Práctica 3. Manejadores de dispositivo

J.C. Pérez, S. Sáez, V. Lorente y L. Pascual

Índice

1. Objetivos

El objetivo de esta práctica es desarrollar un manejador de dispositivo capaz de visualizar en un display de 7 segmentos, conectado a los bits de datos del puerto paralelo, los números del 0 al 9 cuando se vuelcan sobre el fichero especial de caracteres **`/dev/practica`**.

Ese mismo fichero devolverá, cuando se lea, la posición de 4 microinterruptores conectados a las líneas de entrada (control) del puerto paralelo.

El manejador, por tanto, hará accesible el citado hardware del modo habitual en UNIX, a través de un fichero de dispositivo que se podrá abrir desde cualquier programa (con la llamada **`open`**, por ejemplo) y leerse y escribirse con las llamadas **`read`** y **`write`**, o bien desde el mismo shell, con **`echo`**, redirecciones, etc.

Cabe resaltar que la versión del núcleo que explicamos en clase es la 2.6.11 y en el laboratorio usamos la 2.6.20, luego habrá ciertos detalles que cambiarán un poco de una versión a otra. Así que todas las consultas que realicemos sobre el código navegable de la web (lxr) ha de ser sobre la versión 2.6.20.

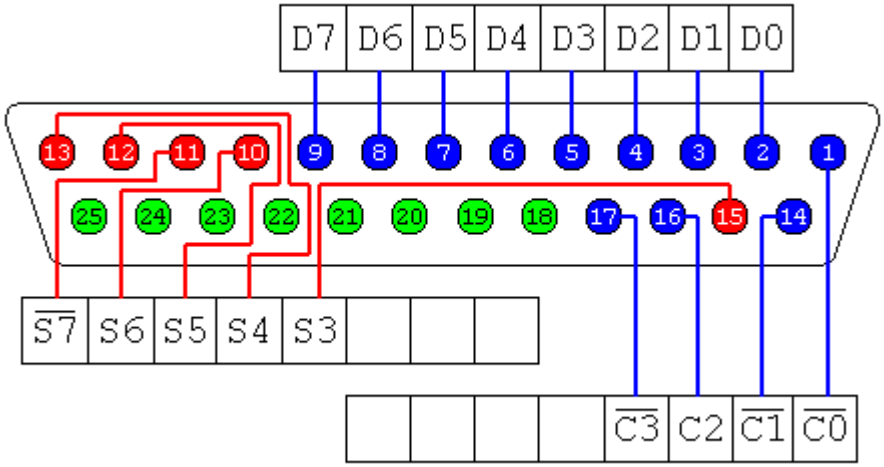
2. Introducción

El puerto paralelo original del IBM-PC (los actuales añaden funcionalidades sobre este diseño básico) dispone de 12 salidas y 5 entradas digitales.

Se accede a ellas a través de 3 registros de 8 bits, consecutivos en el espacio de direccionamiento de E/S del PC. Como se ve en la figura disponemos de 8 salidas a través del registro de DATOS (D0-D7), 5 entradas (una de ellas invertida) accesibles en el registro de ESTADO (S3-S7) y 4 salidas (tres de ellas invertidas) por el registro de CONTROL (C0-C3).

Las 8 patillas restantes en el conector DB-25 hembra estándar están conectadas a MASA (0 V). Los bits restantes de los registros de estado y control no se usan, a excepción del bit 4 del registro de control, C4, que permite activar el envío de una interrupción cuando la patilla 10 recibe un flanco de bajada.

