inicial se numera como 0
- otras alternativas se pueden usar para controlar determinados *endpoints* de modos diferentes (asignando anchos de banda para cada caso específico)

Las interfaces USB se describen en la estructura `struct usb_interface`, que es la que el núcleo USB pasa al driver y a partir de ahí el driver se encarga de controlar

Interfaces USB (cont.)

Los campos más representativos de esta estructura son:

- `struct usb_host_interface *altsetting`
  - array que contiene todos los posibles estados de la interfaz USB
  - cada `struct usb_host_interface` contiene un conjunto de configuraciones de *endpoints* como las definidas en `struct usb_host_endpoint`
- `unsigned num_altsettings`
  - número de estados en el array anterior
- `struct usb_host_interface *cur_altsetting`
  - puntero al array altsetting con el estado actual
- `int minor`
  - si el driver asociado a esta interfaz USB usa un número mayor, éste es el número menor asociado por el núcleo USB a la interfaz

Configuraciones

Las interfaces USB se agrupan en configuraciones

- un dispositivo USB puede tener varias configuraciones y conmutar entre ellas, aunque sólo una está activa en un instante dado
- las configuraciones se describen con la `struct usb_host_config`
- el dispositivo completo se encuentra descrito por la `struct usb_device`
- los drivers generalmente no necesitan usar estas estructuras directamente; se encuantran en `<linux/usb.h>`
- para convertir datos de `struct usb_interface` a `struct usb_device` se usa la función `interface_to_usbdev`

Resumen de las unidades lógicas USB

Los dispositivos tienen una o más configuraciones

Las configuraciones suelen tener una o más interfaces USB

Las interfaces USB tienen uno o más estados

Las interfaces USB tienen cero o más *endpoints*

Representación USB en el sistema de ficheros

El controlador USB aparece normalmente como parte del sistema PCI en su árbol de directorios

Dentro de éste los nombres de los ficheros pueden tener la estructura:

```
root_hub-hub_port:config.interface
```

o bien si hay más hubs

```
root_hub-hub_port-hub_port:config.interface
```

La información de los dispositivos USB también aparece en `/proc/bus/usb` y en el fichero `/proc/bus/usb/devices`

URBs del USB

El código USB en el núcleo de Linux se comunica con los dispositivos utilizando unas estructuras de datos que se llaman URBs (*USB Request Blocks*); en `<linux/usb.h>`

Un URB se usa para enviar o recibir datos de un *endpoint* perteneciente a un dispositivo específico

Un driver puede gestionar tantos URBs como quiera en una cola para un solo *endpoint*, o reutilizarlos para diferentes *endpoints*

El driver puede cancelar URBs en cualquier instante, al igual que el núcleo USB en el caso de que se haya eliminado un dispositivo

Los URBs se crean dinámicamente y llevan un contador interno que permite liberarlos cuando el último usuario los cierra

URBs del USB (cont.)

El ciclo de vida de un URB suele ser:

- lo crea el driver USB
- se asigna a un *endpoint* específico de un dispositivo
- el driver lo remite al núcleo USB
- el núcleo USB lo remite al controlador USB
- el controlador USB lo procesa y realiza la transferencia
- cuando se completa el URB, el controlador USB notifica al driver

URBs del USB (cont.)

Los campos de la `struct urb` importantes para el driver son los siguientes:

- **`struct usb_device *dev`**
  - puntero al dispositivo al que se va a enviar el URB
  - la variable la debe inicializar el driver
- **`unsigned int pipe`**
  - *endpoint* específico del dispositivo al que se quiere enviar el URB
  - también la debe inicializar el driver
  - hay una serie de funciones que permiten establecer los valores del *endpoint* dependiendo del tipo y dirección del tráfico
- **`unsigned int transfer_flags`**
  - valores que programan la transferencia en el URB

URBs del USB (cont.)

