Plataformas de Tiempo Real: Dispositivos y Drivers

Tema 5. Interfaces de E/S de datos

- Interfaz serie
- Programación de la interfaz serie
- Interfaz paralelo
- Programación de la interfaz paralelo

Interfaz serie

Ha sido una de las interfaces más importantes en los computadores, en particular en los PCs:

- para la conexión de dispositivos como ploters, modems, ratones, impresoras, terminales, y un largo etc.
- también se usa como conexión para la comunicación entre dos computadores:
  - aunque hay conexiones mucho más rápidas, para sistemas pequeños sigue siendo una interfaz sencilla de programar
  - por ejemplo la depuración de un sistema MaRTE se hace a través de una línea serie

Los dispositivos que tradicionalmente se conectaban por la interfaz serie y otros más modernos, ahora lo hacen a través de USB (*Universal Serial Bus*)

Funcionamiento de la interfaz serie

La interfaz serie se utiliza para la comunicación punto a punto de datos en serie y asíncrona:

- los bytes se transmiten bit a bit
- no es necesaria una señal de reloj común al transmisor y al receptor
- se utilizan relojes independientes con la misma frecuencia en el transmisor y en el receptor
- se transmite un bit de comienzo (*start bit*), los bits de información y al menos un bit de parada (*stop bit*) que indica que se ha transmitido un dato
- también es posible transmitir información de chequeo de errores antes del bit de finalización, el bit de paridad (*parity bit*)

Funcionamiento de la interfaz serie (cont.)

La paridad sólo puede garantizar la detección de errores de un solo bit:

- para errores múltiples la probabilidad de detección es del 50%
- se puede utilizar un código CRC si se quiere mayor fiabilidad

Las paridades que se pueden utilizar son:

- sin paridad (no se envía el bit)
- par: se busca que haya un número par de unos entre los bits de datos y en el de paridad
- impar: se busca un número impar de unos
- *mark*: bit siempre a 1
- *space*: bit siempre a 0

Funcionamiento de la interfaz serie (cont.)

Un aspecto fundamental es seleccionar la velocidad a la que va a tener lugar la comunicación, o *baud rate*:

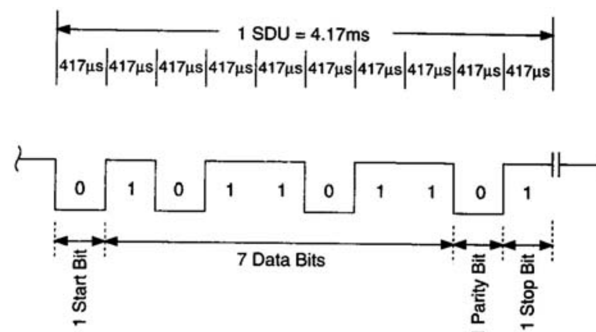
- se mide en bits por segundo (*bps o baudios*)

Antes de comenzar una transferencia de datos el transmisor y el receptor se deben sintonizar con los mismos formatos:

- velocidad de transmisión
- bits de datos: 5, 6, 7 u 8
- paridad: par, impar o ninguna
- bits de parada: 1 ó 2 (uno y medio si se eligen 5 bits de datos)

Transmisión de datos en serie

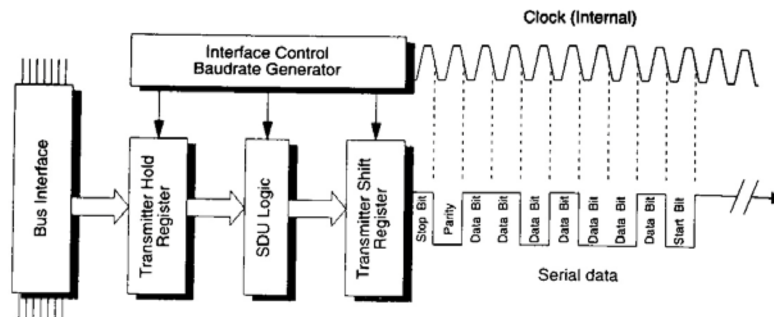
7 bits, paridad impar, 1 bit de parada, y 2400 bps (Messmer[1])



Transmisión de datos en serie (cont.)

UC UNIVERSIDAD DE CANTABRIA

La UART (*Universal Asynchronous Receiver-Transmitter*) es la interfaz encargada de serializar los datos, para transmisión (Messmer[1]):

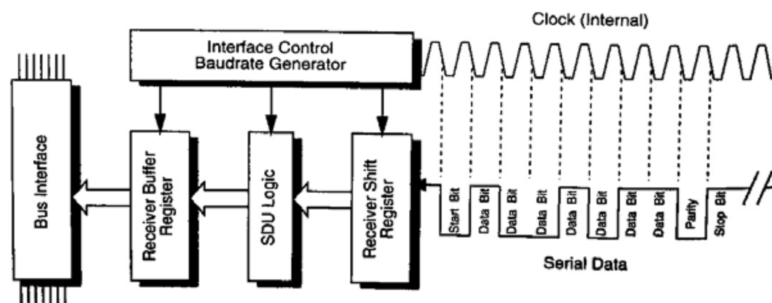


© J. Javier Gutiérrez, Michael González
5/ma/14

Transmisión de datos en serie (cont.)

UC UNIVERSIDAD DE CANTABRIA

Y para recepción (Messmer[1]):



© J. Javier Gutiérrez, Michael González
5/ma/14

Interfaz RS-232C

UC UNIVERSIDAD DE CANTABRIA

Estándar de EIA (*Electronic Industries Association*)

Interfaz eléctrica, mecánica y lógica entre un DTE (*Data Terminal Equipment*, p.e., un computador) y un DCE (*Data Carrier Equipment*, p.e., un módem):

- conexión de 25 pines, la mayoría reservados para la transferencia de datos síncrona
- para el uso mayoritario de esta interfaz en PCs sólo se necesitan 11 pines: tranferencia de datos asíncrona
- IBM definió un conector de sólo 9 pines
- un pin de transmisión, otro de recepción y el resto señales de control

© J. Javier Gutiérrez, Michael González
5/ma/14

Interfaz RS-232C (cont.)

Niveles de tensión y lógica de las señales:

ESTADO BINARIO	1 (MARK)	0 (SPACE)
Nivel de tensión entradas	+3 < N.T. < +15	-15 < N.T. < -3
Nivel de tensión salidas	+5 < N.T. < +15	-15 < N.T. < -5

Para velocidades de transmisión del orden de 20 kbps (máxima del estándar) se recomienda que la longitud del cable no sobrepase los 15 metros

Los circuitos de la interfaz permiten velocidades de 115200 bps

Otros estándares permiten longitudes mayores con comunicaciones más fiables (RS-422, o RS-485)

Distribución de pines en la Interfaz RS-232C

25 pines	9 pines	Señal	Dirección	Descripción
1	-	-	-	Protección de tierra
2	3	TD	DCE<-DTE	Transmisión
3	2	RD	DCE->DTE	Recepción
4	7	RTS	DCE<-DTE	Request to send
5	8	CTS	DCE->DTE	Clear to send
6	6	DSR	DCE->DTE	Data set ready
7	5	-	-	Tierra (común)
8	1	DCD	DCE->DTE	Data carrier detect
20	4	DTR	DCE<-DTE	Data terminal ready
22	9	RI	DCE->DTE	Ring indicator
23	-	DSRD	DCE<->DTE	Data signal rate detector

Distribución de pines en la Interfaz RS-232C (cont.)

TD (*transmitted data*) y RD (*received data*)
- líneas de transmisión y recepción de datos

RTS (*request to send*)
- el DTE le dice al DCE que le quiere enviar un dato

CTS (*clear to send*)
- el DCE le dice al DTE que está preparado para aceptar un dato
- cuando el DTE recibe la señal puede comenzar a enviar el dato

DCD (*data carrier detect*)
- el DCE activa esta señal cuando detecta la portadora y la mantiene activa durante toda la conexión

Distribución de pines en la Interfaz RS-232C (cont.)

