**Code :**

#include <linux/module.h>

#include <linux/fs.h>

#include <linux/cdev.h>

#include <linux/uaccess.h>

#include <linux/device.h>

/\* Define device name and buffer size \*/

#define DEVICE\_NAME "dev1"

#define BUFFER\_SIZE 1024

/\* Define global variables for device number, character device, device class, and buffer \*/

static dev\_t dev\_num;

static struct cdev my\_cdev;

static struct class \*my\_class;

static char kernel\_buffer[BUFFER\_SIZE];

static int open\_count = 0;

/\* Function to handle opening of the device \*/

static int simple\_open(struct inode \*inode, struct file \*file) {

open\_count++;

printk(KERN\_INFO "Device opened %d times\n", open\_count);

return 0;

}

/\* Function to handle closing of the device \*/

static int simple\_release(struct inode \*inode, struct file \*file) {

printk(KERN\_INFO "Device closed\n");

return 0;

}

/\* Function to handle reading from the device \*/

static ssize\_t simple\_read(struct file \*file, char \_\_user \*user\_buffer, size\_t size, loff\_t \*offset) {

size\_t to\_copy, not\_copied;

/\* Calculate the amount of data to copy based on the buffer size and requested size \*/

to\_copy = min(size, BUFFER\_SIZE - (size\_t)(\*offset));

/\* Copy data from kernel space to user space and update the offset \*/

not\_copied = copy\_to\_user(user\_buffer, kernel\_buffer + \*offset, to\_copy);

\*offset += to\_copy - not\_copied;

printk(KERN\_INFO "simple\_char\_dev: Read %zu bytes\n", to\_copy - not\_copied);

printk(KERN\_INFO "simple\_char\_dev: Read %zu bytes: %.\*s\n", to\_copy - not\_copied, (int)(to\_copy - not\_copied), kernel\_buffer + \*offset - (to\_copy - not\_copied));

/\* Return the number of bytes actually read \*/

return to\_copy - not\_copied;

}

/\* Function to handle writing to the device \*/

static ssize\_t simple\_write(struct file \*file, const char \_\_user \*user\_buffer, size\_t size, loff\_t \*offset) {

size\_t to\_copy, not\_copied;

/\* Calculate the amount of data to copy based on the buffer size and requested size \*/

to\_copy = min(size, BUFFER\_SIZE - (size\_t)(\*offset));

/\* Copy data from user space to kernel space and update the offset \*/

not\_copied = copy\_from\_user(kernel\_buffer + \*offset, user\_buffer, to\_copy);

\*offset += to\_copy - not\_copied;

printk(KERN\_INFO "simple\_char\_dev: Written %zu bytes\n", to\_copy - not\_copied);

printk(KERN\_INFO "simple\_char\_dev: Written %zu bytes: %.\*s\n", to\_copy - not\_copied, (int)(to\_copy - not\_copied), kernel\_buffer + \*offset - (to\_copy - not\_copied));

/\* Return the number of bytes actually written \*/

return to\_copy - not\_copied;

}

/\* File operations structure linking function pointers for open, read, write, and release operations \*/

static struct file\_operations fops = {

.owner = THIS\_MODULE,

.open = simple\_open,

.release = simple\_release,

.read = simple\_read,

.write = simple\_write,

};

/\* Module initialization function, called when the module is loaded \*/

static int \_\_init simple\_char\_init(void) {

int ret;

/\* Allocate a major and minor number dynamically for the device \*/

ret = alloc\_chrdev\_region(&dev\_num, 0, 1, DEVICE\_NAME);

if (ret < 0) {

printk(KERN\_ALERT "simple\_char\_dev: Failed to allocate a major number\n");

return ret;

}

/\* Initialize the character device and add it to the system \*/

cdev\_init(&my\_cdev, &fops);

ret = cdev\_add(&my\_cdev, dev\_num, 1);

if (ret < 0) {

unregister\_chrdev\_region(dev\_num, 1);

printk(KERN\_ALERT "simple\_char\_dev: Failed to add cdev\n");

return ret;

