BerryclipDriver

ESTRUCTURA DEL PROGRAMA

// ====== Definiciones previas: #include, #define y variables globales ======

// ====== GLOBAL BUTTONS VARIABLES ======

// ====== LEDS DEVICE FUNCTIONS ======
static ssize_t leds_read(struct file *file, char __user *buf, size_t len, loff_t *ppos) {...]
static ssize_t leds_write(struct file *file, const char __user *buf, size_t len, loff_t *ppos) {...]

- Lectura/escritura en los leds interpretando bits de control (modos 00/01/10/11 sobre los 6 LEDs)
- **leds_read()**: devuelve un byte con los 6 bits inferiores de leds_state (bits 6–7 a 0)
  Se asegura de que solo se lea un byte (Implementa EOF tras 1 byte).
- **leds_write()**: interpreta el byte escrito, aplicando la máscara contenida en los bits 7-8 a los 6 primeros bits (0-5). Actualiza leds_state y dispara cada GPIO para modificar los leds físicos.

//====== BUTTONS DEVICE FUNCTIONS ======
static int buttons_open(struct inode *inode, struct file *flip) {...]
static int buttons_release(struct inode *inode, struct file *flip) {...]
static ssize_t buttons_read(struct file *file, char __user *buf, size_t len, loff_t *ppos) {...]

- buttons_open(): controla exclusión mutua y O_NONBLOCK vs bloqueante.
- buttons_release(): libera la apertura exclusiva.
- buttons_read(): si el buffer está vacío, bloquea (o devuelve -EAGAIN si O_NONBLOCK), luego copia un carácter desde el buffer circular.

//====== SPEAKER DEVICE FUNCTIONS ======
static ssize_t speaker_write(struct file *file, const char __user *buf, size_t len, loff_t *ppos)

- Recibe un byte: si es '0' apaga el GPIO del altavoz; si no, lo enciende.
- /dev/speaker: “solo escritura, '0' apaga, otro valor enciende”.

//====== FILE OPERATIONS ======
static const struct file_operations leds_fops = {...];
static const struct file_operations buttons_fops = {...];
static const struct file_operations speaker_fops = {...];

- Asocia open/read/write/release de cada device con sus funciones.
- Enlaza operaciones de archivo (fops) para cada /dev/....

//====== DEVICE DECLARATION ======
static struct miscdevice leds_dev = {...];
static struct miscdevice buttons_dev = {...];
static struct miscdevice speaker_dev = {...];

- Define cómo se llamarán los nodos en /dev y qué permisos (.mode) tendrán.
- Crea tres devices diferentes: /dev/leds, /dev/buttons, /dev/speaker.

// ====== IRQ AND WORKQUEUE FUNCTIONS ======
static irqreturn_t button1_irq_handler(int irq, void *dev_id) {...]
static irqreturn_t button2_irq_handler(int irq, void *dev_id) {...]
static void debounce_work_func_b1(struct work_struct *work) {...]
static void debounce_work_func_b2(struct work_struct *work) {...]

- Uso de interrupciones y colas de trabajo (técnica deferred job - workqueue).
- Solución a rebotes configurable.
- wake_up_interruptible() para desbloquear read()
- Los Handlers de interrupción sirven para encolar un trabajo diferido (delayed_work) cuando cambia el pin del botón (al pulsar un botón) y filtran rebotes.
- debounce_work_func_*: tras debounce_time_ms, leen el GPIO, meten '1' o '2' en el buffer y despiertan a lectores.

// ====== DEVICES CONFIGURATION ======
static int r_devices_config(void)

- Registra los devices en /dev. Retorna <0 si hubiera error al crearlos.
- Llama a misc_ergister() para cada miscdevice. Si falla uno despierta a los ya creados.