UC UNIVERSIDAD DE CANTABRIA

DSR (*data set ready*)

- el DCE la activa para indicar que está preparado para operar

DTR (*data terminal ready*)

- el DTE indica que ha arrancado y está listo para operar
- normalmente se mantiene activa durante toda la conexión

RI (*ring indicator*)

- el DCE informa al DTE de que quiere conectarse

DSRD (*data signal race detector*)

- permite cambiar entre dos velocidades de transmisión diferentes

Modos de funcionamiento de la interfaz serie

UC UNIVERSIDAD DE CANTABRIA

La interfaz serie puede utilizar tres modos de funcionamiento:

- *Simplex*
  - modo unidireccional en el que sólo el DTE o el DCE pueden transmitir datos
- *Half-duplex*
  - DTE y DCE pueden transmitir datos pero no a la vez, lo hacen sobre una única línea que usan alternativamente
- *Full-duplex*
  - dos líneas de datos separadas sobre las DTE y DCE pueden transmitir simultáneamente

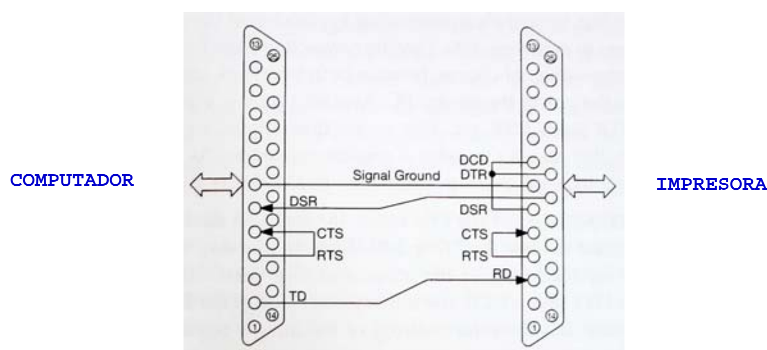
Dispositivos como una impresora utilizan una conexión *simplex*, mientras que un modem u otro DTE suelen utilizar *full-duplex*

Modos de funcionamiento de la interfaz serie (cont.)

UC UNIVERSIDAD DE CANTABRIA

Conexión de una impresora (Messmer [1]):

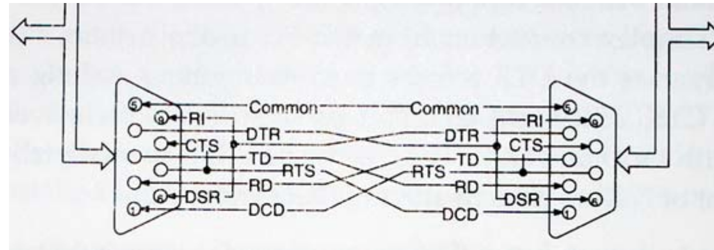
COMPUTADOR



IMPRESORA

Modos de funcionamiento de la interfaz serie (cont.)

Conexión entre dos computadores (Messmer [1]):



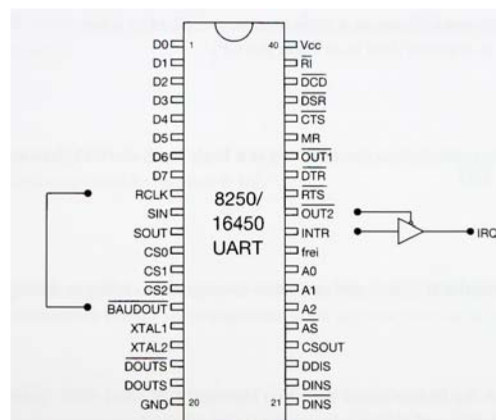
Habitualmente es suficiente con cruzar las líneas de transmisión y recepción, y posiblemente conectar una salida de un computador (RTS) con una entrada del otro (CTS)

La UART en el PC

El PC dispone de un chip para la comunicación serie:

- UART 8250
  - permite una velocidad máxima de transferencia de 9600 bps
- UART 16450
  - permite una velocidad máxima de transferencia de 115200 bps
- UART 16550
  - incorpora un buffer FIFO tanto para los datos de entrada como para los de salida, haciendo que los errores de sobreescritura sean menos frecuentes

Distribución de pines en la UART



Messmer [1]

Distribución de pines en la UART (cont.)

A0-A2, CS0, CS1, $\overline{\text{CS2}}$, CSOUT, $\overline{\text{AS}}$

- líneas de direccionamiento, selección de chip, y reconocimiento de dirección
- junto con el bit DLAB (*divisor latch access bit*) permite la selección de los registros internos

D0-D7, DDIS, DINS, $\overline{\text{DINS}}$, DOUTS, $\overline{\text{DOUTS}}$

- líneas de datos, validez y dirección de los datos

$\overline{\text{OUT1}}$, $\overline{\text{OUT2}}$

- salidas programadas por el usuario

Distribución de pines en la UART (cont.)

INTR

- línea de solicitud de interrupción por parte de la UART

$\overline{\text{BAUDOUT}}$

- señal con una frecuencia que es 16 veces la programada como *baud rate*

RCLK

- reloj de recepción (se debe suministrar una frecuencia 16 veces la del *baud rate* de recepción)
- si se conecta con BAUDOUT la transmisión y la recepción operan a la misma velocidad (caso del PC)

Distribución de pines en la UART (cont.)

$\overline{\text{MR}}$

- reset de todos los registros excepto los buffers de recepción y transmisión y el divisor

XTAL1, XTAL2

- relojes de referencia para la UART

Vcc, GND

- 5 voltios y tierra

$\overline{\text{CTS}}$, $\overline{\text{DSR}}$, $\overline{\text{DTR}}$, $\overline{\text{RI}}$, $\overline{\text{RTS}}$, $\overline{\text{DCD}}$, SIN, SOUT

- líneas habituales de interfaz hacia el periférico, incluyendo la transmisión (SOUT) y la recepción (SIN)

Modelo de programación de la UART

UNIVERSIDAD DE CANTABRIA

La programación de la UART se realiza mediante los registros de los que dispone:

- localizados a partir de una dirección base
- consecutivos con offsets que van de 0-7

Las direcciones base y las interrupciones que se utilizan en un PC son las siguientes:

Interfaz	Dirección base	IRQ
COM1	3f8h	IRQ4
COM2	2f8h	IRQ3
COM3	3e8h	IRQ4 o sólo polling
COM4	2e8h	IRQ3 o sólo polling

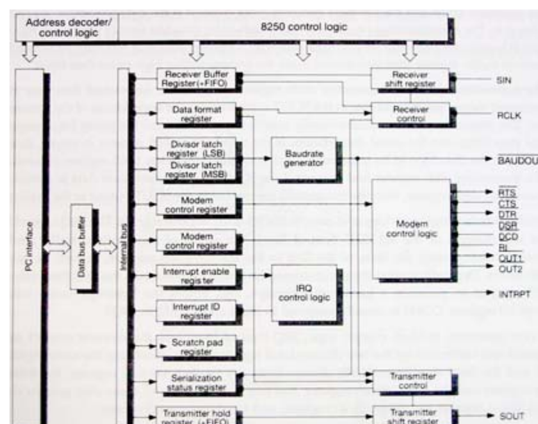
Modelo de programación de la UART (cont.)

UNIVERSIDAD DE CANTABRIA

Registros de la UART	Offset	DLAB	A2	A1	A0
Receiver buffer register	00h	0	0	0	0
Transmitter hold register	00h	0	0	0	0
Interrupt enable register	01h	0	0	0	1
Interrupt identification reg.	02h	-	0	1	0
Data format register	03h	-	0	1	1
Modem control register(output)	04h	-	1	0	0
Serialization status register	05h	-	1	0	1
Modem status register (input)	06h	-	1	1	0
Scratch-pad register	07h	-	1	1	1
Divisor latch register (LSB)	00h	1	0	0	0
Divisor latch register (MSB)	01h	1	0	0	1

Modelo de programación de la UART (cont.)

