

***Universidad Nacional de Córdoba***

*Facultad de Ciencias Exactas, Físicas y Naturales*

*Sistemas de computación*

*TP5: Driver Device*

*Grupo:*

*Epsilon*

*Profesores (por órden alfabético):*

*Jorge, Javier Alejandro*

*Lamberti, Germán Andrés*

*Solinas, Miguel Ángel*

*Alumnos (por órden alfabético):*

*Campos, Mariano*

*Erlicher,Ezequiel*

*González, Damián Marcelo*

[**Consigna: 3**](#_hjxai32q5rgz)

[**Desarrollo: 3**](#_khfqb73i35jq)

[¿Qué es un módulo de Linux? 3](#_1pfibh139hsh)

[Drivers de caracteres 3](#_oz9jxg42cl0h)

[Placa de desarrollo 4](#_93fnv8w6iimv)

[Instalar el sistema operativo: Raspberry Pi OS Lite 4](#_795ic71arbc9)

[Instalar herramientas de desarrollo 6](#_lunm2j6dtzgh)

[Generación de señales digitales 7](#_6zrwf0skn4ay)

# 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 tambien 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 abcisas 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/>

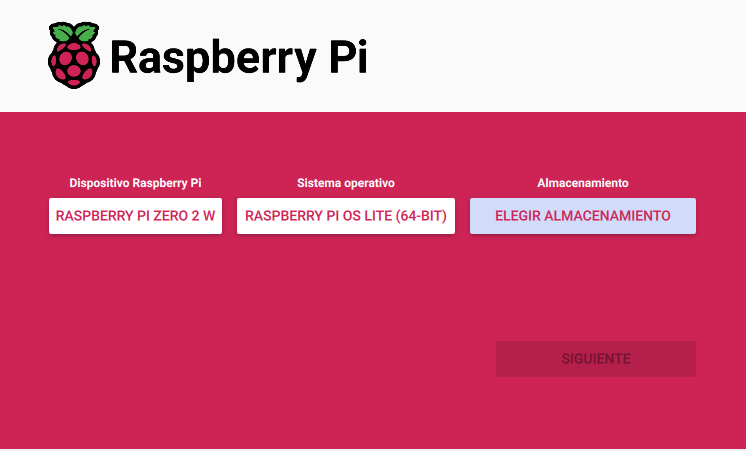
Nos permite:

* 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.

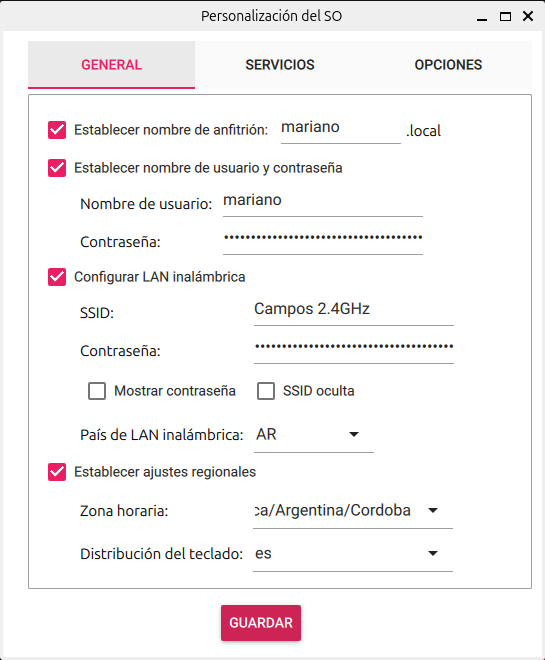
El procedimiento de instalación es el siguiente:

* 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.



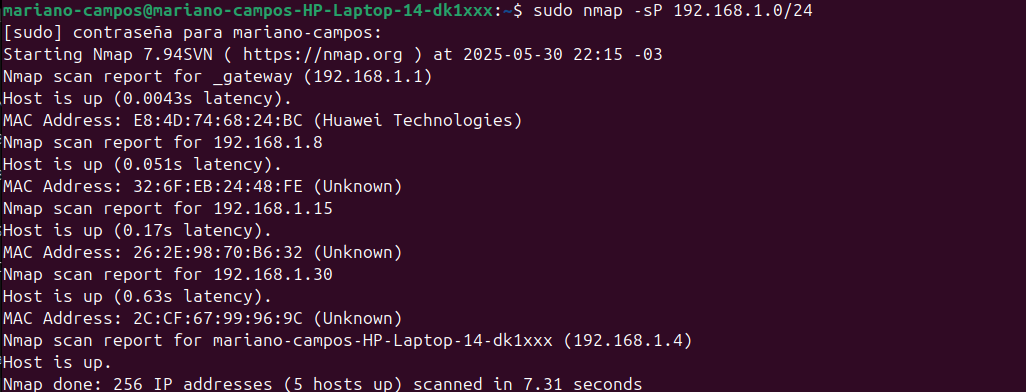
*Figura 1: Raspberry PI imager (flashea microSD para el sistema operativo)*



