



Universidad Nacional de Córdoba

Facultad de Ciencias Exactas, Físicas y Naturales

Sistemas de computación

TP5: Driver Device

Grupo:

Epsilon

Profesores (por orden alfabético):

Jorge, Javier Alejandro

Lamberti, Germán Andrés

Solinas, Miguel Ángel

Alumnos (por orden alfabético):

Campos, Mariano

Erlicher, Ezequiel

González, Damián Marcelo



Consigna:	3
Desarrollo:	3
¿Qué es un módulo de Linux?	3
Drivers de caracteres	3
Placa de desarrollo	4
Instalar el sistema operativo: Raspberry Pi OS Lite	5
Instalar herramientas de desarrollo	7
Generación de señales digitales	8
Driver	12
Aplicación de nivel de usuario	22
Showcase del trabajo	24



[Link a repositorio](#) (a uno de ellos, los otros son fork de este, y están sincronizados).

Consigna:

Para superar este TP tendrán que diseñar y construir un CDD que permita sensar dos señales externas con un periodo de UN segundo. Luego una aplicación a nivel de usuario deberá leer UNA de las dos señales y graficarla en función del tiempo. La aplicación también debe poder indicarle al CDD cuál de las dos señales leer. Las correcciones de escalas de las mediciones, de ser necesario, se harán a nivel de usuario. Los gráficos de la señal deben indicar el tipo de señal que se está sensando, unidades en abscisas y tiempo en ordenadas. Cuando se cambie de señal el gráfico se debe "resetear" y acomodar a la nueva medición. Se recomienda utilizar una Raspberry Pi para desarrollar este TP.

Desarrollo:

¿Qué es un módulo de Linux?

Un módulo de Linux es una pieza de código que se puede cargar y descargar dinámicamente en el kernel sin necesidad de reiniciar el sistema. Su principal función es extender las capacidades del kernel, permitiéndole manejar nuevo hardware o agregar funcionalidades como sistemas de archivos, protocolos de red o controladores de dispositivos. Se utiliza principalmente para crear drivers (por ejemplo, para manejar GPIO, I2C, SPI, ADC, etc.), sistemas de archivos (como ext4, FAT), y soporte para distintos protocolos o periféricos (como Bluetooth, USB, cámaras, etc.). Los módulos pueden clasificarse en tres tipos principales: drivers de dispositivo (como los de carácter, bloque o red), sistemas de archivos y módulos de red. Esta modularidad permite mantener el kernel liviano y flexible, cargando solo lo necesario según el hardware o las necesidades del sistema.

Drivers de caracteres

Los drivers de carácter (Character Device Drivers, CDD) son un tipo específico de controlador en Linux diseñados para manejar dispositivos que transmiten datos carácter por carácter (byte a byte), como si fueran flujos secuenciales de información. A diferencia de los drivers de bloque (como discos duros), que manejan datos en bloques grandes, los CDD son ideales para dispositivos como puertos serie, sensores, ADCs, DACs, pantallas OLED/I2C, teclados o dispositivos GPIO personalizados. Sirve para permitir que programas en espacio de usuario (como



scripts en C, Python, Bash, etc.) interactúen con hardware personalizado mediante operaciones estándar del sistema operativo:

- `open()`: Abrir el dispositivo.
- `read()`: Leer datos desde el dispositivo.
- `write()`: Escribir datos hacia el dispositivo.
- `ioctl()`: Enviar comandos de control personalizados.
- `close()`: Cerrar el dispositivo.

Placa de desarrollo

Para realizar el trabajo práctico se utilizó una placa Raspberry Pi Zero 2W con las siguientes características: Placa reducida (SBC, Single Board Computer) que integra un SoC Broadcom BCM2710A1, el mismo núcleo utilizado en la Raspberry Pi 3, pero con una frecuencia de 1GHz. Este SoC incluye un CPU ARM Cortex-A53 de 64 bits, de cuatro núcleos, que proporciona una mejora sustancial en rendimiento comparado con el modelo Zero original. La placa tiene 512MB de RAM LPDDR2 integrada.

Conectividad:

- WiFi 802.11 b/g/n (2,4GHz) y Bluetooth 4.2 / BLE.
- GPIO de 40 pines con soporte para múltiples interfaces:
- 2× SPI
- 2× I2C
- 2× UART
- 8 canales PWM (dependientes de multiplexación) GPIOs digitales programables con funciones alternativas.
- 1× microUSB OTG para datos (permite conexión de periféricos USB mediante un hub o adaptador).
- 1× microUSB para alimentación.
- 1× mini HDMI para salida de video.
- 1× CSI (Camera Serial Interface) para conectar una cámara oficial de Raspberry Pi.
- Slot para tarjeta microSD, donde se instala el sistema operativo y el almacenamiento principal.

