Sistemas de Computación

-

FACULTAD DE CIENCIAS EXACTAS, FÍSICAS Y NATURALES
UNIVERSIDAD NACIONAL DE CÓRDOBA

TP 5

Drivers

Sistemas de computación - 2025

Integrantes

Franco Gabriel Lopez, 43271762

Kevin Bejarano, 42814288

Elly Indra Esparza, 39437553

Profesor adjunto

Ing. Javier Alejandro Jorge

Índice

1. Introduccion

1.1. Objetivo

El objetivo de este trabajo práctico es desarrollar un controlador de dispositivo (*device driver*) capaz de ejecutarse en una Raspberry Pi y leer las entradas de 2 puertos de entrada y salida digital, y desarrollar una aplicación de usuario que utilice el driver para mostrar en pantalla el valor de una de las entradas como una señal cuadrada en el tiempo.

En el desarrollo de este trabajo, comenzaremos con una descripción detallada sobre los fundamentos esenciales para crear un controlador. Procederemos a implementar una interfaz de usuario diseñada en Python, interconectada con una biblioteca escrita en C++, que ofrezca acceso eficiente al controlador desarrollado. Posteriormente, integraremos un conjunto de funciones específicas en nuestro controlador. Estas funciones se implementarán con el objetivo principal de habilitar y permitir la lectura precisa de los puertos digitales.

2. Desarrollo

2.1. Primer driver

Un driver es en un principio un módulo más como los que ya hemos trabajado en el trabajo anterior. Para refrescar, se crea con un archivo `.c` que contenga una función marcada como `__init` y otra como `__exit`.

Dado que para nuestro ejemplo trabajaremos con un *character device driver* (apropiado para controladores de dispositivos externos), debemos reservar los números MAJOR y MINOR, que son identificadores de dispositivos y controladores. Reservaremos un número MAJOR que represente nuestro controlador y tantos números MINOR como dispositivos vayan a hacer uso del controlador (en nuestro caso 1). Para esto se utiliza la siguiente función:

```
1 dev\_t dev;
2 dev\_t dev\_no;
3 int major;
4
5 alloc\_chrdev\_region(\&dev\_no, BASE\_MINOR, MINOR\_COUNT,
     DEVICE\_NAME);
6 major = MAJOR(dev\_no);
7 dev = MKDEV(major, 0);
```

Listing 1: Reserva de numeros MAJOR y MINOR

La función `alloc_chrdev_region()` realiza una asignación dinámica de un número MAJOR que esté disponible en el sistema, a la vez que reserva una cantidad de números MINOR de acuerdo con el valor especificado por el entero sin signo `MINOR_COUNT`. El valor de `BASE_MINOR` se utiliza para determinar el punto de partida de la secuencia de números MINOR que se han de asignar. Finalmente, el argumento `DEVICE_NAME` es un puntero a un `const char*` y sirve para identificar el controlador del dispositivo. El valor de retorno de `alloc_chrdev_region()` no debe descartarse, ya que nos indica si ha ocurrido algún error.

Para crear el dispositivo necesitamos dos estructuras: una del tipo `cdev`, que se inicializa con `cdev_init()`, y otra del tipo `file_operations`. Esta última estructura contiene las direcciones de las funciones que deben ejecutarse cuando se realizan operaciones sobre el archivo de dispositivo, como `open`, `read`, `write` y `close`.

```
static struct file\_operations dev\_fops =
{
.owner = THIS\_MODULE,
.open = my\_open,
.release = my\_close,
.read = my\_read,
.write = my\_write
};
```

Listing 2: Estructura file_operations

Para simplificar, haremos que la función de escritura asigne el valor de una bandera como '0' o '1', y que la función de lectura retorne el valor actual de dicha bandera:

```
static char curr\_signal = '0';

static ssize\_t my\_read(struct file \*f, char \_\_user \*buf,
    size\_t len, loff\_t \*off)
{
char kbuf\[1];
kbuf\[0] = curr\_signal;

```
if (copy_to_user(buf, kbuf, 1))
    return -EINVAL;

return 1;
```

}

static ssize_t my_write(struct file *f, const char __user *
    buf, size_t len, loff_t *off)
{
char kbuf;

