

WIPRO LINUX DEVICE DRIVER TRAINING

CAPSTONE PROJECTS

PROJECT 1

CHARACTER DEVICE DRIVER

Submitted by,

Ganavi M

Bhavana

Chandana M K

Sl.No	Table of Contents
1	Introduction
2	Objective
3	Software Development Life Cycle
4	Design Phase
	a. High level diagram
	b. Use Case Diagram
	c. Flow Diagram
5	Solid Principle Implementation on Code
6	Code Documentation
	a. Kernel Program
	b. User Program
	c. Makefile
6	API Used
7	Screenshots
8	Key Learnings
9	Conclusion

1. INTRODUCTION

In the realm of kernel development and systems programming, understanding how to interact with hardware devices at a low level is crucial. A fundamental aspect of this understanding lies in the creation and management of device drivers. A device driver acts as a bridge between the operating system's kernel and the hardware it controls, facilitating communication and providing a standardized interface for user-space applications. This document outlines the process of developing, testing, and documenting a character device driver for a simulated hardware device. The objective is to gain insight into essential concepts such as device registration, file operations, and module interaction with the Linux kernel.

Key Concepts:

1. **Kernel Module Programming:** Understanding how to write code that operates within the kernel space is fundamental. Kernel modules are pieces of code that can be dynamically loaded and unloaded into the kernel as needed, providing an efficient way to extend kernel functionality or support new hardware.
2. **Character Device Registration:** Registering a character device involves informing the kernel about the existence and characteristics of the device. This step is crucial for enabling user-space applications to interact with the device through standard file operations like open, read, write, and close.
3. **File Operations:** Implementing file operations allows user-space programs to communicate with the device driver. These operations include opening the device, reading from it, writing to it, and closing it. Each operation involves interactions between the user-space application, the kernel, and the device driver.

2. OBJECTIVE

The primary goal is to develop a character device driver tailored for a simulated hardware device. This driver will mimic the behavior of a real hardware device, allowing us to explore and understand the intricacies of kernel module programming.

Day-by-Day Tasks:

Day 1: Design the Character Device Features and Plan the Module Structure

Day 2: Implement Module Initialization and Cleanup Routines

Day 3: Add File Operations (open, close, read, write)

Day 4: Implement ioctl Command for Device Control

Day 5: Test the Driver with a User-Space Application and Document the Development Process

By following this structured approach, we aim to build a functional and well-documented character device driver, gaining valuable insights into the interaction between user-space applications and the kernel, as well as the intricacies of Linux kernel module development. This project serves as a foundational learning experience for anyone interested in systems programming and driver development.

3. SOFTWARE DEVELOPMENT LIFECYCLE

Requirement Analysis:

- Identify and specify the need for a character device driver.
- Define the file operations (open, close, read, write) and IOCTL commands required.

Design:

- Plan the structure and design of the driver.
- Define how the initialization and cleanup routines will work.
- Design the file operations and IOCTL command handling.

Implementation:

- Write the driver code.
- Implement the module initialization and cleanup routines.
- Add file operations (open, close, read, write).
- Implement IOCTL command handling.

Testing:

- Develop a user-space application to interact with the driver.
- Perform functional testing of all operations (open, read, write, IOCTL).
- Ensure the driver handles errors and edge cases gracefully.

Deployment:

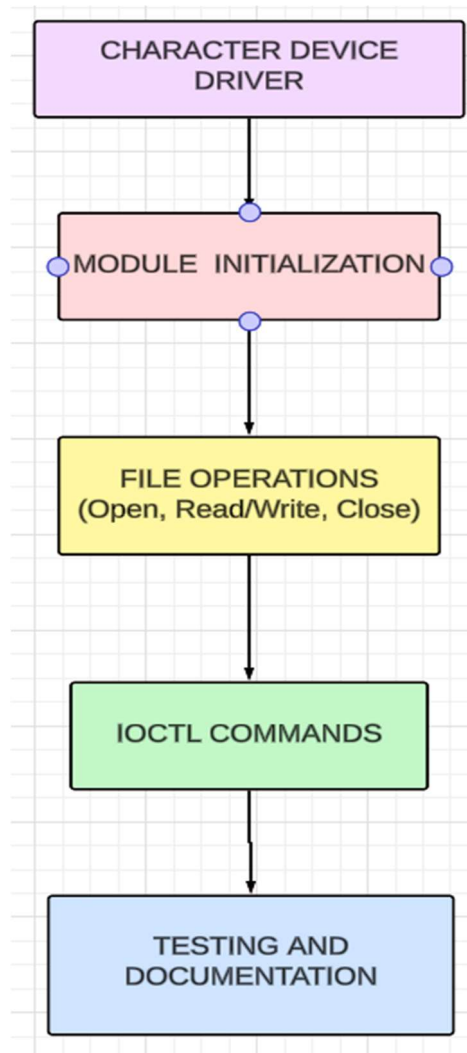
- Load the driver module into the kernel.
- Ensure the driver is correctly registered and available for use.
- Verify the driver works correctly in the target environment.

