Bloque I: Principios de sistemas operativos

Tema 1. Principios básicos de los sistemas operativos
Tema 2. Concurrencia
Tema 3. Ficheros
Tema 4. Sincronización y programación dirigida por eventos
Tema 5. Planificación y despacho
Tema 6. Sistemas de tiempo real y sistemas empotrados
Tema 7. Gestión de memoria
Tema 8. Gestión de dispositivos de entrada-salida

Notas:

Tema 8. Gestión de dispositivos de entrada-salida

- Características de los dispositivos de entrada/salida
- Entrada/salida por consulta
- Entrada/salida por interrupciones
- Entrada/salida por acceso directo a memoria
- Organización de manejadores de dispositivos (*drivers*) de entrada/salida
- Programación de manejadores de dispositivos de entrada/salida
- Implementación de *drivers*

1. Características de los dispositivos de entrada/salida

Los dispositivos de entrada/salida (I/O) forman junto con la CPU y la memoria los elementos más importantes del computador

Uno de sus objetivos principales es la eficiencia en las operaciones de entrada/salida, minimizando el trabajo a realizar por la CPU

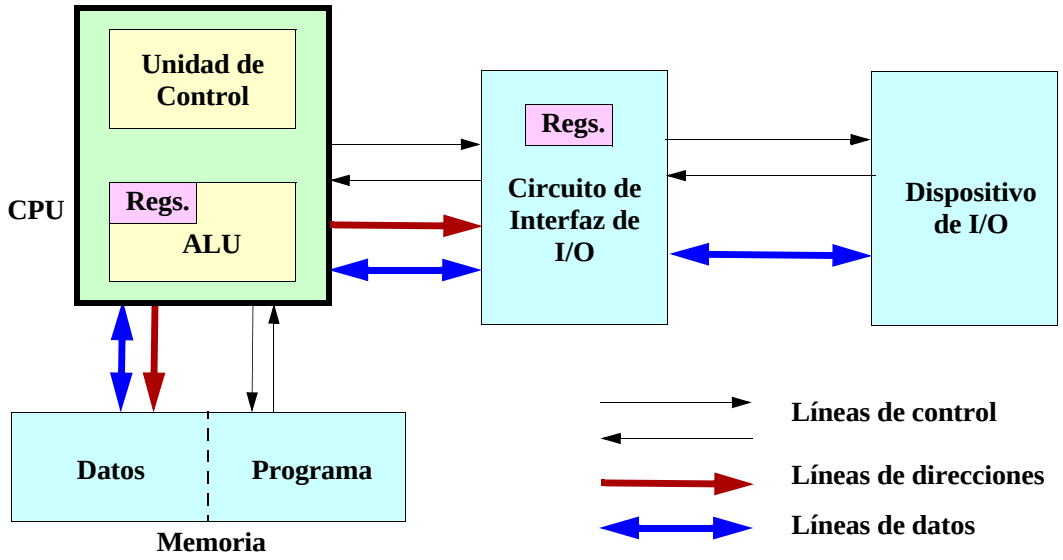
Las velocidades de los dispositivos de I/O son muy variadas:

- dispositivos lentos (p.e., ratón, teclado)
- dispositivos medios (p.e., impresora)
- dispositivos rápidos (p.e., red, disco)

Para acomodar las velocidades se usan circuitos de interfaz

Interfaces de entrada/salida

Se encargan de la comunicación entre la CPU y el dispositivo



Conexión de las interfaces de entrada/salida

Conexión mapeada en memoria

- el circuito de interfaz se conecta como si fuera memoria
- se accede a los registros leyendo o escribiendo una variable en una posición de memoria concreta

Conexión mediante puertos de entrada/salida

- el circuito de interfaz se conecta mediante líneas especiales
- se accede a los registros mediante instrucciones especiales (*in*, *out*), especificando un número de puerto

Tipos de entrada/salida

Entrada/salida por consulta o programada

- la CPU accede a los registros desde programa
- para saber si el dispositivo está listo, se hace una consulta periódica

Entrada/salida por interrupciones

- el dispositivo avisa a la CPU cuando está listo
- la entrada/salida se hace mediante una rutina de servicio de interrupción

Entrada/salida por acceso directo a memoria

- el dispositivo accede directamente a la memoria
- avisa a la CPU del inicio o final de la operación

2. Entrada/salida por consulta

La operación de I/O es controlada por la CPU

Antes de realizar la operación se comprueba el (los) registro(s) de estado, para ver si el dispositivo está listo

Ventajas: sencillez

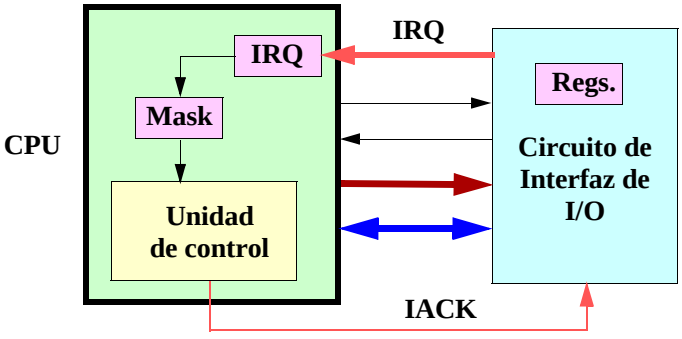
Desventajas:

- ritmo de transferencia limitado por la velocidad de la CPU
- tiempo de respuesta elevado, mayor que el periodo de consulta
- sobrecarga de la CPU para operaciones de consulta que podrían evitarse

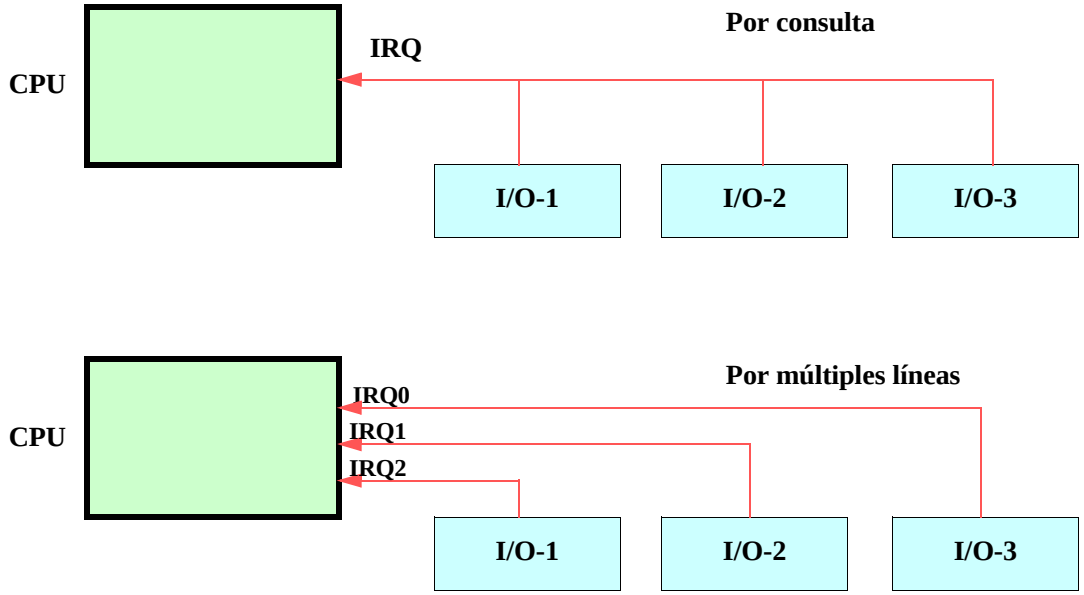
3. Entrada/salida por interrupciones

Permite al dispositivo marcar el instante en que se hace la transferencia de datos

El mecanismo de interrupción está presente en casi todos los computadores



Identificación de la fuente de interrupción



Gestión de interrupciones

Las interrupciones se pueden enmascarar

- se utiliza para evitar la interrupción cuando se accede a datos compartidos con ella

Para cada interrupción se puede instalar una rutina de servicio de interrupción

- al llegar la interrupción, el procesador interrumpe el programa en ejecución y enmascara esa interrupción
- después ejecuta la rutina de servicio de interrupción
- al acabar, el procesador restaura el estado anterior y el programa interrumpido continúa

La rutina de servicio de interrupción

Estructura habitual

- Acceder al dispositivo causante de la interrupción y hacer que cese la petición de interrupción
- Si es necesario, acceder al controlador de interrupciones para hacer lo mismo
- realizar la transferencia de datos

El modelo es el de una tarea concurrente más

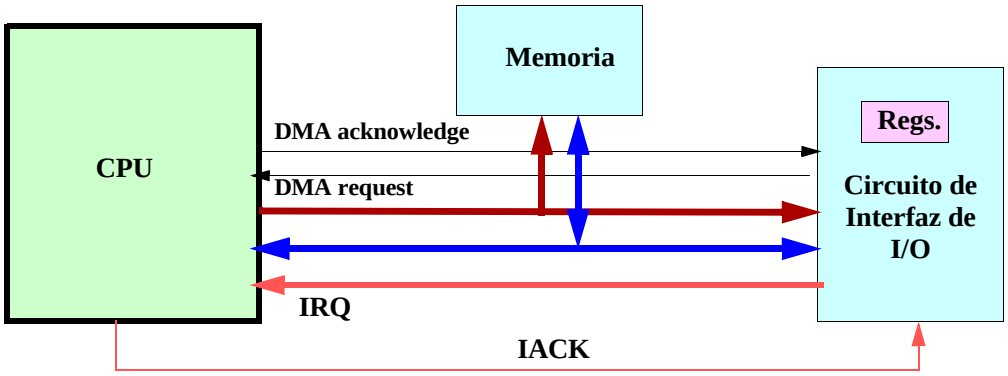
- ejecuta a la máxima prioridad del sistema