- **`void *transfer_buffer`**
  - puntero al buffer de transmisión o recepción
  - debe ser creado con `kmalloc`
- **`dma_addr_t transfer_dma`**
  - buffer si se usa transferencia DMA
- **`int_transfer_buffer_length`**
  - longitud del buffer que se use (uno de los dos anteriores)
  - puede ser 0
- **`unsigned char *setup_packet`**
  - puntero al paquete de inicialización para un URB de control
- **`dma_addr_t setup_dma`**
  - buffer del paquete de inicialización; sólo URB de control

URBs del USB (cont.)

- `usb_complete_t complete`
  - puntero a la función manejadora de terminación

```
typedef void (*usb_complete_t)
             (struct urb *, struct pt_regs *)
```

  - la llama el núcleo USB cuando el URB ha sido transferido o ha ocurrido un error
- `void *context`
  - puntero a datos que retorna la función manejadora de terminación
- `int actual_length`
  - cuando el URB acaba indica los datos realmente transferidos
- `int status`
  - estado del URB cuando acaba
  - un cero indica que ha ido bien, otro valor es un código de error

URBs del USB (cont.)

- `int start_frame`
  - número inicial de marcos en transferencia *isochronous*
- `int interval`
  - intervalo al que se chequea el URB
  - válido para transferencias *interrupt* e *isochronous*
- `int number_of_packets`
  - válido para transferencias *isochronous*
  - especifica el número de buffers que deben ser transferidos
- `int error_count`
  - válido para transferencias *isochronous*
  - especifica el número de transferencias que han tenido algún error
- `struct usb_iso_packet_descriptor iso_frame_desc[0]`
  - define de una vez varias transferencias *isochronous*

Funciones de manejo de endpoints

```
unsigned int usb_sndctrlpipe
     (struct usb_device *dev,unsigned int endpoint)
unsigned int usb_rcvctrlpipe
     (struct usb_device *dev,unsigned int endpoint)
unsigned int usb_sndbulkpipe
     (struct usb_device *dev,unsigned int endpoint)
unsigned int usb_rcvbulkpipe
     (struct usb_device *dev,unsigned int endpoint)
unsigned int usb_sndintpipe
     (struct usb_device *dev,unsigned int endpoint)
unsigned int usb_rcvintpipe
     (struct usb_device *dev,unsigned int endpoint)
unsigned int usb_sndisocpipe
     (struct usb_device *dev,unsigned int endpoint)
unsigned int usb_rcvisocpipe
     (struct usb_device *dev,unsigned int endpoint)
```


Valores del campo `transfer_flags`

```
URB_SHORT_NOT_OK
URB_ISO_ASAP
URB_NO_TRANSFER_DMA_MAP
URB_NO_SETUP_DMA_MAP
URB_ASYNC_UNLINK
URB_NO_FSBR
URB_ZERO_PACKET
URB_NO_INTERRUPT
```

Creación y destrucción de URBs

Para la creación de URBs se utiliza la siguiente función

```
struct urb *usb_alloc_urb
            (int iso_packets, int mem_flags);
```

- `iso_packets`: número de paquetes isochronous que el URB debe contener; si no se usan debe ser 0
- `mem_flags`: los mismos que `kmalloc`
- devuelve un puntero al URB o un valor NULL si hay error

La liberación de URBs se hace con la siguiente función

```
void usb_free_urb (struct *urb);
```

Existen funciones para inicializar los URBs

- no hay función para los *isochronous* y se debe hacer a mano

Inicializa URBs *interrupt*

```
void usb_fill_int_urb
     (struct urb *urb, struct usb_device *dev,
      unsigned int pipe, void *transfer_buffer,
      int buffer_length, usb_complete_t complete,
      void *context, int interval);
```

- `*urb`: puntero al URB
- `*dev`: dispositivo al que se quiere enviar el URB
- `pipe`: *endpoint* del dispositivo al que se quiere enviar el URB; este valor se crea con `usb_sndintpipe` o `usb_rcvintpipe`
- `*transfer_buffer`: puntero al buffer de transferencia de datos
- `buffer_length`: longitud del buffer anterior
- `complete`: manejador de terminación del URB
- `*context`: puntero a los datos que retorna la función anterior
- `interval`: intervalo al que el URB se debe planificar