Maintenance:

- Monitor for any issues or bugs reported by users.
- Update the driver as necessary to maintain compatibility with new kernel versions or hardware.
- Provide patches or updates to fix any identified issues.

4. DESIGN PHASE

1. HIGH LEVEL DIAGRAM



This diagram outlines the development process of a character device driver in a hierarchical structure, illustrating the sequential flow of tasks involved in its creation, implementation, and validation

1. Character Device Driver:

- This is the overarching entity representing the entire driver module.

2. Module Initialization/Cleanup:

- These are the routines responsible for setting up and tearing down the driver module.
- During initialization, resources are allocated, and necessary configurations are made to prepare the driver for operation.
- During cleanup, any allocated resources are released, and the driver's presence in the system is gracefully removed.

3. File Operations:

- This segment encapsulates the functionalities related to basic file operations, such as opening, closing, reading, and writing data to the character device.
- Each operation is handled by specific functions within the driver, ensuring proper communication between the user space and the kernel space.

4. IOCTL Commands:

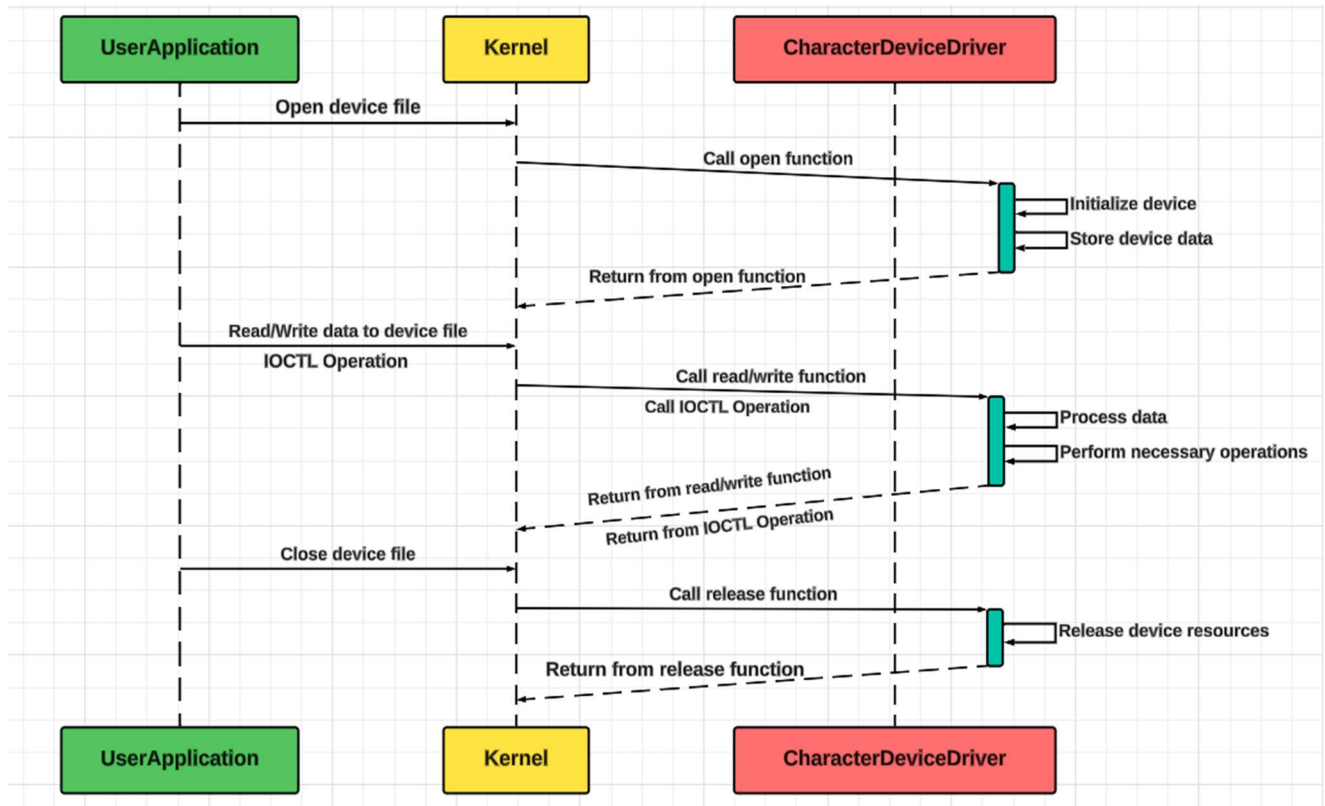
- This section deals with Input/Output Control (IOCTL) commands, which allow for more advanced control and configuration of the device beyond standard read and write operations.
- IOCTL commands provide a mechanism for applications to interact with the driver to perform specific actions or retrieve information about the device.

