Linux Kernel GPIO LED Driver - Complete Code Explanation

Project Overview

This project is a **Linux Kernel Character Device Driver** for Raspberry Pi 4 that controls an LED connected to GPIO pin 23 (Physical Pin 16). It demonstrates fundamental kernel programming concepts including character devices, IOCTL interface, GPIO control, and kernel timers.

Project Structure

```
Linux_Kernel_Practical_1/
├── gpio_led_ioctl.h      - Shared header file (IOCTL definitions)
├── gpio_led_driver.c     - Kernel module/driver
├── gpio_led_app.c        - User-space control application
├── Makefile              - Build system
└── CODE_EXPLANATION.md   - This documentation
```

Hardware Connection

```
Raspberry Pi 4 GPIO Header
┌──────────────────────────┐
│  Physical Pin 16 (BCM 23)  │──── 330Ω Resistor ──── LED Anode
│  GND Pin                   │──── LED Cathode
└──────────────────────────┘
```

File 1: gpio_led_ioctl.h (Shared Header)

This header is shared between the kernel driver and user-space application to ensure consistent definitions.

Complete Source Code

```
/* gpio_led_ioctl.h
 *
 * Shared header between kernel driver and user-space application.
 * Defines IOCTL commands for GPIO LED control on Raspberry Pi 4.
 *
 * Hardware: LED on Physical Pin 16 (BCM GPIO 23)
 */

#ifndef _GPIO_LED_IOCTL_H
#define _GPIO_LED_IOCTL_H

#ifdef __KERNEL__
#include <linux/ioctl.h>
#else
#include <sys/ioctl.h>
#endif

/* Magic number for our driver */
#define GPIO_LED_MAGIC  'L'

/* LED states */
#define LED_OFF         0
#define LED_ON          1

/* IOCTL Commands */
#define GPIO_LED_SET_ON         _IO(GPIO_LED_MAGIC, 0)
#define GPIO_LED_SET_OFF        _IO(GPIO_LED_MAGIC, 1)
#define GPIO_LED_TOGGLE         _IO(GPIO_LED_MAGIC, 2)
#define GPIO_LED_GET_STATE      _IOR(GPIO_LED_MAGIC, 3, int)
#define GPIO_LED_SET_STATE      _IOW(GPIO_LED_MAGIC, 4, int)
#define GPIO_LED_SET_BLINK      _IOW(GPIO_LED_MAGIC, 5, int)
#define GPIO_LED_STOP_BLINK     _IO(GPIO_LED_MAGIC, 6)

#define GPIO_LED_MAX_CMD        6
#define DEVICE_NAME             "gpio_led"
#define DEVICE_PATH             "/dev/" DEVICE_NAME
#define LED_GPIO_BCM            23

#endif /* _GPIO_LED_IOCTL_H */
```

Line-by-Line Explanation

Lines	Code	Explanation
1-7	Comment block	File description and hardware info
9-10	`#ifndef _GPIO_LED_IOCTL_H`	**Include guard** - prevents multiple inclusions of the same header file
12-16	`#ifdef __KERNEL__`	**Conditional compilation** - `__KERNEL__` is defined when compiling kernel code. Uses appropriate header for each context
19	`#define GPIO_LED_MAGIC 'L'`	**Magic number** - unique identifier ('L' = 0x4C) for this driver's IOCTL commands
22-23	`LED_OFF`, `LED_ON`	**State constants** - 0 for OFF, 1 for ON
26-32	IOCTL Command definitions	Uses `_IO`, `_IOR`, `_IOW` macros (explained below)
34	`GPIO_LED_MAX_CMD`	Maximum command number for validation
35-36	Device name and path	Used to create/access `/dev/gpio_led`
37	`LED_GPIO_BCM`	BCM GPIO pin number 23

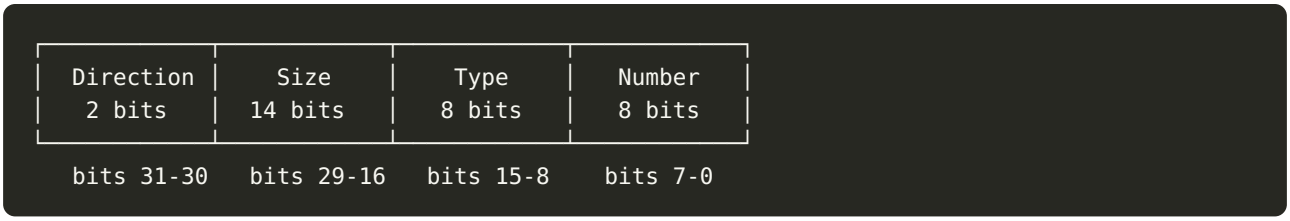
Understanding IOCTL Macros

The `_IO` Macro Family

IOCTL (Input/Output Control) macros create unique 32-bit command numbers.