4. Entrada/salida por acceso directo a memoria

El ritmo de transferencia es superior al de los otros métodos

El instante de I/O lo marca el dispositivo con interrupciones

Hay líneas para desconectar a la CPU de la memoria



Funcionamiento de la I/O directa

Registros del dispositivo:

- ***IODIR*: Dirección de memoria para la operación de I/O**
- ***CONT*: Contador de número de bytes a transferir**

Funcionamiento habitual:

- **La CPU carga los valores en los registros del dispositivo**
- **El dispositivo solicita el uso de la memoria: *DMA-request***
- **La CPU se lo concede: *DMA-acknowledge***
- **El dispositivo transfiere los datos. Para cada uno, incrementa *IODIR* y decrementa *CONT***
- **Cuando *CONT* llega a 0 se devuelve el control de la memoria a la CPU (*DMA-request*) y se envía una interrupción de aviso**

5. Organización de *drivers* de entrada/salida

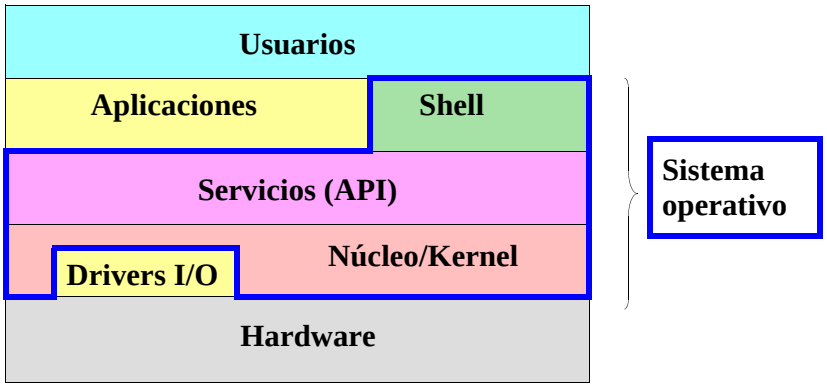
Un driver para un dispositivo es una interfaz entre el sistema operativo y el hardware de ese dispositivo

Los drivers forman parte del núcleo y tienen acceso restringido a estructuras del sistema operativo

El objetivo del driver es ofrecer un mecanismo de uso general, con operaciones como:

- **abrir (*open*) y cerrar (*close*)**
- **leer (*read*)**
- **escribir (*write*)**
- **controlar (*ioctl*)**

El driver en el contexto del sistema operativo



Tipos de dispositivos y módulos

En Linux 2.6 se distinguen tres tipos de dispositivos y módulos:

- *de caracteres*: E/S directa o por interrupciones
- *de bloques*: E/S por acceso directo a memoria
- *de red*: E/S por dispositivos de comunicaciones

Esta clasificación no es rígida y se podría considerar otra ortogonal como:

- módulos USB, módulos serie, módulos SCSI (*Small Computer Systems Interface*), etc.
- cada dispositivo USB estaría controlado por un módulo USB, pero el dispositivo en sí mismo se comportaría como de caracteres (puerto serie USB), de bloques (tarjeta de memoria USB), o una interfaz de red (interfaz Ethernet USB)

Dispositivos de caracteres

Se puede utilizar como si se tratara de un fichero, como una tira o conjunto de bytes

Normalmente implementa las llamadas al sistema: *open*, *close*, *read* y *write*

Ejemplos de dispositivos de caracteres:

- consola (`/dev/console`)
- puertos serie (`/dev/ttys0`)

El acceso a estos dispositivos se realiza a través de nodos (*nodes*) del sistema de ficheros

Módulos de Linux

Los drivers se implementan mediante módulos del núcleo:

- objetos software con una interfaz bien definida, que se cargan dinámicamente en el núcleo del sistema operativo

El módulo en Linux debe tener al menos dos operaciones:

- *init_function*: para instalarlo
  - debe preparar el módulo para la posterior ejecución del resto de las funciones o puntos de entrada del módulo
- *cleanup_function*: para desinstalarlo
  - debe eliminar todo rastro del módulo

El código de un módulo sólo debe llamar a funciones incluidas en el kernel y no en librerías, ya que el módulo se enlaza directamente con el kernel.

Módulos de Linux (cont.)

Los pasos para construir un módulo de Linux serían:

1. editar el fichero con las operaciones de inicialización y finalización del módulo
   - tenemos un `fichero.c`
2. compilar el módulo
   - se puede editar un *Makefile* para ahorrar trabajo
   - obtenemos el `fichero.ko`
3. cargar el módulo
   - con `insmod`

Para descargar el módulo se hace con `rmmod`

Ejemplo de módulo Linux

Fichero *sencillo.c*:

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("GPL");

static int instala (void)
{
  printk(KERN_ALERT "Sencillo instalado\n");
  return 0;
}

static void desinstala (void)
{
  printk (KERN_ALERT "Sencillo desinstalado\n");
}

module_init(instala);
module_exit(desinstala)
```

Estructura de un módulo del núcleo

- "*Includes*" para las funciones de los módulos de núcleo
- `MODULE_LICENSE`: Macro con el tipo de licencia (un texto)
- *Dos funciones*: una de instalación y otra de desinstalación, con los prototipos como en el ejemplo
- `module_init`: con esta macro declaramos cuál de nuestras funciones es la de instalación
- `module_exit`: con esta macro declaramos cuál de nuestras funciones es la de desinstalación

Compilación de módulos de Linux

La compilación:

- Se hace con un `make`, creando un fichero `Makefile` con la siguiente línea (`sencillo.o` es el nombre del fichero objeto):

```
obj-m := sencillo.o
```

- La orden `make` debe incluir el contexto del kernel y una orden que le indique el directorio de trabajo:

```
make -C /lib/modules/linux-2.6.15.23/build M=`pwd` modules
```

- donde hay que reemplazar `/lib/modules/linux-2.6.15.23` por el directorio donde se hallan los fuentes del kernel
  - es necesario disponer de las cabeceras de las fuentes del kernel
- el módulo recibe el nombre `sencillo.ko`

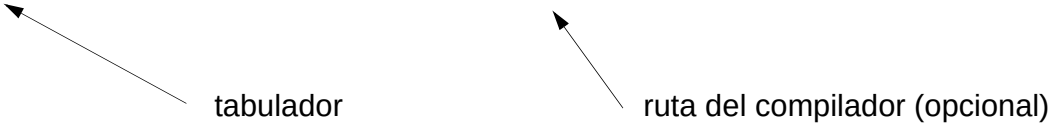
Compilación de módulos de Linux (cont.)

Es posible facilitar la compilación indicando la información anterior dentro del fichero `Makefile`:

```
obj-m := sencillo.o

KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
CC = /usr/bin/gcc-4.0

default:
    $(MAKE) -C $(KDIR) CC=$CC SUBDIRS=$(PWD) modules
```



Cabeceras de las fuentes del kernel

Se suelen encontrar en un paquete aparte

- `linux-headers-2.6.xx.xx`

Comprobar también que están instaladas las utilidades para carga y descarga dinámica de módulos

- `module-init-tools`

Instalación o desinstalación de módulos

- se debe hacer en modo superusuario (*root*)
- se instala con `insmod` (que, además, ejecuta la `init_function`):
- se desinstala mediante `rmmod` (ejecuta la `cleanup_function`):

```
/sbin/insmod sencillo.ko
/sbin/rmmod sencillo
```

- la orden `lsmod` muestra los módulos actualmente instalados
  - lee el fichero `/proc/modules`
  - muestra la siguiente información: nombre, tamaño, contador de uso y lista de módulos referidos

Depuración en módulos de Linux

No se pueden usar las llamadas normales al sistema operativo desde dentro del núcleo

- `printk`: similar al `printf`, pero ejecutable desde el núcleo; escribe en la consola y en `/var/log/messages`
  - la macro `KERN_ALERT` le da prioridad alta al mensaje (**linux/kernel.h**)
  - se puede consultar con