5. Testing & Documentation:

- This phase involves the validation of the driver's functionality and the creation of comprehensive documentation.
- Testing ensures that the driver behaves as expected under various conditions and scenarios, identifying and addressing any potential issues or bugs.
- Documentation captures the development process, including design decisions, implementation details, usage instructions, and any other relevant information for users and developers.

Overall, this hierarchical representation delineates the systematic progression of tasks involved in developing a character device driver, from initialization and file operations to advanced command handling, testing, and documentation.

2. USE CASE DIAGRAM

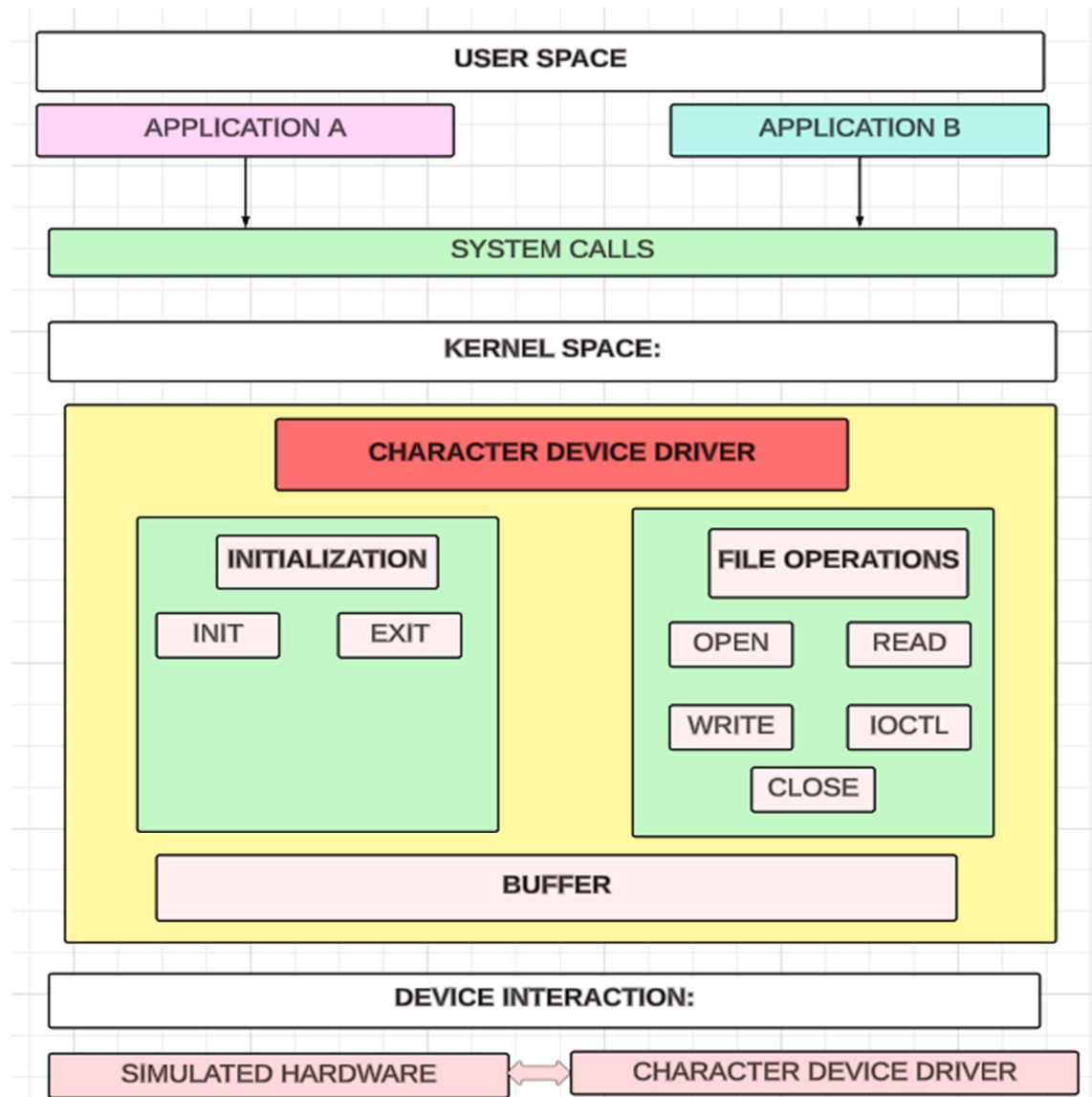


Main Flow:

- a. User Application opens the device file.
- b. Kernel calls the open function in the character device driver.
- c. Character Device Driver initializes the device and stores the device data in the file structure.
- d. User Application reads or writes data to the device file.
- e. Kernel calls the read or write function and ioctl function in the character device driver.
- f. Character Device Driver processes the data and performs the necessary operations.
- g. User Application closes the device file.
- h. Kernel calls the release function in the character device driver.
- i. Character Device Driver releases the device resources.

Postconditions: The character device driver is unloaded from the kernel.

3. FLOW DIAGRAM



1. User Space:

- **Applications:** There are multiple applications running in user space, such as Application A and Application B. These applications need to interact with hardware devices.
- **System Calls:** When an application wants to communicate with a device, it uses system calls. System calls are the way user-space applications request services from the operating system's kernel.

2. Kernel Space:

- a. Character Device Driver: This is the core component in the kernel that handles interactions with the character device.
- b. Initialization:
 - `init()`: This function is called when the device driver is loaded into the kernel. It sets up everything needed for the driver to work.
 - `exit()`: This function is called when the device driver is unloaded from the kernel. It cleans up and releases resources.

- c. File Operations:

These are the main functions that handle various actions performed by user applications:

- `open()`: Called when an application opens the device file. It prepares the device for use.
- `read()`: Called when an application reads data from the device. It transfers data from the device to the application.