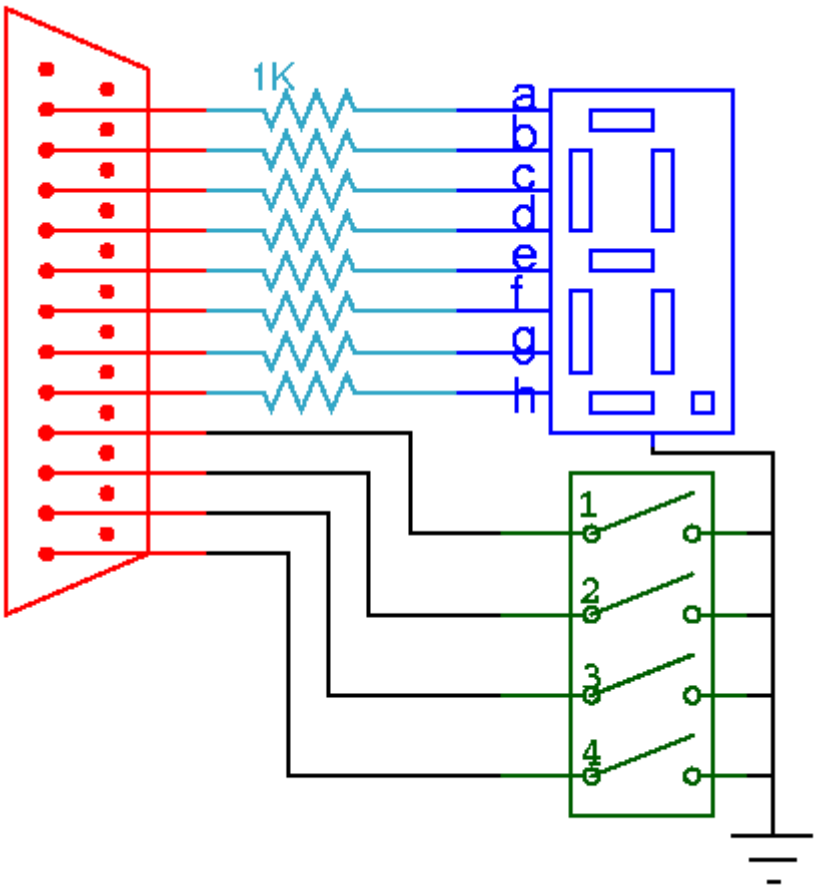
Las direcciones de los tres registros (datos, estado y control) son, como se ha dicho consecutivas, y se denotan en referencia a la dirección del primero de ellos (el de datos), que se denomina dirección base.



Por ejemplo, si en un PC se dice que tenemos el puerto paralelo en la dirección base **`0xc030`** (es el caso de los PC's del laboratorio), eso significa que el registro de datos está en **`0xc030`**, el de estado en **`0xc031`** y el de control en **`0xc032`**.

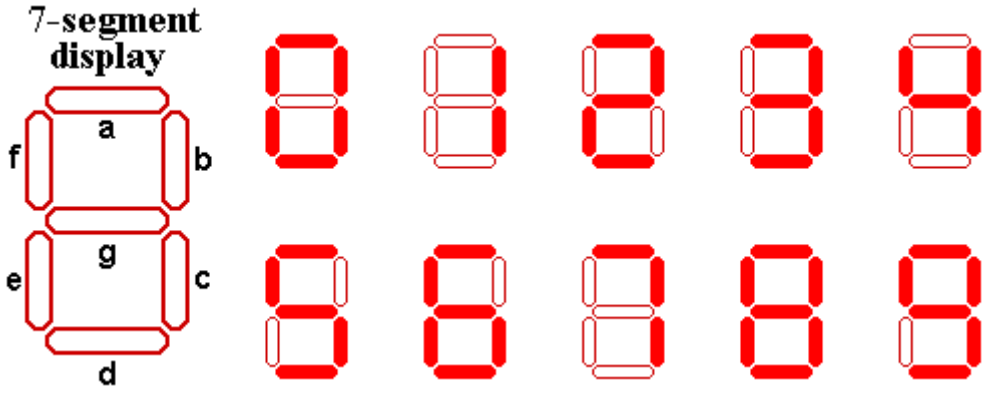
3. El dispositivo

Vemos en la figura el esquema general del dispositivo a controlar.



3.1. El visualizador de 7 segmentos

Los visualizadores de 7 segmentos son componentes muy sencillos que constan únicamente de LEDs conectados directamente a las patillas de conexión.