Inicializa URBs *bulk*

```
void usb_fill_bulk_urb
     (struct urb *urb, struct usb_device *dev,
      unsigned int pipe, void *transfer_buffer,
      int buffer_length, usb_complete_t complete,
      void *context);
```

- los parámetros son como el anterior pero sin el intervalo
- el parámetro `pipe` se ha debido de inicializar con las funciones `usb_sndbulkpipe` o `usb_rcvbulkpipe`
- esta función no establece el campo `transfer_flags` por lo que si se quiere cambiar lo debe hacer el mismo driver

Inicializa URBs *control*

```
void usb_fill_control_urb
     (struct urb *urb, struct usb_device *dev,
      unsigned int pipe, unsigned char *setup_packet,
      void *transfer_buffer, int buffer_length,
      usb_complete_t complete, void *context);
```

- `*setup_packet`: apunta al dato a ser transmitido al *endpoint*
- el parámetro `pipe` se ha debido de inicializar con las funciones `usb_sndctrlpipe` o `usb_rcvctrlpipe`
- esta función no establece el campo `transfer_flags` por lo que si se quiere cambiar lo debe hacer el mismo driver

Finalización de URBs

Hay tres motivos por los que un URB finaliza y llama a la función de finalización:

- el URB se ha transferido adecuadamente y devuelve el correspondiente reconocimiento (la variable `status` se pone a cero)
- ha ocurrido algún tipo de error que se refleja en la variable `status`
- el núcleo USB ha desenlazado el URB
  - ocurre si el driver cancela un URB remitido
  - o cuando el dispositivo se ha retirado del sistema y se envía un URB

Cancelación de URBs

La cancelación de un URB se realiza mediante la llamada a las funciones

```
int usb_kill_urb (struct urb *urb);
```

- esta función se usa cuando el dispositivo se desconecta del sistema

```
int usb_unlink_urb (struct urb *urb);
```

- se usa para decirle al núcleo USB que detenga un URB
- la función no espera a que el URB sea cancelado
- es útil cuando se usa desde un manejador de interrupción o con un spinlock tomado
- el uso de esta función requiere que se haya establecido el bit `URB_ASYNC_UNLINK`

Identificación de un dispositivo USB

Como para los drivers PCI el driver debe registrar un objeto que identifica el fabricante y el dispositivo

La estructura `struct usb_device_id` proporciona la lista de dispositivos que el driver soporta

- esta lista la usa el núcleo USB para decidir el driver que usa para un dispositivo cuado se arranca en caliente

Los campos de esta estructura son:

- `__u16 match_flags`
  - determina cuales de los campos de identificación son válidos
  - normalmente no se establece directamente, sino que lo hacen las macros que se usan para los siguientes campos
  - los valores están en `<linux/mod_device_table.h>`

Identificación de un dispositivo USB (cont.)

- `__16 idVendor`
  - identificador del fabricante, asignado por el forum USB
- `__16 idProduct`
  - identificador del producto, elegido por el fabricante
- `__16 bcdDevice_lo`
- `__16 bcdDevice_hi`
  - define en BCD la versión del dispositivo
- `__8 bDeviceClass`
- `__8 bDeviceSubClass`
- `__8 bDeviceProtocol`
  - definen la clase, sub-clase y protocolo del dispositivo
  - asignados por el forum USB

Identificación de un dispositivo USB (cont.)

- `__8 bInterfaceClass`
- `__8 bInterfaceSubClass`
- `__8 bInterfaceProtocol`
  - definen la clase, sub-clase y protocolo de una interfaz en particular
  - asignados por el forum USB
- `kerner_ulong_t driver_info`
  - sirve para diferenciar dispositivos en la función de *callback* `probe`

Identificación de un dispositivo USB (cont.)