En cuanto a energía, la placa se alimenta típicamente con 5V a través del puerto micro USB. El consumo depende del uso de CPU, Wifi y periféricos, y ronda los 100–200 mA en reposo. No posee RTC ni almacenamiento persistente más allá de la microSD.



Instalar el sistema operativo: Raspberry Pi OS Lite

El primer paso es instalar el sistema operativo en la placa de desarrollo mediante Raspberry Pi Imager, esta es una herramienta oficial desarrollada por la Fundación Raspberry Pi que permite grabar sistemas operativos en tarjetas microSD de forma fácil, rápida y segura. Es compatible con Windows, macOS y Linux.

Este software se puede descargar de: <https://www.raspberrypi.com/software/>

Nos permite:

- Grabar sistemas operativos como Raspberry Pi OS, Ubuntu, LibreELEC, etc, directamente en una microSD o unidad USB.
- Personalizar configuraciones previas al arranque, como habilitar SSH, configurar WiFi, o establecer un hostname.
- Verificar la integridad de la imagen grabada automáticamente.

El procedimiento de instalación es el siguiente:

- Insertá tu microSD en la PC o lector USB.
- Abrí Raspberry Pi Imager y seguí estos pasos:
- Choose OS: seleccioná el sistema operativo (por ejemplo, Raspberry Pi OS Lite).
- Choose Storage: seleccioná la microSD conectada.
- Hacé clic en "Write". El proceso formatea la tarjeta, copia la imagen

Una vez flasheada la imagen en la microSD, se conecta a la alimentación y ya esta lista para ser utilizada. Se accede a la placa mediante SSH para instalar las herramientas necesarias.



Dispositivo Raspberry Pi

RASPBERRY PI ZERO 2 W

Sistema operativo


RASPBERRY PI OS LITE (64-BIT)

Almacenamiento

ELEGIR ALMACENAMIENTO

SIGUIENTE

Figura 1: Raspberry PI imager (flashea microSD para el sistema operativo)



Personalización del SO

GENERAL SERVICIOS OPCIONES

☒ Establecer nombre de anfitrión: .local

☒ Establecer nombre de usuario y contraseña

Nombre de usuario:

Contraseña:

☒ Configurar LAN inalámbrica

SSID:

Contraseña:

☐ Mostrar contraseña ☐ SSID oculta

País de LAN inalámbrica: ▼

☒ Establecer ajustes regionales

Zona horaria: ▼

Distribución del teclado: ▼

GUARDAR

Figura 2: Configuraciones iniciales del sistema operativo (Acceso a la red y SSH)

Instalar herramientas de desarrollo

Para poder desarrollar y compilar drivers en Raspberry Pi OS Lite, es necesario instalar ciertas herramientas de desarrollo. Esto incluye el paquete **build-essential**, que proporciona el compilador **gcc**, **make** y otras utilidades necesarias, y los encabezados del **kernel (raspberrypi-kernel -headers)**, que permiten compilar módulos compatibles con el núcleo actual. Estos se instalan desde la terminal con **sudo apt update && sudo apt install -y build-essential raspberrypi-kernel-headers**. También es útil instalar **git** para clonar repositorios de ejemplo. Una vez instalados, se debe verificar que el directorio **/lib/modules/(uname -r)/build** exista, lo cual indica que los headers están correctamente disponibles. Esta configuración es indispensable para compilar drivers de carácter (CDD) y otros módulos del kernel desde la Raspberry Pi.

```
mariano-campos@mariano-campos-HP-Laptop-14-dk1xxx:~$ sudo nmap -sP 192.168.1.0/24
[sudo] contraseña para mariano-campos:
Starting Nmap 7.94SVN ( https://nmap.org ) at 2025-05-30 22:15 -03
Nmap scan report for _gateway (192.168.1.1)
Host is up (0.0043s latency).
MAC Address: E8:4D:74:68:24:BC (Huawei Technologies)
Nmap scan report for 192.168.1.8
Host is up (0.051s latency).
MAC Address: 32:6F:EB:24:48:FE (Unknown)
Nmap scan report for 192.168.1.15
Host is up (0.17s latency).
MAC Address: 26:2E:98:70:B6:32 (Unknown)
Nmap scan report for 192.168.1.30
Host is up (0.63s latency).
MAC Address: 2C:CF:67:99:96:9C (Unknown)
Nmap scan report for mariano-campos-HP-Laptop-14-dk1xxx (192.168.1.4)
Host is up.
Nmap done: 256 IP addresses (5 hosts up) scanned in 7.31 seconds
```