UNIVERSIDAD DE CANTABRIA



Messmer [1]

Receiver buffer register y Transmitter hold register

Receiver buffer register: recepción de datos

R7	R6	R5	R4	R3	R2	R1	R0

- la lectura de este registro permite la recepción de un nuevo dato, y su llegada puede provocar una interrupción
- si se utilizan tamaños de dato inferiores a 8 bits se deben enmascarar los bits no utilizados

Transmiter hold register: transmisión de datos

T7	T6	T5	T4	T3	T2	T1	T0

- la transferencia del dato al registro de desplazamiento de transmisión puede provocar una interrupción

Interrupt activation register

Programación de los sucesos que pueden generar interrupciones

0	0	0	0	SINP	ERBK	TBE	RxRD

- SINP: a 1 se produce una interrupción si alguna de las líneas de entrada de control cambian de estado:
  - CTS, DSR, DCD, y RI
- ERBK: a 1 produce una interrupción si se produce un error de paridad, sobreescritura, marco o un *break*
- TBE (*Transmitter buffer empty*): a 1 produce una interrupción si se vacía el buffer de transmisión
- RxRD (*Received data ready*): a 1 produce una interrupción si se ha recibido un dato en el buffer de recepción

Interrupt identification register

Identificación de las fuentes que generan la interrupción

0	0	0	0	0	ID1	ID0	$\overline{\text{PND}}$

- los valores de ID1, ID0 identifican la fuente de interrupción y se interpretan como sigue:
  - 00: cambio en una entrada de control (prioridad 3, menor)
  - 01: registro de transmisión vacío (prioridad 2)
  - 10: dato recibido (prioridad 1)
  - 11: error o *break* (prioridad 0, mayor)
- las causas de interrupción se van manejando por orden de prioridad
- PND: a valor 0 indica que hay una interrupción pendiente

Interrupt identification register (cont.)

Las fuentes de interrupción se pueden eliminar del modo siguiente:

- cambio en una entrada de control
  - lectura del registro de estado del modem
- buffer de transmisión vacío
  - escritura del buffer de transmisión (nuevo dato a enviar) o lectura del registro de identificación de interrupciones
- dato recibido
  - lectura del dato del buffer de recepción
- error de recepción o *break*
  - lectura del registro de estado de serialización

Data format register

También denominado *Line Control Register* establece el formato de la transmisión y de la recepción

DLAB	BRK	Paridad (3 bits)	STOP	D-Bit (2 bits)

- DLAB (*divisor latch access bit*): a 1 se permite acceder al divisor, a 0 se accede a los registros de transmisión y recepción, y activación de interrupciones
- BRK:
  - a 1 provoca que la línea SOUT se ponga en estado de *break*, es decir permanentemente a estado lógico 0 (*space*)
  - a 0 hace que SOUT funcione de nuevo para la transmisión de datos
- STOP:
  - a 1 se usan 2 bits de parada; a 0 se usa 1 bit de parada

Data format register (cont.)

- Paridad:
  - 000: sin paridad
  - 001: paridad impar
  - 011: paridad par
  - 101: *mark*
  - 111: *space*
- D-Bit: número de bit de datos
  - 00: 5 bits de datos (la UART selecciona 1 1/2 bits de parada)
  - 01: 6 bits de datos
  - 10: 7 bits de datos
  - 11: 8 bits de datos

Data format register (cont.)

DLAB permite acceder al registro contador de 16 bits con el que la UART genera la frecuencia de referencia

La velocidad de transmisión se obtiene

$$BaudRate = \frac{(FrecuenciaPrincipal)}{16 \cdot Divisor} = \frac{(FrecDeReferencia)}{Divisor}$$

La frecuencia principal en el PC es 1.8432 MHz (relojes conectados a XTAL1 y XTAL2), por lo que la frecuencia de referencia es 115200 Hz

Ej: un valor de 12 en el divisor genera una velocidad de transmisión de 9600 bps

Modem control register

Supervisa la lógica de control del modem

0	0	0	LOOP	$\overline{\text{OUT2}}$	$\overline{\text{OUT1}}$	$\overline{\text{RTS}}$	$\overline{\text{DTR}}$

- **LOOP con valor 1 activa el modo en el que las cuatro salidas de control del modem se cortocircuitan internamente con las entradas:**
  - RTS con $\overline{\text{CTS}}$
  - $\overline{\text{DTR}}$ con $\overline{\text{DSR}}$
  - $\overline{\text{OUT1}}$ con $\overline{\text{RI}}$
  - $\overline{\text{OUT2}}$ con $\overline{\text{DCD}}$
  - la salida SOUT se pone a 1 y la entrada SIN se aisla de la UART
  - los registros de desplazamiento de transmisión y recepción se conectan

Modem control register (cont.)

- **LOOP con valor 0 hace que la UART opere en modo normal; en este caso:**
  - OUT2 y OUT1 controlan los niveles de las correspondientes salidas de la UART
  - en el PC la salida OUT2 se combina con la señal INTR para permitir o no su paso; OUT1 no se usa
  - RTS y DTR controlan las correspondientes salidas $\overline{\text{RTS}}$ y $\overline{\text{DTR}}$ de la UART

Serialization status register

También denominado *Line Status Register* contiene información del estado de la transmisión y de la recepción

0	TXE	TBE	BREK	FRME	PARE	OVRE	RxRD

- **TXE (*transmitter empty*): a 1 indica que tanto el registro de transmisión como el registro de desplazamiento de transmisión están vacíos**
- **TBE (*transmitter buffer empty*): a 1 indica que el registro de transmisión está vacío**
- **BREK: a 1 indica que se ha detectado un *break***
- **RxRD: a 1 indica se ha recibido un dato en el buffer de recepción**

Serialization status register

- **Los otros tres bits a 1 indican que se han producido errores de:**
  - **FRME: marco**
  - **PARE: paridad**
  - **OVRE: sobreescritura**

Modem status register

Contiene el estado de las entradas de la RS-232C

$\overline{\text{DCD}}$	$\overline{\text{RI}}$	$\overline{\text{DSR}}$	$\overline{\text{CTS}}$	DDCD	DRI	DDSR	DCTS

- **un 1 en el bit correspondiente indica que están activas las siguientes señales (un 0 que están inactivas):**
  - **DCD, RI, DSR, y CTS**
- **los otros cuatro bits contienen el estado de cambio en la línea correspondiente desde la última lectura del registro**
  - **un 1 indica que la línea ha cambiado desde la última lectura**
  - **un 0 indica que no ha habido cambio**

Scratch pad register

Buffer temporal de datos

SPR7	SPR6	SPR5	SPR4	SPR3	SPR2	SPR1	SPR0

- se implementa sólo en los modelos 16450/16550
- se puede usar como dato temporal de la rutina de interrupción si no se quiere acceder a la memoria principal
- su uso no afecta en cualquier caso al comportamiento de la UART

Plataformas de Tiempo Real: Dispositivos y Drivers

Tema 5. Interfaces de E/S de datos

- Interfaz serie
- Programación de la interfaz serie
- Interfaz paralelo
- Programación de la interfaz paralelo

Ejemplo de programación de la interfaz serie

Programamos el driver para una dirección base y una interrupción cualquiera; lo especializamos para el COM1

Programamos el puerto con:

- un *baud rate* de 9600 bps
- sin paridad
- un bit de parada
- y 8 bits de datos

Vamos a utilizar el buffer circular proporcionado en `<linux/kfifo.h>`

- proporciona protección de acceso exclusivo con spinlock

Ejemplo de programación de la interfaz serie (cont.)

Utilizamos los datos privados del driver para alojar un buffer intermedio de intercambio de datos:

- se reserva la memoria necesaria en la apertura del dispositivo (no en la instalación del módulo)
- se libera la memoria reservada en el cierre del dispositivo

La técnica de entrada/salida va a ser por interrupciones tanto para la transmisión como para la recepción

- cada carácter recibido o transmitido producirá una interrupción
- la rutina de interrupción deberá atender ambas situaciones y encaminar el byte al buffer correspondiente

Ejemplo de programación de la interfaz serie (cont.)

Los puntos de entrada de lectura y escritura toman o depositan datos de cualquier tamaño de sus respectivos buffers:

- utilizan los buffers intermedios de la estructura de datos privada para intercambiar datos entre las colas circulares y los buffers de usuario

Se puede programar el punto de entrada ioctl para modificar, por ejemplo, las características de la comunicación:

- tamaño de los datos, paridad, bits de parada, o *baud rate*

Driver serie: includes

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include "serie.h"
#include "serie_c.h"    // para la implementacion de ioctl

#include <linux/ioport.h>
#include <linux/interrupt.h>
#include <linux/spinlock.h>
#include <linux/kfifo.h>
#include <asm/uaccess.h>
#include <asm/io.h>

MODULE_LICENSE("GPL");
```

Driver serie: constantes

```
// Direccion I/O base de los puertos serie COM1 y COM2
#define COM1_BASE  0x3f8
#define COM2_BASE  0x2f8

// Direccion base de la instalacion
#define SERIE_BASE  COM1_BASE

// Numero de puertos de la UART
#define NUM_REGS    8

// IRQ puerto serie
#define COM1_IRQ   4
#define COM2_IRQ   3

// IRQ de instalacion
#define SERIE_IRQ COM1_IRQ
```


Driver serie: constantes (cont.)