- `write()`: Called when an application writes data to the device. It transfers data from the application to the device.
- `ioctl()`: Called for device-specific operations. It allows applications to send control commands to the device driver.
- `release()`: Called when an application closes the device file. It finalizes any pending operations and releases resources.

- d. Buffer:

- This is a memory area used to temporarily store data being transferred between the user space and the device.

3. Device Interaction:

- This section shows the communication link between the simulated hardware and the character device driver. The driver acts as a bridge, managing the data exchange and control commands between the applications and the hardware.

5. SOLID PRINCIPLES IMPLEMENTATION ON CODE

1. Single Responsibility Principle (SRP)

Kernel Module:

- Each function in the kernel module has a single responsibility:
- `dev_open` handles device opening.
- `dev_release` handles device release.
- `dev_read` handles reading from the device.
- `dev_write` handles writing to the device (although it's not supported in this case).
- `dev_ioctl` handles the `ioctl` commands.

User Program:

- Functions in the user program are responsible for specific tasks:
- `set_message` sends a message to the device.
- `get_message` retrieves a message from the device.
- `main` handles the overall flow of operations (open, set message, read, get message, close).

2. Open/Closed Principle (OCP)

Kernel Module:

- The kernel module's `file_operations` structure is open for extension. If you need to add more operations, you can extend this structure without modifying existing functions.

User Program:

- If new functionalities need to be added to the user program, such as additional `ioctl` commands, they can be incorporated by adding new functions without modifying existing ones.

3. Liskov Substitution Principle (LSP)

- The Liskov Substitution Principle is less relevant here since we are not using inheritance or polymorphism in a classical sense. However, the general idea of substitutability is present in the sense that the kernel module functions correctly handle their respective tasks and can be "substituted" or called in various contexts without causing errors.

4. Interface Segregation Principle (ISP)

Kernel Module:

- The `file_operations` structure in the kernel module adheres to this principle by only implementing the necessary interfaces (`read`, `write`, `ioctl`, `open`, `release`). If a certain operation (e.g., `write`) is not needed, it returns an appropriate error.