Macro	Meaning	Data Direction	Example
`_IO(type, nr)`	No data transfer	None	`_IO('L', 0)`
`_IOR(type, nr, datatype)`	Read from kernel	Kernel → User	`_IOR('L', 3, int)`
`_IOW(type, nr, datatype)`	Write to kernel	User → Kernel	`_IOW('L', 4, int)`
`_IOWR(type, nr, datatype)`	Read and Write	Both directions	`_IOWR('L', 5, struct data)`

IOCTL Command Structure (32 bits)



Field	Bits	Description
Direction	31-30	00=none, 01=write, 10=read, 11=read/write
Size	29-16	Size of data being transferred (0 for `_IO`)
Type	15-8	Magic number (unique identifier for driver)
Number	7-0	Command sequence number (0, 1, 2, ...)

Helper Macros for Decoding

```
_IOC_TYPE(cmd)  // Extract the magic number (type)
_IOC_NR(cmd)    // Extract the command number
_IOC_SIZE(cmd)  // Extract the data size
_IOC_DIR(cmd)   // Extract the direction
```

Why Use IOCTL Macros?

1. **Uniqueness**: Magic number + command number prevents collisions between drivers
2. **Validation**: Kernel can verify commands belong to correct driver
3. **Self-documenting**: Macro name indicates data direction
4. **Standardization**: Consistent interface across all Linux drivers

---

File 2: gpio_led_driver.c (Kernel Module)

This is the core kernel module that creates a character device for LED control.

Section 1: Headers and Module Metadata

```c
#include <linux/module.h>      // Core module macros
#include <linux/kernel.h>      // Kernel logging (pr_info, pr_err)
#include <linux/init.h>        // __init, __exit macros
#include <linux/fs.h>          // File operations, register_chrdev
#include <linux/cdev.h>        // Character device structure
#include <linux/device.h>      // Device class, device_create
#include <linux/gpio.h>        // Legacy GPIO API
#include <linux/gpio/consumer.h> // Descriptor-based GPIO API
#include <linux/uaccess.h>     // copy_to_user, copy_from_user
#include <linux/timer.h>       // Kernel timers
#include <linux/mutex.h>       // Mutex locks
#include <linux/err.h>         // IS_ERR, PTR_ERR macros

#include "gpio_led_ioctl.h"

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Anjaneya");
MODULE_DESCRIPTION("GPIO LED Driver for Raspberry Pi 4");
MODULE_VERSION("1.0");
```

Header	Purpose
`linux/module.h`	MODULE_LICENSE, MODULE_AUTHOR, module_init, module_exit
`linux/kernel.h`	pr_info(), pr_err() logging functions
`linux/init.h`	`__init` and `__exit` section markers
`linux/fs.h`	struct file_operations, register_chrdev_region
`linux/cdev.h`	struct cdev, cdev_init, cdev_add
`linux/device.h`	class_create, device_create for udev
`linux/gpio.h`	gpio_request, gpio_direction_output (legacy API)
`linux/gpio/consumer.h`	gpiod_set_value (descriptor API)
`linux/uaccess.h`	copy_to_user, copy_from_user
`linux/timer.h`	timer_setup, mod_timer, del_timer_sync
`linux/mutex.h`	mutex_init, mutex_lock, mutex_unlock

Section 2: Driver Private Data Structure

```
struct gpio_led_dev {
    dev_t           devno;          /* Major:Minor device number          */
    struct cdev     cdev;           /* Character device structure          */
    struct class    *class;         /* Device class (for /sys/class entry) */
    struct device   *device;        /* Device (for /dev entry via udev)   */
    struct gpio_desc *gpio;         /* GPIO descriptor (new API)          */

    int             led_state;      /* Current LED state: 0=OFF, 1=ON    */
    struct mutex    lock;           /* Protects led_state and GPIO access */

    /* Blink support */
    struct timer_list blink_timer;  /* Kernel timer for blinking          */
    int             blinking;       /* 1 = blink active, 0 = stopped     */
    unsigned long   blink_period_jiffies; /* Half-period in jiffies       */
};

static struct gpio_led_dev *led_dev;
```

Field	Type	Purpose
`devno`	`dev_t`	Combined major:minor device number
`cdev`	`struct cdev`	Character device kernel structure
`class`	`struct class *`	Creates `/sys/class/gpio_led`
`device`	`struct device *`	Creates `/dev/gpio_led` via udev
`gpio`	`struct gpio_desc *`	GPIO descriptor for modern API
`led_state`	`int`	Cached LED state (0 or 1)
`lock`	`struct mutex`	Thread synchronization
`blink_timer`	`struct timer_list`	Kernel timer for blinking
`blinking`	`int`	Blink active flag
`blink_period_jiffies`	`unsigned long`	Timer period in jiffies