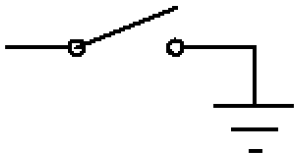


El tipo de visualizador de que se dispone en el laboratorio es de cátodo común, lo que significa que los cátodos de todos los LEDs están unidos y conectados a una misma patilla. Los ánodos de los diodos están accesibles por separado.

La patilla conectada al ánodo de cada LED se etiqueta mediante una letra de **a** a **g**. El punto decimal de nuestro display está conectado a la pata **h**.

3.2. Los microinterruptores

Las conexiones de entrada al puerto paralelo disponen de una resistencia de pull-up interna y, por tanto, para que sea posible detectar un cierre de contacto desde el PC, basta con colocar un interruptor entre una entrada y masa.



Cuando el interruptor está cerrado (hacia arriba - on), se lee un cero y cuando está abierto (hacia abajo - off), la patilla queda al aire exteriormente pero puesta a nivel alto por la resistencia interna, y se lee un uno.

3.3. El manejador de dispositivo

El manejador asociado al este dispositivo se desarrollará en forma de módulo cargable que al insertarse convierta los accesos al fichero especial de dispositivo **`/dev/practica`** en las operaciones correspondientes sobre los puertos de entrada/salida de la máquina.

A continuación se realiza una revisión de la entrada/salida en los sistemas UNIX.

4. Revisión de la E/S en UNIX

La abstracción que ofrece UNIX de los dispositivos físicos conectados al ordenador es la de fichero especial de dispositivo. Los ficheros especiales de dispositivo suelen localizarse en el directorio **`/dev`**, aunque esto no es obligatorio.

En los dispositivos simples, como los puertos serie, terminales, teclados, etc., la interacción se limita a menudo a abrir el fichero con la llamada open, para lectura o escritura y volcar o leer los datos que queremos transmitir o recibir, mediante las llamadas read y write. En muchos dispositivos, sobre todo en los más sofisticados, como tarjetas de sonido, escáneres, tarjetas de vídeo, etc, es necesario interactuar de forma más específica con el hardware. Para ello se recurre a la llamada ioctl, que permite modificar los parámetros del dispositivo (como velocidad, paridad, bits de stop, etc. del puerto serie; bits por muestra o frecuencia de muestreo en una tarjeta de sonido, etc.), así como instruir al periférico para que realice accesos directos a memoria u otras operaciones específicas.

En esta práctica nos limitaremos a implementar un manejador sencillo que muestre en el visualizador de 7 segmentos los valores que se escriban en el fichero especial y que devuelva un valor cuyos bits reflejen el

estado actual de los microinterruptores, cuando se lea del fichero.

Los ficheros especiales (recordemos que en UNIX hay tres tipos de fichero: ficheros normales o "regulares", ficheros especiales y directorios) ocultan una funcionalidad especial bajo la apariencia de ficheros convencionales, con su ruta de acceso, sus atributos, etc.

Distinguimos dos tipos de ficheros especiales de dispositivo:

Dispositivos de bloques
: Son aquéllos que pueden direccionarse por bloques, es decir que proporcionan acceso aleatorio a sectores, típicamente los discos de todo tipo.

Dispositivos de caracteres
: Son los que admiten operaciones secuenciales carácter a carácter y/o pueden direccionarse a nivel de byte.

Ejercicio: Haz **`ls -l /dev`** y comprueba el primer carácter de los atributos de cada fichero. Una "b" indica bloques y una "c" caracteres.

Un detalle importante a tener en cuenta es que el nombre de un fichero especial no influye para nada en su relación con uno u otro dispositivo físico. Por tanto, podemos copiar o renombrar cualquier fichero de este tipo sin ningún problema.

De hecho, estos ficheros no contienen información ni ocupan espacio si los duplicamos. Los únicos datos útiles que contienen son el *major number* y el *minor number*, que identifican unívocamente el dispositivo que representan. En el listado producido por **`ls -l`**, las dos columnas anteriores a la fecha son el *major* y *minor number*.