Figura 4: Búsqueda IP de la Raspberry PI (192.168.1.30)

```
mariano-campos@mariano-campos-HP-Laptop-14-dk1xxx:~$ ssh RBmariano@192.168.1.30
RBmariano@192.168.1.30's password:
Linux RBmariano 6.12.25+rpt-rpi-v8 #1 SMP PREEMPT Debian 1:6.12.25-1+rpt1 (2025-04-30) aarch64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Fri May 30 23:48:54 2025 from 192.168.1.4
RBmariano@RBmariano:~$ cat /proc/cpuinfo
processor       : 0
BogoMIPS      : 38.40
Features       : fp asimd evtstrm crc32 cpuid
CPU implementer : 0x41
CPU architecture: 8
CPU variant    : 0x0
CPU part       : 0xd03
CPU revision   : 4
```

Figura 5: Acceso a la terminal de la placa mediante SSH

Generación de señales digitales

Este programa en C, diseñado para ejecutarse en una Raspberry Pi moderna con sistema operativo basado en Linux (como Raspberry Pi OS Bookworm), genera señales digitales cuadradas en dos pines GPIO utilizando la biblioteca **libgpiod**, que reemplaza el antiguo sistema de control GPIO basado en archivos `/sys/class/gpio`. El código define dos pines de salida (GPIO14 y GPIO15) para generar señales de con periodo arbitrario. Usando funciones de **libgpiod**, el programa accede al chip GPIO principal (`/dev/gpiochip0`), solicita el control exclusivo de las líneas deseadas y luego entra en un bucle infinito donde alterna los valores (encendido/apagado) de cada pin en intervalos definidos por los argumentos de línea de comandos. Esto se



logra utilizando una función `delay_ms()` que implementa retardos precisos con `nanosleep`, y contadores que determinan cuándo invertir el estado lógico de cada salida.

```
#include <gpiod.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

#define GPIO_CHIP "/dev/gpiochip0"

// Pines GPIO (números Broadcom GPIO)
#define GPIO_OUT1 23 // Salida 1Hz
#define GPIO_OUT2 24 // Salida 2Hz

void delay_ms(int ms) {
    struct timespec ts = { ms / 1000, (ms % 1000) * 1000000 };
    nanosleep(&ts, NULL);
}

int main(int argc, char* argv[]) {
    struct gpiod_chip *chip;
    struct gpiod_line *line_out1, *line_out2;
    int ret;
    int out1_state = 0, out2_state = 0;
    int counter_1Hz = 0, counter_2Hz = 0;
    // Abrir el gpiochip0
    chip = gpiod_chip_open(GPIO_CHIP);
    if (!chip) {
        perror("Error al abrir gpiochip");
        return 1;
    }
    // Obtener líneas
    line_out1 = gpiod_chip_get_line(chip, GPIO_OUT1);
    line_out2 = gpiod_chip_get_line(chip, GPIO_OUT2);
    if (!line_out1 || !line_out2) {
        fprintf(stderr, "Error obteniendo líneas GPIO\n");
        gpiod_chip_close(chip);
        return 1;
    }
    // Configurar líneas de salida
```

```
ret = gpiod_line_request_output(line_out1, "gen_signal", 0);  
if (ret < 0) { perror("Error request output line_out1"); return 1; }  
ret = gpiod_line_request_output(line_out2, "gen_signal", 0);  
if (ret < 0) { perror("Error request output line_out2"); return 1; }  
printf("Iniciando loop principal...\n");
```

```
while (1) {
```

```
    // Toggle salida 1 cada X ms
```

```
    if (counter_1Hz >= atoi(argv[1])) {  
        out1_state = !out1_state;  
        gpiod_line_set_value(line_out1, out1_state);  
        printf("GPIO %d set a %d\n", GPIO_OUT1, out1_state);  
        counter_1Hz = 0;  
    }
```

```
}
```

```
    // Toggle salida 2 cada Y ms
```

```
    if (counter_2Hz >= atoi(argv[2])) {  
        out2_state = !out2_state;  
        gpiod_line_set_value(line_out2, out2_state);  
        printf("GPIO %d set a %d\n", GPIO_OUT2, out2_state);  
        counter_2Hz = 0;  
    }
```

```
}
```

```
    delay_ms(10);
```

```
    counter_1Hz += 10;
```

```
    counter_2Hz += 10;
```

```
}
```

```
    // Nunca llega acá pero por limpieza
```

