Breve tutorial para escribir drivers en Linux

Xavier Calbet
GULIC (Grupo de Usuarios de LInux de Canarias)
xcalbet@yahoo.es

16 septiembre 2001

"Do you pine for the nice days of Minix-1.1, when men were men and wrote their own device drivers?"
Linus Torvalds

1 Requisitos preliminares

Para realizar drivers para Linux es necesario unos conocimientos previos mínimos que son:

- **Programar en C**. Es necesario conocer de forma relativamente profunda la programación en C como los punteros, funciones de manipulación de bits, etc.
- **Nociones de funcionamiento de los microprocesadores**. Es necesario tener nociones del funcionamiento interno de los microordenadores tales como el direccionamiento de memoria, interrupciones, etc. Sabiendo programar en ensamblador todos estos conceptos resultarán familiares.

Existen varios tipos de dispositivos diferentes en Linux. Por sencillez, en este breve tutorial veremos los dispositivos tipo **char** cargados como módulos. Se utilizará la versión del kernel 2.2.x (en concreto la 2.2.14), aunque los módulos funcionarán con modificaciones pequeñas o nulas en la versión 2.4.x.

2 Espacio de usuario (“user space”) y espacio de kernel (“kernel space”)

Cuando se escriben drivers es importante distinguir entre el espacio de usuario (“user space”) y el espacio de kernel (“kernel space”).

- **Espacio del kernel (“kernel space”)**. El sistema operativo Linux y en especial su kernel se ocupan de gestionar los recursos de hardware de la máquina de una forma eficiente y sencilla, ofreciendo al usuario una interfaz de programación simple y uniforme. El kernel, y en especial sus drivers, constituyen así un puente o interfase entre el programador de aplicaciones para el usuario final y el hardware. Toda subrutina que forma parte del kernel tales como los módulos o drivers se consideran que están en el espacio del kernel (“kernel space”).

- **Espacio de usuario (“user space”)**. Los programas que utiliza el usuario final, tales como las “shell” u otras aplicaciones con ventanas como por ejemplo “kpresenter”, residen en el espacio de usuario (“user space”). Como es lógico estas aplicaciones necesitan interaccionar con el hardware del sistema, pero no lo hacen directamente, sino a través de las funciones que soporta el kernel.

Todo esto se puede visualizar en la Fig. 1.

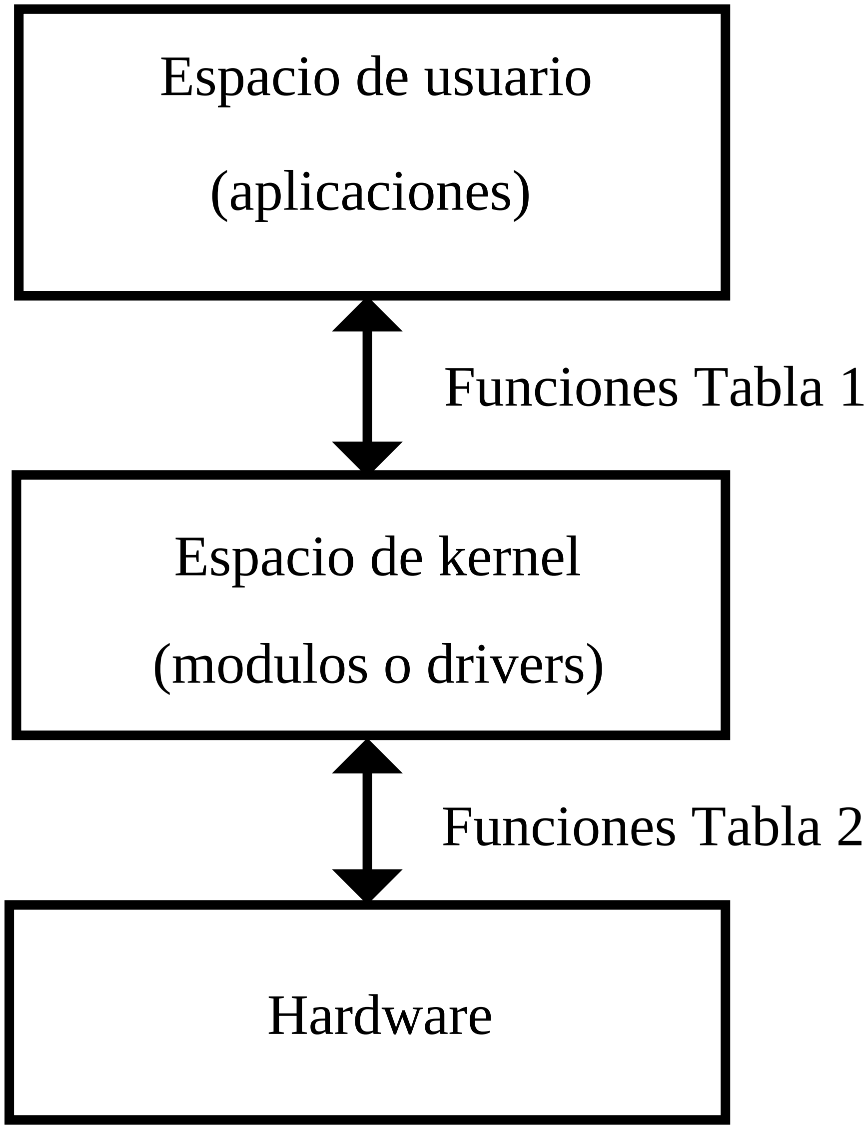

Figura 1. Espacio de usuario donde residen las aplicaciones y espacio de kernel donde residen los módulos o drivers.