- El *major number* indica el tipo de dispositivo (p.ej. 3 es un disco del primer IDE, 22 uno del segundo IDE, 2 es un disco flexible, etc.)
- El *minor number* indica un número de orden dentro de los dispositivos de cada tipo (p.ej., entre los discos de un IDE, el 0 es el primer disco, el 64 el segundo, el 1 es la primera partición del primer disco, el 2 la segunda, etc.).

Ejemplos de dispositivos: En **`/usr/src/linux/Documentation/devices.txt`** puedes encontrar la descripción completa de todos los dispositivos, incluyendo sus *major* y *minor numbers*.

Para crear un fichero especial de dispositivo se usa la orden mknod manual, que recibe un nombre de fichero, un tipo (bloques o caracteres), un *major* y un *minor number*.

- **`/dev/fd0`**: Primer floppy.
- **`/dev/hda`**: Disco master del primer IDE.
- **`/dev/hdd3`**: Tercera partición del disco slave del segundo IDE. (Las particiones primarias van de la 1 a la 4 y las lógicas de la 5 a la 20.)
- **`/dev/sda2`**: Segunda partición del primer disco SCSI.
- **`/dev/scd0`**: Primer CD-ROM SCSI. (Los CD-ROM IDE se ven como discos magnéticos.)
- **`/dev/st0`**: Primera unidad de cinta SCSI.
- **`/dev/tty2`**: Tercera consola.
- **`/dev/ttyS0`**: Primer puerto serie.
- **`/dev/ptyp5`**: Sexta pseudoterminal maestra.
- **`/dev/ttyp5`**: Sexta pseudoterminal esclava (el proceso que crea la terminal abre la maestra y el que la va a usar abre la esclava).

Otros dispositivos físicos:

- **`/dev/lp0`**: Primer puerto paralelo.
- **`/dev/parport0`**: Primer puerto paralelo a nivel binario.
- **`/dev/psaux`**: Puerto de ratón PS/2.

- **/dev/sg2**: Tercer dispositivo genérico SCSI (escáner, p.ej.).
- **/dev/dsp0**: Primera entrada y salida de audio digital.
- **/dev/mixer1**: Segundo mezclador de audio (tarjeta de sonido).
- **/dev/midi3**: Cuarto puerto MIDI (instrumentos musicales).
- **/dev/video0**: Primera tarjeta de captura de vídeo.
- **/dev/radio1**: Segunda tarjeta de radio.
- **/dev/md1**: Segundo "metadisco" (RAID).
- **/dev/ppp**: Dispositivo virtual para ppp.

Dispositivos virtuales:

- **/dev/loop0**: Primer fichero "loopback". Se asocia un fichero normal para poder montarlo como un dispositivo de bloques.
- **/dev/mem**: Permite acceder a la memoria física.
- **/dev/kmem**: Permite acceder a la memoria virtual del núcleo.
- **/dev/null**: Como un agujero negro que se lo "traga" todo.
- **/dev/port**: Permite acceder a los puertos de E/S.
- **/dev/zero**: Suministra incansablemente caracteres "\0".
- **/dev/full**: Devuelve error de dispositivo lleno.
- **/dev/random**: Da auténticos números aleatorios de un pozo de entropía.
- **/dev/urandom**: Devuelve siempre números (pseudo)aleatorios.
- **/dev/stdin**: Entrada estándar del proceso en curso.
- **/dev/stdout**: Salida estándar del proceso en curso.
- **/dev/stderr**: Salida estándar de error del proceso en curso.

5. Acceso a los puertos del PC

Las rutinas para acceder a los puertos de E/S residen en **/usr/src/linux/include/asm/io.h** y son macros, de modo que basta con incluir el fichero para poder usarlas, sin necesidad de enlazar con librería alguna. Debido a una peculiaridad del compilador **gcc**, la optimización debe estar activada cuando se usan estas funciones. Recordad poner el correspondiente **#include <asm/io.h>**.