}

/\* Create a device class for easier access to the device in /dev \*/

my\_class = class\_create(THIS\_MODULE, "simple\_char\_class");

if (IS\_ERR(my\_class)) {

cdev\_del(&my\_cdev);

unregister\_chrdev\_region(dev\_num, 1);

printk(KERN\_ALERT "simple\_char\_dev: Failed to create device class\n");

return PTR\_ERR(my\_class);

}

/\* Create a device node in /dev directory \*/

if (IS\_ERR(device\_create(my\_class, NULL, dev\_num, NULL, DEVICE\_NAME))) {

class\_destroy(my\_class);

cdev\_del(&my\_cdev);

unregister\_chrdev\_region(dev\_num, 1);

printk(KERN\_ALERT "simple\_char\_dev: Failed to create device\n");

return -1;

}

printk(KERN\_INFO "simple\_char\_dev: Device initialized with major %d, minor %d\n", MAJOR(dev\_num), MINOR(dev\_num));

return 0;

}

/\* Module cleanup function, called when the module is removed \*/

static void \_\_exit simple\_char\_exit(void) {

/\* Remove device from /dev \*/

device\_destroy(my\_class, dev\_num);

/\* Destroy the device class \*/

class\_destroy(my\_class);

/\* Remove the character device \*/

cdev\_del(&my\_cdev);

/\* Unregister the device numbers \*/

unregister\_chrdev\_region(dev\_num, 1);

printk(KERN\_INFO "simple\_char\_dev: Device unregistered\n");

}

/\* Register module initialization and cleanup functions \*/

module\_init(simple\_char\_init);

module\_exit(simple\_char\_exit);

/\* Module metadata \*/

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Shivam Kumar");

MODULE\_DESCRIPTION("A simple character device driver for Raspberry Pi 4");

MODULE\_VERSION("1.0");

**Working & Flow :**

**Step 1: Allocate Device Numbers**

ret = alloc\_chrdev\_region(&dev\_num, 0, 1, DEVICE\_NAME);

The alloc\_chrdev\_region() function allocates a range of device numbers dynamically. Let's break down its parameters:

* &dev\_num: Pointer to store the allocated device number
* 0: Starting minor number
* 1: Number of consecutive device numbers to allocate
* DEVICE\_NAME: Name of the device ("dev1")

The function:

1. Finds an available major number
2. Allocates the requested range of minor numbers
3. Stores the result in dev\_num (contains both major and minor numbers)
4. Returns 0 on success, negative on failure

After successful allocation:

* Major number can be retrieved using MAJOR(dev\_num)
* Minor number can be retrieved using MINOR(dev\_num)

**Step 2: Initialize cdev Structure**

cdev\_init(&my\_cdev, &fops);

The cdev structure represents a character device. cdev\_init() initializes this structure:

1. Sets up the cdev structure fields:

struct cdev {

struct kobject kobj; *// Kernel object representation*

struct module \*owner; *// Pointer to the owning module*

const struct file\_operations \*ops; *// File operations*

struct list\_head list; *// Device list*

dev\_t dev; *// Device number*

unsigned int count; *// Number of devices*

};

1. Links the file operations (fops) to the device:

static struct file\_operations fops = {

.owner = THIS\_MODULE, *// Set owner to this module*

.open = simple\_open, *// Function called when device opened*

.release = simple\_release, *// Function called when device closed*

.read = simple\_read, *// Function called when device read*

.write = simple\_write, *// Function called when device written*

};

**Step 3: Add Device to System**

ret = cdev\_add(&my\_cdev, dev\_num, 1);

cdev\_add() adds the initialized character device to the system:

1. Makes the device "live":
   * Registers the device with the kernel
   * Makes it visible to the system
   * Enables it to handle I/O operations
2. Parameters:
   * &my\_cdev: Pointer to initialized cdev structure
   * dev\_num: Device number (major and minor)
   * 1: Number of consecutive minor numbers
3. Internal operations:
   * Adds device to the kernel's device list
   * Sets up the device number
   * Makes the device available for operations