```
if (len != 1)
    return -EINVAL;

if (copy_from_user(&kbuf, buf, 1))
    return -EFAULT;

if (kbuf != '0' && kbuf != '1')
    return -EINVAL;

curr_signal = kbuf;
return len;
```

}
```

Listing 3: Lectura y escritura en el dispositivo

Para crear el dispositivo usamos `cdev_init()` y `cdev_add()`:

```
cdev\_init(\&c\_dev, \&dev\_fops);
cdev\_add(\&c\_dev, dev, COUNT);

cl = class\_create(THIS\_MODULE, CLASS\_NAME);
device\_create(cl, NULL, dev, NULL, DEVICE\_NAME);
```

Listing 4: Creacion del dispositivo

Finalmente, la función `device_create()` se encarga de registrar el dispositivo en `/sys/class` y generar el archivo en `/dev`.

Para remover el módulo, la función `__exit` debe eliminar el dispositivo, liberar la memoria de las estructuras y liberar los números reservados:

```
static void \_\_exit oxcode\_exit(void)
{
device\_destroy(cl, dev);
class\_destroy(cl);
cdev\_del(\&c\_dev);
unregister\_chrdev\_region(dev, 1);
}
```

Listing 5: Remover el modulo

2.2. Acceso al driver

Para acceder al dispositivo, generamos una librería en C++ que interactúa con el archivo en `/dev`. Para mantener compatibilidad con Python, las funciones se declaran `extern Ç"`.

```
bool switch\_signal(const uint signal\_no)
{
if (device < 0  || signal\_no > 1)
{
return false;
}
const std::string value = std::to\_string(signal\_no);
const auto ret = write(device, value.c\_str(), 1);
return ret >= 0;
}

int read\_signal\_values()
{
if (device < 0)
{
return false;
}
char buffer\[1];

if (read(device, buffer, 1) <= 0)
return -1;
```

```
22
23  return buffer\[0] - '0';
24  }
```

Listing 6: Interfaz en C++ con Python

Luego, el script en Python accede a esta librería para obtener nuevos valores de la señal cada 1 segundo y permite cambiar la señal que se observa con *inputs* de usuario. segundo y permite cambiar la señal que se observa con *inputs* de usuario.