Como para el PCI también existen una serie de macros para inicializar la estructura:

```
USB_DEVICE (vendor, product)
USB_DEVICE_VER (vendor, product, lo, hi)
USB_DEVICE_INFO (class, subclass, protocol)
USB_INTERFACE_INFO (class, subclass, protocol)
```

Un ejemplo de la tabla de identificadores puede ser el siguiente:

```
static struct usb_device_id usb_table[] = {
  {USB_DEVICE(USB_VENSOR_ID,USB_PRODUCT_ID)},
  {}
};
MODULE_DEVICE_TABLE(usb,usb_table);
```

La macro `MODULE_DEVICE_TABLE` permite el acceso a la tabla desde el espacio de usuario

Registro de un dispositivo USB

La estructura principal de un driver USB es `struct usb_driver` que tiene los siguientes campos, algunos de los cuales son funciones de *callback*:

```
struct module *owner
const char *name
const struct usb_device_id *id_table
int (*probe) (struct usb_interface *intf,
              const struct usb_device_id *id)
int (*disconnect) (struct usb_interface *intf)
int (*ioctl) (struct usb_interface *intf,
              unsigned long code, void *buf)
int (*suspend) (struct usb_interface *intf, u32 state)
int (*resume) (struct usb_interface *intf)
```


Registro de un dispositivo USB (cont.)

El registro de un dispositivo se hace mediante la llamada a la función `usb_register_driver()`, a la que se le pasa un puntero a la `struct usb_driver`

```
...
int result;
...
result = usb_register_driver(&usb_driver);
if (result)  //comunica error
...
return result;
```

La descarga del driver se realiza con la función

```
usb_deregister(&usb_driver);
```

Función `probe`

La llama el núcleo USB cuando instala el dispositivo y debe chequear la información que se le pasa para decidir si es el driver adecuado para el dispositivo

Esta función se puede dormir pero

- se ejecuta en el contexto de un thread del kernel que es único para todas estas funciones
- se recomienda hacerla lo más pequeña posible, delegando su actividad a la operación `open`
- un dispositivo lento podría producir bloqueos en la detección

Función `probe` (cont.)

Las actividades que se deben hacer en esta función son las siguientes:

- inicializar cualquier estructura de datos local usada para manejar el dispositivo
- salvar cualquier información que vaya a necesitar en estas estructuras locales
  - p.e., detectar las direcciones de los *endpoints* y los tamaños de los buffers que utiliza el dispositivo

El siguiente segmento de código detecta *endpoints in* y *out* de un tipo *bulk* y salva la información en una estructura de datos local

Función probe (cont.)

```
// detecta el primer bulk in y el primer bulkout
...
iface_desc=interface->cur_altsetting;
for (i=0;i<iface_desc->desc.bNum_Endpoints;i++) {
  endpoint=&iface_desc[endpoint]i.desc;
  // bulk in
  if (!dev->bulk_in_endpointAddr &&
      (endpoint->bEndpoint_Address & USB_DIR_IN) &&
      ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)
      == USB_ENDPOINT_XFER_BULK)) {
    buffer_size=endpoint->wMaxPacketSize;
    dev->bulk_in_size=buffer_size;
    dev->bulk_in_endpointAddr=endpoint->bEndpoint_Address;
    dev->bulk_in_buffer=kmalloc(buffer_size,GFP_KERNEL);
    if (!dev->bulk_in_buffer) {
      // error
      goto error;
    }
  }
```


Función probe (cont.)

```
  // bulk out endpoint
  if (!dev->bulk_out_endpointAddr &&
      (endpoint->bEndpoint_Address & USB_DIR_OUT) &&
      ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)
      == USB_ENDPOINT_XFER_BULK)) {
    dev->bulk_out_endpointAddr=endpoint->bEndpoint_Address;
    }
  }
  if (!(dev->bulk_in_endpointAddr && dev->bulk_out_endpointAddr)) {
    // error
    goto error;
  }
}
...
```

Función probe (cont.)