User Program:

- The user program has specific functions (`set_message`, `get_message`) that focus on specific `ioctl` commands rather than having a single function handle all possible operations, making the interface more manageable and less prone to errors.

5. Dependency Inversion Principle (DIP)

- The user program relies on the abstractions provided by the kernel (device file and `ioctl` interface) rather than directly manipulating hardware or kernel internals. This decouples the high-level logic of the user program from low-level kernel operations.

6. CODE IMPLEMENTATION

KERNEL CODE:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/uaccess.h>

#define DEVICE_NAME "simchardev"
#define BUF_LEN 80

// IOCTL Commands
#define IOCTL_SET_MSG_IOW('a', 1, char *)
#define IOCTL_GET_MSG_IOR('a', 2, char *)

static int Major;
static char message[BUF_LEN];
static char *msg_ptr;

// Function Prototypes
static int dev_open(struct inode *, struct file *);
static int dev_release(struct inode *, struct file *);
static ssize_t dev_read(struct file *, char __user *, size_t, loff_t *);
static ssize_t dev_write(struct file *, const char __user *, size_t, loff_t *);
static long dev_ioctl(struct file *, unsigned int, unsigned long);

// File Operations Structure
struct file_operations fops = {
    .read = dev_read,
    .write = dev_write,
    .unlocked_ioctl = dev_ioctl,
    .open = dev_open,
    .release = dev_release
};

// Initialization Function
static int __init chardev_init(void) {
    Major = register_chrdev(0, DEVICE_NAME, &fops);
    if (Major < 0) {
```

```

    printk(KERN_ALERT "Registering char device failed with %d\n", Major);
    return Major;
}

printk(KERN_INFO "I was assigned major number %d. To talk to\n", Major);
printk(KERN_INFO "the driver, create a dev file with\n");
printk(KERN_INFO "'mknod /dev/%s c %d 0'\n", DEVICE_NAME, Major);
return 0;
}

// Cleanup Function
static void __exit chardev_exit(void) {
    unregister_chrdev(Major, DEVICE_NAME);
    printk(KERN_INFO "Goodbye, world!\n");
}

// Open Function
static int dev_open(struct inode *inode, struct file *file) {
    static int counter = 0;
    sprintf(message, "I already told you %d times Hello world!\n", counter++);
    msg_ptr = message;
    try_module_get(THIS_MODULE);
    return 0;
}

// Release Function
static int dev_release(struct inode *inode, struct file *file) {
    module_put(THIS_MODULE);
    return 0;
}

// Read Function
static ssize_t dev_read(struct file *filp, char __user *buffer, size_t length, loff_t * offset) {
    int bytes_read = 0;
    if (*msg_ptr == 0)
        return 0;
    while (length && *msg_ptr) {
        put_user(*(msg_ptr++), buffer++);
    }
}

```

```

        length--;
        bytes_read++;
    }
    return bytes_read;
}

// Write Function
static ssize_t dev_write(struct file *filp, const char __user *buff, size_t len, loff_t * off) {
    printk(KERN_ALERT "Sorry, this operation isn't supported.\n");
    return -EINVAL;
}

// Ioctl Function
static long dev_ioctl(struct file *file, unsigned int cmd, unsigned long arg) {
    switch (cmd) {
        case IOCTL_SET_MSG:
            if (copy_from_user(message, (char __user *)arg, BUF_LEN))
                return -EFAULT;
            msg_ptr = message;
            break;
        case IOCTL_GET_MSG:
            if (copy_to_user((char __user *)arg, message, BUF_LEN))
                return -EFAULT;
            break;
        default:
            return -ENOTTY;
    }
    return 0;
}