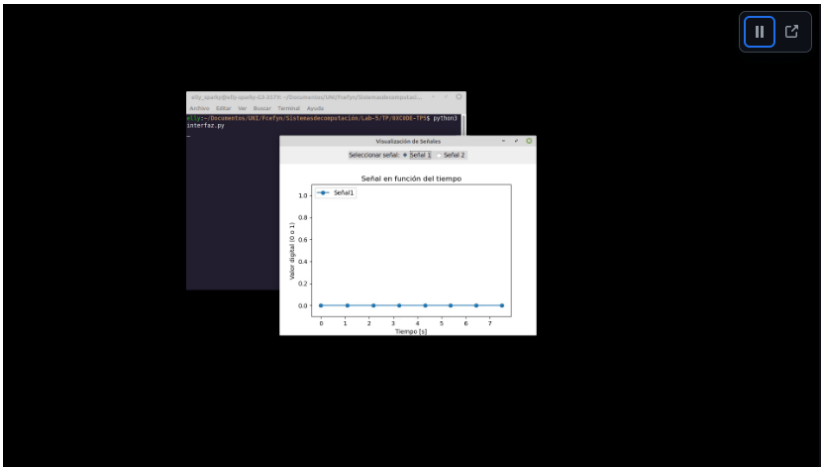

Figura 1: Señal 1

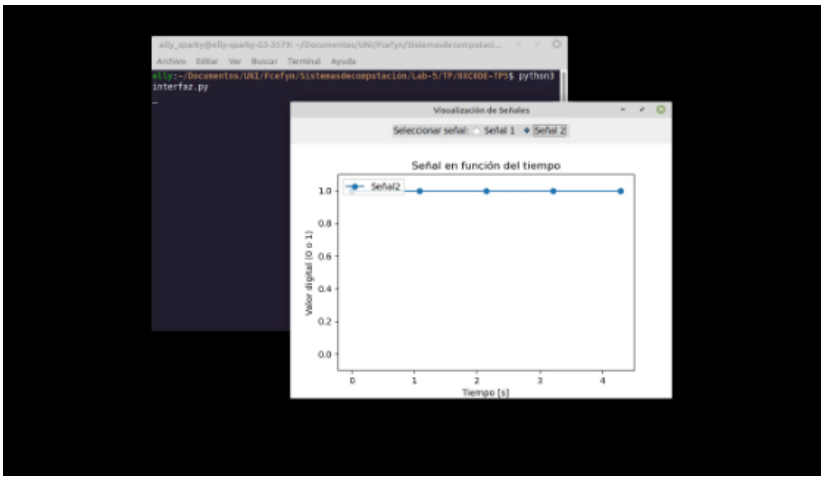

Figura 2: Señal 2

Ahora podemos actualizar el driver para leer valores reales de I/O sin necesidad de actualizar el script de Python ni la librería en C++.

2.3. Acceso a GPIO

Para manipular los puertos GPIO en el driver se utiliza la cabecera `<linux/gpio/consumer.h>` y las siguientes funciones clave:

- `gpio_to_desc()`: Convierte el número GPIO legacy a un descriptor moderno.

- `gpiod_direction_input()`: Configura el pin como entrada.
- `gpiod_get_value()`: Lee el valor lógico del pin.
- `gpiod_put()`: Libera el descriptor del pin.

A continuación se muestra un ejemplo simplificado de uso:

```c
#include <linux/gpio/consumer.h>

static struct gpio_desc *gpio;

static int __init mydriver_init(void) {
    gpio = gpio_to_desc(512 + 17); // GPIO17
    gpiod_direction_input(gpio);
    return 0;
}

static ssize_t my_read(...) {
    int val = gpiod_get_value(gpio);
    // Convertir val a '0' o '1' y copiar al usuario
}

static void __exit mydriver_exit(void) {
    gpiod_put(gpio);
}
```