Esta función también debe a

`usb_set_intfdata(interface,dev);`

- salva el puntero a los datos locales en la interfaz
- acepta un puntero a cualquier tipo de dato y lo guarda en la `struct usb_interface`

Para recuperar los datos se usa `usb_get_intfdata()`

- normalmente en las operaciones `open` y `disconnect`

Esta referencia indirecta a la información del dispositivo permite que un driver soporte un número cualquiera de dispositivos

Un ejemplo de recuperación de datos se muestra en página siguiente

Función probe (cont.)

```
struct usb_data *dev;
struct usb_interface *interface;
int subminor;
int result;

subminor=iminor(inode);

intrerface=usb_find_interface (&usb_driver,subminor);
if (!interface) {  // error
  result=-ENODEV;
  goto error;
}

dev=usb_get_intfdata(interface);
if (!dev) {   // error
  result=-ENODEV;
  goto error;
}
```

Función probe (cont.)

Si el driver USB no está asociado a ningún tipo de subsistema que controle la interacción del usuario con el dispositivo (tty, vídeo, etc.)

- el driver puede usar un número mayor en el modo tradicional de un dispositivo de caracteres
- para ello utiliza la función `usb_register_dev()` en la función `probe`

```
result=usb_register_dev(interface,&usb_class);
if (result) {
  // error
  usb_set_intfdata(interface,NULL);
  goto error;
}
```


Función probe (cont.)

- la función requiere dos punteros uno a una `struct usb_interface` y otro a una `struct usb_class_driver`
- ésta última `struct usb_class_driver` permite pasar parámetros al núcleo USB asociados a un número menor
  - `char *name`
    nombre del dispositivo
  - `struct file_operations *fops`
    puntero a las operaciones del driver
  - `mode_t mode`
    acceso de lectura o escritura modo del dispositivo
  - `int minor_base`
    número menor de comienzo para el rango de este driver (hasta 16 dispositivos)

Función `disconnect`

La llama el núcleo USB cuando el driver ya no debe controlar más el dispositivo

Debe liberar los recursos que se hayan tomado

- en particular si se ha llamado a `usb_register_dev()`, se debe llamar a `usb_deregister_dev()`, ej:

```
static void usb_disconnect(struct usb_interface *interface)
{
  struct usb_data *dev;
  int minor=interface->minor;
  lock_kernel(); // previene la llamada al open
  dev=usb_get_intfdata(interface);
  usb_set_intfdata(interface,NULL);
  usb_deregister_dev(interface,&usb_class);
  unlock_kernel();
}
```


Envío y control de URBs

Cuando un driver tiene que enviar datos al dispositivo (función `write`) debe tener alojado un URB, ej:

```
urb=usb_alloc_urb(0,GFP_KERNEL);
if (!urb) {
  result=-ENOMEM;
  goto error;
}
```

Después se puede por ejemplo crear un buffer DMA:

```
buf=usb_buffer_alloc(dev->udev,count,GFP_KERNEL,&urb->treansfer_dma);
if (!urb) {
  result=-ENOMEM;
  goto error;
}
```

Envío y control de URBs (cont.)

A continuación se copian los datos a este buffer:

```
if (copy_from_user(buf,user_buf,count)) {
  result=-ENOMEM;
  goto error;
}
```

Después se inicializa el URB y se remite al núcleo USB:

```
usb_fill_bulk_urb(urb, dev->udev,
                  usb_sndbulkpipe(dev->udev,dev->bulk_out_endpointAddr),
                    buf,count,usb_write_callback,dev);
urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;

result=usb_submit_urb(urb,GFP_KERNEL);
if (result) {    //error
  goto error;
}
```


Envío y control de URBs (cont.)

El núcleo USB llama a la función de *callback* cuando ha acabado o ha ocurrido un error:

```
static void usb_write_bulk_callback
                (struct urb *urb, struct pt_regs *regs)
{
  if (urb->status)  {
    // algo ha fallado
  }

  usb_buffer_free (urb->dev, urb->transfer_buffer_length,
                   urb->transfer_buffer, urb_transfer_dma);
}
```

Transferencias sin URBs

También hay funciones que permiten transferir datos USB sin crear un URB:

- *bulk*

```
int usb_bulk_msg
    (struct usb_device *dev, unsigned int pipe,
     void *data, int len, int *actual_length,
     int timeout);
```