Section 3: GPIO Helper Functions

```c
static void gpio_led_set(struct gpio_led_dev *dev, int state)
{
    dev->led_state = !!state;  /* Normalize to 0 or 1 */
    gpiod_set_value(dev->gpio, dev->led_state);
}

static int gpio_led_get(struct gpio_led_dev *dev)
{
    return dev->led_state;
}

static void gpio_led_toggle(struct gpio_led_dev *dev)
{
    gpio_led_set(dev, !dev->led_state);
}
```

Function	Purpose	Notes
`gpio_led_set()`	Sets LED state	`!!state` converts any non-zero to 1
`gpio_led_get()`	Returns current state	Reads cached value
`gpio_led_toggle()`	Inverts LED state	Calls `gpio_led_set()` with inverted value

Section 4: Blink Timer

```c
static void blink_timer_callback(struct timer_list *t)
{
    struct gpio_led_dev *dev = from_timer(dev, t, blink_timer);

    /* Timer callbacks run in softirq context — cannot use mutex */
    dev->led_state = !dev->led_state;
    gpiod_set_value(dev->gpio, dev->led_state);

    /* Re-arm the timer if still blinking */
    if (dev->blinking)
        mod_timer(&dev->blink_timer, jiffies + dev->blink_period_jiffies);
}

static void start_blink(struct gpio_led_dev *dev, int period_ms)
{
    dev->blink_period_jiffies = msecs_to_jiffies(period_ms / 2);
    if (dev->blink_period_jiffies < 1)
        dev->blink_period_jiffies = 1;

    dev->blinking = 1;
    mod_timer(&dev->blink_timer, jiffies + dev->blink_period_jiffies);
}

static void stop_blink(struct gpio_led_dev *dev)
{
    dev->blinking = 0;
    del_timer_sync(&dev->blink_timer);
}
```

Key Concepts:

Concept	Explanation
`from_timer()`	Macro to get container structure from timer pointer
`jiffies`	Kernel's internal time counter (ticks since boot)
`msecs_to_jiffies()`	Converts milliseconds to jiffies
`mod_timer()`	Modifies/arms timer with new expiration time
`del_timer_sync()`	Deletes timer and waits for callback to finish
`softirq context`	Timer callbacks run in interrupt context (cannot sleep)

Section 5: File Operations

open() and release()

```
static int gpio_led_open(struct inode *inode, struct file *filp)
{
    filp->private_data = led_dev;
    pr_info("gpio_led: Device opened\n");
    return 0;
}

static int gpio_led_release(struct inode *inode, struct file *filp)
{
    pr_info("gpio_led: Device closed\n");
    return 0;
}
```

read() - Returns LED state as string

```
static ssize_t gpio_led_read(struct file *filp, char __user *buf,
                             size_t count, loff_t *ppos)
{
    struct gpio_led_dev *dev = filp->private_data;
    char state_str[4];
    int len;

    if (*ppos > 0)
        return 0;  /* EOF on subsequent reads */

    mutex_lock(&dev->lock);
    len = snprintf(state_str, sizeof(state_str), "%d\n", dev->led_state);
    mutex_unlock(&dev->lock);

    if (count < len)
        return -EINVAL;

    if (copy_to_user(buf, state_str, len))
        return -EFAULT;

    *ppos += len;
    return len;
}
```

Explanation:

- `*ppos > 0`: Returns 0 (EOF) on subsequent reads
- `copy_to_user()`: Safely copies data from kernel to user space
- Returns `-EFAULT` if copy fails (bad user pointer)

write() - Sets LED state from character

```
static ssize_t gpio_led_write(struct file *filp, const char __user *buf,
                              size_t count, loff_t *ppos)
{
    struct gpio_led_dev *dev = filp->private_data;
    char val;

    if (count < 1)
        return -EINVAL;

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

    mutex_lock(&dev->lock);
    if (val == '1')
        gpio_led_set(dev, LED_ON);
    else if (val == '0')
        gpio_led_set(dev, LED_OFF);
    else {
        mutex_unlock(&dev->lock);
        return -EINVAL;
    }
    mutex_unlock(&dev->lock);

    return count;
}
```

Explanation:

- `copy_from_user()`: Safely copies data from user to kernel space
- Accepts '0' or '1' character only
- Returns `-EINVAL` for invalid input

ioctl() - Main control interface

```c
static long gpio_led_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
    struct gpio_led_dev *dev = filp->private_data;
    int state;
    int period_ms;
    int ret = 0;

    /* Validate magic number and command range */
    if (_IOC_TYPE(cmd) != GPIO_LED_MAGIC)
        return -ENOTTY;
    if (_IOC_NR(cmd) > GPIO_LED_MAX_CMD)
        return -ENOTTY;

    mutex_lock(&dev->lock);

    switch (cmd) {
    case GPIO_LED_SET_ON:
        stop_blink(dev);
        gpio_led_set(dev, LED_ON);
        break;

    case GPIO_LED_SET_OFF:
        stop_blink(dev);
        gpio_led_set(dev, LED_OFF);
        break;

    case GPIO_LED_TOGGLE:
        stop_blink(dev);
        gpio_led_toggle(dev);
        break;

    case GPIO_LED_GET_STATE:
        state = gpio_led_get(dev);
        if (copy_to_user((int __user *)arg, &state, sizeof(int)))
            ret = -EFAULT;
        break;

    case GPIO_LED_SET_STATE:
        if (copy_from_user(&state, (int __user *)arg, sizeof(int))) {
            ret = -EFAULT;
            break;
        }
        if (state != LED_ON && state != LED_OFF) {
            ret = -EINVAL;
            break;
        }
        stop_blink(dev);
        gpio_led_set(dev, state);
        break;

    case GPIO_LED_SET_BLINK:
        if (copy_from_user(&period_ms, (int __user *)arg, sizeof(int))) {
            ret = -EFAULT;
            break;
        }
        if (period_ms < 50 || period_ms > 10000) {
            ret = -EINVAL;
```

```
            break;
        }
        start_blink(dev, period_ms);
        break;

    case GPIO_LED_STOP_BLINK:
        stop_blink(dev);
        break;

    default:
        ret = -ENOTTY;
        break;
    }

    mutex_unlock(&dev->lock);
    return ret;
}
```

IOCTL Command Summary:

Command	Action	Data Transfer
`GPIO_LED_SET_ON`	Turn LED on, stop blink	None
`GPIO_LED_SET_OFF`	Turn LED off, stop blink	None
`GPIO_LED_TOGGLE`	Toggle LED, stop blink	None
`GPIO_LED_GET_STATE`	Return current state	`copy_to_user()`
`GPIO_LED_SET_STATE`	Set specific state	`copy_from_user()`
`GPIO_LED_SET_BLINK`	Start blinking (50-10000ms)	`copy_from_user()`
`GPIO_LED_STOP_BLINK`	Stop blinking	None

File Operations Structure

```
static const struct file_operations gpio_led_fops = {
    .owner          = THIS_MODULE,
    .open           = gpio_led_open,
    .release        = gpio_led_release,
    .read           = gpio_led_read,
    .write          = gpio_led_write,
    .unlocked_ioctl = gpio_led_ioctl,
};
```

Section 6: Module Initialization

```c
static int __init gpio_led_init(void)
{
    int ret;

    /* Step 1: Allocate driver private data */
    led_dev = kzalloc(sizeof(*led_dev), GFP_KERNEL);
    if (!led_dev)
        return -ENOMEM;

    mutex_init(&led_dev->lock);

    /* Step 2: Request and configure GPIO */
    if (!gpio_is_valid(LED_GPIO_BCM)) {
        ret = -ENODEV;
        goto err_free_dev;
    }

    ret = gpio_request(LED_GPIO_BCM, "led_gpio_23");
    if (ret)
        goto err_free_dev;

    ret = gpio_direction_output(LED_GPIO_BCM, 0);
    if (ret)
        goto err_free_gpio;

    /* Step 3: Get GPIO descriptor */
    led_dev->gpio = gpio_to_desc(LED_GPIO_BCM);
    if (!led_dev->gpio) {
        ret = -ENODEV;
        goto err_free_gpio;
    }

    led_dev->led_state = LED_OFF;

    /* Step 4: Allocate major:minor number */
    ret = alloc_chrdev_region(&led_dev->devno, 0, 1, DEVICE_NAME);
    if (ret < 0)
        goto err_free_gpio;

    /* Step 5: Initialize and add character device */
    cdev_init(&led_dev->cdev, &gpio_led_fops);
    led_dev->cdev.owner = THIS_MODULE;

    ret = cdev_add(&led_dev->cdev, led_dev->devno, 1);
    if (ret < 0)
        goto err_unreg_chrdev;

    /* Step 6: Create device class */
    led_dev->class = class_create(DEVICE_NAME);
    if (IS_ERR(led_dev->class)) {
        ret = PTR_ERR(led_dev->class);
        goto err_cdev_del;
    }

    /* Step 7: Create device node */
    led_dev->device = device_create(led_dev->class, NULL,
                                    led_dev->devno, NULL, DEVICE_NAME);
```

```
    if (IS_ERR(led_dev->device)) {
        ret = PTR_ERR(led_dev->device);
        goto err_class_destroy;
    }

    /* Step 8: Initialize blink timer */
    timer_setup(&led_dev->blink_timer, blink_timer_callback, 0);
    led_dev->blinking = 0;

    pr_info("gpio_led: Driver loaded. Device: /dev/%s\n", DEVICE_NAME);
    return 0;

/* Error cleanup (reverse order) */
err_class_destroy:
    class_destroy(led_dev->class);
err_cdev_del:
    cdev_del(&led_dev->cdev);
err_unreg_chrdev:
    unregister_chrdev_region(led_dev->devno, 1);
err_free_gpio:
    gpio_free(LED_GPIO_BCM);
err_free_dev:
    kfree(led_dev);
    return ret;
}
```