3 Funciones de intercambio entre el espacio de usuario y el espacio de kernel

El kernel ofrece una serie de subrutinas o funciones en el espacio de usuario que permiten al programador de aplicaciones finales interaccionar con el hardware. Habitualmente, en sistemas UNIX o Linux, este diálogo se hace a través de las funciones o subrutinas para leer y escribir en ficheros, ya que en estos sistemas los propios dispositivos se ven desde el punto de vista del usuario como ficheros.

Por otro lado, en el espacio del kernel, Linux también ofrece una serie de funciones o subrutinas para por un lado interaccionar directamente, a bajo nivel, con los dispositivos hardware y por otro permite el paso de información desde el kernel al espacio de usuario.

Es común que para cada función en el espacio de usuario que permite el uso de dispositivos o ficheros haya su equivalente en el espacio de kernel que permite el trasvase de información del espacio de kernel al de usuario y viceversa. Esto se puede apreciar en la Tabla 1, la cual de momento se quedará vacía y se irá rellenando a medida que se vayan introduciendo los diferentes conceptos de los drivers.

Eventos	Funciones de usuarios	Funciones del kernel
Carga de módulo		
Abrir dispositivo		
Leer dispositivo		
Escribir dispositivo		
Cerrar dispositivo		
Quitar módulo		

Tabla 1. Eventos de los drivers y sus funciones asociadas de intercambio entre el espacio de kernel y el espacio de usuario.

4 Funciones de intercambio entre el espacio de kernel y el dispositivo hardware

De forma análoga existen funciones en el espacio de kernel que sirven para controlar el dispositivo o trasvasar información entre el kernel y el hardware. La Tabla 2 nos ilustra estos eventos, la cual iremos rellenando a medida que se vayan introduciendo los conceptos.

Eventos	Funciones del kernel
Leer datos	
Escribir datos	

Tabla 2. Eventos de los drivers y sus funciones asociadas entre el espacio de kernel y el dispositivo hardware.

5 El primer driver: carga y descarga del driver en el espacio de usuario

Veremos ahora un driver con el que empezar. Se introducirá dentro del kernel como un módulo.

Para ello se escribe este programa en un fichero llamado **nada.c**

6 ⟨*nada.c* 6⟩≡

```
#define MODULE
#include <linux/module.h>
```

Lo compilamos con el siguiente comando:

$ gcc -c nada.c

(en realidad, para que funcione siempre sin problemas el comando para compilar debería ser

$ gcc -I/usr/src/linux/include -O -Wall -c nada.c ,

donde detrás de **-I** debe aparecer el directorio donde se encuentren los ficheros **include** del kernel)

Este elemental módulo pertenece al espacio de kernel y entrará a formar parte de él cuando se cargue.

Dentro del espacio de usuario podemos cargar el módulo en el kernel en la línea de comandos como usuario **root** con

\# **insmod nada.o**

(si éste no funciona se puede probar **insmod -f nada.o**).

El comando **insmod** permite instalar nuestro módulo en el kernel, aunque éste en concreto no tenga ninguna utilidad.

Podemos comprobar que el módulo ha sido instalado mediante el comando que lista todos los módulos instalados:

\# **lsmod**

Finalmente podemos eliminar el módulo del kernel con el comando

\# **rmmod nada**

Podemos comprobar que el módulo ya no está instalado de nuevo con el comando **lsmod**.

Podemos ver un resumen de todo esto en la Tabla 3.

Eventos	Funciones de usuarios	Funciones del kernel
Carga de módulo	insmod	
Abrir dispositivo		
Leer dispositivo		
Escribir dispositivo		
Cerrar dispositivo		
Quitar módulo	rmmod	

Tabla 3. Eventos de los drivers y sus funciones asociadas de intercambio entre el espacio de kernel y el espacio de usuario.

6 El driver “Hola mundo”: carga y descarga del driver en el espacio de kernel

Normalmente cuando se carga un módulo de un driver de un dispositivo en el kernel se suelen realizar una serie de tareas preliminares como reiniciar el dispositivo, reservar RAM, reservar interrupciones, reservar los puertos de entrada/salida del dispositivo, etc.

Para ello existen dos funciones, **init_module** y **cleanup_module**, dentro del espacio de kernel correspondientes a las del espacio de usuario, **insmod** y **rmmod**, que se utilizan cuando se instala o quita un módulo. Dichas funciones son llamadas por el kernel cuando se realizan estas operaciones.