- *control*

```
int usb_control_msg
    (struct usb_device *dev, unsigned int pipe,
     __u8 request, __u8 requesttype,
     __16 value, __16 index,
     void *data, __16 size, int timeout);
```


Interfaces de E/S de datos

Bloque II

- Bus USB
- Programación del bus USB
- Firewire

Introducción al FireWire

Estándar IEEE 1394 para un bus serie de altas prestaciones

- conocido como FireWire y desarrollado originariamente por Apple y Texas Instruments a mediados de los 90
- también conocido como i.Link desarrollado por Sony

Características principales:

- flexibilidad de la conexión
- proporciona una interfaz única de E/S con un conector sencillo que puede manejar varios dispositivos a través de un único puerto
- capacidad de conectar un máximo de 63 nodos con dispositivos u otros computadores

Introducción al FireWire (cont.)

- elevada velocidad de transferencia de información; dos versiones
  - FireWire 400 (IEEE 1394a): ancho de banda de 400 Mbits/s, similar al USB 2.0 (480 Mbits/s)
  - FireWire 800 (IEEE 1394b) o FireWire 2: duplica la velocidad del FireWire 400
  - versión en desarrollo de 3.2 Gbits/s, comparable al USB 3.0 de 4.8 Gbits/s
- su velocidad y sencillez hace que sea una interfaz muy utilizada para audio y vídeo digital
  - se usa mucho en cámaras de vídeo, discos duros, impresoras, reproductores de vídeo digital, sistemas domésticos para el ocio, sintetizadores de música y escáneres

Configuraciones de FireWire

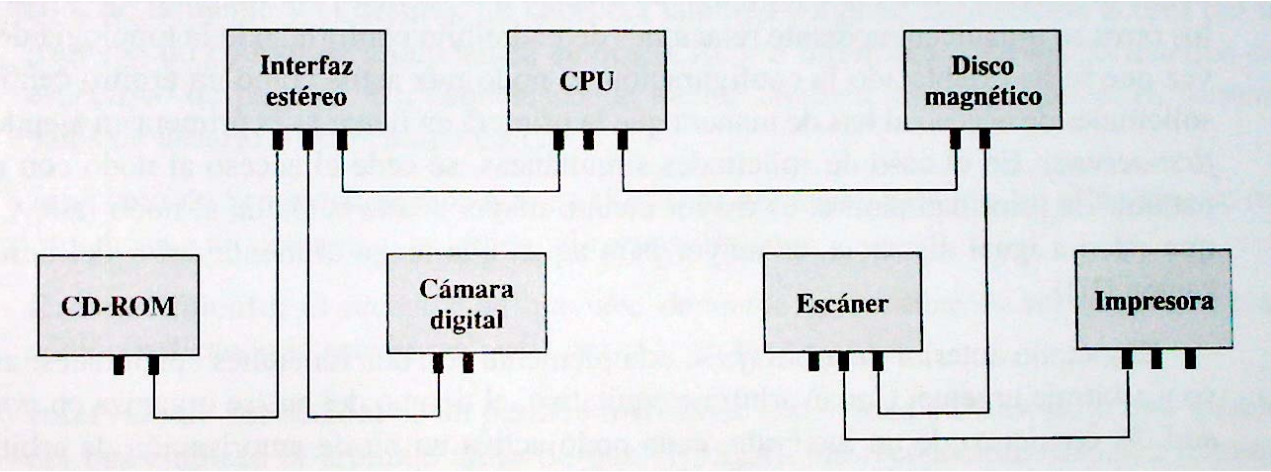
Utiliza una configuración de conexión en cadena lineal (*daisy chain*) que permite conectar hasta 63 nodos (hasta 16 dispositivos por nodo)

Se pueden conectar hasta 1023 buses FireWire mediante adaptadores (*bridges*)

Permite conectar y desconectar dispositivos en caliente, y la configuración automática

- no hace falta asignar manualmente los identificadores de los dispositivos ni tener en cuenta su posición relativa
- no hay terminadores y el sistema realiza automáticamente la configuración para asignar direcciones

Configuraciones de FireWire (cont.)