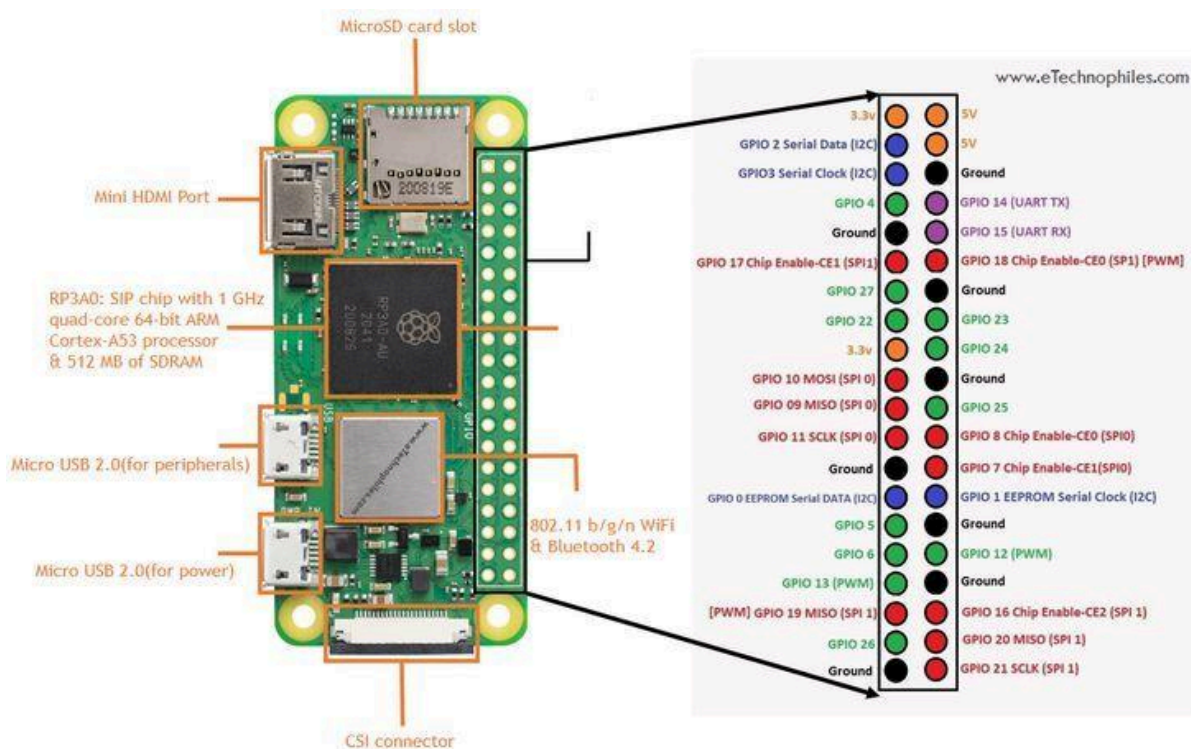
```
    gpiod_chip_close(chip);
```

```
    return 0;
```

```
}
```

```
RBmariano@RBmariano:~ $ nano gen_signal.c
RBmariano@RBmariano:~ $ gcc gen_signal.c -o gen_signal -lgpiod
RBmariano@RBmariano:~ $ ./gen_signal 1000 500
Iniciando loop principal...
GPIO 15 set a 1
GPIO 14 set a 1
GPIO 15 set a 0
GPIO 15 set a 1
GPIO 14 set a 0
GPIO 15 set a 0
GPIO 15 set a 1
GPIO 14 set a 1
GPIO 15 set a 0
GPIO 15 set a 1
^C
RBmariano@RBmariano:~ $
```

Figura 6: Compilación y prueba del programa



Implementación del driver

Este módulo del kernel de Linux , implementa un driver de muestreo de señales que lee periódicamente la entrada digital de dos pines GPIO (PIN_A y PIN_B) y hace que estas muestras estén disponibles para el espacio del usuario a través de (/dev/signals_reader). El funcionamiento general del mismo es el siguiente:

- Primero se definen las variables y estructuras que se utilizan en todo el código:

```
/* GPIO pins */
#define PIN_A 526 // pin 14
#define PIN_B 527 // pin 15
#define SAMPLE_BUFFER_SIZE 50

/* Globals */
static char sample_buffer_PIN_A[SAMPLE_BUFFER_SIZE];
static char sample_buffer_PIN_B[SAMPLE_BUFFER_SIZE];
static int sample_index = 0;
static int sample_count = 0;
static dev_t first; // first device number
static struct cdev c_dev; // character device structure
static struct class *cl; // device class
static struct delayed_work my_work; // Delayed work structure to poll GPIO periodically
static struct workqueue_struct *my_wq; // Workqueue structure
static int read_period = HZ/5; //2 second delay
```

-PIN_A y PIN_B son los números de pin GPIO (524 y 525), estos son específicos del hardware.