**Step 4: Create Device Class**

my\_class = class\_create(THIS\_MODULE, "simple\_char\_class");

The device class is created for udev device management:

1. Creates a new class in sysfs (/sys/class/simple\_char\_class)
2. Parameters:
   * THIS\_MODULE: Owner module
   * "simple\_char\_class": Class name
3. Internal operations:
   * Creates class directory in sysfs
   * Sets up class attributes
   * Enables udev to manage devices of this class

**Step 5: Create Device Node**

device\_create(my\_class, NULL, dev\_num, NULL, DEVICE\_NAME);

Creates the actual device file in /dev:

1. Parameters:
   * my\_class: Device class created earlier
   * NULL: No parent device
   * dev\_num: Device number
   * NULL: No additional data
   * DEVICE\_NAME: Name of device file ("dev1")
2. Internal operations:
   * Creates device entry in sysfs
   * Triggers udev to create device node in /dev
   * Sets up device attributes and permissions

**Data Flow During Operations**

**When Device is Opened:**

fd = open("/dev/dev1", O\_RDWR);

1. Kernel looks up the device number
2. Finds corresponding cdev structure
3. Calls simple\_open() from fops
4. Increments open\_count
5. Returns file descriptor

**During Write Operation:**

write(fd, "hello", 5);

1. Kernel validates the file descriptor
2. Calls simple\_write() from fops
3. Process:

to\_copy = min(size, BUFFER\_SIZE - (size\_t)(\*offset));

not\_copied = copy\_from\_user(kernel\_buffer + \*offset, user\_buffer, to\_copy);

* + Calculates available buffer space
  + Safely copies data from user space to kernel buffer
  + Updates offset
  + Returns number of bytes written

**During Read Operation:**

read(fd, buffer, 100);

1. Kernel validates file descriptor
2. Calls simple\_read() from fops
3. Process:

to\_copy = min(size, BUFFER\_SIZE - (size\_t)(\*offset));

not\_copied = copy\_to\_user(user\_buffer, kernel\_buffer + \*offset, to\_copy);

* + Calculates available data
  + Safely copies data from kernel buffer to user space
  + Updates offset
  + Returns number of bytes read

Other

**Module Initialization (**simple\_char\_init**function)**

* **Code Location**:

static int \_\_init simple\_char\_init(void) { ... }

* **Explanation**: This is the module’s entry point, which is called when you load the driver (e.g., using insmod).
* **Relevant Code Steps**:
  1. alloc\_chrdev\_region(&dev\_num, 0, 1, DEVICE\_NAME);: Allocates a major and minor number for the device.
  2. cdev\_init(&my\_cdev, &fops); and cdev\_add(&my\_cdev, dev\_num, 1);: Initializes and adds the character device to the kernel with the specified file operations (fops).
  3. class\_create**and**device\_create: Creates a class and a device, making the device node /dev/dev1 automatically available. This allows user-space applications to access this device by opening /dev/dev1.

When the device\_create function is called, udev**(the device manager in Linux)** automatically creates the device node, which will be visible in /dev.

* **Interaction with Raspberry Pi 4**:
  + Here, the kernel, device management system (udev), and your driver interact to allocate the device and expose it to user space. This step sets up the driver without engaging any hardware directly, as this is a virtual character device not mapped to physical hardware.

**2. File Operations**

These are the functions defined in the struct file\_operations structure (fops). Each function in fops is invoked when an application interacts with /dev/dev1.

**Open Operation (**simple\_open**function)**

* **Code Location**:

static int simple\_open(struct inode \*inode, struct file \*file) { ... }

* **Explanation**: The simple\_open function is triggered when a user-space application opens /dev/dev1.
  + **Example Trigger**: Running cat /dev/dev1 or echo "data" > /dev/dev1 would invoke simple\_open.
* **Interaction with Raspberry Pi 4**:
  + This increments open\_count and logs the number of times the device has been opened.
  + No hardware on the Pi 4 is directly involved in this virtual operation, as the character driver here is purely software-based.