```
// Offset de los registros del puerto serie
#define RX   0   // Registro de recepcion de dato
#define TX   0   // Registro de transmision de dato
#define DLL  0   // Divisor de frecuencia menos significativo
#define DLM  1   // Divisor de frecuencia mas significativo
#define IER  1   // Registro de habilitacion de interrupciones
#define IIR  2   // Registro de identificacion de interrupciones
#define LCR  3   // Registro de control de linea
#define MCR  4   // Registro de control de Modem
#define LSR  5   // Registro de estado de linea
#define MSR  6   // Registro de estado de Modem

// Dimension maxima de los buffers
#define MAX 100

// Dimension maxima de las kfifos
#define LONG_FIFO 1000
```


Driver serie: constantes (cont.)

```
// Palabras de control
#define C9600H  0x00   // Cuenta para 9600 baudios (Reg. DLM)
#define C9600L  0x0c   // Cuenta para 9600 baudios (Reg. DLL)
#define NP1S8B  0x03   // Sin paridad, 1 bit stop y 8 bits (Reg. LCR)
#define RCsp28  0x07   // Sin paridad, 2 bit stop y 8 bits (Reg. LCR)
#define DLAB    0x80   // Acceso a DLL y DLM (Reg. LCR)
#define EDA     0x03   // Habilita interrupciones de transmision
                       // y recepcion
#define OUT2E   0x08   // Habilita paso de interrupciones

// Mascaras del registro de estado de linea
#define TR_EMPTY     0x20    // registro de transmision vacio
#define RECEIVED     0x01    // registro de recepcion lleno
```

Driver serie: datos del dispositivo

```
// Datos del dispositivo, incluyendo la estructura cdev
struct datos_serie {
  struct cdev *cdev;         // Character device structure
  dev_t dev;                 // informacion con el numero mayor y menor
  struct kfifo *fifo_rec;    // kfifo de recepcion
  spinlock_t lock_rec;       // lock para la kfifo de recepcion
  struct kfifo *fifo_trans;  // kfifo de transmision
  spinlock_t lock_trans;     // lock para la kfifo de transmision
};

static struct datos_serie datos;

// Datos privados del dispositivo
struct datos_privados {
  char buffer_rec[MAX];      // buffer intermedio de recepcion
  char buffer_trans[MAX];    // buffer intermedio de transmision
};
```


Driver serie: instalación

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

  // ponemos los puntos de entrada
  serie_fops.open=serie_open;
  serie_fops.release=serie_release;
  serie_fops.write=serie_write;
  serie_fops.read=serie_read;
  serie_fops.ioctl=serie_ioctl;

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


Driver serie: instalación (cont.)

```
  //Inicializamos las kfifos
  datos.fifo_rec=kfifo_alloc(LONG_FIFO,0,&datos.lock_rec);
  kfifo_reset(datos.fifo_rec);
  datos.fifo_trans=kfifo_alloc(LONG_FIFO,0,&datos.lock_trans);
  kfifo_reset(datos.fifo_trans);

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

Driver serie: instalación (cont.)

```
// Reserva el rango de direcciones de I/O para el COM1 o COM2
if (check_region (SERIE_BASE, NUM_REGS)!=0) {
  printk(KERN_WARNING "serie> (init_module) Borrando region I/O\n");
  release_region (SERIE_BASE, NUM_REGS);
}
if (request_region (SERIE_BASE, NUM_REGS, "serie")==0) {
  result = -EFAULT;
  printk(KERN_WARNING
         "serie> (init_module) Error direcciones I/O\n");
  goto error_reserva_IO;
}

printk(KERN_INFO
         "serie> (init_module) Res. Dir. I/O. Rango:%x..%x\n",
         SERIE_BASE, SERIE_BASE + NUM_REGS - 1);
```


Driver serie: instalación (cont.)

```
// Inicializa el puerto serie
outb_p(NP1S8B + DLAB, SERIE_BASE + LCR);
outb_p(C9600L, SERIE_BASE + DLL);
outb_p(C9600H, SERIE_BASE + DLM);
outb_p(NP1S8B, SERIE_BASE + LCR);
basura = inb_p (SERIE_BASE + RX);

// Instala manejador de interrupcion
result = request_irq(SERIE_IRQ, manejador_interrupcion,
                     0, "serie", NULL);
if (result) {
  printk(KERN_WARNING "serie> (init_module) Error en manejador\n");
  goto error_request_irq;
}
printk(KERN_INFO "serie> (init_module) Instalado manejador %d\n",
         SERIE_IRQ);
```


Driver serie: instalación (cont.)

```
// Habilita interrupciones en la interfaz
outb_p(EDA, SERIE_BASE + IER);
outb_p(OUT2E, SERIE_BASE + MCR);

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

Driver serie: instalación/errores

```
 // Errores
 error_request_irq:
  release_region (SERIE_BASE, NUM_REGS);

 error_reserva_IO:
  cdev_del(datos.cdev);

 error_instalacion_dev:
  unregister_chrdev_region(datos.dev,1);
  kfifo_free(datos.fifo_rec);
  kfifo_free(datos.fifo_trans);

 error_reserva_numeros:
  return result;
}
```


Driver serie: desinstalación

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

  // Libera memoria de las kfifos
  kfifo_free(datos.fifo_rec);
  kfifo_free(datos.fifo_trans);

  // Inhabilita interrupciones en la interfaz
  outb_p(0, SERIE_BASE + IER);

  // Libera direcciones de I/O
  release_region (SERIE_BASE, NUM_REGS);

  // Desinstala manejador de interrupcion
  free_irq(SERIE_IRQ, NULL);

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


Driver serie: open/release

```
int serie_open(struct inode *inodep, struct file *filp) {
  // Reserva de memoria para los buffers en area privada
  filp->private_data=
    kmalloc(sizeof(struct datos_privados),GFP_KERNEL);
  if (filp->private_data==NULL) {
    printk( KERN_WARNING
             "serie> (init_module) Error, no hay memoria\n");
    return -ENOMEM;
  }

  printk(KERN_INFO "serie> (open) Abierto dispositivo\n");
  return 0;
}

int serie_release(struct inode *inodep, struct file *filp) {
  kfree(filp->private_data);
  printk(KERN_INFO "serie> (release) Cerrado dispositivo\n");
  return 0;
}
```

Driver serie: read

```
ssize_t serie_read (struct file *filp, char *buff,
                     size_t count, loff_t *offp)
{
  ssize_t cuenta;
  unsigned int copied;
  unsigned long not_copied;
  struct datos_privados *buffers=
          (struct datos_privados *)filp->private_data;

  printk(KERN_INFO "serie> (read) count=%d\n",(int) count);
  cuenta=(ssize_t) count;
  if (cuenta>MAX) {
    cuenta=MAX;
  }

  // Se copian los datos de la kfifo al buffer privado
  copied=kfifo_get(datos.fifo_rec,buffers->buffer_rec,
                  (unsigned int)cuenta);
  cuenta= (ssize_t)copied;
```


Driver serie: read (cont.)

```
  // Se pasan los datos del buffer privado al del usuario
  not_copied=copy_to_user(buff,buffers->buffer_rec,
                          (unsigned long)cuenta);
  if (not_copied!=0) {
    printk(KERN_WARNING
          "serie> (read) AVISO, no se leyeron los datos\n");
    return(-EFAULT);
  }

  return cuenta;
}
```


Driver serie: write

```
ssize_t serie_write (struct file *filp, const char *buff,
                     size_t count, loff_t *offp)
{
  ssize_t cuenta;
  unsigned int copied;
  unsigned long not_copied;
  unsigned int long_init;
  char byte;
  struct datos_privados *buffers=
              (struct datos_privados *)filp->private_data;

  printk(KERN_INFO "serie> (write) count=%d\n",(int) count);
  cuenta=(ssize_t) count;
  if (cuenta>MAX) {
    cuenta=MAX;
  }
```

Driver serie: write (cont.)