// Register Module Initialization and Cleanup Functions
module_init(chardev_init);
module_exit(chardev_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A Simple Character Device Driver");

```

USER CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <sys/ioctl.h>

#define DEVICE_FILE_NAME "/dev/simchardev"
#define BUF_LEN 80

// IOCTL Commands
#define IOCTL_SET_MSG_IOW('a', 1, char *)
#define IOCTL_GET_MSG_IOR('a', 2, char *)

void set_message(int file_desc, char *message) {
    int ret_val = ioctl(file_desc, IOCTL_SET_MSG, message);
    if (ret_val < 0) {
        printf("ioctl_set_msg failed:%d\n", ret_val);
        exit(-1);
    }
}

void get_message(int file_desc) {
    char message[BUF_LEN];
    int ret_val = ioctl(file_desc, IOCTL_GET_MSG, message);
    if (ret_val < 0) {
        printf("ioctl_get_msg failed:%d\n", ret_val);
        exit(-1);
    }
    printf("The message is: %s\n", message);
}

int main() {
    int file_desc;
    char read_buffer[BUF_LEN];
    int read_bytes;
```



```

// Open the device file
file_desc = open(DEVICE_FILE_NAME, O_RDWR);
if (file_desc < 0) {
    printf("Can't open device file: %s\n", DEVICE_FILE_NAME);
    exit(-1);
}

// Set a message using ioctl
set_message(file_desc, "Hello from user program!");

// Read message from the device
read_bytes = read(file_desc, read_buffer, BUF_LEN);
if (read_bytes < 0) {
    printf("Failed to read from device\n");
    close(file_desc);
    exit(-1);
}

read_buffer[read_bytes] = '\0'; // Null-terminate the string
printf("Read from device: %s\n", read_buffer);

// Get the message using ioctl
get_message(file_desc);

// Close the device file
close(file_desc);

return 0;
}

```

MAKEFILE:

```

obj-m += uid.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

```

7. API 's USAGE IN CODE

User Program

- **Open** - Opens a file or device.
Parameters: `const char *pathname` (path to the file), `int flags` (access mode).
- **Ioctl** - Performs a device-specific input/output operation.
Parameters: `int fd` (file descriptor), `unsigned long request` (operation code), additional arguments based on the request.
- **Read** - Reads data from a file descriptor.
Parameters: `int fd` (file descriptor), `void *buf` (buffer to store data), `size_t count` (number of bytes to read).
- **Close** - Closes a file descriptor.
Parameters: `int fd` (file descriptor).

Kernel Module

- **register_chrdev** - Registers a character device driver.
Parameters: `unsigned int major` (major device number), `const char *name` (device name), `const struct file_operations *fops` (file operations structure).
- **unregister_chrdev** - Unregisters a character device driver.
Parameters: `unsigned int major` (major device number), `const char *name` (device name).
- **try_module_get** - Increments the module's reference count.
Parameters: `struct module *module` (module structure, usually `THIS_MODULE`).
- **module_put** - Decrements the module's reference count.
Parameters: `struct module *module` (module structure, usually `THIS_MODULE`).
- **copy_from_user** - Copies data from user space to kernel space.
Parameters: `void *to` (destination buffer), `const void __user *from` (source buffer), `unsigned long n` (number of bytes to copy).

- **copy_to_user** - Copies data from kernel space to user space.
Parameters: void __user *to (destination buffer), const void *from (source buffer), unsigned long n (number of bytes to copy).
- **module_init** - Specifies the function to be executed when the module is loaded.
Parameters: module_init_func (pointer to the initialization function).
- **module_exit** - Specifies the function to be executed when the module is unloaded.
Parameters: module_exit_func (pointer to the cleanup function).