Veamos un ejemplo práctico con el clásico programa “Hola mundo”:

7 ⟨*hola.c* 7⟩≡

```
#define MODULE
#include <linux/module.h>

int init_module(void) {
  printk("<1>Hola mundo\n");
  return 0;
}

void cleanup_module(void) {
  printk("<1>Adios mundo cruel\n");
}
```

También se ha introducido la función **printk**, la cual es muy similar a la conocida **printf** sólo que opera dentro del kernel. El símbolo **<1>** indica la prioridad del mensaje, se ha especificado una alta prioridad (bajo número) para que el mensaje aparezca por pantalla y no se quede en los ficheros de mensajes del kernel.

Cuando se cargue y descargue el módulo aparecerán en la consola los mensajes que hemos escrito dentro de **printk**. Si no los vemos inmediatamente en la consola podemos escribir el comando **dmesg** en la línea de comandos para verlos o mostrando el fichero de mensajes del sistema con **cat /var/log/syslog**.

En la Tabla 4 se pueden ver estas dos nuevas funciones.

Eventos	Funciones de usuarios	Funciones del kernel
Carga de módulo	insmod	init_module
Abrir dispositivo		
Leer dispositivo		
Escribir dispositivo		
Cerrar dispositivo		
Quitar módulo	rmmod	cleanup_module

Tabla 4. Eventos de los drivers y sus funciones asociadas de intercambio entre el espacio de kernel y el espacio de usuario.

7 El driver completo "memoria": parte inicial del driver

Ahora realizaremos un driver completo, **memoria.c**, utilizando la memoria del ordenador que nos permitirá escribir y leer un carácter en memoria. Este dispositivo, aunque no muy útil, es muy ilustrativo dado que es un driver completo y fácil de implementar ya que no se necesita un dispositivo real.

Para realizar un driver, en la parte inicial de él, tendremos que definir las constantes **MODULE** y **__KERNEL__**. Además tendremos que incluir, con **#include**, una serie de ficheros habituales en los drivers:

8 ⟨*memoria inicio* 8⟩≡

```
/* Definiciones e includes necesarios para los drivers */
#define MODULE
#define __KERNEL__
#include <linux/config.h>
#include <linux/module.h>
#include <linux/kernel.h> /* printk() */
#include <linux/malloc.h> /* kmalloc() */
#include <linux/fs.h> /* everything... */
#include <linux/errno.h> /* error codes */
#include <linux/types.h> /* size_t */
#include <linux/proc_fs.h>
#include <linux/fcntl.h> /* O_ACCMODE */
#include <asm/system.h> /* cli(), *_flags */
#include <asm/uaccess.h> /* copy_from/to_user */

/* Declaracion de funciones de memoria.c */
int memoria_open(struct inode *inode, struct file *filp);
int memoria_release(struct inode *inode, struct file *filp);
ssize_t memoria_read(struct file *filp, char *buf,
  size_t count, loff_t *f_pos);
ssize_t memoria_write(struct file *filp, char *buf,
  size_t count, loff_t *f_pos);
void cleanup_module(void);

/* Estructura que declara las funciones tipicas */
/* de acceso a ficheros */
struct file_operations memoria_fops = {
  read: memoria_read,
  write: memoria_write,
  open: memoria_open,
  release: memoria_release
};                                                    (15)

/* Variables globales del driver */
/* Numero mayor */
int memoria_major = 60;
/* Buffer donde guardar los datos */
char *memoria_buffer;
```

Detrás de los ficheros **#include**, aparecen las declaraciones de las funciones que definiremos en el programa más adelante. Posteriormente aparece la definición de la estructura **file_operations** que define las funciones típicas que se utilizan al manipular ficheros y que veremos después. Finalmente están las variables globales del driver, una de ellas es el NÚMERO MAYOR del dispositivo y la otra un puntero a la región de memoria, **memoria_buffer**, que utilizaremos como almacén de datos del driver.

8 El driver "memoria": conexión de dispositivos con sus ficheros

En UNIX y Linux se accede a los dispositivos desde el espacio de usuario de idéntica forma a como se hace con un fichero. Dichos ficheros suelen colgar del directorio **/dev**.

Para ligar ficheros con dispositivos se utilizan dos números: NÚMERO MAYOR y NÚMERO MENOR. El NÚMERO MAYOR es el que utiliza el kernel para relacionar el fichero con su driver. El NÚMERO MENOR es para uso interno del dispositivo y por simplicidad no lo veremos aquí.

Para conseguir este propósito primero se tiene que crear el fichero que sirva como dispositivo con el comando, como usuario **root**,

 # mknod /dev/memoria c 60 0

donde la **c** significa que se trata de un dispositivo tipo **char**, el **60** es el NÚMERO MAYOR y el **0** el NÚMERO MENOR.