```
  not_copied=copy_from_user(buffers->buffer_trans,buff,
                            (unsigned long)cuenta);
  if (not_copied!=0) {
   printk(KERN_WARNING "serie> (write) AVISO, no se escribio bien\n");
    return(-EFAULT);
  }
  // Posible uso de un mutex
  long_init=kfifo_len(datos.fifo_trans);
  copied=kfifo_put(datos.fifo_trans,buffers->buffer_trans,
                   (unsigned int)cuenta);
  cuenta= (ssize_t)copied;

  // Si la cola estaba vacia se envia el primer byte
  if (long_init==0) {
    kfifo_get(datos.fifo_trans,&byte,1);
    outb_p(byte, SERIE_BASE + TX);
  }
  return cuenta;
}
```


Driver serie: manejador de interrupción

```
irqreturn_t manejador_interrupcion (int irq, void *dev_id,
                                    struct pt_regs *pt)
{
  char byte;

  switch (inb_p (SERIE_BASE + IIR)&0x07){

    case 0x06:
      printk("--->Interrupcion %d (error)...\n", irq);
      break;
    case 0x04:
      // recepcion
      byte = inb_p (SERIE_BASE + RX);
      kfifo_put(datos.fifo_rec,&byte,1);
      // Escribe el caracter recibido
      printk("--->Interrupcion %d (recibido puerto serie)...%c\n",
             irq,byte);
      break;
```


Driver serie: manejador de interrupción (cont.)

```
    case 0x02:
      // transmision
      if (kfifo_len(datos.fifo_trans)!=0) {
        kfifo_get(datos.fifo_trans,&byte,1);
        outb_p(byte, SERIE_BASE + TX);
        // Escribe el caracter transmitido
       printk("--->Interrupcion %d (transmitido puerto serie)...%c\n",
              irq,byte);
      }
      break;
    default:
      printk("--->Interrupcion %d (cambio en linea)...\n", irq);
  }
  printk("--->Interrupcion %d (puerto serie)...\n", irq);
  return IRQ_HANDLED;
}
```

Programas de prueba

Lee

- lee del dispositivo strings de hasta 20 caracteres a intervalos de 2 segundos
- finaliza con una señal (p.e., Ctrl+c)
    - se debe cerrar el fichero

Escribe

- muestra un diálogo en el que se pide al usuario un string de hasta 100 caracteres y lo escribe en el dispositivo
- cuando el primer carácter es un ’9’ finaliza, en caso contrario pide un nuevo string

Comentarios sobre el ejemplo

Se implementa la línea serie como un canal de envío y recepción de bytes

- el uso de las colas circulares hace que no se pierdan datos

Se realiza una doble copia de los datos: de la cola circular al buffer intermedio y de éste al buffer de usuario

- se podría hacer una implementación más eficiente evitando el uso de las *kfifos*
- se podría utilizar una cola circular propia en la que tuviéramos acceso a los punteros para hacer la copia directa hacia o desde el espacio de usuario

Comentarios sobre el ejemplo (cont.)

Otra posibilidad es que el driver esté hecho a medida para alguna aplicación concreta en la que, por ejemplo, se transmitan mensajes con un protocolo de comienzo y finalización

- el driver podría utilizar colas de mensajes
- *read* y *write* sólo podrían manejar los mensajes completos
- podría tener alguna opción bloqueante de envío y recepción de mensajes
    - esperando a que haya un mensaje para leer
    - esperando a que haya espacio para escribir
    - se podría hacer a través de *ioctl*

Ejemplo de uso de un tasklet

UNIVERSIDAD DE CANTABRIA

Tomando como base el driver para el puerto serie previamente realizado, vamos a introducir las siguientes modificaciones:

- descargar el código correspondiente a la recepción de un carácter en un *tasklet*:
  - antes de encolar el carácter comprueba si es una letra (mayúscula o minúscula), un número o un espacio
  - si no es uno de los caracteres permitidos lo sustituye por un guión
  - además cambia todas las letras a mayúsculas
- hacer que la operación de lectura sea bloqueante
- el *tasklet* debe despertar a cualquier proceso de usuario que esté esperando en una lectura

Mostramos sólo las diferencias con el driver serie

Serie con tasklet: datos del dispositivo

UNIVERSIDAD DE CANTABRIA

```
// Datos del dispositivo, incluyendo la estructura cdev
struct datos_serie {
  struct cdev *cdev;          // Character device structure
  dev_t dev;                  // informacion con el numero mayor y menor
  struct kfifo *fifo_rec;     // kfifo de recepcion
  spinlock_t lock_rec;        // lock para la kfifo de recepcion
  struct kfifo *fifo_trans;   // kfifo de transmision
  spinlock_t lock_trans;      // lock para la kfifo de transmision
  struct tasklet_struct tk;   // tasklet
  char byte_tk;               // dato para el tasklet
  wait_queue_head_t sync;     // cola de espera de lectura
};
```

Los datos privados del dispositivo se mantienen igual

Serie con tasklet: instalación

UNIVERSIDAD DE CANTABRIA

```
static int modulo_instalacion(void) {
  // ponemos los puntos de entrada
  // reserva dinamica del número mayor del módulo
  // Inicializamos las kfifos
  // Inicializamos la cola de espera de lectura
  init_waitqueue_head(&datos.sync);
  // instalamos driver
  // Reserva el rango de direcciones de I/O para el COM1 o COM2
  // Inicializa el puerto serie
  // Inicializa el tasklet
  tasklet_init(&datos.tk,serie_tasklet,(unsigned long)&datos.byte_tk);
  // Instala manejador de interrupcion
  // Habilita interrupciones en la interfaz
  // Todo correcto: mensaje y salimos
  // Errores
  ...
  tasklet_kill(&datos.tk);
  ...
}
```

Serie con tasklet: desinstalación/ open/release

```
static void modulo_salida(void) {

  // Elimina dispositivo

  // Elimina el tasklet
  tasklet_kill(&datos.tk);

  // Libera memoria de las kfifos

  // Inhabilita interrupcion en la interfaz

  // Libera direcciones de I/O

  // Desinstala manejador de interrupcion

}
```

Los puntos de entrada *open* y *release* no cambian

Serie con tasklet: read/write

```
ssize_t serie_read (struct file *filp, char *buff,
                    size_t count, loff_t *offp)
{
  ssize_t cuenta;
  unsigned int copied;
  unsigned long not_copied;
  struct datos_privados *buffers=
          (struct datos_privados *)filp->private_data;

  // Comprobacion de la espera sobre los datos de la kfifo
  wait_event_interruptible(datos.sync, kfifo_len(datos.fifo_rec)!=0);

  printk(KERN_INFO "serie> (read) count=%d\n",(int) count);
  cuenta=(ssize_t) count;
  if (cuenta>MAX) {
    cuenta=MAX;
  }
```


Serie con tasklet: read/write (cont.)

```
  // Se copian los datos de la kfifo al buffer privado
  copied=kfifo_get(datos.fifo_rec,buffers->buffer_rec,
                  (unsigned int)cuenta);
  cuenta= (ssize_t)copied;
  // Se pasan los datos del buffer privado al del usuario
  not_copied=copy_to_user(buff,buffers->buffer_rec,
                          (unsigned long)cuenta);
  if (not_copied!=0) {
    printk(KERN_WARNING
          "serie> (read) AVISO, no se leyeron los datos\n");
    return(-EFAULT);
  }

  return cuenta;
}
```

El punto de entrada *write* no cambia

Serie con tasklet: manejador de interrupción

```
irqreturn_t manejador_interrupcion (int irq, void *dev_id,
                                    struct pt_regs *pt)
{
  ...
  switch (inb_p (SERIE_BASE + IIR)&0x07){
  ...
    case 0x04:
      // recepcion
      datos.byte_tk = inb_p (SERIE_BASE + RX);
      // planifica el tasklet
      tasklet_hi_schedule(&datos.tk);
     printk("--->Interrupcion %d (recibido puerto serie)...%c\n", irq,
             datos.byte_tk);
      break;
  ...
  return IRQ_HANDLED;
}
```


Serie con tasklet: el tasklet