// ====== GPIO AND IRQS CONFIGURATION ======
static void free_gpios(void)
static void free_irqs(void)
static int GPIO_config(void)
static int irqs_config(void)

- Se configuran GPIOs como entradas/salidas. Se solicitan y liberan IRQs.
- **GPIO_config()**: solicita cada pin GPIO.
  Fija dirección: salida para leds/speaker, entrada para botones).
- **irqs_config()**: obtiene números de IRQ con gpio_to_irq(), hace request_irq() para ambos botones, crea la workqueue y los delayed_work.
- **free_gpios() / free_irqs()**: liberan recursos en caso de cleanup_driver() o fallo en init.

//====== INIT AND CLEANUP FUNCTIONS ======
static int init_driver(void)
static void cleanup_driver(void)
module_init (init_driver);
module_exit (cleanup_driver);

- Indica qué funciones se encargan de la carga y descarga del módulo.
- Ambas funciones se encargan de cargar el módulo y de una limpieza completa de recursos al descargarlo.

MODULE_LICENSE("GPL");
MODULE_AUTHOR(DRIVER_AUTHOR);
MODULE_DESCRIPTION(DRIVER_DESC);

- Estas líneas sirven para definir el módulo de kernel.
- Contienen metadatos que convierten el .ko en un módulo válido y declaran licencia GPL.

RESUMEN ESQUEMÁTICO DE LOS REQUISITOS DE LA PRÁCTICA Y LAS FUNCIONES USADAS PARA IMPLEMENTARLOS

1.- LEDs (/dev/leds)

Requisitos: Leer y escribir un byte donde los bits 5–0 marcan el estado de 6 LEDs y los bits 7–6 determinan la operación (directo, OR, AND-complemento, XOR).

Implementación:

- **`static ssize_t leds_read(...)`**

Devuelve leds_state & 0x3F, un byte con los 6 bits inferiores.

- **`static ssize_t leds_write(...)`**

extrae `(val & 0xC0)>>6` para el modo y val `& 0x3F` para la máscara, actualiza `leds_state` y llama a `gpio_set_value()`.

Estructuras clave:

- `leds_state`,
- `leds_fops`,
- `leds_dev`,
- `gpio_request`
- `gpio_direction_output (en GPIO_config()).`

2.- Speaker (/dev/speaker)

Requisitos: Sólo escritura; un byte '0' apaga el GPIO, cualquier otro lo enciende.

Implementación:

```
static ssize_t speaker_write(...) →
      copy_from_user(&val, buf,1);
      gpio_set_value(GPIO_DEFAULT+GPIO_SPEAKER, val!='0');
```

Estructuras clave:

- `speaker_fops`,
- `speaker_dev`,
- `configuración GPIO (en GPIO_config()).`

3.- Botones (/dev/buttons)

Requisitos:

1.- Buffer circular de pulsaciones ('1' o '2') → buffer[64], buffer_head/tail, protegido con spinlock.

2.- Lectura bloqueante hasta nueva pulsación →

```
      if (head==tail) {
          if (O_NONBLOCK) return -EAGAIN;
          if (wait_event_interruptible(buttons_waitqueue, head!=tail))
              return -ERESTARTSYS;
      }
```

3.- Interrupciones + debounce →

- request_irq(..., IRQF_TRIGGER_FALLING, buttonX_irq_handler)
- queue_delayed_work(buttons_wq, &debounce_work_bX, msecs_to_jiffies(debounce_ms));
- En el trabajo diferido: gpio_get_value() → insert '1'/'2' en el buffer + wake_up_interruptible().

4.- Apertura exclusiva → DEFINE_SEMAPHORE(buttons_sem,1), en open():

```
      if (O_NONBLOCK) down_trylock()?:-EBUSY;
      else            down_interruptible()?:-ERESTARTSYS;
```

en release(): up(&buttons_sem).

5.- Registro y limpieza → misc_register(&buttons_dev) en
r_devices_config();
cleanup_driver():
free_irqs(), free_gpios(),
misc_deregister(&buttons_dev).