Para ligar un driver con su fichero **/dev** correspondiente se utiliza la función **register_chrdev** que tiene como argumento el NÚMERO MAYOR del dispositivo. Esta función se llama con tres argumentos: NÚMERO MAYOR, cadena de caracteres indicando el nombre del módulo y una estructura **file operations** que asocia esta llamada con las funciones aplicables a ficheros definidas dentro de ella. Se invoca, al instalar el módulo, de esta forma:

9 ⟨*memoria init module* 9⟩≡

```
int init_module(void) {
  int result;

  /* Registrando dispositivo */
  result = register_chrdev(memoria_major, "memoria",
      &memoria_fops);
  if (result < 0) {
    printk(
      "<1>memoria: no puedo obtener numero mayor %d\n",
      memoria_major);
    return result;
  }

  /* Reservando memoria para el buffer */
  memoria_buffer = kmalloc(1, GFP_KERNEL);
  if (!memoria_buffer) {
    result = -ENOMEM;
    goto fallo;
  }
  memset(memoria_buffer, 0, 1);

  printk("<1>Insertando modulo\n");
  return 0;

  fallo:
    cleanup_module();
    return result;
}
```

Además reservamos espacio en memoria para el buffer de nuestro dispositivo, **memoria_buffer**, a través de la función **kmalloc**, la cual es muy similar a la común **malloc**. Finalmente actuamos en consecuencia ante (15)
posibles errores al registrar el NÚMERO MAYOR o al reservar memoria.

9 El driver “memoria”: eliminando el módulo

Para eliminar el módulo, dentro de la función **cleanup_module**, insertamos la función **unregister_chrdev** para liberar el NÚMERO MAYOR dentro del kernel.

10 ⟨*memoria cleanup module* 10⟩≡ (15)

```
void cleanup_module(void) {

  /* Liberamos numero mayor */
  unregister_chrdev(memoria_major, "memoria");

  /* Liberamos memoria del buffer */
  if (memoria_buffer) {
    kfree(memoria_buffer);
  }

  printk("<1>Quitando modulo\n");
}
```

En esta subrutina también liberamos la memoria del buffer del dispositivo para dejar el kernel limpio al quitar el módulo.

10 El driver "memoria": abriendo el dispositivo como fichero

La función en el espacio de kernel correspondiente a la apertura de un fichero en el espacio de usuario (**fopen**) es el miembro **open:** de la estructura **file operations** en la llamada a **register_chrdev**. En este caso se trata de **memoria_open**. Tiene como argumentos una estructura **inode** que pasa información del kernel al driver tal como el NÚMERO MAYOR y el NÚMERO MENOR y una estructura **file** con información relativa a las distintas operaciones que se pueden realizar con el fichero. Ninguna de estas dos funciones las veremos en profundidad aquí.

El kernel lleva un contador de cuantas veces está siendo utilizado un driver. El valor para cada driver se puede ver en la última columna numérica del comando **lsmod**. Cuando se abre un dispositivo para leer o escribir en él, la cuenta de uso se debe incrementar, tal y como aparece en la función **memoria_open**.

Además de esta operación, en la apertura de un fichero, se suelen iniciar las variables pertinentes al driver y el propio dispositivo en si. En este ejemplo, y debido a su extrema sencillez, no realizaremos operaciones de este tipo en dicha función.

Podemos ver la función **memoria_open** a continuación:

11 ⟨*memoria open* 11⟩≡ (15)

```
int memoria_open(struct inode *inode, struct file *filp) {
  /* Aumentamos la cuenta de uso */
  MOD_INC_USE_COUNT;

  /* Exito */
  return 0;

}
```

En la Tabla 5 se puede ver esta nueva función.

Eventos	Funciones de usuarios	Funciones del kernel
Carga de módulo	insmod	init_module
Abrir dispositivo	fopen	file operations: open
Leer dispositivo		
Escribir dispositivo		
Cerrar dispositivo		
Quitar módulo	rmmod	cleanup_module

Tabla 5. Eventos de los drivers y sus funciones asociadas de intercambio entre el espacio de kernel y el espacio de usuario.

11 El driver “memoria”: cerrando el dispositivo como fichero

La función correspondiente a cerrar el fichero en el espacio de usuario (**fclose**), es el miembro **release:** de la estructura **file operations** en la llamada a **register_chrdev**. En este caso se trata de la función **memoria_release**. Tiene como argumentos la estructura **inode** y la estructura **file** anteriores.

Al liberar un fichero del espacio de usuario, se debe decrementar la cuenta de uso para restablecerla a su valor original. El módulo no se podrá descargar del kernel si dicha cuenta es distinta de cero.

Además de esta operación, cuando se cierra un fichero, se suele liberar memoria y variables relacionadas con la apertura del dispositivo. En este caso, a causa de su simplicidad, no se hacen este tipo de operaciones.

La función **memoria_release** aparece a continuación:

12 ⟨*memoria release* 12⟩≡ (15)

```
int memoria_release(struct inode *inode, struct file *filp) {

  /* Decrementamos la cuenta de uso */
  MOD_DEC_USE_COUNT;

  /* Exito */
  return 0;
}
```

En la Tabla 6 se puede ver esta nueva función.

Eventos	Funciones de usuarios	Funciones del kernel
Carga de módulo	insmod	init_module
Abrir dispositivo	fopen	file operations: open
Leer dispositivo		
Escribir dispositivo		
Cerrar dispositivo	fclose	file operations: release
Quitar módulo	rmmod	cleanup_module

Tabla 6. Eventos de los drivers y sus funciones asociadas de intercambio entre el espacio de kernel y el espacio de usuario.