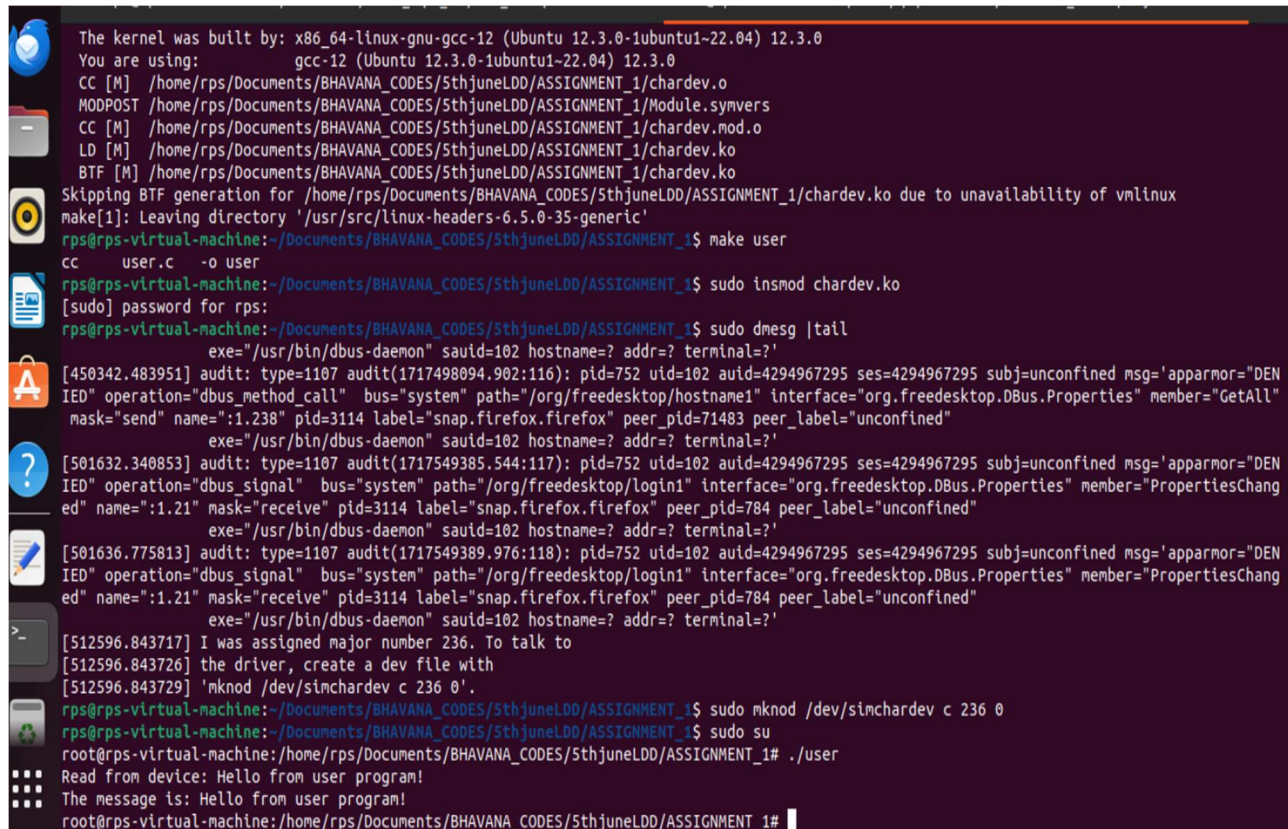
File Operations Structure Functions

- **.read** - Handles reading from the device.
Parameters: struct file *filp, char __user *buffer, size_t length, loff_t *offset.
- **.write** - Handles writing to the device.
Parameters: struct file *filp, const char __user *buff, size_t len, loff_t *off.
- **.unlocked_ioctl** - Handles IOCTL commands.
Parameters: struct file *file, unsigned int cmd, unsigned long arg.
- **.open** - Handles opening the device.
Parameters: struct inode *inode, struct file *file.
- **.release** - Handles closing the device.
Parameters: struct inode *inode, struct file *file.

IOCTL Command Macros

- **_IOW** - Defines a command to write data to the device.
Parameters: type (magic number), nr (command number), data (type of data).
- **_IOR** - Defines a command to read data from the device.
Parameters: type (magic number), nr (command number), data (type of data).