EXPLICACIÓN DETALLADA DE LA IMPLEMENTACIÓN DE LA PRÁCTICA

Realizar un módulo de kernel, que funcione como driver para la placa auxiliar de E/S de la Raspberry

El *driver* debe crear tres *devices* diferentes, con las siguientes características:

1.- DEFINICIÓN COMO MÓDULO DE KERNEL

Para definirlo como módulo de kernel al final de fichero hay tres líneas que convierten el código en un módulo válido de Linux:

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR(DRIVER_AUTHOR);
MODULE_DESCRIPTION(DRIVER_DESC);
```

Hay dos funciones muy importantes cuya función es cargar y descargar el módulo:

`module_init(init_driver);`
Marca la función `init_driver()` como la que el kernel llamará al cargar el módulo:

`module_exit(cleanup_driver);`
Indica que, al descargar el módulo, el kernel debe llamar a `cleanup_driver()` para liberar recursos.

2.- CREACIÓN DE LOS TRES DEVICES (`static struct miscdevice`)

Para cada dispositivo se usa la infraestructura de **`miscdevice`**, con la que creamos un nodo en `/dev`. Con ella definimos cómo y con qué permisos se expondrán los tres dispositivos: número de minor dinámico, nombre del nodo, tabla de operacioens asociada y permisos read/write.

Para los leds, lo creamos en **/dev/leds**:

```
static struct miscdevice leds_dev = {
    .minor = MISC_DYNAMIC_MINOR,
    .name  = DEVICE_LEDS,        // "/dev/leds"
    .fops  = &leds_fops,         // Ata funciones leds_read/write
    .mode  = S_IRUGO | S_IWUGO,  // Permisos read/write para todos
};
```

Para los buttons, lo creamos en **/dev/buttons**:

```
static struct miscdevice buttons_dev = {
    .minor = MISC_DYNAMIC_MINOR,
    .name  = DEVICE_BUTTONS,     // "/dev/buttons"
    .fops  = &buttons_fops,      // Ata open/read/release
    .mode  = S_IWUGO,            // Permiso sólo lectura
};
```

Para el speaker, lo creamos en **/dev/speaker**:

```
static struct miscdevice speaker_dev = {
    .minor = MISC_DYNAMIC_MINOR,
    .name  = DEVICE_SPEAKER,     // "/dev/speaker"
    .fops  = &speaker_fops,      // Ata función speaker_write
    .mode  = S_IWUGO,            // Permiso sólo escritura
};
```

En FILE OPERATIONS se definen las operacioens asociadas a coda nodo y las funciones que lo implementan:

```
static const struct file_operations leds_fops =
{
    .owner  = THIS_MODULE,
    .read   = leds_read,
    .write  = leds_write,
};
```

```
static const struct file_operations buttons_fops =
{
    .owner      = THIS_MODULE,
    .open       = buttons_open,
    .release    = buttons_release,
    .read       = buttons_read,
};
static const struct file_operations speaker_fops =
{
    .owner      = THIS_MODULE,
    .write      = speaker_write,
};
```

3.- INICIALIZACIÓN Y REGISTRO DE LOS DEVICES (`r_devices_config`)

En la fase de inicialización (`init_driver`) se realizan tres operaciones fundamentales:

1. Se configuran las GPIOs con **`GPIO_config()`**,
2. Se configuran las IRQs con **`irqs_config()`** y
3. Se llama a **`r_devices_config()`**, que le pide al kernel que cree tres nodos en `/dev/leds`, `/dev/buttons` y `/dev/speaker`:

```
static int r_devices_config(void)
{
    int ret = 0;

    ret = misc_register(&leds_dev);
    if(ret < 0){
        printk(KERN_ERR "misc_register for leds device failed\n");
        return ret;
    }else{
        printk(KERN_NOTICE "misc_register OK... leds device minor=%d\n",
       leds_dev.minor);
    }

    ret = misc_register(&buttons_dev);
    if(ret < 0){
        printk(KERN_ERR "misc_register for buttons device failed\n");
        misc_deregister(&leds_dev);
        return ret;
    }else{
        printk(KERN_NOTICE "misc_register OK... buttons device minor=%d\n",
        buttons_dev.minor);
    }