Para acceder a los puertos desde un programa C fuera del núcleo, hay que ser superusuario y, además, requerir del sistema operativo el permiso para hacerlo. Para ello se dispone de la función **ioperm()** o **iopl()**. Desde dentro del núcleo (o de un módulo, que es lo mismo), no hace falta llamar a estas funciones.

La E/S propiamente dicha se realiza a través de la función **inb(port)**, que devuelve el valor (de 8 bits) al que se encuentra el puerto correspondiente, y de la función **outb(valor,puerto)**, que vuelca un valor de 8 bits sobre un puerto. **Atención** al orden de los parámetros que es distinto al adoptado por algunos compiladores comunes en el mundo de DOS y Windows.

Aunque no nos hará falta, conviene saber para leer un byte del puerto **p** y otro del **p+1**, formando una palabra de 16 bits, se puede usar **inw(p)**. Para volcar una palabra de 16 bits, tenemos **outw(valor,p)**. Existen también variantes de estas funciones (**inb_p()**, **outb_p()**, **inw_p()**, **outw_p()**) que contienen un retardo de alrededor de un microsegundo después del acceso al puerto.

Desde el núcleo de Linux, el acceso a los puertos de E/S está regulado mediante un mecanismo de reserva de rangos de direcciones. Esto permite que los manejadores sepan si un rango de puertos está siendo utilizado por otros manejadores. Para ello se dispone de tres funciones que debemos emplear para no colisionar con otros manejadores. Podemos saber en un momento dado cuáles son los rangos de puertos reservados por los diferentes manejadores que se hallan instalados en el sistema visualizando el fichero **/proc/ioports**.

- **struct resource * request_region (unsigned long port, unsigned long range, const char *name)** [include/linux/ioport.h]: Es una macro que llama a **__request_region** [kernel/resource.c] que reserva un rango de range puertos a partir de **port** identificándolo con el

nombre **name**. Este identificador es el que aparecerá en **/proc/ioports** y debe servir al usuario para entender qué manejador ha reservado esos puertos. Si la llamada no tiene éxito, el valor de retorno será **NULL**.

- **void release_region (unsigned long port, unsigned long range)** [include/linux/ioport.h]: Es una macro que llama a **__release_region** [kernel/resource.c] que libera un rango de range puertos a partir de **port**, reservados anteriormente.

Recordad poner el correspondiente **#include <asm/io.h>**.

En esta práctica, el módulo que queremos desarrollar ha de controlar un dispositivo de caracteres, es decir, que el núcleo debe saber que cuando un proceso acceda a cierto fichero especial (que tendrá asignado un *major number* concreto), debe enviar la petición correspondiente a nuestro código.

Para notificar esto al núcleo, hemos de llamar nada más insertar el módulo, a la función **register_chrdev(unsigned int major, const char *name, struct file_operations *fops)** [fs/char_dev.c], que recibe como parámetros el *major number* que queremos atender, un nombre que identifique al dispositivo y una estructura que indicará al núcleo qué función tiene que invocar para cada tipo de acceso al dispositivo (una para **open()**, otra para **close()**, etc.).

Si se registra correctamente el *major number*, **register_chrdev()** nos devuelve un 0 o mayor, sino, un código de error negativo. Utilizaremos en esta práctica el *major number* 55, que no está utilizado en el sistema (corresponde al dispositivo **/dev/dsp56k**, que es una placa con el procesador digital de señal DSP56001, de la que no disponemos en el laboratorio)

Para crear un fichero especial de caracteres que tenga asociado este número, se utilizará la orden **mknod** cuyos argumentos hay que consultar en el manual. El fichero especial puede llamarse, como hemos dicho antes, **/dev/practica**, aunque cualquier otro nombre de fichero es igualmente válido. Acordaos de borrar el fichero al terminar cada sesión.

Habréis comprobado que no es necesario pasar a **register_chrdev()** ninguna información sobre *minor numbers*. Esto es así porque el núcleo no hace caso del *minor number* de los dispositivos, simplemente se lo pasa al manejador cada vez que hay una operación sobre uno de ellos.