```
tail /var/log/messages
dmesg
```

Las funciones que se pueden usar desde el kernel se declaran en los directorios: include/linux y en include/asm

situados en las fuentes del kernel, normalmente en:
- **/usr/src/linux-xx-xx**

Drivers de Linux

Se crean como un caso particular de los módulos

Los drivers utilizan números especiales para su identificación que tienen que ser registrados en la instalación y borrados en la desinstalación

La instalación de un driver no da acceso a ningún dispositivo concreto:

- el acceso se obtiene con la creación posterior de un *fichero de dispositivo*:
  - forma parte del sistema de ficheros
  - sobre él se podrán ejecutar las operaciones ofrecidas por el driver

Identificación de dispositivos y drivers

Los dispositivos se representan mediante nombres especiales en el sistema de ficheros (habitualmente en `/dev`)

Usan números de identificación

- *Número mayor* (*major*): identifica a un driver, para el acceso a una unidad funcional separada
- *Número menor* (*minor*):
  - identifica subunidades de un dispositivo mayor
  - o diferentes modos de uso del dispositivo

Drivers: se identifican con un nombre y un dato del tipo `dev_t`

- El fichero `/proc/devices` muestra los dispositivos mayores

El tipo dev_t

Los números de dispositivo se guardan en un dato del tipo `dev_t`, para el que hay dos macros:

```
int MAJOR(dev_t dev);
int MINOR(dev_t dev);
```

- la primera nos devuelve el número mayor, y la segunda el menor

Se puede construir un valor del tipo `dev_t` con la macro:

```
dev_t MKDEV(unsigned int major, unsigned int minor);
```

Los números mayores se pueden asignar de forma estática o dinámica.

Asignación estática de números mayores

Si deseamos un número mayor concreto, podemos usar:

```
int register_chrdev_region
  (dev_t first,        //número mayor y primer menor
   unsigned int count,  //cuántos núm. menores
   char *name)          //nombre del dispositivo
```

- Reserva el número mayor indicado y unos cuantos números menores
- Si hay error, retorna un entero distinto de cero

Peligro de colisiones de número

Asignación dinámica de números mayores

La asignación de número mayor para un dispositivo de caracteres se hace en la `init_function` mediante la función definida en `<linux/fs.h>`:

```
int alloc_chrdev_region
  (dev_t *dev,          //retorna número mayor y menor
   unsigned int firstminor, // primer número menor
   unsigned int count,  //cuántos núm. menores
   char *name)          //nombre del dispositivo
```

- Reserva un número mayor para el driver y unos cuantos números menores
- Devuelve en `dev` el número mayor y el primer número menor
- Si hay error, retorna un entero distinto de cero

Creación del fichero de dispositivo

El nombre del dispositivo aparecerá en `/proc/devices` y `sysfs`

Luego creamos el fichero especial de dispositivo

- habitualmente en `/dev`
- se crean mediante la utilidad `mknod`

Por ejemplo, así se crea un fichero para números mayor y menor 123 y 0, respectivamente

```
mknod /dev/nombre_disp c 123 0
```

Puede ser necesario cambiar los permisos de acceso, para el uso por usuarios normales

Desinstalación del driver

Se hace desde la `cleanup_function`:

```
void unregister_chrdev_region
  (dev_t first,
   unsigned int count);
```

- `First` indica el número mayor y el primer número menor
- `Count` indica cuántos números menores hay que desinstalar

6. Programación de *drivers* de entrada/salida

El driver es un módulo software que debe responder a una interfaz bien definida con el kernel, con determinados *puntos de entrada* (funciones C):

`open`	abrir el dispositivo para usarlo
`release`	cerrar el dispositivo
`read`	leer bytes del dispositivo
`write`	escribir bytes en el dispositivo
`ioctl`	orden de control sobre el dispositivo

Las que no se usen no se implementan

Programación de drivers de entrada/ salida (cont.)

Otros puntos de entrada menos frecuentes:

`llseek`	posicionar el puntero de lectura/escritura
`aio_read`	lectura asíncrona (en paralelo al proceso)
`aio_write`	escritura asíncrona
`readdir`	leer información de directorio (no se usa en dispositivos)
`poll`	preguntar si se puede leer o escribir
`mmap`	mapear el dispositivo en memoria
`flush`	volcar los buffers al dispositivo
`fsync`	sincronizar el dispositivo
`aio_fsync`	sincronizar el dispositivo en paralelo al proceso
`fasync`	notificación de operación asíncrona

Programación de drivers de entrada/ salida (cont.)

Otros puntos de entrada menos frecuentes:

`lock`	reservar el dispositivo
`readv`	leer sobre múltiples áreas de memoria
`writev`	escribir sobre múltiples áreas de memoria
`sendfile`	enviar un fichero a un dispositivo
`sendpage`	enviar datos, página a página
`get_unmapped_area`	obtener zona de memoria
`check_flags`	comprobar las opciones pasadas a fcntl()
`dir_notify`	fcntl() requiere aviso de modificaciones a directorio

Interfaces de los puntos de entrada más frecuentes

Convenio de nombres: anteponer el nombre del dispositivo al del punto de entrada; por ejemplo, para el dispositivo `ptr`

```
int ptr_open (struct inode *inode, struct file *filp)

void ptr_release (struct inode *inode, struct file *filp)

ssize_t ptr_read (struct file *filp, char __user *buff,
                  size_t count, loff_t *offp)

ssize_t ptr_write (struct file *filp, const char __user *buff,
                   size_t count, loff_t *offp)

int ptr_ioctl (struct inode *inode, struct file *filp,
               unsigned int cmd, unsigned long arg)
```

La notación `__user` indica que el dato no puede usarse directamente desde el driver; es ignorada por el compilador

Estructuras básicas de los drivers

Definidas en `<linux/fs.h>`:

- `struct file_operations`
  - contiene una tabla de punteros a las funciones del driver
- `struct file`
  - sirve para representar en el kernel un fichero abierto
- `struct inode`
  - estructura con la que el kernel representa internamente los ficheros

Tabla de punteros

```
struct file_operations {
    struct module *owner;
    ... // punteros a puntos de entrada
}
```

El campo `owner` es un puntero al módulo propietario del driver

- Se suele poner al valor `THIS_MODULE`

Tabla de punteros (cont.)

Los restantes campos de esta estructura son punteros a las funciones del driver, en este orden:

`llseek`	`mmap`	`readv`
`read`	`open`	`writev`
`aio_read`	`flush`	`sendfile`
`write`	`release`	`sendpage`
`aio_write`	`fsync`	`get_unmapped_area`
`readdir`	`aio_fsync`	`check_flags`
`poll`	`fasync`	`dir_notify`
`ioctl`	`lock`	

Estructura de fichero

La información relevante para los drivers es la siguiente:

```
struct file {
  mode_t f_mode;
  loff_t f_pos;
  unsigned int f_flags;
  struct file_operations *f_op;
  void *private_data;
  struct dentry *f_dentry;  ...
}
```

- La crea el kernel en la operación `open` (puede haber muchas para el mismo dispositivo)
- Se le pasa a cualquier función que opera con el dispositivo

Estructura de fichero (cont.)

La descripción de los campos de `struct file` es:

- `f_mode`: Modo de lectura (`FMODE_READ`), escritura (`FMODE_WRITE`) o ambos
- `f_pos`: posición actual para lectura o escritura
  - valor de 64 bits, `long long`
  - se puede leer pero no se debería modificar
  - si `read` o `write` necesitan actualizar la posición deberían usar el parámetro que se les pasa
- `f_flags`: Opciones del fichero usadas al abrir
  - Ej.: `O_RDONLY`, `O_NONBLOCK`, `O_SYNC`
  - los permisos de lectura escritura se deberían chequear en `f_mode`

Estructura de fichero (cont.)

- `f_op`: tabla de punteros a los puntos de entrada del driver (ver página 39)
  - esta tabla se podría cambiar completamente en el **open** dependiendo por ejemplo del número menor para identificar un modo de funcionamiento
- `private_data`: puntero para guardar información específica del driver
  - al principio vale `null`; el **open** debe asignarle memoria
  - el `release` debe liberar esa memoria
- `f_dentry`: datos internos del sistema operativo
  - el punto más importante de esta estructura es que se puede acceder al inodo `filp->f_dentry->d_inode`

Estructura de inodos

Contiene información útil para los ficheros, tal como el estado del fichero, hora de creación y modificación, etc., pero la información importante para los drivers es:

```
struct inode {
    dev_t i_rdev;
    struct cdev *i_cdev;
    ...
}
```

- `i_rdev`: identificador del dispositivo
- `i_cdev`: apunta a la estructura interna del kernel para dispositivos de caracteres

Estructura de inodos (cont.)

Es única para el dispositivo

Podemos usar estas macros para averiguar los números mayor y menor:

```
unsigned int imajor(struct inode *inode)
unsigned int iminor(struct inode *inode)
```


Registro del driver de caracteres

Antes de usar el driver, es preciso alojar en memoria, inicializar y registrar la estructura `cdev`

- con las funciones de `<linux/cdev.h>`

Alojar:

```
struct cdev *cdev_alloc(void);
```

Inicializar

```
void cdev_init(struct cdev *dev,
               struct file_operations *fops);
```

Además, hay que inicializar el campo `owner`:

```
cdev->owner=THIS_MODULE;
```

Registro del driver de caracteres (cont.)

Añadir el driver al kernel

```
int cdev_add(struct cdev *dev,
             dev_t num,            // id. dispositivo
             unsigned int count); // cuántos disp.
```

- si falla, retorna un número distinto de cero, y no se añade el dispositivo al sistema
- si va bien, el dispositivo se dice que está *vivo*

Eliminarlo

```
void cdev_del(struct cdev *dev);
```


Punto de entrada *open*

Interfaz:

```
int ptr_open (struct inode *inodep,
              struct file *filp)
```

Abre el dispositivo:

- Chequea errores del dispositivo
- Inicializa el dispositivo si es la primera vez que se abre
- Crea y rellena si hace falta `filp->private_data`
- Identifica el número menor y actualiza `f_op` si procede

Punto de entrada *release*

Interfaz:

```
void ptr_release (struct inode *inodep,
                  struct file *filp)
```

Cierra el dispositivo:

- Libera si hace falta la memoria asignada a `filp->private_data`
- Cierra el dispositivo si es preciso

Punto de entrada *read*

Interfaz:

```
ssize_t ptr_read (struct file *filp,
                  char __user *buff,
                  size_t count, loff_t *offp)
```

Lee datos del dispositivo:

- `count` es la cantidad de bytes a leer
- `buff` es el buffer de usuario en el que se depositan los datos
  - está en el espacio del usuario
- `offp` la posición del fichero a la que se accede
- retorna el número de bytes leídos o un valor negativo si hay algún error

Punto de entrada *write*

Interfaz:

```
ssize_t ptr_write (struct file *filp,
                   const char __user *buff,
                   size_t count, loff_t *offp)
```

Escribe datos del dispositivo:

- `count` es la cantidad de bytes a escribir
- `buff` es el buffer de usuario en el que están los datos
  - está en el espacio del usuario
- `offp` la posición del fichero a la que se accede
- retorna el número de bytes escritos o un valor negativo si hay algún error

Punto de entrada *ioctl*

Interfaz:

```
int ptr_ioctl (struct inode *inodep,
               struct file *filp,
               unsigned int cmd, unsigned long arg)
```

Envía una orden de control al dispositivo:

- `cmd` es la orden
- `arg` es un argumento para ejecutar la orden

La interpretación de la orden es dependiente del dispositivo

Retorna cero si va bien, o un código de error si va mal.

Ejemplo: Driver de prueba

```
//  basico.h

int basico_open(struct inode *inodep, struct file *filp);

int basico_release(struct inode *inodep, struct file *filp);

ssize_t basico_read (struct file *filp, char *buff,
                     size_t count, loff_t *offp);

ssize_t basico_write (struct file *filp, const char *buff,
          size_t count, loff_t *offp);
```


Driver de prueba: includes

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include "basico.h"

MODULE_LICENSE("GPL");
```

Driver de prueba: variables globales

```
static struct file_operations basico_fops = {
  THIS_MODULE, // owner
  NULL,        // lseek
  NULL,        // read
  NULL,        // aio_read
  NULL,        // write
  NULL,        // aio_write
  NULL,        // readdir
  NULL,        // poll
  NULL,        // ioctl
  NULL,        // mmap
  NULL,        // open
  NULL,        // flush
  NULL,        // release
  NULL,        // fsync
  // lo mismo para todos los demás puntos de entrada
  NULL         // dir_notify
};
```


Driver de prueba: variables globales (cont.)

```
// Datos del dispositivo, incluyendo la estructura cdev

struct basico_datos {
  struct cdev *cdev;  // Estructura para dispositivos de caracteres
  dev_t dev;          // información con el numero mayor y menor
  int dato_cualquiera;
};

static struct basico_datos datos;
```

Driver de prueba: instalación y desinstalación del driver

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

  // ponemos los puntos de entrada
  basico_fops.open=basico_open;
  basico_fops.release=basico_release;
  basico_fops.write=basico_write;
  basico_fops.read=basico_read;

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


Driver de prueba: instalación y desinstalación del driver (cont.)

```
  // iniciamos datos
  datos.dato_cualquiera=33; // por ejemplo
  datos.cdev=cdev_alloc();
  cdev_init(datos.cdev, &basico_fops);
  datos.cdev->owner = THIS_MODULE;
  result= cdev_add(datos.cdev, datos.dev, 1);
  if (result!=0) {
    printk(KERN_WARNING "basico> (init_module) Error%d al añadir",
           result);
    // deshacer la reserva y salir
    unregister_chrdev_region(datos.dev,1);
    return result;
  }

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

Driver de prueba: instalación y desinstalación del driver

```
static void modulo_salida(void) {
  cdev_del(datos.cdev);
  unregister_chrdev_region(datos.dev,1);
  printk( KERN_INFO "basico> (cleanup_module) descargado OK\n");
}


module_init(modulo_instalacion);
module_exit(modulo_salida);
```


Driver de prueba: puntos de entrada

```
int basico_open(struct inode *inodep, struct file *filp) {
  int menor= iminor(inodep);
  printk(KERN_INFO "basico> (open) menor= %d\n",menor);
  return 0;
}


int basico_release(struct inode *inodep, struct file *filp) {
  int menor= iminor(inodep);
  printk(KERN_INFO "basico> (release) menor= %d\n",menor);
  return 0;
}
```

Driver de prueba: puntos de entrada (cont.)

```
ssize_t basico_read (struct file *filp, char *buff,
        size_t count, loff_t *offp)
{
  printk(KERN_INFO "basico> (read) count=%d\n",(int) count);
  return (ssize_t) 0;
}


ssize_t basico_write (struct file *filp, const char *buff,
         size_t count, loff_t *offp)
{
  printk(KERN_INFO "basico> (write) count=%d\n",(int) count);
  return (ssize_t) count;
}
```


Programa de prueba para el driver

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

int main() {
  int fd,err;
  char *mistring="Hola yo soy un mensaje";
  char strleido[100];
  ssize_t leidos, escritos;
  strleido[0]=0; // lo hago vacío

  fd=open("/dev/basico0", O_RDWR);
  if (fd==-1) {
    perror("Error al abrir el fichero"); exit(1);
  }
```

Programa de prueba (cont.)

```
  printf("Fichero abierto\n");

  escritos=write(fd,mistring,strlen(mistring)+1);
  printf("Bytes escritos: %d\n",escritos);

  leidos=read(fd,&strleido,100);
  strleido[leidos]='\0';
  printf("Bytes leidos: %d\n",leidos);
  printf("String leido: %s\n",strleido);

  err=close(fd);
  if (err==-1) {
    perror("Error al cerrar el fichero");
    exit(1);
  }
  printf("Fichero cerrado\n");

  exit(0);
}
```


7. Implementación de drivers

Reserva de memoria para variables dinámicas: en <linux/slab.h>

- dentro del kernel debe usarse `kmalloc` y `kfree`:

```
void *kmalloc(size_t size, int flags);
void kfree(void *obj);
```

- `kmalloc` reserva un número de bytes >= `size` y nos retorna su dirección; `flags` controla el comportamiento de la función
- `kfree` libera la memoria reservada con `kmalloc`

El tamaño máximo que puede ser asignado por `kmalloc` está limitado (siempre menor que 128KB)

- existen otras funciones para conseguir más memoria como `vmalloc`

Copia de datos del espacio del usuario al del kernel

El espacio de direcciones del kernel es diferente al de los procesos (espacio de usuario)

Las funciones *read* y *write* necesitan copiar datos respectivamente hacia y desde el espacio de usuario

Las funciones para estas copias están en <asm/uaccess.h>:

```
unsigned long copy_to_user(void *to, const void *from,
                           unsigned long count);

unsigned long copy_from_user(void *to, const void *from,
                             unsigned long count);
```

- retornan el número de bytes no copiados
- o un error, si el puntero de usuario no es válido por ejemplo

Ejemplo: buffer virtual

```
//  buffervirtual.h

#define MAX 100

int buffervirtual_open(struct inode *inodep, struct file *filp);

int buffervirtual_release(struct inode *inodep,
                          struct file *filp);

ssize_t buffervirtual_read (struct file *filp, char *buff,
                            size_t count, loff_t *offp);

ssize_t buffervirtual_write (struct file *filp, const char *buff,
                             size_t count, loff_t *offp);
```

Buffer virtual: includes

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include "buffervirtual.h"
#include <linux/slab.h>
#include <asm/uaccess.h>

MODULE_LICENSE("GPL");
```


Buffer virtual: variables globales

```
static struct file_operations buffervirtual_fops = {
  THIS_MODULE, // owner
  NULL,        // lseek
  NULL,        // read
  NULL,        // aio_read
  NULL,        // write
  NULL,        // aio_write
  NULL,        // readdir
  NULL,        // poll
  NULL,        // ioctl
  NULL,        // mmap
  NULL,        // open
  NULL,        // flush
  NULL,        // release
  NULL,        // fsync
  // lo mismo para todos los demás puntos de entrada
  NULL         // dir_notify
};
```

Buffer virtual: variables globales

```
// Datos del dispositivo, incluyendo la estructura cdev

struct bv_datos {
  struct cdev *cdev; // Estructura de dispositivos de caracteres
  dev_t dev;         // información con el numero mayor y menor
  char *buffer;      // este sera nuestro buffer
};

static struct bv_datos datos;
```


Buffer virtual: instalación y desinstalación del driver

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

  // ponemos los puntos de entrada
  buffervirtual_fops.open=buffervirtual_open;
  buffervirtual_fops.release=buffervirtual_release;
  buffervirtual_fops.write=buffervirtual_write;
  buffervirtual_fops.read=buffervirtual_read;

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

Buffer virtual: instalación y desinstalación del driver (cont.)