-Dos búferes que almacenan 50 muestras para cada uno de los pines

-sample_index es el índice actual para escribir la siguiente muestra en los buffers .

-sample_count registra cuántas muestras válidas se han almacenado .

-first: Esta variable almacena el número de dispositivo asignado al driver. Es una combinación de números mayores y menores que se utiliza para identificar de forma única el dispositivo en el sistema.

-c_dev: Es una estructura que permite configurar las operaciones de archivo que el dispositivo de caracteres de este driver admite. Permite que los programas de



usuario interactúen con el controlador mediante operaciones de archivo estándar (abrir, leer, liberar, etc.).

-*cl: Puntero a una estructura de tipo `class` que se utiliza para crear device nodes en `/dev/` y `/sys/class/`

-my_work: Es una estructura de tipo `delayed_work` que define la función que se llamará luego de un cierto delay. En nuestro caso, se usa para muestrear periódicamente los pines GPIO.

-my_wq: Puntero a la `workqueue` que gestiona la ejecución del `delayed_work`. Es un hilo del kernel que se utiliza para “liberar” a la ruta de ejecución principal del kernel de la tarea de “polling” de los pines de GPIO .

Funciones:

-Las funciones `my_open()` y `my_release()` solo registran cuándo se abre o se cierra el dispositivo.

-`take_sample()`: Utiliza `gpio_get_value()` para leer la entrada digital en los pines, convierte dichos valores a '1' o '0', almacena los valores en el búfer circular y poner en cola a la estructura `delayed_work` para que luego de 200ms vuelva a ejecutarse.

-`my_read()`: Se copia un string de 100 caracteres a un buffer en el espacio de usuario, donde los primeros 50 representan las últimas 50 muestras de `PIN_A` y los siguientes 50 caracteres, las últimas 50 muestras correspondientes a `PIN_B`

-`my_exit()`: Esta función, la cual se llama cuando se desea quitar el módulo, destruye la `workqueue`, elimina el dispositivo de caracteres junto con la clase y libera los GPIO.