7. SCREENSHOTS



```
The kernel was built by: x86_64-linux-gnu-gcc-12 (Ubuntu 12.3.0-1ubuntu1-22.04) 12.3.0
You are using: gcc-12 (Ubuntu 12.3.0-1ubuntu1-22.04) 12.3.0
CC [M] /home/rps/Documents/BHAVANA_CODES/5thjuneLDD/ASSIGNMENT_1/chardev.o
MODPOST /home/rps/Documents/BHAVANA_CODES/5thjuneLDD/ASSIGNMENT_1/Module.symvers
CC [M] /home/rps/Documents/BHAVANA_CODES/5thjuneLDD/ASSIGNMENT_1/chardev.mod.o
LD [M] /home/rps/Documents/BHAVANA_CODES/5thjuneLDD/ASSIGNMENT_1/chardev.ko
BTF [M] /home/rps/Documents/BHAVANA_CODES/5thjuneLDD/ASSIGNMENT_1/chardev.ko
Skipping BTF generation for /home/rps/Documents/BHAVANA_CODES/5thjuneLDD/ASSIGNMENT_1/chardev.ko due to unavailability of vmlinux
make[1]: Leaving directory '/usr/src/linux-headers-6.5.0-35-generic'
rps@rps-virtual-machine:~/Documents/BHAVANA_CODES/5thjuneLDD/ASSIGNMENT_1$ make user
cc user.c -o user
rps@rps-virtual-machine:~/Documents/BHAVANA_CODES/5thjuneLDD/ASSIGNMENT_1$ sudo insmod chardev.ko
[sudo] password for rps:
rps@rps-virtual-machine:~/Documents/BHAVANA_CODES/5thjuneLDD/ASSIGNMENT_1$ sudo dmesg |tail
exe="/usr/bin/dbus-daemon" sauid=102 hostname=? addr=? terminal=?'
[450342.483951] audit: type=1107 audit(1717498094.902:116): pid=752 uid=102 auid=4294967295 ses=4294967295 subj=unconfined msg='apparmor="DEN
IED" operation="dbus_method_call" bus="system" path="/org/freedesktop/hostname1" interface="org.freedesktop.DBus.Properties" member="GetAll"
mask="send" name=":1.238" pid=3114 label="snap.firefox.firefox" peer_pid=71483 peer_label="unconfined"
exe="/usr/bin/dbus-daemon" sauid=102 hostname=? addr=? terminal=?'
[501632.340853] audit: type=1107 audit(1717549385.544:117): pid=752 uid=102 auid=4294967295 ses=4294967295 subj=unconfined msg='apparmor="DEN
IED" operation="dbus_signal" bus="system" path="/org/freedesktop/login1" interface="org.freedesktop.DBus.Properties" member="PropertiesChang
ed" name=":1.21" mask="receive" pid=3114 label="snap.firefox.firefox" peer_pid=784 peer_label="unconfined"
exe="/usr/bin/dbus-daemon" sauid=102 hostname=? addr=? terminal=?'
[501636.775813] audit: type=1107 audit(1717549389.976:118): pid=752 uid=102 auid=4294967295 ses=4294967295 subj=unconfined msg='apparmor="DEN
IED" operation="dbus_signal" bus="system" path="/org/freedesktop/login1" interface="org.freedesktop.DBus.Properties" member="PropertiesChang
ed" name=":1.21" mask="receive" pid=3114 label="snap.firefox.firefox" peer_pid=784 peer_label="unconfined"
exe="/usr/bin/dbus-daemon" sauid=102 hostname=? addr=? terminal=?'
[512596.843717] I was assigned major number 236. To talk to
[512596.843726] the driver, create a dev file with
[512596.843729] 'mknod /dev/sinchardev c 236 0'.
rps@rps-virtual-machine:~/Documents/BHAVANA_CODES/5thjuneLDD/ASSIGNMENT_1$ sudo mknod /dev/sinchardev c 236 0
rps@rps-virtual-machine:~/Documents/BHAVANA_CODES/5thjuneLDD/ASSIGNMENT_1$ sudo su
root@rps-virtual-machine:/home/rps/Documents/BHAVANA_CODES/5thjuneLDD/ASSIGNMENT_1# ./user
Read from device: Hello from user program!
The message is: Hello from user program!
root@rps-virtual-machine:/home/rps/Documents/BHAVANA_CODES/5thjuneLDD/ASSIGNMENT_1#
```

8. KEY LEARNINGS

1. Kernel Module Programming:

The project introduced fundamental aspects of kernel module programming, including module initialization and cleanup functions. Understanding how to properly load and unload modules is crucial for maintaining system stability.

2. Character Device Registration:

We explored the process of registering a character device with the kernel, allocating major and minor numbers, and creating device nodes. This foundational step is essential for the kernel to recognize and manage the device.

3. File Operations:

Implementing file operations such as open, read, write, and release provided a hands-on experience with how user-space applications interact with device drivers. This understanding is vital for developing efficient and reliable drivers that facilitate seamless communication between hardware and software.

4. Kernel Interaction:

The project emphasized the importance of safely interacting with kernel resources. Techniques such as using kernel memory allocation functions, handling concurrency with proper synchronization mechanisms, and ensuring data integrity are critical for robust driver development.

8. CONCLUSION

Practical Implications:

- Developing a simulated character device driver serves as a stepping stone for creating drivers for real hardware devices. The skills and knowledge gained from this project are directly applicable to more complex and real-world scenarios.
- This exercise enhances problem-solving abilities and deepens the understanding of Linux kernel internals, which are highly valued skills in systems programming, embedded systems development, and various other technology domains

Future Directions:

- Extending the driver to handle more complex operations and integrating it with real hardware devices can further solidify the concepts learned.
- Exploring additional kernel subsystems such as networking, block devices, or USB can provide a broader perspective on driver development and kernel module interactions.

Conclusion:

The development, testing, and documentation of a character device driver for a simulated hardware device not only demystifies the complexities of kernel module programming but also lays a strong foundation for advanced studies and professional work in the field of systems programming. This project underscores the importance of meticulous design, rigorous testing, and thorough documentation in creating reliable and efficient device drivers.