12 El driver “memoria”: leyendo del dispositivo

Para leer el dispositivo mediante la función de usuario **fread** o similar se utiliza el miembro **read:** de la estructura **file operations** en la llamada a **register_chrdev**. En este caso es la función **memoria_read**. Tiene como argumentos una estructura tipo **file**, un buffer, **buf**, del cual leerá la función del espacio de usuario (**fread**), un contador del número de bytes a transferir, **count**, que tiene el mismo valor que el contador habitual de la función de lectura del espacio de usuario (**fread**) y finalmente la posición del fichero donde empezar a leer, **f_pos**.

En este caso simple, la función **memoria_read** procede a transferir el byte del buffer del driver, **memoria_buffer**, al espacio de usuario, mediante la función **copy_to_user**:

13 ⟨*memoria read* 13⟩≡ (15)

```
ssize_t memoria_read(struct file *filp, char *buf,
  size_t count, loff_t *f_pos) {

  /* Transferimos datos al espacio de usuario */
  copy_to_user(buf,memoria_buffer,1);

  /* Cambiamos posición de lectura segun convenga */
  if (*f_pos == 0) {
    *f_pos+=1;
    return 1;
  } else {
    return 0;
  }
}
```

Además cambiamos la posición de lectura en el fichero, **f_pos**. Si dicha posición está en el inicio del fichero, la aumentamos en una unidad y damos como valor de retorno el número de bytes leídos correctamente, 1. Si no estamos en el inicio del fichero devolvemos un fin de fichero, 0, ya que el fichero o dispositivo únicamente almacena un byte.

En la Tabla 7 se puede ver esta nueva función.

Eventos	Funciones de usuarios	Funciones del kernel
Carga de módulo	insmod	init_module
Abrir dispositivo	fopen	file operations: open
Leer dispositivo	fread	file operations: read
Escribir dispositivo		
Cerrar dispositivo	fclose	file operations: release
Quitar módulo	rmmod	cleanup_module

Tabla 7. Eventos de los drivers y sus funciones asociadas de intercambio entre el espacio de kernel y el espacio de usuario.

13 El driver "memoria": escribiendo al dispositivo

Para escribir en el dispositivo mediante la función de usuario **fwrite** o similar se utiliza el miembro **write:** que aparece en la estructura **file operations** en la llamada a **register_chrdev**. En este ejemplo es la función **memoria_write**. Tiene como argumentos una estructura tipo **file**, un buffer, **buf**, al cual escribirá la función del espacio de usuario (**fwrite**), un contador del número de bytes a transferir, **count**, que tiene el mismo valor que el contador habitual de la función de escritura del espacio de usuario (**fwrite**) y finalmente la posición del fichero donde empezar a escribir, **f_pos**.

14 ⟨*memoria write* 14⟩≡ (15)

```
ssize_t memoria_write( struct file *filp, char *buf,
  size_t count, loff_t *f_pos) {
  char *tmp;
  tmp=buf+count-1;
  copy_from_user(memoria_buffer,tmp,1);
  return 1;
}
```

Además resulta útil la función **copy_from_user** que transfiere datos del espacio de usuario al espacio de kernel.

En la Tabla 8 se puede ver esta nueva función.

Eventos	Funciones de usuarios	Funciones del kernel
Carga de módulo	insmod	init_module
Abrir dispositivo	fopen	file operations: open
Leer dispositivo	fread	file operations: read
Escribir dispositivo	fwrite	file operations: write
Cerrar dispositivo	fclose	file operations: release
Quitar módulo	rmmod	cleanup_module

Tabla 8. Eventos de los drivers y sus funciones asociadas de intercambio entre el espacio de kernel y el espacio de usuario.

14 El driver "memoria" al completo

Juntando todas las operaciones que hemos visto obtenemos el driver completo

15 ⟨*memoria.c* 15⟩≡
⟨*memoria inicio* 8⟩
⟨*memoria init module* 9⟩
⟨*memoria cleanup module* 10⟩
⟨*memoria open* 11⟩
⟨*memoria release* 12⟩
⟨*memoria read* 13⟩
⟨*memoria write* 14⟩

Para utilizar este módulo, primero lo compilamos de la misma forma que el **hola.c**. Posteriormente cargamos el módulo con

insmod memoria.o

Conviene también desproteger el dispositivo

chmod 666 /dev/memoria

A partir de entonces, y si todo ha ido bien, tendremos un dispositivo **/dev/memoria** al cual le podemos escribir una cadena de caracteres y guardará el último de ellos. Podemos realizar esta operación así

$ echo -n abcdef > /dev/memoria

Para ver el contenido del dispositivo podemos usar un simple **cat**:

$ cat /dev/memoria

Este carácter no cambiará en memoria hasta que no se sobreescriba o se desinstale el módulo.

15 El driver real “puertopar”: descripción del puerto paralelo

Procederemos ahora a modificar el anterior driver “memoria” para realizar uno que haga una tarea real sobre un dispositivo real. Utilizaremos el ubicuo y sencillo puerto paralelo del ordenador y el módulo se llamará “puertopar”.