Initialization Steps:

```
┌──────────────────────────────────────────────────┐
│ Step 1: kzalloc() - Allocate driver structure    │
├──────────────────────────────────────────────────┤
│ Step 2: gpio_request() - Request GPIO pin        │
├──────────────────────────────────────────────────┤
│ Step 3: gpio_direction_output() - Set as output  │
├──────────────────────────────────────────────────┤
│ Step 4: gpio_to_desc() - Get GPIO descriptor     │
├──────────────────────────────────────────────────┤
│ Step 5: alloc_chrdev_region() - Get major:minor number │
├──────────────────────────────────────────────────┤
│ Step 6: cdev_init() + cdev_add() - Register char device │
├──────────────────────────────────────────────────┤
│ Step 7: class_create() - Create /sys/class/gpio_led │
├──────────────────────────────────────────────────┤
│ Step 8: device_create() - Create /dev/gpio_led via udev │
├──────────────────────────────────────────────────┤
│ Step 9: timer_setup() - Initialize blink timer   │
└──────────────────────────────────────────────────┘
```

Error Handling Pattern (goto chain):

On any error, the code jumps to a cleanup label that undoes all previous steps in reverse order. This is standard Linux kernel practice.

Section 7: Module Exit

```c
static void __exit gpio_led_exit(void)
{
    /* Stop blink timer */
    stop_blink(led_dev);

    /* Turn LED off before unloading */
    gpiod_set_value(led_dev->gpio, 0);

    /* Tear down in reverse order of init */
    device_destroy(led_dev->class, led_dev->devno);
    class_destroy(led_dev->class);
    cdev_del(&led_dev->cdev);
    unregister_chrdev_region(led_dev->devno, 1);
    gpio_free(LED_GPIO_BCM);
    kfree(led_dev);

    pr_info("gpio_led: Driver unloaded\n");
}

module_init(gpio_led_init);
module_exit(gpio_led_exit);
```

Cleanup is always in reverse order of initialization.

File 3: gpio_led_app.c (User-Space Application)

Section 1: Device Opening

```c
static int open_device(void)
{
    int fd = open(DEVICE_PATH, O_RDWR);
    if (fd < 0) {
        perror("Failed to open " DEVICE_PATH);
        fprintf(stderr, "Make sure the driver is loaded\n");
    }
    return fd;
}
```

Opens `/dev/gpio_led` with read/write access. Returns file descriptor or -1 on error.

Section 2: IOCTL Wrapper Functions

```c
static int led_set_on(int fd)
{
    if (ioctl(fd, GPIO_LED_SET_ON) < 0) {
        perror("ioctl GPIO_LED_SET_ON");
        return -1;
    }
    printf("LED turned ON\n");
    return 0;
}

static int led_get_state(int fd)
{
    int state;
    if (ioctl(fd, GPIO_LED_GET_STATE, &state) < 0) {
        perror("ioctl GPIO_LED_GET_STATE");
        return -1;
    }
    printf("LED state: %s (%d)\n", state ? "ON" : "OFF", state);
    return state;
}

static int led_set_blink(int fd, int period_ms)
{
    if (ioctl(fd, GPIO_LED_SET_BLINK, &period_ms) < 0) {
        perror("ioctl GPIO_LED_SET_BLINK");
        return -1;
    }
    printf("LED blinking with period %d ms\n", period_ms);
    return 0;
}
```

IOCTL Wrapper Summary:

Function	IOCTL Command	Data Passed
`led_set_on()`	`GPIO_LED_SET_ON`	None
`led_set_off()`	`GPIO_LED_SET_OFF`	None
`led_toggle()`	`GPIO_LED_TOGGLE`	None
`led_get_state()`	`GPIO_LED_GET_STATE`	`&state` (receives value)
`led_set_state()`	`GPIO_LED_SET_STATE`	`&state` (sends value)
`led_set_blink()`	`GPIO_LED_SET_BLINK`	`&period_ms`
`led_stop_blink()`	`GPIO_LED_STOP_BLINK`	None

Section 3: Read/Write Interface (Alternative to IOCTL)

```c
static int led_read_state(int fd)
{
    char buf[4];
    lseek(fd, 0, SEEK_SET);  /* Reset file position */
    read(fd, buf, sizeof(buf) - 1);
    printf("LED state (via read): %s", buf);
    return 0;
}

static int led_write_state(int fd, const char *val)
{
    write(fd, val, 1);
    printf("LED set to %s via write()\n", (*val == '1') ? "ON" : "OFF");
    return 0;
}
```

Section 4: Command-Line Mode

```c
static int handle_cli(int fd, int argc, char *argv[])
{
    if (strcmp(argv[1], "on") == 0)
        return led_set_on(fd);
    else if (strcmp(argv[1], "off") == 0)
        return led_set_off(fd);
    else if (strcmp(argv[1], "toggle") == 0)
        return led_toggle(fd);
    else if (strcmp(argv[1], "state") == 0)
        return led_get_state(fd);
    else if (strcmp(argv[1], "blink") == 0) {
        if (argc < 3) {
            fprintf(stderr, "Usage: %s blink <period_ms>\n", argv[0]);
            return -1;
        }
        return led_set_blink(fd, atoi(argv[2]));
    } else if (strcmp(argv[1], "stop") == 0)
        return led_stop_blink(fd);
    // ...
}
```

Command-Line Usage:

```
sudo ./gpio_led_app on          # Turn LED on
sudo ./gpio_led_app off         # Turn LED off
sudo ./gpio_led_app toggle      # Toggle LED state
sudo ./gpio_led_app state       # Get current state
sudo ./gpio_led_app blink 500   # Blink with 500ms period
sudo ./gpio_led_app stop        # Stop blinking
```

Section 5: Interactive Menu Mode

```
static void print_menu(void)
{
    printf("╔═══════════════════════════════════════════╗\n");
    printf("║     GPIO LED Controller (BCM Pin 23)      ║\n");
    printf("╠═══════════════════════════════════════════╣\n");
    printf("║  1. Turn LED ON          (ioctl)          ║\n");
    printf("║  2. Turn LED OFF         (ioctl)          ║\n");
    printf("║  3. Toggle LED           (ioctl)          ║\n");
    printf("║  4. Get LED state        (ioctl)          ║\n");
    printf("║  5. Set LED state        (ioctl)          ║\n");
    printf("║  6. Start blink          (ioctl)          ║\n");
    printf("║  7. Stop blink           (ioctl)          ║\n");
    printf("║  8. Read state           (read syscall)   ║\n");
    printf("║  9. Write state          (write syscall)  ║\n");
    printf("║  0. Exit                                  ║\n");
    printf("╚═══════════════════════════════════════════╝\n");
}
```

Section 6: Main Function

```
int main(int argc, char *argv[])
{
    int fd = open_device();
    if (fd < 0)
        return EXIT_FAILURE;

    if (argc > 1)
        ret = handle_cli(fd, argc, argv);   /* Command-line mode */
    else
        interactive_mode(fd);               /* Interactive menu */

    close(fd);
    return (ret < 0) ? EXIT_FAILURE : EXIT_SUCCESS;
}
```

File 4: Makefile (Build System)

Complete Makefile

```
# Makefile for GPIO LED Driver

obj-m := gpio_led_driver.o gpio_led_platform_driver.o

KERNEL_DIR ?= /lib/modules/$(shell uname -r)/build
DTC ?= dtc
PWD := $(shell pwd)

# Build kernel modules
all:
    $(MAKE) -C $(KERNEL_DIR) M=$(PWD) modules

# Build ONLY the legacy driver
legacy:
    $(MAKE) -C $(KERNEL_DIR) M=$(PWD) modules obj-m=gpio_led_driver.o

# Build ONLY the platform driver
platform:
    $(MAKE) -C $(KERNEL_DIR) M=$(PWD) modules obj-m=gpio_led_platform_driver.o

# Compile Device Tree overlay
dtbo: gpio_led_overlay.dts
    $(DTC) -@ -I dts -O dtb -o gpio_led_overlay.dtbo gpio_led_overlay.dts

# Build user-space application
app: gpio_led_app.c gpio_led_ioctl.h
    $(CC) -Wall -Wextra -O2 -o gpio_led_app gpio_led_app.c

# Build everything
all-targets: all dtbo app

# Install platform driver
install-platform: platform dtbo
    sudo cp gpio_led_overlay.dtbo /boot/overlays/
    $(MAKE) -C $(KERNEL_DIR) M=$(PWD) modules_install obj-m=gpio_led_platform_driver.o
    depmod -a

# Clean
clean:
    $(MAKE) -C $(KERNEL_DIR) M=$(PWD) clean
    rm -f gpio_led_app gpio_led_overlay.dtbo

.PHONY: all legacy platform dtbo app all-targets install-platform clean
```