Código:

```
/**
 * @file gpio_driver.c
 * @brief GPIO driver for Linux kernel
 */

#include <linux/init.h>
```



```
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/fs.h>
#include <linux/gpio.h>
#include <linux/kdev_t.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/types.h>
#include <linux/uaccess.h>
#include <linux/version.h>
#include <linux/workqueue.h>

/* GPIO pins */
#define PIN_A 526 // pin 14
#define PIN_B 527 // pin 15
#define SAMPLE_BUFFER_SIZE 50

/* Globals */
static char sample_buffer_PIN_A[SAMPLE_BUFFER_SIZE];
static char sample_buffer_PIN_B[SAMPLE_BUFFER_SIZE];
static int sample_index = 0;
static int sample_count = 0;
static dev_t first; // first device number
static struct cdev c_dev; // character device structure
static struct class *cl; // device class
static struct delayed_work my_work; // Delayed work structure
to poll GPIO periodically
static struct workqueue_struct *my_wq; // Workqueue structure
static int read_period = HZ/5; // 2 second delay

/* ----- */
/* File operations */
/* ----- */
/**
 * @brief Release function for the device
 * @param inode Pointer to the inode structure
 * @param file Pointer to the file structure
 * @return 0 on success
 */
```



```
static int my_release(struct inode *inode, struct file *file) {
    printk("Closing signals sampler\n");
    return 0;
}

/**
 * @brief Open function for the device
 * @param i Pointer to the inode structure
 * @param f Pointer to the file structure
 * @return 0 on success
 */
static int my_open(struct inode *i, struct file *f) {
    printk("Opening signals sampler\n");
    return 0;
}

/**
 *
 * Copies the sampled values to userspace.
 * @param file Pointer to the file structure
 * @param buf A pointer to the user-space buffer where the data
    should be copied to.
 * @param num_of_bytes Number of bytes to read
 * @param offset The current file offset; used to prevent multiple
    reads returning the same data.
 * @return Number of bytes read on success, negative value on
    failure
 */
ssize_t my_read(struct file *file, char __user *buf, size_t
num_of_bytes, loff_t *offset) {

    char temp_buffer[2 * SAMPLE_BUFFER_SIZE] = {0};
    int i, index;

    if (*offset > 0)
        return 0; // Only allow one read per open

    for (i = 0; i < sample_count; i++) {
```



```
    index = (sample_index + SAMPLE_BUFFER_SIZE - sample_count + i)
% SAMPLE_BUFFER_SIZE;
    temp_buffer[i] = sample_buffer_PIN_A[index];           // First
signal
                    temp_buffer[SAMPLE_BUFFER_SIZE      +      i]      =
sample_buffer_PIN_B[index]; // Second signal
}

if (copy_to_user(buf, temp_buffer, 2 * sample_count))
    return -EFAULT;

*offset += 2 * sample_count;

return 2 * sample_count;

}

/** File operations structure */
static struct file_operations my_fops = {
    .read = my_read,
    .open = my_open,
    .release = my_release,
};

/**
 * Called every second 200ms .Reads current GPIOs values and stores
them .
 * Re-enqueues itself for continuous polling.
 *
 * @param work Pointer to the work structure
 */
static void take_sample(struct work_struct *work) {

    u8 value_a = gpio_get_value(PIN_A);
    u8 value_b = gpio_get_value(PIN_B);

    sample_buffer_PIN_A[sample_index] = value_a ? '1' : '0';
    sample_buffer_PIN_B[sample_index] = value_b ? '1' : '0';
```




```
sample_index = (sample_index + 1) % SAMPLE_BUFFER_SIZE;

if (sample_count < SAMPLE_BUFFER_SIZE)
    sample_count++;

queue_delayed_work(my_wq, &my_work, read_period);
}

/* ----- */
/* Lifecycle functions */
/* ----- */

/**
GPIO Setup:

    -Request IO_1 and IO_2 using gpio_request.
    -Set them as input using gpio_direction_input.

Character Device Setup:

    -Allocate a major/minor number with alloc_chrdev_region.
    -Create device class and /dev/signals_reader.
    -Register cdev with my_fops.

Workqueue Setup:

    -Create a single-threaded workqueue.
    -Initialize and schedule the first delayed_work.

* @return 0 on success, negative value on failure
*/
int __init my_init(void) {
    printk("Initializing ports\n");

    int status;
    int ret;
```



```
struct device *dev_ret;

// request GPIO pins

status = gpio_request(PIN_A, "PIN_A");
if (status != 0) {
    printk("Error requesting PIN_A\n");
    return status;
}

status = gpio_request(PIN_B, "PIN_B");
if (status != 0) {
    printk("Error requesting PIN_B\n");
    gpio_free(PIN_A);
    return status;
}

// set pins as inputs
status = gpio_direction_input(PIN_A);
if (status != 0) {
    printk("Error setting direction input PIN_A\n");
    gpio_free(PIN_A);
    gpio_free(PIN_B);
    return status;
}

status = gpio_direction_input(PIN_B);
if (status != 0) {
    printk("Error setting direction input PIN_B\n");
    gpio_free(PIN_A);
    gpio_free(PIN_B);
    return status;
}

/**
 * Dynamically allocates a major number (and one minor number
starting from 0)
 * for the character device.
```



```
* first: where the assigned device number will be stored.
* 0: starting minor number.
* 1: number of devices.
* "signals_reader": name for /proc/devices.
*/
if ((ret = alloc_chrdev_region(&first, 0, 1, "signals_reader")) <
0) {
    gpio_free(PIN_A);
    gpio_free(PIN_B);
    return ret;
}

// Creates a device class which will appear under
/sys/class/signals_reader.
// Purpose: This class is used later to create the actual device
node in /dev/.
if (IS_ERR(cl = class_create(THIS_MODULE, "signals_reader"))) {
    unregister_chrdev_region(first, 1);
    gpio_free(PIN_A);
    gpio_free(PIN_B);
    return PTR_ERR(cl);
}

/**
 * Creates /dev/signals_reader and links it to the driver.
 *
 * cl: the previously created class
 * first: device number.
 * "signals_reader": the filename in /dev.
 *
 * Result: Now user programs can open /dev/signals_reader and
interact with the driver
 * using read, write, etc.
*/
if (IS_ERR(dev_ret = device_create(cl, NULL, first, NULL,
"signals_reader"))) {
    class_destroy(cl);
    unregister_chrdev_region(first, 1);
    gpio_free(PIN_A);
```



```
    gpio_free(PIN_B);
    return PTR_ERR(dev_ret);
}

/**
 * cdev_init: Initializes the cdev structure with custom
file_operations (read, write, etc.).
 * cdev_add: Adds the character device to the kernel so it can
start handling syscalls.
 * If it fails it destroys the device node, class, unregister the
device number, and free the GPIOs.
 */

cdev_init(&c_dev, &my_fops);
if ((ret = cdev_add(&c_dev, first, 1)) < 0) {
    device_destroy(cl, first);
    class_destroy(cl);
    unregister_chrdev_region(first, 1);
    gpio_free(PIN_A);
    gpio_free(PIN_B);
    return ret;
}

// Create workqueue
my_wq = create_singlethread_workqueue("my_wq");
if (!my_wq) {
    cdev_del(&c_dev);
    device_destroy(cl, first);
    class_destroy(cl);
    unregister_chrdev_region(first, 1);
    gpio_free(PIN_A);
    gpio_free(PIN_B);
    return -ENOMEM;
}

// Initialize delayed work
INIT_DELAYED_WORK(&my_work, take_sample);

// Queue the first work
queue_delayed_work(my_wq, &my_work, read_period);
```



```
    return 0;
}

/**
 * @brief Exit function
 */
void __exit my_exit(void) {
    printk("Goodbye! \n");

    // Cancel the delayed work and destroy the workqueue
    cancel_delayed_work_sync(&my_work);
    destroy_workqueue(my_wq);
    // Unregister device
    cdev_del(&c_dev);
    device_destroy(cl, first);
    class_destroy(cl);
    unregister_chrdev_region(first, 1);
    // free gpios
    gpio_free(PIN_A);
    gpio_free(PIN_B);
}

// register functions
module_init(my_init);
module_exit(my_exit);

/* Module information */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Grupo:Epsilon");
MODULE_DESCRIPTION("Signal sampler driver");
```