    ret = misc_register(&speaker_dev);
    if(ret < 0){
        printk(KERN_ERR "misc_register for speaker device failed\n");
        misc_deregister(&buttons_dev);
        misc_deregister(&leds_dev);
        return ret;
    }else{
        printk(KERN_NOTICE "misc_register OK... speaker device minor=%d\n",
        speaker_dev.minor);
    }

    return ret;
}
```

Cada `misc_register(&xxx_dev)` crea el nodo `/dev/xxx` asociado a ese device y enlaza las operaciones de archivo (`fops`) que hayamos definido en `leds_fops, buttons_fops` y `speaker_fops`. Devolverá un número >=0 si el registro fue correcto y un número < 0 si dio error.
En caso de error se escribirá un mensaje en el **registro de eventos** (log) interno del kernel de Linux.
Se asegura que si falla el registro de cualquiera, se desregistra sólo lo creado hasta ese punto.
¿Cómo se hace esto?.

leds (/dev/leds)

El *device* llamado leds, debe permitir modificar y consultar qué leds están encendidos o apagados. La lectura retornará un *byte* con los 6 bits menos significativos representando el estado de los leds (los dos bits más significativos a cero). Para la escritura de los leds se utilizará un *byte*. Los 6 bits menos significativos serán los valores a escribir en los LEDs de la placa y deben interpretarse, dependiendo de los valores de los dos bits más significativos del byte escrito, de la siguiente forma:

- Si el bit 7 y 6 están a cero, los bits de 5 a 0 serán los valores que deben tomar los LEDS: 1 encender y 0 apagar.
- Si el bit 6 está a 1 y el 7 a 0, sólo se tomarán en cuenta los bits a 1 (del 5 a 0) para poner los LEDs correspondientes a 1, el resto quedarán con el valor anterior.
- Si en bit 7 está a 1 y el 6 a 0, sólo se tomarán en cuenta los bits a 1 (del 5 a 0) para poner los LEDs correspondientes a 0, el resto quedarán con el valor anterior.
- Si el bit 6 y 7 están ambos a 1, sólo se tomarán en cuenta los bits a 1 (del 5 a 0) para cambiar el valor de los LEDs correspondientes, el resto quedará con el valor anterior.

Para consultar el estado de los leds y modificarlos usamos las funciones `leds_read` y `leds_write`.

`leds_read()`

```
static ssize_t leds_read(struct file *file, char __user *buf, size_t len, loff_t *ppos)
{
    uint8_t val = leds_state;

    if(*ppos == 0) *ppos+=1;
    else return 0;

    // se copia byte al espacio usuario con comprobación de fallo
    return copy_to_user(buf, &val, 1) ? -EFAULT : 1;
}
```

Se encarga de devolver un byte con los **6 bits inferiores** del estado actual (leds_state & 0x3F), dejando **bits 6 y 7 a 0**. Se ponen a 0 los bits 7 y 8 con la máscara `0x3F`.
Gestiona además que solo se entregue **un único byte** por llamada ya que, si el mismo proceso vuelve a llamar a `read()` sin reabrir el fichero, `*ppos` (que maneja la posición dentro del fichero) ya vale 1, por lo que entra en el **else** y retorna 0, que es la señal de "fin de fichero" (EOF).

`leds_write()`

```
static ssize_t leds_write(struct file *file, const char __user *buf, size_t len, loff_t
*ppos)
{
    uint8_t val;

    if(*ppos == 0) *ppos+=1;
    else return len;

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

    uint8_t type = (val & 0xC0) >> 6;
    uint8_t bits = val & 0x3F;

    switch(type){
    case 0: leds_state = bits; break;
    case 1: leds_state |= bits; break;
    case 2: leds_state &= ~bits; break;
    case 3: leds_state ^= bits; break;
    }

    for(int i = 0; i < ARRAY_SIZE(leds_pin); ++i){
        gpio_set_value(GPIO_DEFAULT + leds_pin[i], (leds_state >> i) & 1);
    }