```
  // reservamos MAX bytes de espacio para el buffer
  datos.buffer = kmalloc(MAX, GFP_KERNEL);
  if (datos.buffer==NULL) {
    result = -ENOMEM;
    unregister_chrdev_region(datos.dev,1);
    printk( KERN_WARNING "bv> (init_module) Error, no hay mem.\n");
    return result;
  }

  // instalamos driver
  datos.cdev=cdev_alloc();
  cdev_init(datos.cdev, &buffervirtual_fops);
  datos.cdev->owner = THIS_MODULE;
  result= cdev_add(datos.cdev, datos.dev, 1);
  if (result!=0) {
    printk(KERN_WARNING "bv> (init_module) Error %d al añadir",
           result);
    // deshacer la reserva y salir
    kfree(datos.buffer);
```


Buffer virtual: instalación y desinstalación del driver (cont.)

```
    unregister_chrdev_region(datos.dev,1);
    return result;
  }

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

static void modulo_salida(void) {
  kfree(datos.buffer);
  cdev_del(datos.cdev);
  unregister_chrdev_region(datos.dev,1);
  printk( KERN_INFO "bv> (cleanup_module) descargado OK\n");
}

module_init(modulo_instalacion);
module_exit(modulo_salida);
```

Buffer virtual: puntos de entrada del driver (cont.)

```
int buffervirtual_open(struct inode *inodep, struct file *filp) {
  int menor= iminor(inodep);
  printk(KERN_INFO "bv> (open) menor= %d\n",menor);
  return 0;
}

int buffervirtual_release(struct inode *inodep, struct file *filp) {
  int menor= iminor(inodep);
  printk(KERN_INFO "bv> (release) menor= %d\n",menor);
  return 0;
}
```


Buffer virtual: puntos de entrada del driver (cont.)

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

  printk(KERN_INFO "bv> (read) count=%d\n",(int) count);
  cuenta=(ssize_t) count;
  if (cuenta>MAX) {
    cuenta=MAX;
  }
  not_copied=copy_to_user(buff,datos.buffer,
                          (unsigned long)cuenta);
  if (not_copied!=0) {
    printk(KERN_WARNING "bv> (read) AVISO, no se leyeron datos\n");
    return(-EFAULT);
  }
  return cuenta;
}
```

Buffer virtual: puntos de entrada del driver (cont.)

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

  printk(KERN_INFO "bv> (write) count=%d\n",(int) count);
  cuenta=(ssize_t) count;
  if (cuenta>MAX) {
    cuenta=MAX;
  }
  not_copied=copy_from_user(datos.buffer,buff,
                            (unsigned long)cuenta);
  if (not_copied!=0) {
    printk(KERN_WARNING "bv> (write) AVISO, no escribió bien\n");
    return(-EFAULT);
  }
  return cuenta;
}
```


Acceso al hardware

Se dispone en `<asm/io.h>` de funciones para acceso a los puertos de entrada/salida:

```
unsigned inb (unsigned port);
void outb (unsigned char byte, unsigned port);
```

Estas funciones leen o escriben, respectivamente, un byte (8 bits) en el puerto indicado

Hay funciones también para leer enteros de 16 o 32 bits.

Hay funciones equivalentes que hacen una pausa después de leer o escribir:

```
unsigned inb_p (unsigned port);
void outb_p (unsigned char byte, unsigned port);
```

Temporizadores del kernel

Los temporizadores se usan para planificar la ejecución de una función en un instante de tiempo dado

- de forma asíncrona a los threads de usuario
- tienen la resolución de un *jiffie*
- la función que se ejecuta es en respuesta a una interrupción software
  - hay que prever la sincronización (p.e., con *spinlocks* o variables atómicas)
  - no se puede dormir o llamar al planificador (`schedule()`)
  - no puede usar funciones bloqueantes (p.e., `kmalloc()`)
- puede volver a registrarse para ejecutar más tarde (p.e., actividad periódica)

Temporizadores del kernel (cont.)

La estructura de datos de un timer se encuentra en `<linux/timer.h>`:

```
struct timer_list {
   /*...*/                  // otros miembros internos
   unsigned long expires; //valor absoluto de jiffies
   void (*function)(unsigned long);
                          // función manejadora
   unsigned long data;    // argumento para el manejador
             // para más datos hacer cast de un puntero
};
```

Temporizadores del kernel (cont.)

Inicializar el temporizador

```
void init_timer(struct timer_list *timer);
```

Añadir a la lista de temporizadores activos

```
void add_timer(struct timer_list *timer);
```

Cambia el instante de expiración

```
int mod_timer(struct timer_list *timer,
              unsigned long expires);
```


Temporizadores del kernel (cont.)

Elimina el temporizador antes de que el tiempo expire

```
int del_timer(struct timer_list *timer);
```

Elimina el temporizador, garantizando que al finalizar la función no se está ejecutando