Aplicación de nivel de usuario

Para la aplicación de alto nivel se decidió utilizar **Ruby** como lenguaje de programación. Ruby es una herramienta que viene por defecto en la mayoría de las distribuciones GNU/Linux basadas en Debian, por lo que no hay que instalar mas que las libraries externas utilizadas.

Las libraries externas utilizadas son las siguientes:

```
1  ruby '~>3.0'
2
3  source 'https://rubygems.org'
4
5  group :production do
6    gem 'rack', '~> 3.1', '>= 3.1.15'
7    gem 'puma', '~> 6.6'
8    gem 'net-ssh', '~> 7.3'
9  end
```

Figura 8: Ruby's Gemfile

- **rack**: de su [información oficial](#): “Rack provides a minimal, modular and adaptable interface for developing web applications in Ruby. By wrapping HTTP requests and responses in the simplest way possible, it unifies and distills the API for web servers, web frameworks, and software in between (the so-called middleware) into a single method call.”
- **puma**: de su [información oficial](#): “Puma is a simple, fast, multi-threaded, and highly parallel HTTP 1.1 server for Ruby/Rack applications. Puma is intended for use in both development and production environments. It's great for highly parallel Ruby implementations such as JRuby and TruffleRuby as well as as providing process worker support to support CRuby well.”
- **net-ssh**: de su [información oficial](#): “Net::SSH: a pure-Ruby implementation of the SSH2 client protocol. It allows you to write programs that invoke and interact with processes on remote servers, via SSH2.”

Como se puede advertir, la aplicación de alto nivel es un servidor web HTTP.

La comunicación con la placa Raspberry Pi se hace mediante SSH, wireless, ya que la misma posee capacidades de WiFi.

Además de toda la algarabía que hace al front-end de la aplicación (vista, estilo, javascript's scripts) que es irrelevante para el foco de este trabajo, está la parte mas importante, que es aquella que realiza la comunicación contra la placa:

```

33 # @param req [Rock::Request]
34 # @return [Array(Integer, Hash<String, String>, Array<String>)]
35 def route(req)
36   if((@req = req).get?)
37     case(req.path)
38       when('/')
39         # according to `cat CDD_PATH_ON_RASPBERRY_PI` expecting binary string i.e.: "01010101011001010101001"
40         raw_signals_data = \
41           Net::SSH.start(ENV['RASPBERRY_PI_IP'], ENV['RASPBERRY_PI_HOST_NAME'], password: ENV['RASPBERRY_PI_PASSWORD']) do |ssh|
42             ssh.exec!(`cat #{CDD_PATH_ON_RASPBERRY_PI}`)
43           end
44         @signals_data = [raw_signals_data.chars[0...SAMPLES_PER_SIGNAL].map {|c| c.to_i}, raw_signals_data.chars[SAMPLES_PER_SIGNAL..-1].map {|c| c.to_i}]
45         [200, HTTP_HEADER_TEMPLATE, [render('main')]]
46       when('/alive')
47         # alive check, test/debugging purpose
48         [200, {'Content-Type' => 'text/plain'}, ['Alive']]
49       else
50         DEFAULT_RESPONSE
51     end
52   else
53     DEFAULT_RESPONSE
54   end
55 end