    return 1;
}
```

Se asegura de que solo se lea el primer byte del fichero `*file` y lo copia del espacio del usuario en `val` (con detección de errores).
Decodifica los bits y aplica la máscara indicada en los bits 7-6 (val & 0xC0) a los bits 5-0 (val & 0x3F). Por último se escribe el estado de cada bit en el led físico correspondiente.
Retorna 1 si todo el proceso ha sido correcto.

speaker (/dev/speaker)

Este *device* será solo de escritura y permitirá activar y desactivar la salida conectada al altavoz piezoeléctrico. El usuario será responsable de, escribiendo en este *device*, generar ondas que produzcan sonidos. Un byte = '0' escrito en el *device*, pondrá a 0 la salida conectada al altavoz. Un byte != '0', pondrá a 1 la salida del *device*.

```
static ssize_t speaker_write(struct file *file, const char __user *buf, size_t
len, loff_t *ppos)
{
    char val;

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

    gpio_set_value(GPIO_DEFAULT + GPIO_SPEAKER, (val != '0') ? 1 : 0);

    return 1;
}
```

Solo hay función de escritura. Copia solo un byte del espacio del usuario: si es cero, apaga el altavoz, cualquier otro valor lo enciende.
En la fase de inicialización, en `r_device_config()`, ejecutamos `misc_register(&speaker_dev)`, que creó el nodo `/dev/speaker` asociado a ese device:

```
static struct miscdevice speaker_dev = {
    .minor = MISC_DYNAMIC_MINOR,
    .name  = DEVICE_SPEAKER,  // "speaker"
    .fops  = &speaker_fops,
    .mode  = S_IWUGO,         // Solo escritura
};
```

y enlazó las operaciones de archivo (`fops`) definidas en la estructura:

```
static const struct file_operations speaker_fops =
{
    .owner      = THIS_MODULE,
    .write      = speaker_write,
};
```

Buttons (/dev/buttons)

1.- Este *device* será de lectura y permitirá leer las pulsaciones de los dos botones como si se tratase de un teclado. Para ello, el *driver* tendrá que manejar un ***buffer* interno donde se almacenen las pulsaciones** de teclas. Y cuando un usuario lea del device servirle dichas pulsaciones de forma que si se pulsó el botón 1, se lea el carácter que representa al número uno '1' y si se pulsó dos, se lea '2'.

1.- **Buffer interno de pulsaciones**

Manejaremos un buffer circular interno de 64 posiciones donde se almacenen las pulsaciones. La pulsación de cada botón se almacena como '1' o '2' y representa una pulsación pendiente de atender.

```
// === GLOBAL BUTTONS VARIABLES ===
#define BUTTONS_BUFF_SIZE 64
static char   buffer[BUTTONS_BUFF_SIZE];
static int    buffer_head = 0;
static int    buffer_tail = 0;
DEFINE_SPINLOCK(buffer_lock);
```

Explicación de las estructuras:
buffer[]: array circular que almacena los caracteres '1' o '2'.
buffer_head: apunta a la **próxima casilla libre** donde grabaremos un nuevo carácter.
buffer_tail: apunta a la **siguiente casilla** que está pendiente de ser leída por el usuario..
Cuando head == tail el buffer está vacío.
DEFINE_SPINLOCK(buffer_lock):
Dado que las pulsaciones llegan **asíncronamente** desde el hardware (en un contexto de interrupción o workqueue) y el usuario podría estar leyendo al mismo tiempo, debemos impedir que dos partes del código modifiquen el buffer simultáneamente y produzcan datos corruptos.
`DEFINE_SPINLOCK(buffer_lock)` es un spinlock para proteger la sección crítica en la que los *workers* diferidos insertan caracteres en buffer[]. Cada vez que un trabajo diferido pone un carácter en buffer[buffer_head], hace:

```
spin_lock(&buffer_lock);
buffer[buffer_head++] = '1' o '2';
buffer_head %= BUTTONS_BUFF_SIZE;
spin_unlock(&buffer_lock);
```

Así garantizamos que **solo un contexto** pueda actualizar los índices y el contenido del buffer a la vez.

2.- Si cuando el usuario realiza una lectura del *device* no hay pulsaciones en el *buffer*, la lectura debe ser bloqueada en espera de una nueva pulsación. Se despertará la lectura del *device* cuando se reciba una pulsación de botón y se pueda servir la lectura al usuario.

2.- **Lecturas bloqueantes hasta nueva pulsación**

Cuando un programa hace read() de /dev/buttons, esperamos que reciba **la siguiente pulsación** que aún esté en nuestro buffer. Pero, ¿ y si en ese momento **no hay nada** que leer?. Si el usuario lanza la lectura y el buffer está vacío, el driver puede comportarse de dos maneras:

1. **Modo no bloqueante** (O_NONBLOCK):
   Devolvemos inmediatamente **-EAGAIN**, señalando al proceso que:

   “no hay datos disponibles ahora mismo, inténtalo más tarde”

   así el llamador no se queda esperando.
2. **Modo bloqueante** (por defecto):
   Aquí queremos que el proceso **se duerma** hasta que llegue la próxima pulsación. Para ello usamos la cola de espera que declaramos con:

   `DECLARE_WAIT_QUEUE_HEAD(buttons_waitqueue);`

   En el interior de buttons_read() comprobamossi el buffer está vacío (buffer_head == buffer_tail), y Si **no** estamos en nonblock, invocamos

wait_event_interruptible(buttons_waitqueue, buffer_head != buffer_tail)

que suspende el proceso hasta que la condición (que haya al menos un carácter disponible) se cumpla. Si, mientras está dormido, el proceso recibe una señal (por ejemplo, Ctrl+C), la llamada devuelve **distinto de cero** y devolvemos -ERESTARTSYS. Eso hace que, tras gestionar la señal, el propio kernel reintente la operación de lectura.

Si salimos de ese bloque, ya **sabemos** que buffer_head != buffer_tail (es decir, hay algo para leer). Entonces copiamos ese byte al espacio de usuario (si la dirección de destino en el espacio de usuario no es válida, devolvemos error). Avanzamos el valor de buffer_tail para indicar que el carácter ha sido leído y retornamos 1 para indicar que el proceso se ha realizado correctamente.

```
static ssize_t buttons_read(struct file *file, char __user *buf, size_t len, loff_t
*ppos)
{
    // no hay datos en el buffer
    if(buffer_head == buffer_tail){
        // no bloquear en modo nonblock
        if(file->f_flags & O_NONBLOCK) return -EAGAIN;

        // bloquear hasta que haya pulsación = buffer_head cambie
        if(wait_event_interruptible(buttons_waitqueue, buffer_head != buffer_tail))
            return -ERESTARTSYS;
    }

    // ya hay pulsaciones → copiar un carácter al espacio usuario
    if(len < 1) return -EINVAL;
    if(copy_to_user(buf, &buffer[buffer_tail], 1)) return -EFAULT;

    // avanzar cola circular
    buffer_tail = (buffer_tail + 1) % sizeof(buffer);

    return 1;
}
```

3.- Deben usarse interrupciones para controlar la pulsación de teclas. Utilizar una técnica de *deferred job* (tasklet, timer, threaded_irq, workqueue) para una implementación eficiente de las interrupciones. Adoptar una solución a los rebotes, ofreciendo como parámetro del módulo *driver* la cantidad de milisegundos que se usará para filtrar los rebotes, estableciendo un valor por defecto adecuado.

3.- **Interrupciones + y trabajo diferido (deferred job) para detectar pulsaciones y filtrar rebotes**

Usar interrupciones para los botones; emplear una técnica deferred (tasklet, timer, workqueue) para antirrebote; parámetro configurable.