En la función que pasemos a la macro **module_init()**, además de todo lo anterior, habrá que comprobar la disponibilidad del rango de puertos de E/S que usaremos del puerto paralelo (2 bytes a partir de **0xc030**) y reservarlo.

Cuando se elimina el módulo que maneja un dispositivo, se debe efectuar la operación inversa a la de registrado del *major number*. Esta operación la lleva a cabo la función **unregister_chrdev(unsigned int major, const char * name)** [fs/char_dev.c, L323], a la que hay que pasarle el *major number* y el nombre que se utilizaron al registrarse. Además habrá que liberar también el rango de puertos de E/S que se reservó en el inicio.

6. Funciones de manejo de dispositivos

Vamos a examinar las operaciones sobre dispositivos y cómo se indica al núcleo qué funciones llamar para cada operación.

El último parámetro que se le pasa a la función **register_chrdev()** es **struct file_operations** [include/linux/fs.h]. También deberemos incluir éste fichero (**#include <linux/fs.h>**).

Los campos de esta estructura indican al núcleo qué funciones llamar cuando se producen las operaciones de apertura (**open()**), cierre (**close()**), lectura (**read()**), escritura (**write()**), etc., sobre un fichero especial de dispositivo que tenga el *major number* correspondiente.

En nuestro caso, sólo implementaremos las funciones citadas (existen algunas otras, como **select()**, **ioctl()**), etc., con lo cual la estructura apuntada por **fops** puede tener el aspecto siguiente:

```
struct file_operations practica_fops = {
 .read= practica_read,
 .write= practica_write,
 .open= practica_open,
 .release= practica_release,
};
```

Vamos a comentar a continuación la implementación de las cuatro funciones de manejo del dispositivo que necesitamos para esta práctica:

- **`int practica_open (struct inode *inode, struct file *filp)`**: En un dispositivo más complejo inicializaría adecuadamente el hardware, tomaría nota de que está siendo usado por algún proceso, etc. En nuestro caso basta con que llame a **`try_module_get(THIS_MODULE)`** para indicar que algún proceso está usando el módulo, ya que se ha abierto un dispositivo que éste está controlando. Esto evitará que se descargue el módulo mientras el dispositivo está siendo usado. Efectivamente, esta macro incrementa un contador asociado al módulo, lo que impedirá que el módulo pueda eliminarse del sistema con rmmod. Si no se han hecho tantas **`module_put(THIS_MODULE)`** como **`try_module_get(THIS_MODULE)`**, el código del núcleo que sirve la llamada de **`rmmod`** dará un mensaje de *módulo en uso*.

  La función debe devolver un cero para indicar que se ha abierto el dispositivo con éxito.

  Como puede observarse, no consultamos el *minor number*, ya que sólo vamos a usar el primer puerto paralelo. Si quisiéramos distinguir entre varios puertos, podríamos consultar el *minor number* en el campo **`i_rdev`** de **`*inode`**.

- **`int practica_release (struct inode *inode, struct file *filp)`**: En un caso real debería dar algún tipo de orden de "reposo" al hardware y notificarle que no está siendo ya usado. En nuestro caso, basta con que haga un **`module_put(THIS_MODULE)`** para decrementar el contador de dispositivos abiertos (en su caso, podría haber varios con distintos *minor numbers*, por ejemplo). Cuando ese contador vale cero, se puede eliminar el módulo con **`rmmod`**.

  La función debe devolver 0 si todo ha ido bien.

- **`ssize_t practica_read (struct file *filp, char *buf, size_t count, loff_t *offp)`**: Debe leer del dispositivo y enviar la información correspondiente al proceso demandante. Para ello, en nuestro caso, ha de hacer un **`inb()`** del puerto de estado para averiguar el estado de los microinterruptores y devolverlo como una cadena de 5 bytes en buf.

  Devolverá como resultado de la función el número de bytes leídos (siempre 5, a no ser que en count el usuario nos especifique que el buffer tiene **`b<5`** bytes reservados, en cuyo caso rellenará y devolverá sólo los b primeros). El formato de la cadena de 5 caracteres debe ser de ceros y unos en ASCII, por ejemplo "0101\n", reflejando la posición de los microinterruptores.