```
void serie_tasklet (unsigned long data)
{
  // Se podria usar directamente datos.byte_tk
  char *byte=(char*)data;
  // Controla el dato y lo mete en la kfifo
  if ((*byte>='a')&&(*byte<='z')) {
    *byte=*byte+('A'-'a');
  } else {
    if (!(((*byte>='0')&&(*byte<='9'))||((*byte>='A')&&(*byte<='Z'))||
          (*byte==' '))) {
      *byte='-';
    }
  }
  kfifo_put(datos.fifo_rec,byte,1);
  // Despierta al lector si se da el caso
  wake_up_interruptible(&datos.sync);
  // Escribe el caracter recibido
  printk("--->Tasklet (recibido puerto serie)...%c\n", *byte);
}
```


Comentarios sobre el ejemplo

El ***tasklet*** permite descargar al manejador de interrupción, sin embargo

- si el ritmo de recepción de caracteres es alto, es posible que se puedan producir sobreescrituras en el byte de intercambio de datos entre el manejador y el ***tasklet***
- se podría solucionar utilizando una cola circular
  - habría que sopesar la sobrecarga introducida

El tratamiento de los caracteres que se hace en este ejemplo, es posible que no justifique el uso de un tasklet

- el procesado de los caracteres se podría hacer en la lectura antes de la copia al espacio de usuario

Plataformas de Tiempo Real: Dispositivos y Drivers

Tema 5. Interfaces de E/S de datos

- Interfaz serie
- Programación de la interfaz serie
- Interfaz paralelo
- Programación de la interfaz paralelo

Interfaz paralelo

El uso tradicional del puerto paralelo ha sido la conexión de impresoras:

- como un canal unidireccional de 8 bits en paralelo
- con algunas señales de protocolo en ambas direcciones
  - en particular la posibilidad de generar una interrupción a instancia del dispositivo
- con longitudes de cable menores que las permitidas por el puerto serie (hasta 5 m.), pero a mayores velocidades (desde 150 KBps)

Los puertos paralelos actuales se pueden configurar para la transmisión bidireccional

El estándar IEEE 1284

Es un estándar del puerto paralelo aprobado en 1994 que define sus características físicas (mecánicas y eléctricas), e incluso los modos de transferencia de datos:

- puede soportar modos de operación bidireccionales de 4 y 8 bits
- estandariza el protocolo entre el PC y el periférico que controla (principalmente la impresora)
- sólo estandariza el hardware: conectores de 25 y 36 pines, señales TTL (0-5 v.)

Otros comités de estandarización han ido más allá tratando de estandarizar el software que se ejecuta sobre el puerto paralelo:

- p.e., el EPP (*Enhanced Parallel Port*) propone el uso del puerto paralelo a través de la BIOS del PC

El estándar IEEE 1284 (cont.)

UC UNIVERSIDAD DE CANTABRIA

Este estándar define cuatro tipos de puerto:

Tipo	Modo In	Modo Out	Características
`SPP (Standard Parallel Port)`	`Nibble`	`Compatible`	`4 bit In/8 bits Out`
`Bidireccional`	`Byte`	`Compatible`	`8 bits de I/O`
`EPP (Enhanced Parallel Port)`	`EPP`		`8 bits de I/O`
`ECP (Enhanced Capabilities P)`	`ECP`		`8 bits de I/O, DMA`

- El significado de los modos es el siguiente:
  - Nibble: 4 bits, sólo entrada, a 50 KBps
  - Byte: 8 bits, sólo entrada, a 150 KBps
  - Compatible: 8 bits, sólo salida, a 150 KBps
  - EPP: 8 bits I/O, a 500 KBps-2.77 MBps
  - ECP: 8 bits I/O, a 500 KBps-2.77 MBps

El estándar IEEE 1284 (cont.)

UC UNIVERSIDAD DE CANTABRIA

SPP:

- coincide con el puerto paralelo original del PC (Centronics)
- 8 bits de datos unidireccionales de salida
- se pueden usar 4 líneas de control como entradas de datos (modo *Nibble*)

Bidireccional:

- se le conoce también como tipo PS/2 o tipo extendido
- permite una comunicación bidireccional real con el dispositivo
- define algunos pines no usados del conector paralelo SPP
- define un bit de estado que indica la dirección de la comunicación

El estándar IEEE 1284 (cont.)

UC UNIVERSIDAD DE CANTABRIA

EPP:

- también conocido como puerto de modo rápido (*fast mode*)
- normalmente se integra en el chip *Super I/O* o en el *South Bridge* de la placa base
- opera a velocidades parecidas al bus ISA por lo que permite la conexión de dispositivos más rápidos:
  - discos, cintas, e incluso adaptadores de red

El estándar IEEE 1284 (cont.)

ECP:

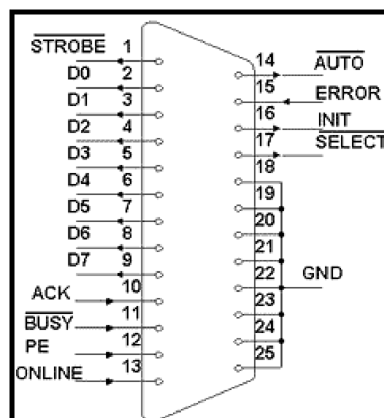
- pensado para soportar periféricos que requieren mayor rendimiento que los tradicionales:
  - impresoras de alta velocidad, scanners, unidades Zip, o CD-ROMs
- la mayoría de los chips que integran los PCs actuales soportan ECP
  - alternativamente permiten elegir el modo de uso del puerto paralelo como EPP o Bidireccional
  - la recomendación es que se use como ECP

A continuación veremos los significados de las líneas del puerto paralelo básico en términos de control de una impresora

- para otros dispositivos pueden tener otro significado

Líneas del puerto paralelo

UC UNIVERSIDAD DE CANTABRIA



Líneas del puerto paralelo (cont.)

UC UNIVERSIDAD DE CANTABRIA

Entradas:

- ACK: con una transición de alto a bajo la impresora reconoce la recepción de un caracter
  - puede provocar una interrupción
- BUSY: si está activa indica que la impresora no puede recibir más caracteres
- PE: indica que la impresora no tiene papel
- ONLINE: indica si la impresora está en línea
- ERROR: con valor bajo indica que la impresora tiene algún tipo de error

Líneas del puerto paralelo (cont.)

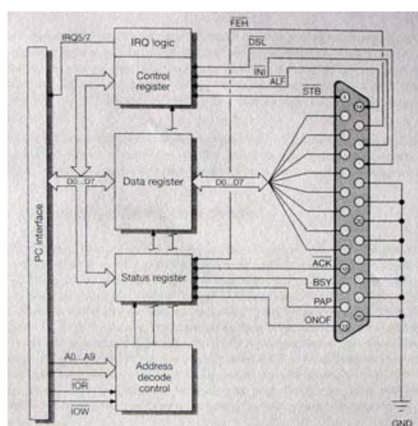
UNIVERSIDAD DE CANTABRIA

Salidas:

- STROBE: una transición de alto a bajo indica a la impresora que acepte un carácter de las líneas de datos
- Datos de salida: pines 2-9
- AUTO: a nivel bajo indica a la impresora que haga un salto de línea automático después de cada línea impresa (CR recibido)
- INIT: una transición de alto a bajo indica a la impresora que debe hacer un inicialización
- SELECT: indica si el dispositivo está seleccionado
  - la mayoría de las impresoras no lo usan y se consideran siempre seleccionadas

Modelo de programación del puerto paralelo

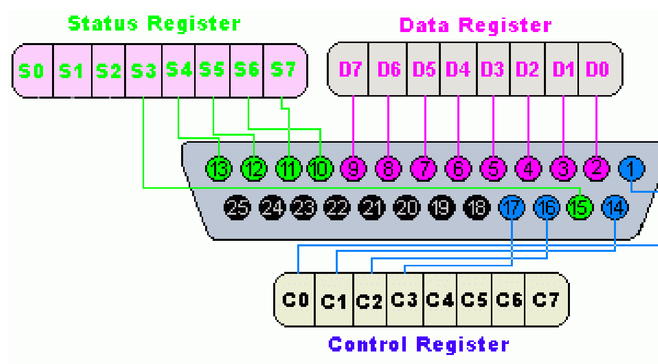
UNIVERSIDAD DE CANTABRIA



Messmer [1]

Modelo de programación del puerto paralelo (cont.)

UNIVERSIDAD DE CANTABRIA



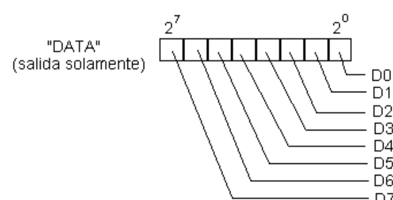
Recursos del puerto paralelo

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

Registro de datos



- en el modo SPP es un registro unidireccional de salida que mantiene los valores de las líneas de datos
- en los otros modos tiene sentido tanto su lectura como su escritura

Registro de control

			IRQ	DSL	$\overline{\text{INI}}$	ALF	STR

- es un registro de lectura y escritura
  - en la lectura sólo se usan los cuatro bits menos significativos
- los cuatro bits menos significativos implementan directamente la funcionalidad de las salidas del puerto
  - STR: a 1 hay transferencia a la impresora
  - ALF: a 1 salto de línea automático
  - INI: a 1 operación normal, a 0 inicialización
  - DSL: a 1 la impresora está seleccionada
- IRQ: a valor 1 habilita la generación de interrupción por parte del puerto con un flanco de bajada de la línea ACK

Registro de estado

$\overline{\text{BSY}}$	$\overline{\text{ACK}}$	PAP	ON	$\overline{\text{ERR}}$			

- los cinco bits más significativos implementan directamente la funcionalidad de las entradas del puerto
  - BSY: a valor 0 indica que la impresora está ocupada
  - ACK: a valor 0 indica que la impresora ha recibido el caracter; si está habilitada la interrupción, además se habrá producido una, si no lo está, se debe utilizar este bit de estado