El puerto paralelo es en realidad un dispositivo que permite la entrada y salida de información digital. Externamente tiene un conector hembra D-25 con veinticinco patillas. Internamente, desde el punto de vista de la CPU, ocupa tres bytes de memoria. La dirección base, es decir, la del primer byte del dispositivo, es habitualmente la **0x378** en un PC. En este ejemplo sencillo usaremos únicamente el primer byte. el cual consta enteramente de salidas digitales.

La conexión de dicho byte con el patillaje del conector exterior aparece en la Fig. 2.

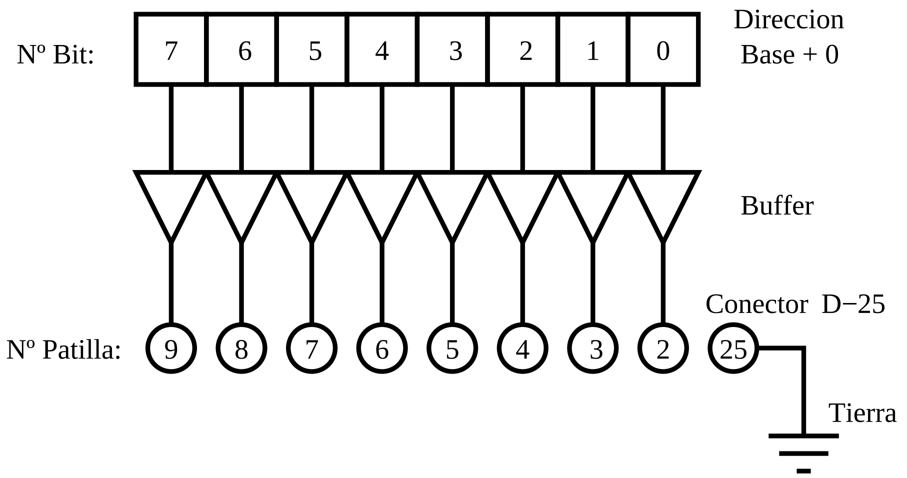

Figura 2. Esquema del primer byte del puerto paralelo y su conexión con el patillaje al conector exterior hembra D-25.

16 El driver “puertopar”: inicio del módulo

La función **init_module** anterior del módulo “memoria” habrá que modificarla sustituyendo la reserva de memoria RAM por la reserva de la dirección de memoria del puerto paralelo, es decir, la **0x378**. Para ello utilizaremos la función de chequeo de disponibilidad de la región de memoria, **check_region**, y la función de reserva de una región de memoria para este dispositivo, **request_region**. Ambas tienen como argumentos la dirección base de la región de memoria y su longitud. La función **request_region** además admite una cadena de caracteres que define el módulo.

17 ⟨*puertopar modificacion init module* 17⟩≡ (22)

```
/* Registrando puerto */
port = check_region(0x378, 1);
if (port) {
  printk("<1>puertopar: no puedo reservar 0x378\n");
  result = port;
  goto fallo;
}
request_region(0x378, 1, "puertopar");
```

17 El driver “puertopar”: eliminando el módulo

Será muy parecida al módulo “memoria” pero sustituyendo la liberación de memoria por la eliminación de la reserva de memoria del puerto paralelo. Esto se realiza con la función **release_region**, la cual tiene los mismos argumentos que **check_region**.

18 ⟨*puertopar modificacion cleanup module* 18⟩≡ (23a)

```
/* Liberando puerto */
if (!port) {
  release_region(0x378,1);
}
```

18 El driver "puertopar": leyendo del dispositivo

En este caso tenemos que añadir la verdadera lectura del dispositivo para luego pasar este dato al espacio de usuario. La función **inb** consigue este resultado, admitiendo como argumento la dirección del puerto y devolviendo como resultado el contenido del puerto.

19 ⟨*puertopar inport* 19⟩≡ (24a)

```
/* Leyendo del puerto */
puertopar_buffer = inb(0x378);
```

La Tabla 9, equivalente a la Tabla 2, nos muestra esta nueva función.

Eventos	Funciones del kernel
Leer datos	inb
Escribir datos	

Tabla 9. Eventos de los drivers y sus funciones asociadas entre el espacio de kernel y el dispositivo hardware.

19 El driver "puertopar": escribiendo al dispositivo

De nuevo tenemos que añadir la escritura al dispositivo para luego pasar este dato al espacio de usuario. La función **outb** consigue este resultado, admitiendo como argumento el contenido a escribir en el puerto y su dirección.

20 ⟨*puertopar outport* 20⟩≡ (24b)

```
/* Escribiendo al puerto */
outb(puertopar_buffer,0x378);
```

La Tabla 10, equivalente a la Tabla 2, nos muestra esta nueva función.

Eventos	Funciones del kernel
Leer datos	inb
Escribir datos	outb

Tabla 10. Eventos de los drivers y sus funciones asociadas entre el espacio de kernel y el dispositivo hardware.