```
int del_timer_sync(struct timer_list *timer);
```

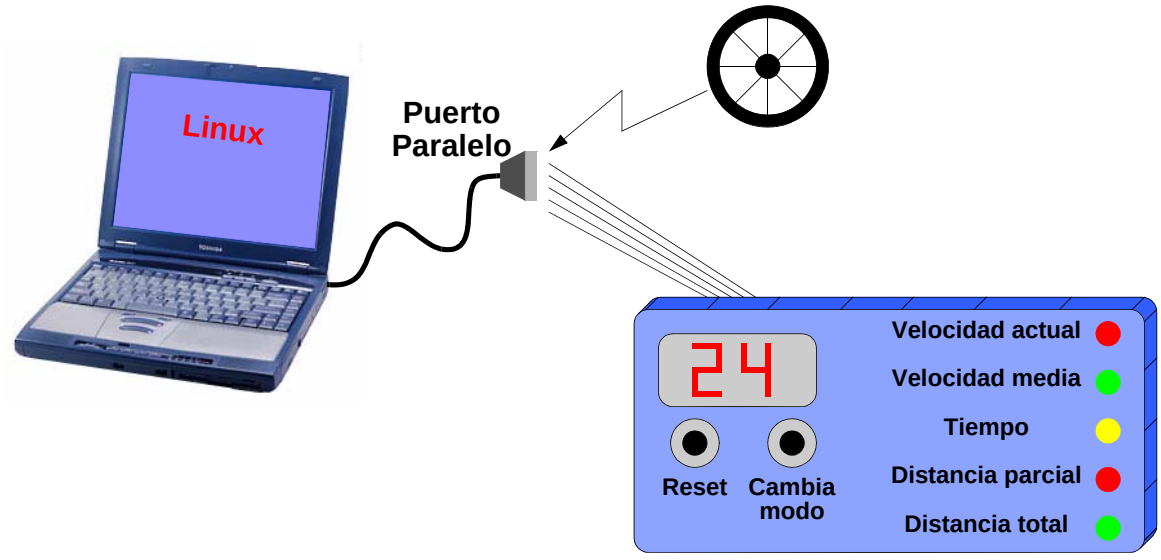
- debe asegurarse que la función del timer no usa *add_timer* sobre sí misma al eliminarlo
- es la función preferida para eliminar

Ejemplo: entrada/salida digital

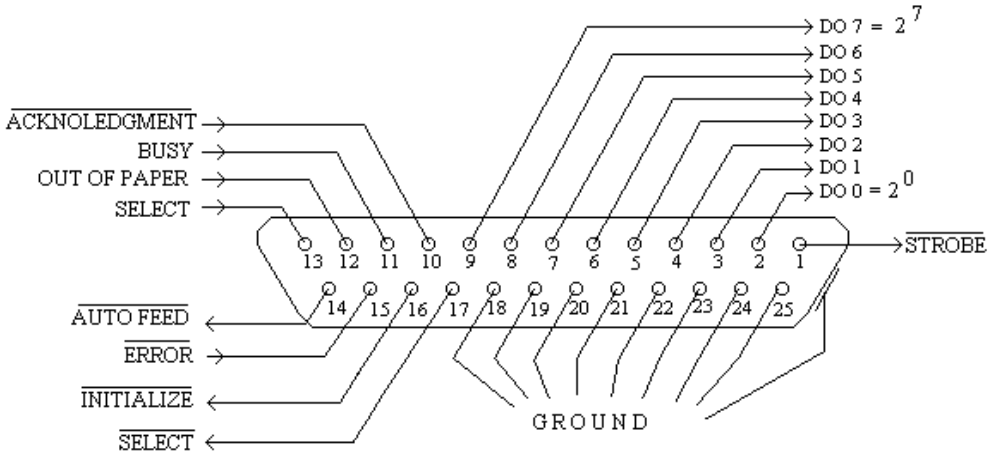
Es una de las aplicaciones que tiene el puerto paralelo; nosotros vamos a construir y programar un *dispositivo velocímetro* que:

- **puede medir la velocidad de giro de una rueda, y por tanto la velocidad del vehículo que la lleva**
- **puede presentar en un display:**
  - **la velocidad instantánea**
  - **la velocidad media (desde la inicialización)**
  - **la distancia recorrida (desde la inicialización)**
  - **la distancia total recorrida (desde el arranque del sistema)**
  - **y el tiempo (desde la inicialización)**
- **dispone además de dos botones para cambiar el modo de presentación en el display, y para inicializar el dispositivo**
- **también tiene 5 leds que permiten identificar el modo elegido**

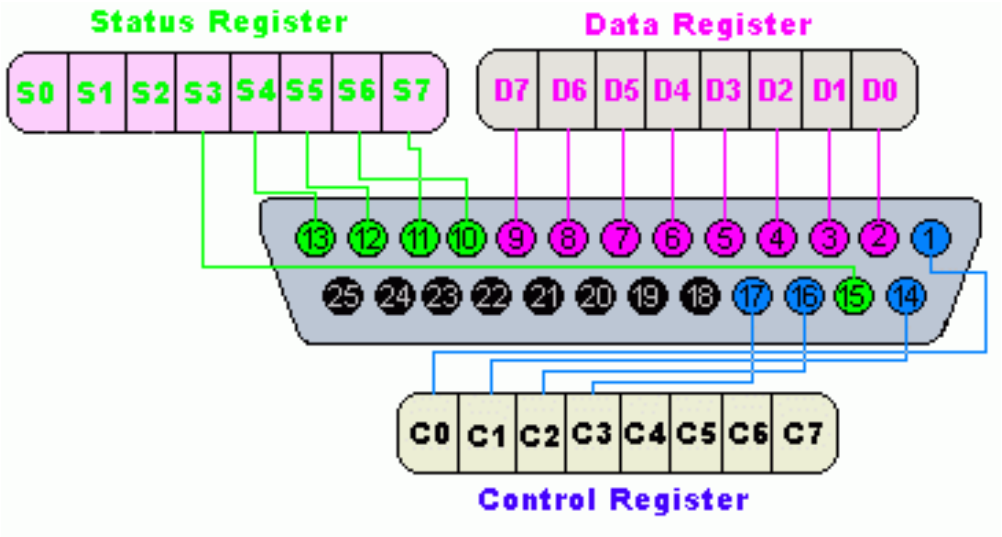
Esquema de conexión del dispositivo velocímetro



Líneas del puerto paralelo



Modelo de programación del puerto paralelo



Recursos del puerto paralelo

En el PC las direcciones de los registros y los niveles de interrupción utilizados son los siguientes:

Puerto	LPT1	LPT2	LPT3
Datos	0x0378	0x0278	0x03BC
Estado	0x0379	0x0279	0x03BD
Control	0x037A	0x027A	0x03BE
Interrupción	IRQ7	IRQ5	IRQ7

Habrá que configurar la BIOS del PC para elegir tanto el puerto que se utiliza como el modo de operación

Hardware del dispositivo velocímetro

Medidor del movimiento de la rueda:

- un fototransitor y un diodo led enfrentados permiten detectar el paso de los radios de la rueda
- la señal del fototransitor se digitaliza mediante un comparador
- la salida del comparador se conecta a la línea ACK del puerto paralelo (permite generar una interrupción cada vez que pasa un radio)
- este mecanismo permite contar radios:
  - o lo que es lo mismo, vueltas, distancia ...
  - si se anotan además los tiempos en los que se detecta el paso del radio se obtienen las velocidades

Hardware del dispositivo velocímetro (cont.)

Botones:

- se conectan directamente a dos entradas del puerto paralelo
  - Reset a BUSY
  - Cambio de modo a PE
- habrá que leer periódicamente los valores de las entradas para saber si se han pulsado

Display:

- consta de dos displays led de 7 segmentos conectados a sendos convertidores BCD a 7 segmentos
- los convertidores se controlan con las 8 líneas de datos en las que se podrán escribir valores de 0 a 99 (en BCD)
- los displays están conectados al revés

Hardware del dispositivo velocímetro (cont.)

Leds de modo seleccionado:

- se conectan a las salidas del puerto paralelo mediante un decodificador de 3 a 8

El control desde los registros del puerto queda del modo siguiente

LED	Reg. Control
Velocidad actual	0x03
Velocidad media	0x02
Tiempo transcurrido	0x01
Distancia parcial	0x00
Distancia total	0x04

Botón	Reg. Estado
Reset	0x80
Cambia modo	0x20

Hardware del dispositivo velocímetro (cont.)

Desde el punto de vista del uso del puerto como entradas salidas digitales, no nos interesa la información de protocolo

- únicamente importa la relación entre los bits de los registros y las líneas de control y estado
  - registro de control

			IRQ	$\overline{P17}$	P16	$\overline{P14}$	$\overline{P1}$

  - registro de estado

$\overline{P11}$	P10	P12	P13	P15			

  - el registro de datos mantiene directamente los valores escritos

Diseño del driver: Velocímetro

Vamos a contemplar dos aproximaciones en el diseño del driver:

- opción 1:
  - hacer el driver lo más sencillo posible con la funcionalidad imprescindible: acceso a los registros y control de las interrupciones
  - dejar el resto a la aplicación
- opción 2:
  - hacer que el driver se encarge del control total del dispositivo
  - ofrecer a la aplicación operaciones de consulta y control alternativo al que ofrece el propio dispositivo

Diseño del driver: Velocímetro (cont.)

- **Solución a la opción 1:**
  - hacer que la operación `read` lea el registro de estado con la información de los botones y `write` escriba en el registro de datos la información del display
  - si se usan interrupciones, los controles de distancia, tiempo y velocidad quedarían dentro del driver (paso del radio)
  - el `ioctl` escribe el registro de control para encencer el led correspondiente al modo, y ofrece operaciones de consulta
  - el control de los botones pulsados lo tendrá que hacer la aplicación y llevará registro del modo actual; también presentará la información correspondiente en el display
  - si no se usan interrupciones, la aplicación también se encargará de calcular las distancias, tiempos y velocidades

Diseño del driver: Velocímetro (cont.)

- **Solución a la opción 2:**
  - tanto si se usa la interrupción como si no, el driver controla completamente la medida de la distancia, el tiempo y la velocidad
  - también se encarga de controlar si los botones se han pulsado, ejecutando las acciones oportunas, con activación de los led y presentación de datos en el display
  - la operación `read` no es relevante para el control y se puede usar para obtener información de estado del velocímetro
  - la operación `write` tampoco es relevante para el control y puede suministrar algún tipo de información al driver
  - el `ioctl` puede ejecutar comandos de cambio de modo o inicialización alternativos a los botones

Esta es la opción que vamos a implementar

Detalles del diseño: Velocímetro

¿Se van a usar interrupciones?

- Sí: la detección del paso del radio es inmediata
- No: habrá que programar una actividad periódica dependiente de la velocidad máxima que se pretenda medir

Aparición de datos en el display:

- estamos ante una interfaz humana y el periodo de presentación de datos es importante
  - si para ser muy exactos lo hacemos con un periodo bajo, es posible que los cambios muy rápidos hagan que no se vea nada
  - si se hace con un periodo muy alto, los cambios por ejemplo de la velocidad, pueden dar saltos muy grandes entre valores
- un valor razonable puede estar en 500 ms

Detalles del diseño: Velocímetro (cont.)

Atención al pulsado de los botones:

- de nuevo estamos ante una interfaz humana, y en este caso lo importante es el periodo de atención a que el ususario pulse uno de los botones
  - un periodo bajo, aunque se consiga detectar que el botón ha sido pulsado, puede dar una sensasión de que el sistema no reacciona
- un periodo razonable para dar sensación de instantaneidad puede ser 50 ms

Vamos a implementar la versión de *polling* programando una actividad periódica que atienda al cambio en la línea que provoca la interrupción

Diseño detallado: Velocímetro polling

***Open*:**

- **Controla el acceso al velocímetro como un recurso exclusivo**

***Read*:**

- **Obtiene en formato de texto la información interna del estado del velocímetro en el instante de la llamada: velocidad actual, velocidad media, distancia parcial, distancia total y tiempo**

***Write*:**

- **Graba en el driver un mensaje que va a ser mostrado como introducción a la información de estado que suministra la operación de lectura**

Diseño detallado: Velocímetro polling (cont.)

***Ioctl*:**

- **Establece el nuevo modo de presentación, como alternativa al uso del botón de cambio de modo**

Tres temporizadores para la realización de las actividades periódicas:

- **temporizador de control de botones**
- **temporizador de control de interrupción**
- **temporizador de presentación de datos**

Velocímetro: fichero de cabeceras

```
//  velocimetro.h

int velocimetro_open(struct inode *inodep, struct file *filp);

int velocimetro_release(struct inode *inodep, struct file *filp);

ssize_t velocimetro_read (struct file *filp, char *buff, size_t count,
                          loff_t *offp);

ssize_t velocimetro_write (struct file *filp, const char *buff,
                           size_t count, loff_t *offp);

int velocimetro_ioctl (struct inode *inodep, struct file *filp,
                       unsigned int cmd, unsigned long arg);
```


Velocímetro: comandos de ioctl

```
//  velocimetro_ioctl.h

#define SELECT_V_ACT    0x03
#define SELECT_V_MED    0x02
#define SELECT_TIEMPO   0x01
#define SELECT_D_PAR    0x00
#define SELECT_D_TOT    0x04
```

Velocímetro: includes

```
//  velocimetro.c driver

#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 <asm/atomic.h>
#include "velocimetro.h"

MODULE_LICENSE("GPL");
```


Velocímetro: constantes

```
// Direccion base del puerto paralelo LPT1
#define LPT1_BASE    0x378

// Offset de los registros del puerto paralelo
#define DATOS     0
#define ESTADO    1
#define CONTROL   2

// Palabras de control
#define V_ACT    0x03
#define V_MED    0x02
#define TIEMPO   0x01
#define D_PAR    0x00
#define D_TOT    0x04

// Mascaras de estado
#define RESET    0x80
#define MODO     0x20
#define INTER    0x40
```

Velocímetro: constantes (cont.)