  - PAP: a 1 indica que no hay papel
  - ON: a 1 indica que la impresora está en línea
  - ERR: a 0 indica un error en la impresora

Utilización del puerto paralelo

Aunque su uso principal es la conexión de impresoras, su sencillez y el control total y directo de sus líneas, abre la posibilidad a otro tipo de aplicaciones:

- intercambio de datos entre computadores
  - conectando directamente las líneas de datos
  - implementando un protocolo asíncrono de envío y reconocimiento con las líneas STROBE, INIT y SELECT conectadas a ACK, ONLINE y BUSY
- aplicaciones de control de cualquier dispositivo externo
  - utilizado como una interfaz de entradas y salidas digitales con posibilidad de generar interrupciones

Otros modos de operación

Las líneas del puerto paralelo y los bits de los registros de control y estado tienen diferentes significados en los distintos modos de operación del puerto:

- Nibble
- Bidireccional
- Compatible (es el que hemos visto)
- EPP
- ECP

Modo Nibble

UNIVERSIDAD DE CANTABRIA

Registro de estado

Dato 3	PTR Clk	Dato 2	Dato 1	Dato 0			

Registro de control

			IRQ	Active		Host busy	

El registro de datos no se usa ya que es una transferencia unidireccional hacia el PC

El significado de los bits corresponde a las líneas: *Host busy* (AUTO), *PTR Clk* (ACK), y *Active* (SELECT)

Modo bidireccional

UNIVERSIDAD DE CANTABRIA

Registro de estado

PTR busy	PTR Clk	Ack Data		Data Avail.			

Registro de control

		Data Direct.	IRQ	Active		Host busy	Host Clk

El registro de datos es bidireccional y se interpreta de acuerdo con la programación del bit *Data Direct.* del registro de control

- **a 0 se usa como salida y a 1 como entrada**

Modo bidireccional (cont.)

UNIVERSIDAD DE CANTABRIA

Los significados de los bits corresponden a las líneas del puerto, que ahora implementan un protocolo diferente:

- ***Ptr Clk*** **(ACK)**
- ***Ptr busy*** **(BUSY)**
- ***Ack Data*** **(PE)**
- ***Data Avail.*** **(ERROR)**
- ***Active*** **(SELECT)**
- ***Host busy*** **(AUTO)**
- ***Host Clk*** **(STROBE)**

Modo EPP

Permite hacer transferencias bidireccionales de 8 bits tanto de datos como de direcciones

- se podría acceder hasta a 256 dispositivos

Se puede usar directamente como conexión entre dos dispositivos

Los accesos de lectura y escritura desde el PC son controlados directamente por la interfaz con un protocolo particular EPP

Para direccionar un dispositivo se utiliza el registro de datos

- pero se activa una línea diferente hacia el dispositivo indicando que es una dirección

Modo EPP (cont.)

Los tres registros de datos, estado y control se usan en el modo habitual del SPP (modo compatible) pero el usuario no controla el protocolo

Tiene además 5 registros adicionales con offsets de 03h a 07h

- offset 03h: registro de direcciones EPP
- offset 04h: registro de datos 0 EPP (8, 16, 32 bits)
- offset 05h: registro de datos 1 EPP (16, 32 bits)
- offset 06h: registro de datos 2 EPP (32 bits)
- offset 07h: registro de datos 3 EPP (32 bits)

Modo ECP

Como el EPP puede direccionar varios dispositivos y sus principales características son:

- hasta 128 dispositivos
- utiliza compresión de datos
- dispone de una FIFO de 16K con DMA y capacidad de interrupción
- el protocolo implementa ciclos de transferencia de datos y de comandos en ambas direcciones
  - el tipo de ciclo lo distingue una línea del protocolo
- la transferencia de datos usa un protocolo diferente a los otros modos

Modo ECP (cont.)

UC UNIVERSIDAD DE CANTABRIA

- los comandos son dos y consisten en:
  - el direccionamiento de una unidad conectada
  - o el envío del contador para la compresión/descompresión
- el bit más significativo del registro de datos distingue entre los dos comandos y los restantes bits representan la longitud (a 0) o el canal de direccionamiento del dispositivo (a 1)
- el ECP permite implementar también otros modos de operación
- el modelo de registros es el siguiente
  - tiene los tres habituales del SPP, aunque su significado y uso depende del modo de operación
  - y otros 6 adicionales seleccionados con offsets de 400h a 402h

Modo ECP (cont.)

UC UNIVERSIDAD DE CANTABRIA

Direcciones y significado de los registros

Registro	Offset	Acceso	Modo-ECP	Función
Datos	000h	R/W	0-1	Registro de datos
ECP-A-FIFO	000h	R/W	3	Direcciones de la FIFO
DSR	001h	R/W	Todos	Registro de estado
DCR	002h	R/W	Todos	Registro de control
C-FIFO	400h	R/W	2	Datos del puerto paralelo FIFO
ECP-D-FIFO	400h	R/W	3	Datos de la FIFO
T-FIFO	400h	R/W	6	Test de la FIFO
CNFG-A	400h	R	7	Registro de configuración A
CNFG-B	401h	R/W	7	Registro de configuración A
ECR	402h	R/W	Todos	Registro de control extendido

Modo ECP (cont.)

UC UNIVERSIDAD DE CANTABRIA

Extended control register

Bit	Significado
7:5	000 SPP (Compatible ISA)
	001 Bidireccional (compatible PS/2)
	010 Centronics rápido (Compatible ISA FIFO)
	011 ECP
	100 Reservado (EPP)
	101 Reservado
	110 Prueba
	111 Configuración
4	Deshabilita interrupciones de ERROR
3	Habilita DMA
2	Deshabilita el servicio de interrupción FIFO/Terminal Count
1	Sólo lectura (FIFO lleno)
0	Sólo lectura (FIFO vacío)

Plataformas de Tiempo Real: Dispositivos y Drivers

UC UNIVERSIDAD DE CANTABRIA

Tema 5. Interfaces de E/S de datos

- Interfaz serie
- Programación de la interfaz serie
- Interfaz paralelo
- Programación de la interfaz paralelo

Ejemplo: entrada/salida digital

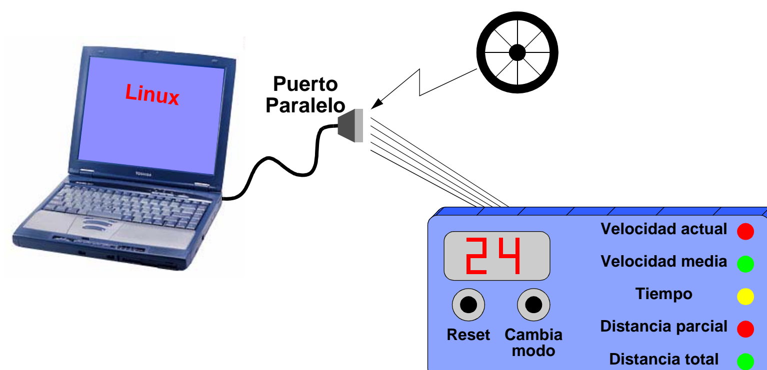
UC UNIVERSIDAD DE CANTABRIA

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

UC UNIVERSIDAD DE CANTABRIA



Hardware del dispositivo velocímetro

UC UNIVERSIDAD DE CANTABRIA

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.)

UC UNIVERSIDAD DE CANTABRIA

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.)

UC UNIVERSIDAD DE CANTABRIA

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.)

UC UNIVERSIDAD DE CANTABRIA

- **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 sensación de que el sistema no reacciona
- **un periodo razonable para dar sensación de instantaneidad puede ser 50 ms**

Vamos a implementar dos versiones:

- **una haciendo uso de la interrupción**
- **la otra por *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)1000)/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

Diseño detallado: Velocímetro con interrupciones

Los puntos de entrada del driver son iguales

Dos temporizadores para la realización de las actividades periódicas:

- temporizador de control de botones
- temporizador de presentación de datos

Un manejador de interrupción con la funcionalidad que tenía el temporizador de la versión de *polling*

- la línea que sale del comparador genera *glitches* y produce varias interrupciones por cada paso de un radio de la rueda
  - se puede solucionar en el hardware
  - se puede solucionar inhibiendo interrupciones por un tiempo y programando un trabajo en una cola que las vuelva a habilitar

Velocímetro-int

UNIVERSIDAD DE CANTABRIA

Sólo mostramos los cambios respecto del caso en el que no tenemos interrupciones

En los accesos al registro de control habrá que tener en cuenta si las interrupciones están habilitadas o no (operación *or* con una variable con el bit activado o no)

Permanecen sin cambios:

- el temporizador de control de los botones
- el temporizador de presentación
- las funciones auxiliares
- todos los puntos de entrada

Velocímetro-int: includes

UNIVERSIDAD DE CANTABRIA