20 El driver “puertopar” al completo

Procederemos ahora a ver todo el código del módulo “puertopar”. En general habrá que cambiar en todo el código, respecto al módulo **memoria.c**, las palabras “memoria” por “puertopar”. El driver final queda como sigue:

21a ⟨*puertopar.c* 21a⟩≡
⟨*puertopar inicio* 21b⟩
⟨*puertopar init module* 22⟩
⟨*puertopar cleanup module* 23a⟩
⟨*puertopar open* 23b⟩
⟨*puertopar release* 23c⟩
⟨*puertopar read* 24a⟩
⟨*puertopar write* 24b⟩

20.1 Sección inicial

En la parte inicial del driver utilizaremos un NÚMERO MAYOR diferente, el 61, cambiaremos la variable global **memoria_buffer** por **port** e incluiremos con **#include** los ficheros **ioport.h** y **io.h**.

21b ⟨*puertopar inicio* 21b⟩≡

```
/* Definiciones e includes necesarios para los drivers */
#define MODULE
#define __KERNEL__
#include <linux/config.h>
#include <linux/module.h>
#include <linux/kernel.h> /* printk() */
#include <linux/malloc.h> /* kmalloc() */
#include <linux/fs.h> /* everything... */
#include <linux/errno.h> /* error codes */
#include <linux/types.h> /* size_t */
#include <linux/proc_fs.h>
#include <linux/fcntl.h> /* O_ACCMODE */
#include <linux/ioport.h>
#include <asm/system.h> /* cli(), *_flags */
#include <asm/uaccess.h> /* copy_from/to_user */
#include <asm/io.h> /* inb, outb */

/* Declaracion de funciones de puertopar.c */
int puertopar_open(struct inode *inode, struct file *filp);
int puertopar_release(struct inode *inode, struct file *filp);
ssize_t puertopar_read(struct file *filp, char *buf,
  size_t count, loff_t *f_pos);
ssize_t puertopar_write(struct file *filp, char *buf,
  size_t count, loff_t *f_pos);
void cleanup_module(void);
```

```
/* Estructura que declara las funciones tipicas */
/* de acceso a ficheros */
struct file_operations puertopar_fops = {
  read: puertopar_read,
  write: puertopar_write,
  open: puertopar_open,
  release: puertopar_release
};

/* Variables globales del driver */
/* Numero mayor */
int puertopar_major = 61;

/* Variable de control para la reserva */
/* de memoria del puerto paralelo*/
int port;
```

20.2 Inicio del módulo

En la rutina de inicio del módulo incluiremos la reserva de la dirección de memoria del puerto paralelo como describimos antes.

22 ⟨*puertopar init module* 22⟩≡

```
int init_module(void) {
  int result;

  /* Registrando dispositivo */
  result = register_chrdev(puertopar_major, "puertopar",
      &puertopar_fops);
  if (result < 0) {
    printk(
      "<1>puertopar: no puedo obtener numero mayor %d\n",
      puertopar_major);
    return result;
  }

  ⟨puertopar modificacion init module 17⟩

  printk("<1>Insertando modulo\n");
  return 0;

  fallo:
    cleanup_module();
    return result;
}
```

20.3 Eliminación del módulo

La rutina de eliminación del módulo incluirá las modificaciones antes reseñadas.

23a ⟨*puertopar cleanup module* 23a⟩≡ (21a)

```
void cleanup_module(void) {

  /* Liberamos numero mayor */
  unregister_chrdev(puertopar_major, "memoria");

  ⟨puertopar modificacion cleanup module 18⟩

  printk("<1>Quitando modulo\n");
}
```

20.4 Abriendo el dispositivo como fichero

Esta rutina es idéntica a la del driver “memoria”.

23b ⟨*puertopar open* 23b⟩≡ (21a)

```
int puertopar_open(struct inode *inode, struct file *filp) {
  /* Aumentamos la cuenta de uso */
  MOD_INC_USE_COUNT;

  /* Exito */
  return 0;

}
```

20.5 Cerrando el dispositivo como fichero

De nuevo la similitud es exacta.

23c ⟨*puertopar release* 23c⟩≡

```
int puertopar_release(struct inode *inode, struct file *filp) {

  /* Decrementamos la cuenta de uso */
  MOD_DEC_USE_COUNT;

  /* Exito */
  return 0;
}
```

20.6 Leyendo del dispositivo

La función de lectura es análoga a la del "memoria" con las consiguientes modificaciones de lectura del puerto del dispositivo.

24a ⟨*puertopar read* 24a⟩≡

```
ssize_t puertopar_read(struct file *filp, char *buf,
  size_t count, loff_t *f_pos) {

  /* Buffer para leer el dispositivo */
  char puertopar_buffer;

  ⟨puertopar inport 19⟩

  /* Transferimos datos al espacio de usuario */
  copy_to_user(buf,&puertopar_buffer,1);

  /* Cambiamos posición de lectura segun convenga */
  if (*f_pos == 0) {
    *f_pos+=1;
    return 1;
  } else {
    return 0;
  }
}
```

20.7 Escribiendo al dispositivo

Es igual que el del "memoria" pero añadiendo la escritura al puerto del dispositivo.