```
// Numero de registros de puerto paralelo
#define NUM_REGS  3

// Periodo del temporizador
#define PERIODO   50

// Periodo del temporizador de interrupcion
#define PERIODO_INT   5

// Periodo del temporizador de presentacion de datos
#define PERIODO_PRES  500

// Longitud maxima del mensaje
#define MAX 30
```


Velocímetro: datos del dispositivo

```
struct veloc_datos {
  struct cdev *cdev;      // Character device structure
  dev_t dev;              // informacion con el numero mayor y menor
  char modo;              // modo de la aplicacion
  atomic_t distancia;     // distancia recorrida: contador de int.
  atomic_t distancia_t;   // distancia total
  atomic_t distancia_ant;// para calcular la velocidad actual
  unsigned long t_inic;   // instante inicial
  unsigned long t_ant;    // instante previo
  unsigned long t_act;    // instante actual
  int v_act;              // velocidad actual
  int v_media;            // velocidad media
  char *mensaje;             // mensaje de presentación de informacion
  char *buffer;              // buffer de informacion
  struct semaphore acceso;   // controla el acceso al driver
  spinlock_t uso_datos;      // controla acceso a los parametros
};

static struct veloc_datos datos;
```

Velocímetro: datos del dispositivo (cont.)

```
// Datos que se le pasan a la funcion manejadora del temporizador,
// incluyendo el propio temporizador
// Lee cada PERIODO ms el valor de los botones para ver si han
// sido pulsados

struct datos_temporizador {
  char estado_reset;              // -
  char estado_reset_prev;         // controlan el reset
  char estado_modo;               // -
  char estado_modo_prev;          // controlan el cambio de modo
  unsigned long jiffies_previos;
  struct timer_list timer;
  atomic_t apagando;
};

static struct datos_temporizador datos_timer;
```


Velocímetro: datos del dispositivo (cont.)

```
// Datos que se le pasan a la funcion manejadora del temporizador,
// que emula la interrupción incluyendo el propio temporizador
// Lee cada PERIODO_INT ms el valor de la interrupción
struct datos_temporizador_int {
  char estado_int;                // -
  char estado_int_prev;           // controlan el flanco de la int.
  unsigned long jiffies_previos;
  struct timer_list timer;
  atomic_t apagando;
};
static struct datos_temporizador_int datos_int;
// Datos que se le pasan a la funcion manejadora del temporizador
// Presenta cada PERIODO_PRES ms los datos en el display
struct datos_temporizador_pres {
  unsigned long jiffies_previos;
  struct timer_list timer;
  atomic_t apagando;
};
static struct datos_temporizador_pres datos_pres;
```

Velocímetro: funciones auxiliares

```
// Calcula la velocidad actual y la instantánea cada periodo
// del timer de presentación de datos
static void calcula_velocidades(void)
{
  long op;

  // actualiza el instante actual
  datos.t_act=jiffies;

  // velocidad actual en radios/segundo
  // radio como unidad de distancia, 5 radios/vuelta
  op=(long)datos.t_act-(long)datos.t_ant;
  if (op!=0)
    datos.v_act=(int)(((long)(atomic_read(&datos.distancia)-
          atomic_read(&datos.distancia_ant))*(long)PERIODO_PRES)/op);
```


Velocímetro: funciones auxiliares (cont.)

```
  // actualiza distancia e instante anteriores
  atomic_set(&datos.distancia_ant,atomic_read(&datos.distancia));
  datos.t_ant=datos.t_act;

  // velocidad media en radios/segundo
  op=(long)datos.t_act-(long)datos.t_inic;
  if (op!=0) datos.v_media= (int)
     (((long)atomic_read(&datos.distancia)*(long)1000)/op);

}
```

Velocímetro: funciones auxiliares (cont.)

```
// Muestra el parámetro apropiado en el display
static void display(void)
{
  int dato=0;
  char msh,lsh;

  switch (datos.modo) {
    case V_ACT:
      // velocidad en radios/segundo
      dato=datos.v_act;
      break;
    case V_MED:
      // velocidad en radios/segundo
      dato=datos.v_media;
      break;
    case TIEMPO:
      // tiempo en segundos
      dato=((long)jiffies-(long)datos.t_inic)/1000;
      break;
```


Velocímetro: funciones auxiliares (cont.)

```
    case D_PAR:
      // distancia en radios
      dato=atomic_read(&datos.distancia);
      break;
    case D_TOT:
      // distancia en radios
      dato=atomic_read(&datos.distancia_t);
      break;
    default:
      printk(KERN_WARNING "veloc> (display) error \n");
  }
  // corrige el orden de los displays de 7 segmentos
  // colocados al reves en la placa y extrae BCD
  msh= (dato/10)%10;
  lsh= (dato%10)<<4;
  outb(msh|lsh,LPT1_BASE+DATOS);
}
```

Velocímetro: instalación

```
static int modulo_instalacion(void) {
  int result;
  struct resource * region;

  // ponemos los puntos de entrada
  velocimetro_fops.open=velocimetro_open;
  velocimetro_fops.release=velocimetro_release;
  velocimetro_fops.write=velocimetro_write;
  velocimetro_fops.read=velocimetro_read;
  velocimetro_fops.ioctl=velocimetro_ioctl;

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


Velocímetro: instalación (cont.)

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

Velocímetro: instalación (cont.)

```
// reserva el rango de direcciones de I/O
if (check_region (LPT1_BASE, NUM_REGS)!=0) {
  printk(KERN_WARNING "veloc> (init_module) Borrando region I/O\n");
  release_region (LPT1_BASE, NUM_REGS);
}
region=request_region (LPT1_BASE, NUM_REGS, "velocimetro");
if (region==NULL) {
  result=-ENODEV;
  printk(KERN_WARNING
         "veloc> (init_module) direcc. I/O no disp.!!\n");
  goto error_reserva_IO;
}
printk(KERN_INFO
       "veloc> (init_module) Reservadas I/O. Rango:%x..%x\n",
       LPT1_BASE, LPT1_BASE + NUM_REGS -1);
```


Velocímetro: instalación (cont.)

```
// inicializamos datos del temporizador
datos_timer.jiffies_previos=jiffies;
atomic_set(&datos_timer.apagando,0);
datos_timer.estado_reset = 1;
datos_timer.estado_reset_prev = 1;
datos_timer.estado_modo = 1;
datos_timer.estado_modo_prev = 1;

// inicializamos datos del temporizador de interrupcion
datos_int.jiffies_previos=jiffies;
atomic_set(&datos_int.apagando,0);
datos_int.estado_int = 1;
datos_int.estado_int_prev = 1;

// inicializamos datos del temporizador de presentacion
datos_pres.jiffies_previos=jiffies;
atomic_set(&datos_pres.apagando,0);
```

Velocímetro: instalación (cont.)

```
// inicializamos datos del dispositivo
datos.modo=3;
atomic_set(&datos.distancia,0);
atomic_set(&datos.distancia_t,0);
atomic_set(&datos.distancia_ant,0);
datos.t_inic=jiffies;
datos.t_ant=datos.t_inic;
datos.t_act=datos.t_inic;
datos.v_act=0;
datos.v_media=0;
sema_init(&datos.acceso,1);
spin_lock_init(&datos.uso_datos);
if ((datos.mensaje= kmalloc(MAX,GFP_KERNEL))==NULL) {
  printk( KERN_WARNING
          "veloc> (init_module) Error, no hay memoria\n");
  result = -ENOMEM;
  goto error_mensaje;
}
```


Velocímetro: instalación (cont.)

```
datos.mensaje[0]=0;
if ((datos.buffer= kmalloc(3*MAX,GFP_KERNEL))==NULL) {
  printk( KERN_WARNING
          "veloc> (init_module) Error, no hay memoria\n");
  result = -ENOMEM;
  goto error_buffer;
}

// creamos el temporizador y lo activamos
init_timer(&datos_timer.timer);
datos_timer.timer.expires=jiffies+PERIODO;
datos_timer.timer.function=manejador;
datos_timer.timer.data=(unsigned long) &datos_timer; // cast
add_timer(&datos_timer.timer);
```

Velocímetro: instalación (cont.)

```
  // creamos el temporizador de interrupcion y lo activamos
  init_timer(&datos_int.timer);
  datos_int.timer.expires=jiffies+PERIODO_INT;
  datos_int.timer.function=manejador_int;
  datos_int.timer.data=(unsigned long) &datos_int; // cast
  add_timer(&datos_int.timer);
  // creamos el temporizador de presentacion y lo activamos
  init_timer(&datos_pres.timer);
  datos_pres.timer.expires=jiffies+PERIODO_PRES;
  datos_pres.timer.function=manejador_presenta_datos;
  datos_pres.timer.data=(unsigned long) &datos_pres; // cast
  add_timer(&datos_pres.timer);
  // inicializa registros de control y datos
  outb(datos.modo,LPT1_BASE+CONTROL);
  display();
  // todo correcto: mensaje y salimos
  printk( KERN_INFO "veloc> (init_module) OK con major %d\n",
          MAJOR(datos.dev));
  return 0;
```