```

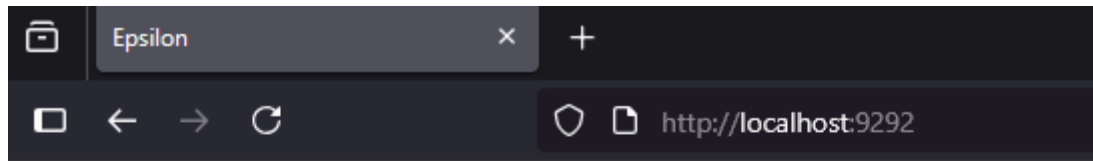
Figura 9: Método route() del servidor HTTP.

Lo mas interesante se encuentra entre las **lineas 39 y 44**. Allí se inicia una sesión SSH contra la placa Raspberry Pi. Los valores de autenticación fueron cargados con antelación via variables de ambiente. Una vez *dentro de la placa*, se ejecuta el comando:

``cat CDD_PATH_ON_RASPBERRY_PI``, donde ese *placeholder* toma el valor del path al CDD driver, por ej.: `"/dev/signal_reader"`.

Aquello retornado por el `cat`, se nombra *raw_signals_data*. Se espera obtener, en forma de string, los dos buffers correspondientes a cada señal. Esa string luego es *parseada*, convirtiéndola en un útil array de valores enteros o flotantes en *@signals_data*.

Por último lo que sucede es el renderizado de la vista, que toma la información contenida en *@signals_data* y la muestra de forma *bonita*. La interfaz gráfica también permite switchear entre visualizar una u otra señal. Para refrescar los datos (sampleo) de las señales, vale con hacer un refresco de la página principal.



FCEFN UNC

Sistemas de Computación

TP 5: user level app

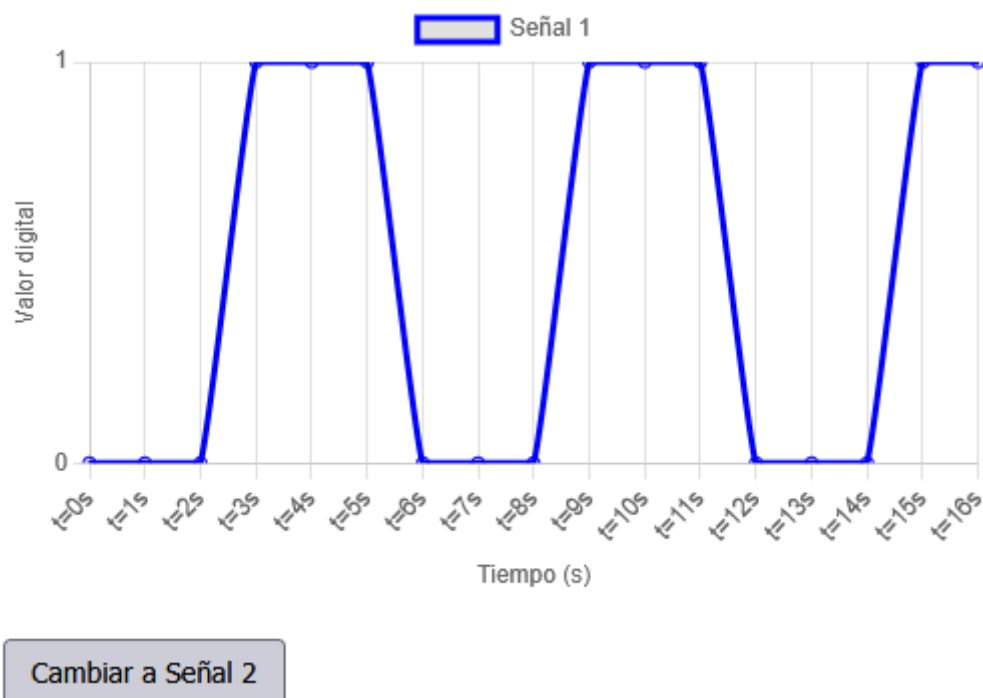


Figura 10: Vista (GUI) de la aplicación de nivel de usuario.

Showcase del trabajo

- [Parte 1.](#)
- [Parte 2.](#)