24b ⟨*puertopar write* 24b⟩≡ (21a)

```
ssize_t puertopar_write( struct file *filp, char *buf,
  size_t count, loff_t *f_pos) {

  char *tmp;

  /* Buffer para leer el dispositivo */
  char puertopar_buffer;

  tmp=buf+count-1;
  copy_from_user(&puertopar_buffer,tmp,1);

  ⟨puertopar outport 20⟩

  return 1;
}
```

21 LEDs para comprobar el uso del puerto paralelo

Vamos a realizar en esta sección un "hardware" para poder visualizar el estado del puerto paralelo con unos simples LEDs.

ADVERTENCIA: Conectar dispositivos al puerto paralelo puede dañar su ordenador. Asegurése de que su cuerpo está conectado a una buena toma de tierra y su ordenador apagado al conectar un dispositivo. Cualquier problema que surja es totalmente responsabilidad suya.

El circuito a construir aparece en la Fig. 3. Como referencia bibliográfica se puede consultar Zoller (1997).

Para utilizarlo, primero se debe comprobar que todas las conexiones del montaje son correctas. Posteriormente se apaga el PC y se conecta el módulo al puerto paralelo. Luego se arranca el PC y se desinstalan todos los módulos habituales relacionados con el puerto paralelo, como por ejemplo, **lp**, **parport**, **parport_pc**, etc. Si no existe el dispositivo **puertopar**, se crea, como **root**, con el comando

 # mknod /dev/puertopar c 61 0

y se le otorgan permisos de escritura y lectura universales

 # chmod 666 /dev/puertopar

Se instala el módulo que hemos realizado, **puertopar**, pudiendo comprobar que efectivamente reserva la posición de puertos de entrada/salida **0x378** mediante el comando

$ cat /proc/ioports

Para encender los LEDs y verificar que el sistema funciona ejecutamos el comando

$ echo -n A > /dev/puertopar

que debe tener como consecuencia que se encienda el LED cero y el seis, estando apagados todos los demás.

Podemos ver el estado del puerto paralelo con el comando

$ cat /dev/puertopar

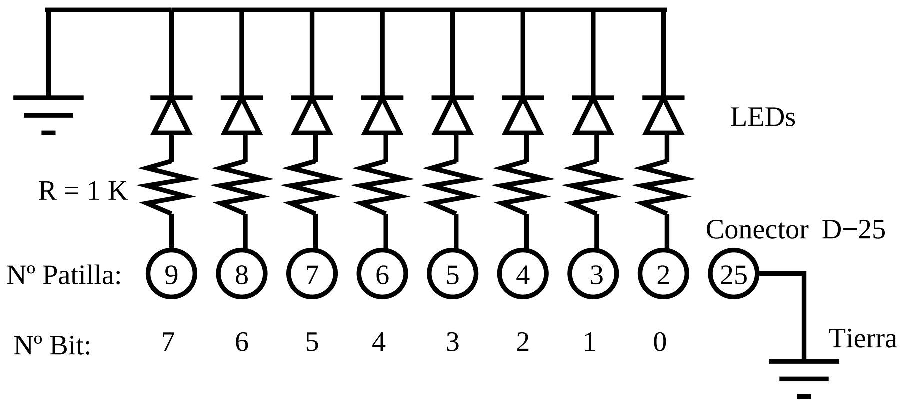

Figura 3. Esquema electrónico de la matriz de LEDs para monitorizar el puerto paralelo.

22 Aplicación final: luces centelleantes

Finalmente realizaremos una bonita aplicación que consiste en que se enciendan sucesivamente los LEDs en secuencia. Para ello realizaremos un programa en el espacio de usuario (el normal) con el que escribiremos al dispositivo **/dev/puertopar** un sólo bit de forma sucesiva.

26 ⟨*luces.c* 26⟩≡

```
#include <stdio.h>
#include <unistd.h>


int main() {
  unsigned char byte,dummy;
  FILE * PUERTOPAR;

  /* Abrimos el dispositivo puertopar */
  PUERTOPAR=fopen("/dev/puertopar","w");
  /* Eliminamos el buffer del fichero */
  setvbuf(PUERTOPAR,&dummy,_IONBF,1);

  /* Iniciamos la variable a uno */
  byte=1;

  /* Realizamos el bucle indefinido */
  while (1) {
    /* Escribimos al puerto paralelo */
    /* para encender un LED */
    printf("Byte vale %d\n",byte);
    fwrite(&byte,1,1,PUERTOPAR);
    sleep(1);

    /* Actualizamos el valor de byte */
    byte«=1;
    if (byte == 0) byte = 1;
  }

  fclose(PUERTOPAR);

}
```

Lo compilamos de la forma habitual
$ gcc -o luces luces.c
y lo ejecutamos con el comando
$ luces
¡Las luces se encenderán una por una secuencialmente!

23 Bibliografía

1. A. Rubini, J. Corbert. 2001. Linux device drivers (second edition). Ed. O'Reilly. Es un libro libre disponible en http://www.xml.com/ldd/chapter/book/.

2. B. Zoller. 1997. Circuitos electrónicos con el PC. Ed. Marcombo.

3. M. Waite, S. Prata. 1990. Programación en C. Ed. Anaya.