- **`ssize_t practica_write (struct file *filp, const char *buf, size_t count, loff_t *offp)`**: Debe escribir en el dispositivo los count bytes de datos que el usuario envía en buf. En nuestro caso, sólo admitiremos un byte cada vez y haremos un outb() al puerto de datos para iluminar los segmentos del display correspondientes al primer caracter de buf.

  Devolverá como resultado de la función el número de bytes escritos (siempre 1).

  Si el carácter no es un dígito o el punto decimal en ASCII, no se debe iluminar ningún led del display, pero no se devolverá error.

7. Acceso a la memoria del proceso usuario desde el núcleo

Desde el núcleo se puede leer o escribir sobre la memoria del proceso en modo usuario simplemente accediendo a ella (en nuestro caso, leyendo o escribiendo directamente en **`buf[]`**), pero es recomendable por diversos motivos utilizar las funciones **`copy_to_user`** y **`copy_from_user`**.

- **`unsigned long copy_from_user(void *to, const void *from, unsigned long n)`** [include/asm/uaccess.h] : Es una macro que copia n bytes de from (memoria del usuario) a to (memoria del núcleo) comprobando si la dirección origen es correcta y se halla dentro del mapa de memoria del proceso.
- **`unsigned long copy_to_user(void *to, const void *from, unsigned long n)`** [include/asm/uaccess.h]: Es una macro que copia n bytes de from (memoria del núcleo) a to (memoria del usuario) comprobando si la dirección destino es correcta y se halla dentro del mapa de memoria del proceso.

Recordad poner el correspondiente **`#include <asm/uaccess.h>`**.

8. Pruebas de funcionamiento

Para comprobar el funcionamiento del visualizador de 7 segmentos podéis volcar cualquier fichero sobre el dispositivo con **`cat fichero >/dev/practica`**. El problema es que la visualización será tan rápida que sólo aparecerá visible el último carácter válido del fichero.

Una opción más adecuada es la de utilizar **`echo -n 5 >/dev/practica`**, que visualiza un cinco en el display (el -n es para evitar que se envíe un carácter de nueva línea después del cinco). Para ver una secuencia de cifras del 0 al 9, podéis usar la secuencia de órdenes siguiente:

```
for i in 0 1 2 3 4 5 6 7 8 9 ; do
        echo -n $i >/dev/practica;
        sleep 0.3;
done
```

O, para que no acabe nunca, la siguiente:

```
i=0;
while true; do
        echo -n $i >/dev/practica ;
        i=$[ ($i + 1) % 10 ] ;
        sleep 0.2;
done
```

Por otro lado, para comprobar la lectura de los microinterruptores, bastará con hacer

```
$ cat /dev/practica
```

y veremos aparecer continuamente la representación de ceros y unos correspondiente a la posición actual de los contactos. Para parar, bastará con pulsar Ctrl-c. (En las terminales "konsole" no funciona demasiado bien el scroll. Puedes probar en una consola no gráfica [Ctrl-Alt-F1] o con un terminal gráfico "xterm")

```
1011
1011
1011
1011
1011
1011
...
```

9. Entrega de la práctica

Para entregar la práctica deberéis hacer un fichero *tar* comprimido con **`gzip`** que contenga única y exclusivamente el fichero fuente con el manejador de dispositivo (v.g. **`dispositivo.c`**) y el fichero **`Makefile`**. Un ejemplo de la orden a utilizar sería:

```
$ tar cvzf practica3.tgz dispositivo.c Makefile
```

No os preocupéis por el nombre del fichero ya que el sistema de entrega lo renombra.

Para entregar la práctica deberéis rellenar el formulario que se encuentra en el enlace http://futura.disca.upv.es/eso/practicas/index.php. La clave solicitada debe ser introducida por el profesor de prácticas.

10. Bibliografía

Si necesitáis más información acerca de cómo diseñar módulos, podéis visitar: http://www.tldp.org/LDP/lkmpg/2.6/html/