Makefile Variables

Variable	Default Value	Purpose
`obj-m`	Module list	Kernel modules to build
`KERNEL_DIR`	`/lib/modules/$(uname -r)/build`	Path to kernel headers
`DTC`	`dtc`	Device tree compiler
`PWD`	Current directory	Working directory

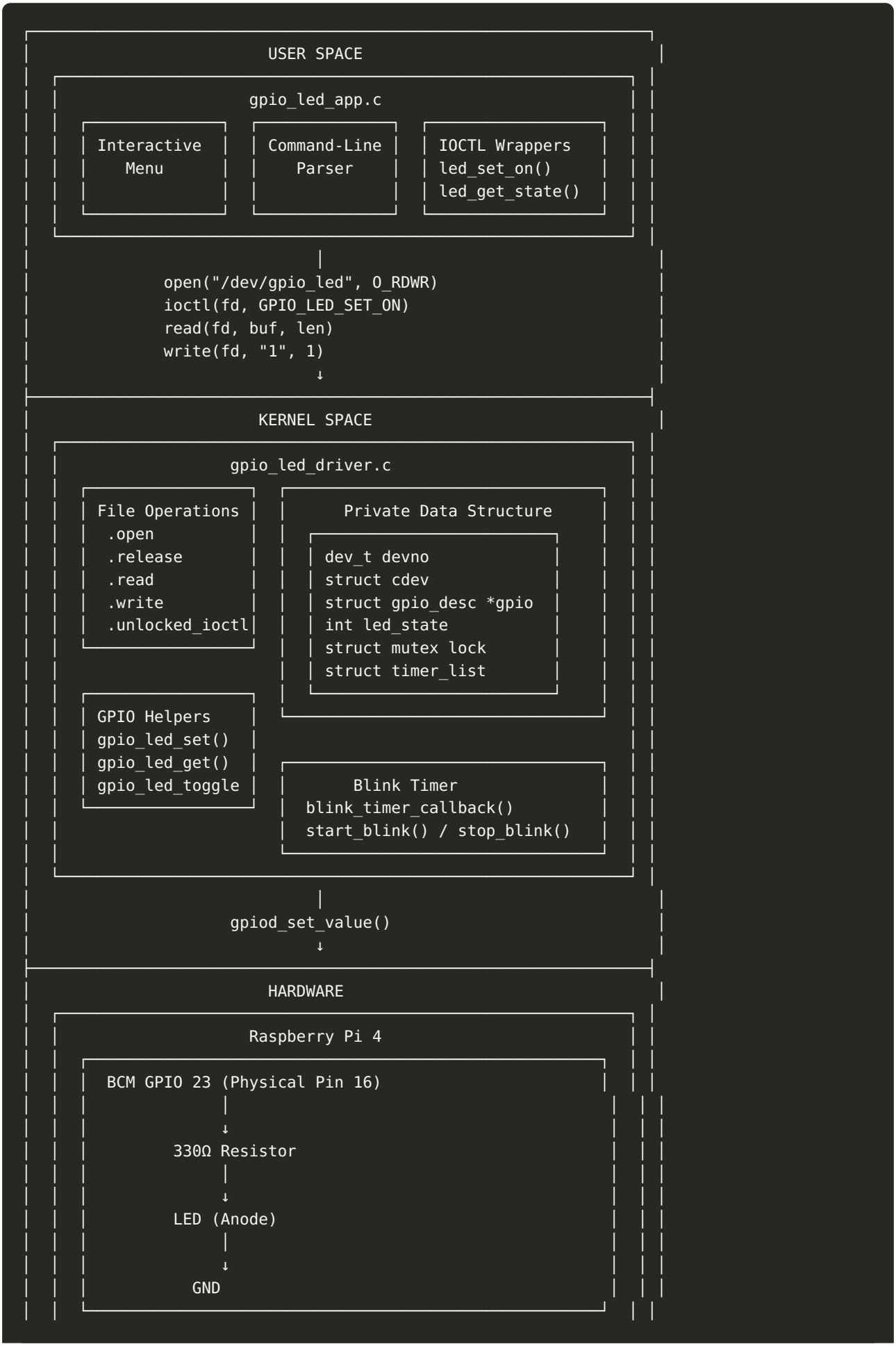
Makefile Targets

Target	Command	Purpose
`all`	`make`	Build all kernel modules
`legacy`	`make legacy`	Build only gpio_led_driver.ko
`platform`	`make platform`	Build platform driver
`dtbo`	`make dtbo`	Compile device tree overlay
`app`	`make app`	Build user-space application
`all-targets`	`make all-targets`	Build everything
`install-platform`	`make install-platform`	Install platform driver
`clean`	`make clean`	Remove build artifacts