Velocímetro: instalación (cont.)

```
  // Errores

 error_buffer:
  kfree(datos.mensaje);

 error_mensaje:

 error_reserva_IO:
  cdev_del(datos.cdev);

 error_instalacion_dev:
  unregister_chrdev_region(datos.dev,1);

 error_reserva_numeros:
  return result;
}
```

Velocímetro: desinstalación

```
static void modulo_salida(void) {
  cdev_del(datos.cdev);
  unregister_chrdev_region(datos.dev,1);
  // libera memoria
  kfree(datos.mensaje);
  kfree(datos.buffer);
  // libera el temporizador
  atomic_set(&datos_timer.apagando,1);
  del_timer_sync(&datos_timer.timer);
  // libera el temporizador de interrupcion
  atomic_set(&datos_int.apagando,1);
  del_timer_sync(&datos_int.timer);
  // libera el temporizador de presentacion de datos
  atomic_set(&datos_pres.apagando,1);
  del_timer_sync(&datos_pres.timer);
  // Libera direcciones de I/O
  release_region (LPT1_BASE, NUM_REGS);
  printk( KERN_INFO "veloc> (cleanup_module) descargado OK\n");
}
```


Velocímetro: open/release

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

  // intenta acceder al driver de acceso exclusivo
  if (down_trylock(&datos.acceso)!=0) {
    printk(KERN_WARNING "veloc> (open) Esta en uso ... \n");
    return -EBUSY;
  }

  return 0;
}

int velocimetro_release(struct inode *inodep, struct file *filp) {
  int menor= MINOR(inodep->i_rdev);
  printk(KERN_INFO "veloc> (release) menor= %d\n",menor);
  up(&datos.acceso);
  return 0;
}
```

Velocímetro: read

```
ssize_t velocimetro_read (struct file *filp, char *buff,
          size_t count, loff_t *offp)
{
  unsigned long not_copied=1;

  printk(KERN_INFO "veloc> (read) count=%d\n",(int) count);

  spin_lock_bh(&datos.uso_datos);
  sprintf(datos.buffer,
         "%s \nD   = %d \nDT  = %d \nT   = %d \nVi  = %d \nVmed= %d \n",
          datos.mensaje,
          atomic_read(&datos.distancia),
          atomic_read(&datos.distancia_t),
          (int)((datos.t_act-datos.t_inic)/1000),
          datos.v_act,
          datos.v_media);
  spin_unlock_bh(&datos.uso_datos);
```


Velocímetro: read (cont.)

```
  not_copied=copy_to_user(buff,datos.buffer,(unsigned long)(3*MAX));
  if (not_copied!=0) {
    printk(KERN_WARNING
           "veloc> (read) AVISO, no se leyeron los datos\n");
    return(-EFAULT);
  }
  return 3*MAX;
}
```

Velocímetro: write

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

  printk(KERN_INFO "veloc> (write) count=%d\n",(int) count);
  cuenta=(ssize_t) count;
  if (cuenta>MAX) {
    cuenta=MAX;
  }
  not_copied=copy_from_user(datos.mensaje,buff,(unsigned long)cuenta);
  if (not_copied!=0) {
    printk(KERN_WARNING "veloc> (write) AVISO, no se escribio bien\n");
    return(-EFAULT);
  }
  return cuenta;
}
```


Velocímetro: ioctl

```
int velocimetro_ioctl (struct inode *inodep, struct file *filp,
          unsigned int cmd, unsigned long arg)
{
  printk(KERN_INFO "veloc> (ioctl) cmd=%d\n",(int) cmd);
  datos.modo=(char)cmd%5;
  outb(datos.modo,LPT1_BASE+CONTROL);
  return 0;
}
```

Velocímetro: timer control botones

```
void manejador (unsigned long arg) {
  struct datos_temporizador *dat =(struct datos_temporizador *) arg;

  char byte;
  unsigned long j=jiffies;
  char DEBUG = 0;

  // comprueba si hay cambio en los botones
  byte=inb(LPT1_BASE+ESTADO);

  if (DEBUG) printk(KERN_INFO
                    "veloc> (timer) estado 0x%x \n",byte&0xff);
```


Velocímetro: timer control botones (cont.)

```
  dat->estado_modo_prev=dat->estado_modo;
  if (byte & MODO) {
    dat->estado_modo=1;
  } else {
    dat->estado_modo=0;
  }

  if ((dat->estado_modo_prev==1) && (dat->estado_modo==0)) {
    // cambio de modo
    if (DEBUG) printk(KERN_INFO "veloc> (timer) cambio de modo \n");
    datos.modo=(datos.modo+1) % 5;
    outb(datos.modo,LPT1_BASE+CONTROL);
  }
```

Velocímetro: timer control botones (cont.)

```
  dat->estado_reset_prev=dat->estado_reset;
  if (byte & RESET) {
    dat->estado_reset=1;
  } else {
    dat->estado_reset=0;
  }
  if ((dat->estado_reset_prev==1) && (dat->estado_reset==0)) {
    // pasa estado de reset
    if (DEBUG) printk(KERN_INFO "veloc> (timer) reset \n");
    datos.modo=3;
    outb(datos.modo,LPT1_BASE+CONTROL);
    datos.t_inic=jiffies;
    datos.t_ant=datos.t_inic;
    datos.t_act=datos.t_inic;
    atomic_set(&datos.distancia,0);
    atomic_set(&datos.distancia_ant,0);
  }
```


Velocímetro: timer control botones (cont.)

```
  // programa de nuevo el timer
  dat->timer.expires+=PERIODO;
  dat->jiffies_previos=j;
  if(atomic_read(&dat->apagando)==0) {
    add_timer(&dat->timer); //solo si no estamos apagando
  }
}
```

Velocímetro: timer control interrupción

```
void manejador_int (unsigned long arg) {
  struct datos_temporizador_int *dat =
              (struct datos_temporizador_int *) arg;

  char byte;
  unsigned long j=jiffies;
  char DEBUG = 0;

  // comprueba si hay cambio en la interrupcion
  byte=inb(LPT1_BASE+ESTADO);

  if (DEBUG) printk(KERN_INFO "veloc> (int) contador %d \n",
                    atomic_read(&datos.distancia));
  dat->estado_int_prev=dat->estado_int;
  if (byte & INTER) {
    dat->estado_int=1;
  } else {
    dat->estado_int=0;
  }
```


Velocímetro: timer control interrupción

```
  if ((dat->estado_int_prev==1) && (dat->estado_int==0)) {
    // anota otra interrupción
    if (DEBUG) printk(KERN_INFO "veloc> (timer) reset \n");
    atomic_inc(&datos.distancia);
    atomic_inc(&datos.distancia_t);
  }

  // programa de nuevo el timer
  dat->timer.expires+=PERIODO_INT;
  dat->jiffies_previos=j;
  if(atomic_read(&dat->apagando)==0) {
    add_timer(&dat->timer); //solo si no estamos apagando
  }
}
```

Velocímetro: timer presentación

```
void manejador_presenta_datos (unsigned long arg) {
  struct datos_temporizador_pres *dat =
                (struct datos_temporizador_pres *) arg;

  unsigned long j=jiffies;

  // calcula las velocidades y muestra el parametro acorde al modo
  spin_lock_bh(&datos.uso_datos);
  calcula_velocidades();
  spin_unlock_bh(&datos.uso_datos);
  display();

  // programa de nuevo el timer
  dat->timer.expires+=PERIODO_PRES;
  dat->jiffies_previos=j;
  if(atomic_read(&dat->apagando)==0) {
    add_timer(&dat->timer); //solo si no estamos apagando
  }
}
```


Programa de prueba

Dado que el driver incorpora toda la funcionalidad de control del velocímetro desde el momento de la instalación el hardware está operativo y funcionando

- no es necesario un programa de prueba

El programa de prueba

- escribe el mensaje para la información
- lee la información de estado del dispositivo y la muestra en pantalla cada segundo
- finaliza con una señal (p.e., Ctrl+c)
  - se pone el dispositivo en modo de muestra del tiempo
  - se cierra el dispositivo

Comentarios sobre el ejemplo

Hemos utilizado el puerto paralelo como una interfaz de entradas/ salidas digitales para una aplicación de control

Hemos optado por un diseño en el que se ofrece a la aplicación una funcionalidad de alto nivel

Hemos corregido por software dos problemas detectados en el hardware:

- posición de los displays de 7 segmentos
- las variaciones en la línea que detecta el paso del radio de la rueda

Comentarios sobre el ejemplo (cont.)

Hemos usado mecanismos de sincronización:

- un semáforo para controlar el uso exclusivo del driver por parte de las aplicaciones
- variables atómicas para los contadores de distancia
- un spinlock para el uso agrupado de los datos del driver

Hemos usado mecanismos de temporización:

- temporizador de control de botones
- temporizador de control de la línea de interrupción
- temporizador de presentación de datos