```
// === GPIO AND IRQS CONFIGURATION ===
static int irqs_config(void)
{
    // obtener números de IRQ
    irq_b1 = gpio_to_irq(GPIO_DEFAULT + GPIO_BUTTON1);
    irq_b2 = gpio_to_irq(GPIO_DEFAULT + GPIO_BUTTON2);

    // registrar handlers
    request_irq(irq_b1, button1_irq_handler,
                IRQF_TRIGGER_FALLING, "btn1_irq", NULL);
    request_irq(irq_b2, button2_irq_handler,
                IRQF_TRIGGER_FALLING, "btn2_irq", NULL);

    // crear trabajos diferidos
    INIT_DELAYED_WORK(&debounce_work_b1, debounce_work_func_b1);
    INIT_DELAYED_WORK(&debounce_work_b2, debounce_work_func_b2);
    buttons_wq = create_singlethread_workqueue("buttons_wq");
    return 0;
}
```

gpio_to_irq() convierte el pin GPIO al número de IRQ que el kernel utiliza para esa línea.
request_irq(..., IRQF_TRIGGER_FALLING) vincula la rutina de atención al flanco de bajada de la señal, *garantizando que capturamos la pulsación, no el rebote inicial*.
Con **INIT_DELAYED_WORK()** preparamos dos estructuras de trabajo diferido, asociadas a sus funciones de debounce, y con **create_singlethread_workqueue()** obtenemos una cola de ejecución dedicada.

```
// === IRQ HANDLERS ===
static irqreturn_t button1_irq_handler(int irq, void *dev_id)
{
    queue_delayed_work(buttons_wq, &debounce_work_b1,
                       msecs_to_jiffies(debounce_time_ms));
    return IRQ_HANDLED;
}
static irqreturn_t button2_irq_handler(int irq, void *dev_id)
{
    queue_delayed_work(buttons_wq, &debounce_work_b2,
                       msecs_to_jiffies(debounce_time_ms));
    return IRQ_HANDLED;
}
```

Estos handlers son muy simples: no acceden al buffer ni duermen, solo encolan el trabajo diferido. El retardo **msecs_to_jiffies(debounce_time_ms)** se establece inicialmente en 50 y sirve para filtrar rebotes mecánicos.

```
// === DEFERRED WORK (debounce) ===
static void debounce_work_func_b1(struct work_struct *work)
{
    if (!gpio_get_value(GPIO_DEFAULT + GPIO_BUTTON1)) {
        spin_lock(&buffer_lock);
        buffer[buffer_head++] = '1';
        buffer_head %= BUTTONS_BUFF_SIZE;
        spin_unlock(&buffer_lock);
        wake_up_interruptible(&buttons_waitqueue);
    }
}
```

Tras el retardo, comprobamos si el botón sigue pulsado con **gpio_get_value()**. Solo si el nivel sigue bajo (botón pulsado), consideramos que el evento es válido e inserta un "1" en el buffer bajo spinlock.
Con el **spinlock** protegemos la escritura en el buffer circular.
Finalmente, con **wake_up_interruptible()** despertamos a cualquier proceso bloqueado en **read()**, habilitando la entrega inmediata de ese carácter.

4.- Hay que asegurarse de que el dispositivo buttons se usa en exclusividad, esto es, el segundo "open()" al dispositivo (sin un "close()" intermedio) no puede retornar con éxito. Durante la apertura del dispositivo con "open()" se comprobará si el flag O_NONBLOCK está o no activo. Si el flag O_NONBLOCK está activo y el dispositivo está en uso se retornará el error de "dispositivo ocupado", si no está activo, el proceso quedará en espera de que se cierre para continuar correctamente.

```
if (filp->f_flags & O_NONBLOCK) {...} else {...}
```

4.- **Apertura exclusiva y modos** `O_NONBLOCK` **vs** `bloqueante`

Para que **solo un proceso** pueda usar /dev/buttons en un momento dado, usamos un **semáforo** binario:

```
DEFINE_SEMAPHORE(buttons_sem, 1);
```

Al abrir el dispositivo (open), comprobamos si el llamador pidió **modo no bloqueante** (O_NONBLOCK) o **bloqueante** (por defecto):

- **down_trylock()**: intenta decrementar sin dormir; si el semáforo vale 0 (otro proceso ya está dentro), devuelve fallo inmediato (-EBUSY).
- **down_interruptible()**: si el semáforo vale 0, **duerme** hasta que alguien haga up(), pero si llega señal interrumpe y retorna -ERESTARTSYS.