Cross-Compilation

```
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- KERNEL_DIR=/path/to/rpi-linux
```

Architecture Diagram


USER SPACE
gpio_led_app.c
Interactive Menu
Command-Line Parser
IOCTL Wrappers
led_set_on()
led_get_state()
open("/dev/gpio_led", O_RDWR)
ioctl(fd, GPIO_LED_SET_ON)
read(fd, buf, len)
write(fd, "1", 1)
KERNEL SPACE
gpio_led_driver.c
File Operations
.open
.release
.read
.write
.unlocked_ioctl
Private Data Structure
dev_t devno
struct cdev
struct gpio_desc *gpio
int led_state
struct mutex lock
struct timer_list
GPIO Helpers
gpio_led_set()
gpio_led_get()
gpio_led_toggle
Blink Timer
blink_timer_callback()
start_blink() / stop_blink()
gpiod_set_value()
HARDWARE
Raspberry Pi 4
BCM GPIO 23 (Physical Pin 16)
330Ω Resistor
LED (Anode)
GND

Key Kernel Programming Concepts

1. Character Device Driver Framework

```
alloc_chrdev_region() → Get major:minor numbers
          ↓
     cdev_init()       → Initialize cdev structure
          ↓
      cdev_add()       → Register with kernel
          ↓
    class_create()     → Create /sys/class entry
          ↓
  device_create()      → Create /dev entry via udev
```

2. User-Kernel Data Transfer

Function	Direction	Usage
`copy_to_user(to, from, n)`	Kernel → User	Returning data to user space
`copy_from_user(to, from, n)`	User → Kernel	Getting data from user space

Always check return value - returns number of bytes NOT copied (0 = success).

3. Synchronization

- **Mutex**: Used for protecting shared state (`led_state`, GPIO access)
- **Timer context**: Cannot use mutex in timer callbacks (softirq context)

4. Error Handling Pattern

```
ret = function1();
if (ret < 0)
    goto err_cleanup1;

ret = function2();
if (ret < 0)
    goto err_cleanup2;

return 0;

err_cleanup2:
    undo_function1();
err_cleanup1:
    return ret;
```

Usage Instructions

Building the Driver

```
# Build kernel module
make legacy

# Build user-space application
make app
```

Loading the Driver

```
# Load the module
sudo insmod gpio_led_driver.ko

# Verify it's loaded
lsmod | grep gpio_led

# Check kernel messages
dmesg | tail

# Verify device node exists
ls -la /dev/gpio_led
```

Using the Application

```
# Interactive mode
sudo ./gpio_led_app

# Command-line mode
sudo ./gpio_led_app on
sudo ./gpio_led_app off
sudo ./gpio_led_app toggle
sudo ./gpio_led_app state
sudo ./gpio_led_app blink 500
sudo ./gpio_led_app stop
```

Direct Device Access (Shell)

```
# Read LED state
cat /dev/gpio_led

# Set LED on
echo 1 > /dev/gpio_led

# Set LED off
echo 0 > /dev/gpio_led
```

Unloading the Driver

```
sudo rmmod gpio_led_driver
```

Error Codes Reference

Error	Value	Meaning
`-ENOMEM`	-12	Out of memory
`-ENODEV`	-19	No such device
`-EINVAL`	-22	Invalid argument
`-EFAULT`	-14	Bad address (copy_to/from_user failed)
`-ENOTTY`	-25	Invalid ioctl command

Summary

This project demonstrates:

1. **Linux kernel module development** - Module init/exit, licensing, metadata
2. **Character device drivers** - cdev framework, file operations
3. **GPIO subsystem** - Legacy and descriptor-based APIs
4. **IOCTL interface** - Command definition and handling
5. **Kernel timers** - Periodic callbacks for blinking
6. **Synchronization** - Mutex for thread safety
7. **User-kernel communication** - copy_to_user, copy_from_user
8. **Error handling** - goto-based cleanup chains
9. **Build system** - Kernel module Makefile

Document generated for Linux Kernel Practical 1 - GPIO LED Driver Project