Listing 7: Uso básico de GPIO en un driver

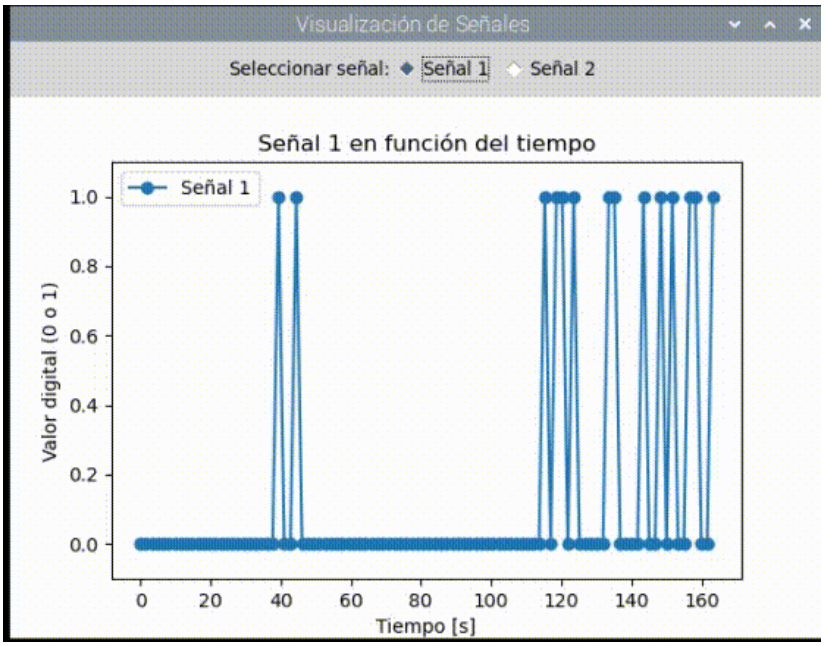

Figura 3: Señal 1

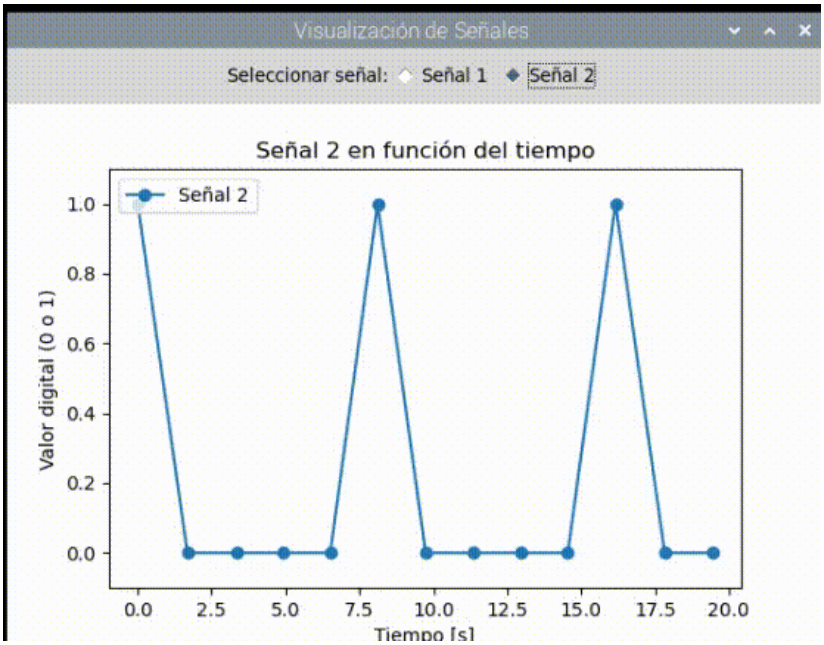

Figura 4: Señal 2

2.4. Extra: Resolución de conflictos con GPIO

Durante el desarrollo del driver para la lectura de pines GPIO en la Raspberry Pi, la mayor dificultad que experimentamos fue el constante fallo al intentar obtener descriptores válidos para los pines GPIO 17 y 27. Probamos diversas interfaces provistas por los módulos del kernel de Linux, como `gpio_request`, `gpiod_get` y `gpiod_get_index`. Todas ellas generaban errores al insertar el módulo, indicando que no se encontraba el símbolo correspondiente.

Inicialmente asumimos que los pines GPIO estaban asociados al módulo `gpiochip0`, el cual viene por defecto en Linux. Sin embargo, tras múltiples intentos fallidos y lecturas erróneas, llegamos a la conclusión de que el problema no residía tanto en el código del driver, sino en cómo estaban mapeados los GPIOs en el kernel.

Finalmente, al probar con la función `gpio_to_desc()`, el módulo pudo cargarse sin errores. Aun así, no obteníamos lecturas válidas. Recurriendo al comando:

```
1 cat /sys/kernel/debug/gpio
```

descubrimos que el controlador de pines GPIO estaba registrado como `gpiochip512`. Esto implicaba que los pines físicos, como el GPIO17 y GPIO27, debían ser referenciados como `512 + 17 = 529` y `512 + 27 = 539`, respectivamente.

Tras realizar este ajuste, el driver fue capaz de obtener los descriptores correctos y leer los pines sin inconvenientes.

Esta experiencia demuestra que no basta con conocer la API del kernel de Linux: también es crucial entender cómo está mapeado el hardware en memoria, especialmente en sistemas embebidos como la Raspberry Pi.

3. Conclusión

El desarrollo de este trabajo práctico nos permitió adentrarnos en el mundo de los drivers de dispositivos en Linux, un componente fundamental pero muchas veces invisible del sistema operativo. Al escribir un controlador desde cero y enfrentarnos a problemas reales como el mapeo incorrecto de los GPIOs, comprendimos que trabajar con drivers no se trata solamente de programar, sino también de entender cómo interactúan el hardware y el sistema operativo.

Aprendimos a utilizar interfaces modernas como `gpiod`, a lidiar con errores de carga de módulos, y a interpretar herramientas del sistema como `/sys/kernel/debug/gpio`. Estos desafíos nos obligaron a investigar más allá del código y a tener en cuenta aspectos como la configuración del sistema y el diseño del kernel.

En definitiva, este TP nos dejó como aprendizaje que desarrollar drivers implica trabajar a bajo nivel, donde cada detalle importa, y que una comprensión profunda del entorno de ejecución es tan importante como la implementación misma.

Referencias

[1] `https://github.com/KevinGTH/0XC0DE-TP5`