```
//  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 <linux/interrupt.h>
#include <linux/workqueue.h>
#include "velocimetro.h"

MODULE_LICENSE("GPL");
```


Velocímetro-int: constantes

UNIVERSIDAD DE CANTABRIA

```
// 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

// Palabras de control para la interrupción
#define INT      0x10
#define NO_INT      0x00
```

Velocímetro-int: constantes (cont.)

```
// IRQ puerto paralelo
#define LPT1_IRQ    7

// Mascaras de estado
#define RESET    0x80
#define MODO     0x20

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

// Periodo del temporizador
#define PERIODO    50
// Periodo del temporizador de presentacion de datos
#define PERIODO_PRES   500
// Retraso de habilitacion de la interrupcion
#define RETRASO_INT   10

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


Velocímetro-int: 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
  char estado_int;             // habilita o inhibe interr.en disp.
  struct workqueue_struct *q;  // cola de trabajo para activar la int.
  struct work_struct w;        // trabajo para la cola
  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;
```


Modo y estado de interrupción

En todas las escrituras sobre el registro de control habrá que usar la combinación

- modo + estado de interrupción

```
outb(datos.modo|datos.estado_int,LPT1_BASE+CONTROL);
```

Velocímetro-int: instalación

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

  // ponemos los puntos de entrada

  // reserva dinamica del número mayor del módulo
  // numero menor = cero, numero de dispositivos =1

  // instalamos driver

  // reserva el rango de direcciones de I/O

  // inicializamos datos del temporizador

  // inicializamos datos del temporizador de presentacion
```


Velocímetro-int: 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-int: instalación (cont.)

```
  datos.estado_int=INT;
  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

  // creamos el temporizador de presentacion y lo activamos

  // Inicializa la cola de trabajos
  datos.q=create_workqueue("cola_velocimetro");
  INIT_WORK(&datos.w,manejador_cola,NULL);
```

Velocímetro-int: instalación (cont.)

```
  // Instala manejador de interrupcion
  result = request_irq(LPT1_IRQ, manejador_interrupcion,
                       0, "velocimetro", NULL);
  if (result) {
    printk(KERN_WARNING "veloc> (init_module) Error en manejador\n");
    goto error_request_irq;
  }
  printk(KERN_INFO "veloc> (init_module) Instalado manejador %d\n",
         LPT1_IRQ);

  // inicializa registros de control y datos
  outb(datos.modo|datos.estado_int,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-int: instalación (cont.)

```
  // Errores
error_request_irq:
  atomic_set(&datos_timer.apagando,1);
  del_timer_sync(&datos_timer.timer);
  atomic_set(&datos_pres.apagando,1);
  del_timer_sync(&datos_pres.timer);
  destroy_workqueue(datos.q);
error_buffer:
  kfree(datos.mensaje);
error_mensaje:
  release_region (LPT1_BASE, NUM_REGS);
error_reserva_IO:
  cdev_del(datos.cdev);
error_instalacion_dev:
  unregister_chrdev_region(datos.dev,1);
error_reserva_numeros:
  return result;
}
```


Velocímetro-int: 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 presentacion de datos
  atomic_set(&datos_pres.apagando,1);
  del_timer_sync(&datos_pres.timer);
  // Elimina la cola de trabajos
  destroy_workqueue(datos.q);
  // Libera direcciones de I/O
  release_region (LPT1_BASE, NUM_REGS);
  // Desinstala manejador de interrupcion
  free_irq(LPT1_IRQ, NULL);
  printk( KERN_INFO "veloc> (cleanup_module) descargado OK\n");
}
```

Velocímetro: manejador de interrupción

```
irqreturn_t manejador_interrupcion
               (int irq, void *dev_id, struct pt_regs *pt)
{
  char DEBUG = 1;

  // inhibimos la interrupcion en el dispositivo
  datos.estado_int=NO_INT;
  outb(datos.modo|datos.estado_int,LPT1_BASE+CONTROL);

  // incrementamos contador de distancia
  atomic_inc(&datos.distancia);
  atomic_inc(&datos.distancia_t);

  if  (DEBUG) printk("--->Interrupcion %d (velocimetro) d = %d...\n",
                     irq, atomic_read(&datos.distancia));
```


Velocímetro: manejador de interrupción (cont.)

```
  // planifica el trabajo para la cola retrasado
  if (queue_delayed_work(datos.q,&datos.w,RETRASO_INT)==0) {
    if (DEBUG)
      printk("--->Interrupcion (velocimetro) Error en encolado...\n");
  }

  return IRQ_HANDLED;
}
```


Velocímetro: cola de trabajos

```
void manejador_cola (void *data)
{
  char DEBUG = 1;

  if (DEBUG)
    printk(KERN_INFO "veloc> (ejecuta trabajo: habilita_int)\n");

  // activa de nuevo la interrupcion
  datos.estado_int=INT;
  outb(datos.modo|datos.estado_int,LPT1_BASE+CONTROL);
}
```

Comentarios sobre el ejemplo

Los comentarios hechos para el caso de *polling* sirven también para éste

Aquí hemos sustituido un temporizador por una interrupción y su función manejadora:

- para evitar la anotación de múltiples interrupciones correspondientes al mismo paso de la rueda, se utiliza un filtro software
- en este caso se inhiben las interrupciones en el dispositivo y se programa un trabajo retrasado en una cola de trabajos
- el trabajo consiste en habilitar de nuevo las interrupciones del dispositivo

Comentarios sobre el ejemplo (cont.)

Otra alternativa de filtro software podría permitir que sucedieran todas las interrupciones

- anotar sólo las buenas
  - incrementar los contadores de paso sólo si ha transcurrido un tiempo mínimo a partir de la última cuenta anotada
  - el tiempo se establecería en función de la velocidad máxima de paso y diferencia entre dos radios
  - no haría falta la cola de trabajos

El manejador de interrupción sería como sigue:

Comentarios sobre el ejemplo (cont.)

```
irqreturn_t manejador_interrupcion
               (int irq, void *dev_id, struct pt_regs *pt)
{
  char DEBUG = 1;

  // actualizamos tiempos para filtro
  datos.ti=jiffies;

  // incrementamos contador de distancia
  if ((datos.ti-datos.ti_ant)> FILTRO){
    datos.ti_ant=datos.ti;
    atomic_inc(&datos.distancia);
    atomic_inc(&datos.distancia_t);
    ...
  }

  return IRQ_HANDLED;
}
```

Conclusión

Para este caso en el que el hardware tiene problemas en el mecanismo de generación de interrupciones

- el método de *polling* se presenta como la mejor solución

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] http://www.national.com/ds/PC/PC16550D.pdf

Bibliografía (cont.)

[6] http://www.beyondlogic.org/spp/parallel.pdf

[7] http://www.beyondlogic.org/epp/epp.pdf

[8] http://www.beyondlogic.org/ecp/ecp.pdf

[9] http://www.beyondlogic.org/serial/serial.pdf