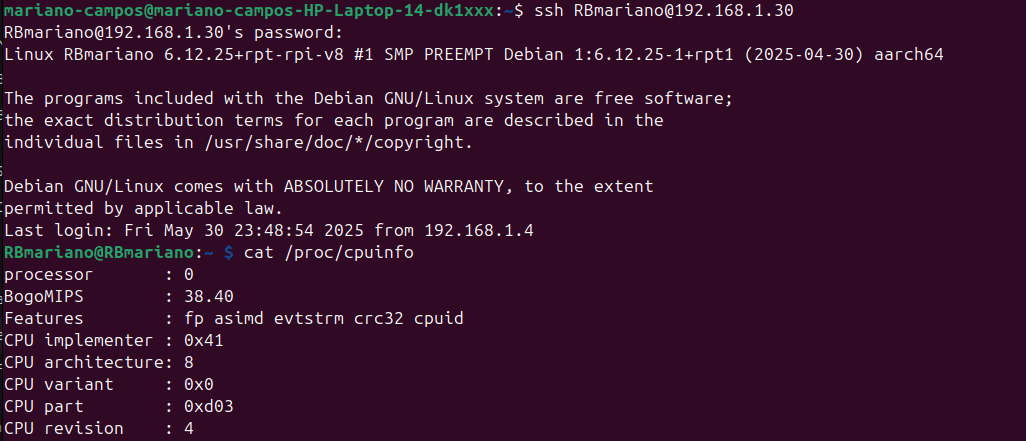
*Figura 2: Configuraciones iniciales del sistema operativo (Acceso a la red y SSH)*

## 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 **sudoaptupdate&&sudoaptinstall −ybuild − essentialraspberrypi − 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.



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



*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;

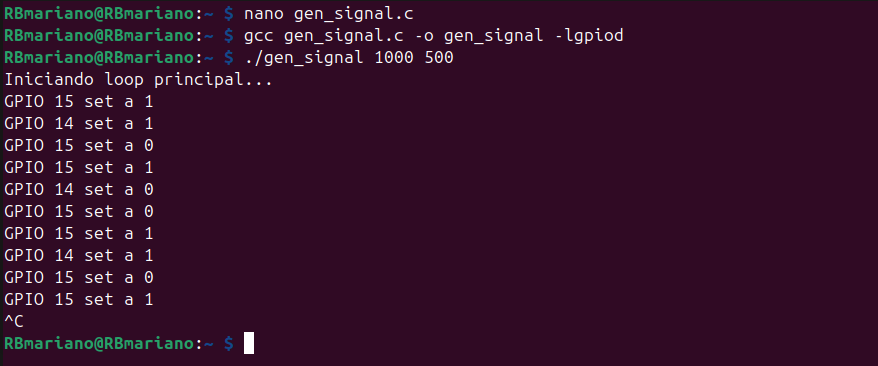
}

// Nunca llega acá pero por limpieza

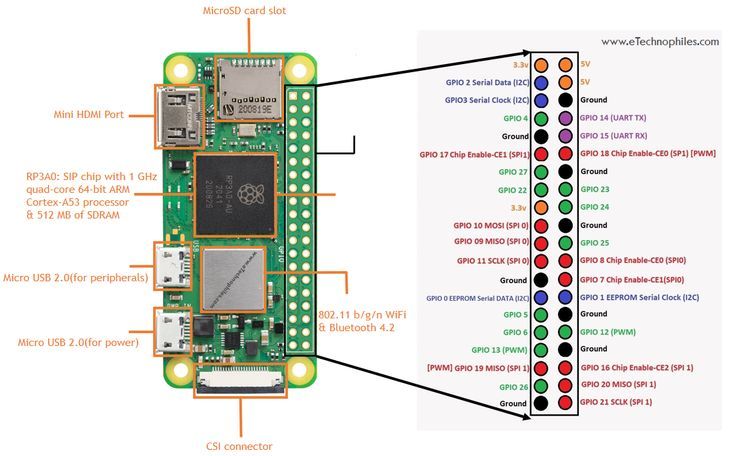
gpiod\_chip\_close(chip);

return 0;

}



*Figura 6: Compilación y prueba del programa*

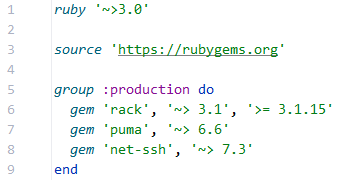


*Figura 7: pinout de la placa*

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:

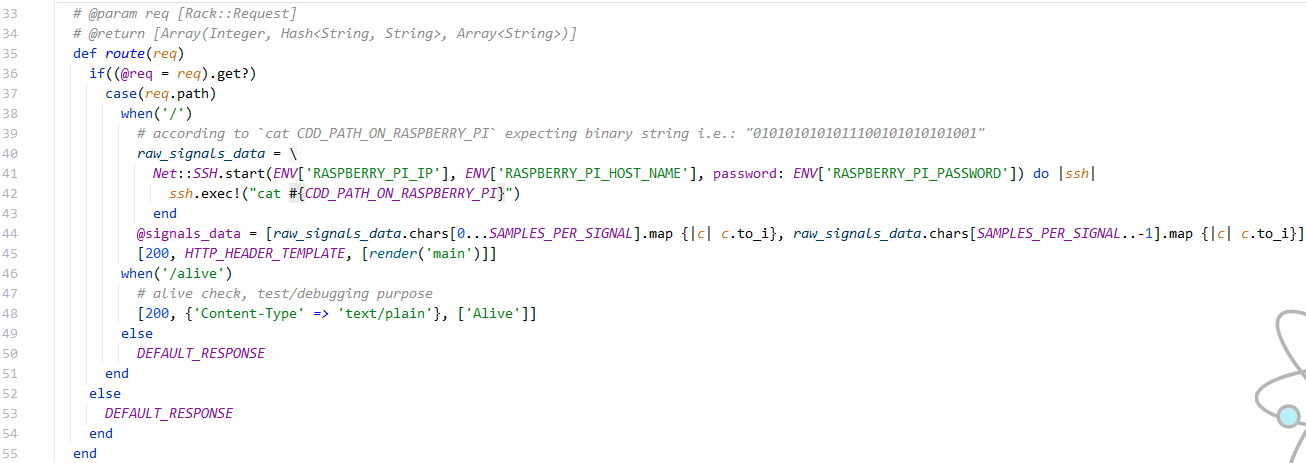
  
*Figura 7: Ruby’s Gemfile*

* **rack**: de su [información oficial](https://rubygems.org/gems/rack): *“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](https://rubygems.org/gems/puma): *“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](https://rubygems.org/gems/net-ssh): *“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.

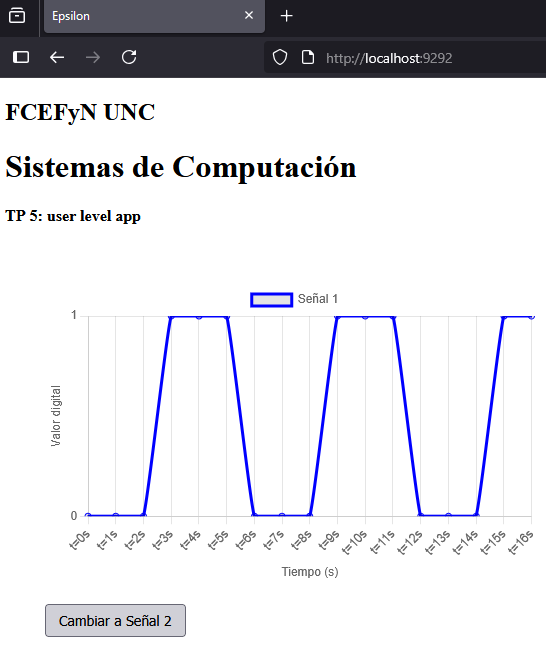
Además de toda la algarabía que hace al front-end de la aplicación (vista, estilo, javascript’s scripts) que es irrelevante para el foco de este trabajo, está la parte mas importante, que es aquella que realiza la comunicación contra la placa:

  
*Figura 8: 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 autentificación 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 nomencla *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.

*****Figura 9: Vista (GUI) de la aplicación de nivel de usuario.*

/\*\*

\* @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 int chosen\_pin = PIN\_A; // Selected GPIO to read

static struct delayed\_work my\_work; // Delayed work structure to poll GPIO periodically

static struct workqueue\_struct \*my\_wq; // Workqueue structure

static char last\_value[8] = "0\n"; // Buffer to store the last read value

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;

}

/\*\*

\*

\*

\* @param file Pointer to the file structure

\* @param buf User buffer to store data

\* @param num\_of\_bytes Number of bytes to read

\* @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;

}

/\*\*

\* It allows userspace programs to choose between two GPIO pins (IO\_A or IO\_B)

\* to read values from

\*

\* @param file Pointer to the file structure

\* @param buf User buffer containing data to write

\* @param num\_of\_bytes Number of bytes to write

\* @param off Offset within the file

\* @return Number of bytes written on success, negative value on failure

\*/

static ssize\_t my\_write(struct file \*file, const char \_\_user \*buf, size\_t num\_of\_bytes,

loff\_t \*off) {

char kbuf[1];

if (num\_of\_bytes == 0)

return 0;

if (copy\_from\_user(kbuf, buf, 1)) {

return -EFAULT;

}

printk("Write called with value: %c\n", kbuf[0]);

return num\_of\_bytes;

}

/\*\* File operations structure \*/

static struct file\_operations my\_fops = {

.read = my\_read,

.write = my\_write,

.open = my\_open,

.release = my\_release,

};

/\*\*

\* Called every second.Reads current GPIO value and stores it in last\_value.

\* 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 your 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 your 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 your 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");