[6]

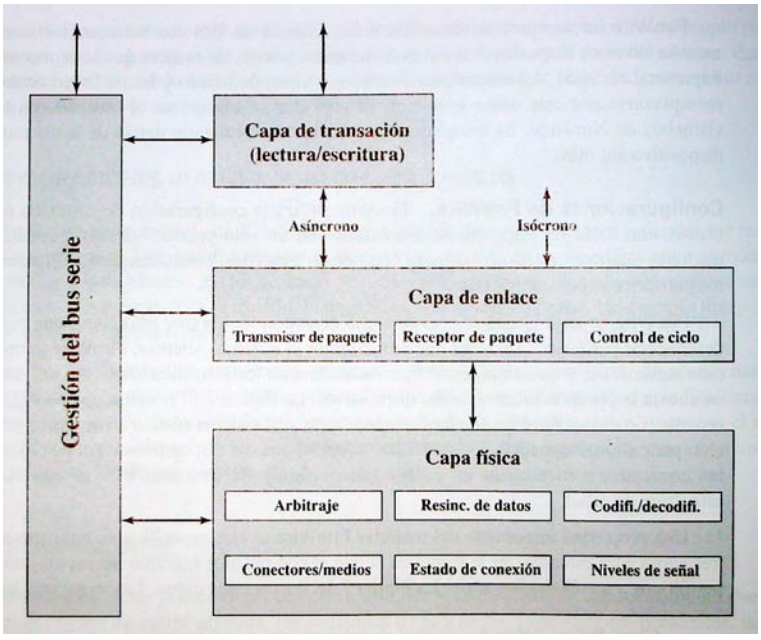
Configuraciones de FireWire (cont.)

La conexión de los dispositivos no es estríctamente en cadena sino más bien en forma de árbol

El estándar especifica un conjunto de protocolos en tres capas

- capa física
  - define los medios de transmisión permitidos y las señales y características eléctricas
- capa de enlace
  - describe la transmisión de datos en los paquetes
- capa de transacción
  - define un protocolo de petición y respuesta que oculta a las aplicaciones las capas inferiores

Configuraciones de FireWire (cont.)



[6]

Capa física

Se definen varios medios de transmisión y sus conectores, así como las velocidades a las que se puede operar desde 25 hasta 400 Mbits/s

Esta capa proporciona también el arbitrio que garantiza que hay un solo dispositivo transmitiendo datos en cada momento

Dos mecanismos de arbitrio:

- la forma más simple se basa en una estructura en árbol
- el otro mecanismo es la estructura lineal en cadena

Arbitraje en árbol

Se incluye la lógica para que un nudo sea designado como raíz del árbol y los demás se organicen en la estructura padre/hijo hasta completar la topología del árbol

El nodo raíz es el árbitro y procesa las peticiones en orden FIFO

Para peticiones simultáneas

- se atiende a la mayor prioridad, dada por la cercanía a la raíz
- para igual distancia a la raíz el que tenga menor número de identificación

Este método de arbitraje se complementa con modos adicionales:

- arbitraje equitativo y arbitraje urgente

Arbitraje en árbol

Arbitraje equitativo

- el tiempo del bus se organiza en intervalos de equidad
- al comienzo de un intervalo cada nodo activa un bit de autorización de arbitraje
- durante el intervalo cada nodo puede competir por el acceso al bus
- cuando un nudo gana el acceso deactiva su bit de autorización y no compite más en el intervalo actual
- se evita que pocos dispositivos de alta prioridad monopolicen el uso del bus

Arbitraje urgente

- algunos dispositivos se pueden configurar con esta prioridad y pueden ganar el bus varias veces a lo largo de un intervalo
- los nodos urgentes sólo pueden acaparar el 75% de uso del bus

Capa de enlace

Define la transmisión de datos en forma de paquetes

Se permiten dos tipos de transmisión:

- asíncrona
  - se transmite una cantidad variable de datos más varios bytes de protocolo a una dirección explícita
  - se devuelve información de reconocimiento
  - se utiliza para datos que no tienen una velocidad de transferencia fija
- isócrona
  - se transmite una cantidad variable de datos pero en una secuencia de paquetes de tamaño fijo y a intervalos regulares
  - se usa un direccionamiento simplificado y no necesita reconocimiento

Transacción asíncrona

El método de arbitraje implícito es el equitativo, aunque también se puede usar el urgente

El proceso de enviar un paquete se denomina subacción (*subaction*) y consta de cinco periodos de tiempo:

- secuencia de arbitraje: intercambio de señales para acceder al control del bus
- transmisión de paquete
- intervalo de reconocimiento: espera al reconocimiento
- reconocimiento: el receptor del paquete devuelve el código del resultado de la transmisión
- intervalo de subacción: inactividad forzosa hasta un nuevo arbitraje

Transacción asíncrona (cont.)

El formato del paquete consta de:

- una cabecera con
  - identificadores de fuente y destino
  - tipo de paquete
  - código de redundancia cíclica (CRC)
  - parámetros del tipo específico de paquete
- bloque de datos de usuario
- otro código CRC

En el envío de reconocimiento no se realiza arbitraje porque el nudo que recibe el paquete sabe que dispone del bus

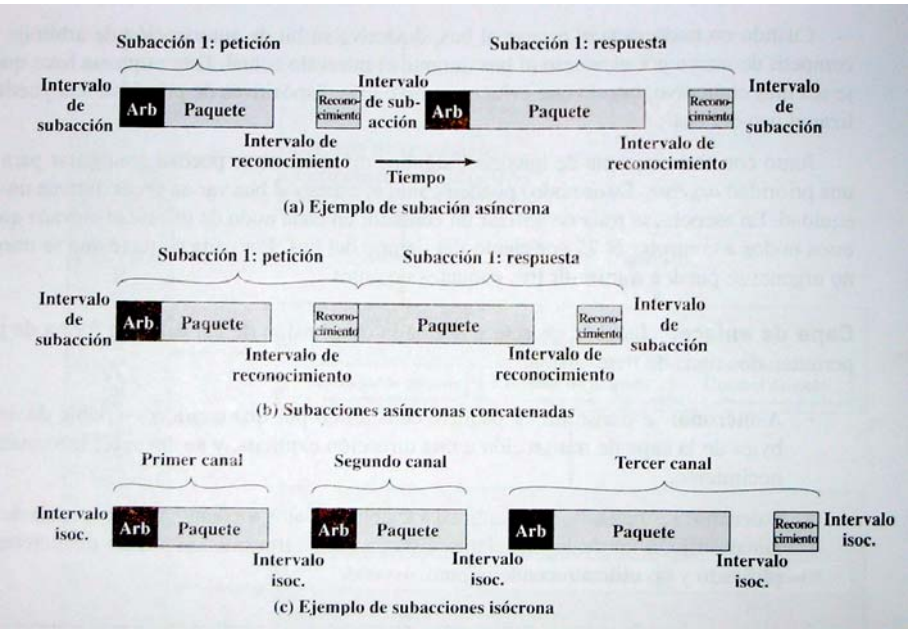
Otras transacciones

La transacción isócrona permite a los dispositivos mantener transferencias regulares de datos; muy útil en transmisión de audio o vídeo por ejemplo

También es posible tener tráfico mixto:

- **uno de los nodos se designa como maestro del ciclo y genera periódicamente un paquete de comiezo**
- **todos saben que ha comenzado un ciclo isócrono en el que sólo se envían este tipo de paquetes**
  - **cada nodo transmite cuando gana el bus sin envío de reconocimiento y con la espera de un intervalo isócrono (menor que el asíncrono)**
  - **después del último envío isócrono las fuentes asíncronas transmiten hasta el siguiente ciclo**

Subacciones FireWire



[6]

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] Especificaciones USB: http://www.usb.org/

[6] William Stallings, "Organización y arquitectura de computadores", 7ª Ed., Pearson, 2006.