**Release Operation (**simple\_release**function)**

* **Code Location**:

static int simple\_release(struct inode \*inode, struct file \*file) { ... }

* **Explanation**: This is called when the file descriptor for /dev/dev1 is closed (e.g., when cat /dev/dev1 finishes reading).
* **Interaction with Raspberry Pi 4**:
  + Like simple\_open, this function only logs an event, releasing the software resource, with no interaction with the Raspberry Pi 4 hardware.

**Read Operation (**simple\_read**function)**

* **Code Location**:

static ssize\_t simple\_read(struct file \*file, char \_\_user \*user\_buffer, size\_t size, loff\_t \*offset) { ... }

* **Explanation**: When an application reads from /dev/dev1, this function is triggered.
  + **Example Trigger**: Running cat /dev/dev1 or dd if=/dev/dev1 of=output.txt bs=1024 count=1 would invoke simple\_read.
  + **Details**:
    - simple\_read calculates how many bytes to read (to\_copy) and then uses copy\_to\_user to transfer data from kernel\_buffer to user\_buffer.
    - copy\_to\_user ensures safe data transfer between kernel space and user space.
* **Interaction with Raspberry Pi 4**:
  + No direct hardware interaction occurs. The read function simply provides data from kernel\_buffer.

**Write Operation (**simple\_write**function)**

* **Code Location**:

static ssize\_t simple\_write(struct file \*file, const char \_\_user \*user\_buffer, size\_t size, loff\_t \*offset) { ... }

* **Explanation**: This function is triggered when an application writes to /dev/dev1.
  + **Example Trigger**: Running echo "Hello" > /dev/dev1 would invoke simple\_write.
  + **Details**:
    - It calculates how many bytes to write (to\_copy) and uses copy\_from\_user to transfer data from user\_buffer to kernel\_buffer.
    - copy\_from\_user ensures safe data transfer from user space to kernel space.
* **Interaction with Raspberry Pi 4**:
  + Similar to simple\_read, this function works entirely in software and doesn’t interact with hardware.

**3. Module Exit (**simple\_char\_exit**function)**

* **Code Location**:

static void \_\_exit simple\_char\_exit(void) { ... }

* **Explanation**: This function is called when the module is removed (e.g., using rmmod).
  + **Relevant Code Steps**:
    - device\_destroy(my\_class, dev\_num);: Removes the device node /dev/dev1.
    - class\_destroy(my\_class);: Destroys the device class.
    - cdev\_del(&my\_cdev); and unregister\_chrdev\_region(dev\_num, 1);: Remove the character device and release the major/minor numbers.
* **Interaction with Raspberry Pi 4**:
  + This function removes the software constructs created by the driver, but as a virtual device, there is no hardware interaction.

**Full Flow Summary**

1. **Loading the Module**:
   * Creates and registers the device node /dev/dev1.
   * No direct hardware interaction.
2. **Interacting with**/dev/dev1:
   * Opening, reading, writing, and closing /dev/dev1 interacts with kernel-space data (kernel\_buffer) only, without triggering any hardware on the Pi 4.
   * Functions like copy\_to\_user and copy\_from\_user handle safe data transfer between kernel and user spaces.
3. **Unloading the Module**:
   * Cleans up resources, deletes /dev/dev1, and removes kernel registrations.

**Makefile**

obj-m += simple\_char\_driver.o

all:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean

**Steps to Compile and Run**

1. Save the driver code as simple\_char\_driver.c and the Makefile in the same directory.
2. Run make to build the kernel module.
3. Insert the module using:

sudo insmod simple\_char\_driver.ko

1. Check the device creation by looking in /dev, or use dmesg to see kernel logs.
2. To remove the module:

sudo rmmod simple\_char\_driver