```
static int buttons_open(struct inode *inode, struct file *flip)
{
    if (flip->f_flags & O_NONBLOCK) {
        if (down_trylock(&buttons_sem))  // si semáforo ocupado
            return -EBUSY;               // nonblock → error inmediato
    } else {
        if (down_interruptible(&buttons_sem))
            return -ERESTARTSYS;         // bloqueante → espera y respeta señales
    }
    return 0;
}
```

Al cerrar el dispositivo (release), liberamos el semáforo para que otro proceso pueda abrirlo:

```
static int buttons_release(struct inode *inode, struct file *flip)
{
    up(&buttons_sem);   // libera el semáforo, permitiendo a otro open()
    return 0;
}
```

5.- Para que la llamada al device buttons sea bloqueante utilizar una cola de espera. Este es el interfaz necesario para gestionar el bloqueo de las lecturas:

```
DECLARE_WAIT_QUEUE_HEAD()       // declara una cola de espera en el kernel
wait_event_interruptible()      // el proceso actual se bloquea en una cola de espera según una condición,
normalmente                     // en la función read() del dispositivo, cuando hayq ue esperar que se complete la
                                // operación de E/S
wake_up_interruptible()         // desbloquea a los procesos que esperan en una cola de espera, normalmente desde
                                // un tasklet arrancado después de la interrupción que nos informa de que ya se puede
                                // continuar la operación de E/S
```

Las funciones que bloquean a un proceso de forma interruptible, pueden ser interrumpidas por una señal recibida por el proceso. Esto permite que un proceso bloqueado responda a señales (CTRL+C por ejemplo). Para controlar esta situación hay que comprobar si estas funciones que se bloquean devuelven cero, si no lo hacen hay que devolver el error -ERESTARTSYS . Algo como:

if (wait_event_interruptible(...)) return -ERESTARTSYS;

Esto provocará que el sistema reintente la ejecución de la función donde se produzca este error después del tratamiento de la señal del proceso interrumpido durante su bloqueo.

Referencia del interfaz de E/S bloqueante: https://bootlin.com/doc/books/ldd3.pdf chapter 6, section 2.

5. Cola de espera y limpieza simétrica

5.1.- Cola de espera en read()

Para completar el manejo correcto de /dev/buttons, usaremos también las **colas de espera** del kernel y nos aseguraremos de liberar todo al desmontar el módulo.

Creamos una cola de espera en read() declarando al inicio:

```
static DECLARE_WAIT_QUEUE_HEAD(buttons_waitqueue);
```

Esto crea la estructura necesaria para que procesos bloqueados puedan dormir y luego despertarse. En buttons_read():

```
if (wait_event_interruptible(buttons_waitqueue, buffer_head != buffer_tail))
    return -ERESTARTSYS;
```

Donde **wait_event_interruptible()** duerme al proceso hasta que la condición (buffer no vacío) sea cierta. Si llega **una señal** (por ejemplo Ctrl+C), la función devuelve distinto de cero y devolvemos -ERESTARTSYS, permitiendo que el propio kernel reintente la llamada tras el manejo de la señal.

Cuando un trabajo diferido inserta un carácter en el buffer:

```
wake_up_interruptible(&buttons_waitqueue);
```

wake_up_interruptible() despierta a todos los procesos que duermen en esa cola, de modo que read() continúa y puede devolver el dato nuevo.

5.1.- **Registro y limpieza de** `/dev/buttons`

En **init_driver(),** tras GPIO_config() y irqs_config() se llama a r_devices_config() para su registro:

```
r_devices_config();  // misc_register(&buttons_dev)
```

La limpieza de recursos se realiza en **cleanup_driver()**:

```
free_irqs();     // free_irq + destroy_workqueue
free_gpios();    // gpio_free para todos los pines
misc_deregister(&buttons_dev);
```

Así garantizamos que, al descargar el módulo, se liberen **IRQ**, **GPIO**, **workqueue** y se desregistre **el device**.