OS ASSIGNMENT 2

Praval Pattam
B220057CS
CS04

1. Problem Statement

The aim of this assignment is to create a character device driver with the following functionalities:

1. Kernel Version Check: The driver must accept an array parameter called kernel_version, specifying the current kernel version. The driver should only be inserted if the provided kernel version matches the version used to compile the module.
2. Driver Insertion: Upon successful insertion, the driver should print the assigned major and minor numbers in the kernel log.
3. Device Read/Write Operations: After insertion, the user can write to it and read from it using two different methods: the echo command for writing and the cat command for reading or a user program written in C or any high-level language.

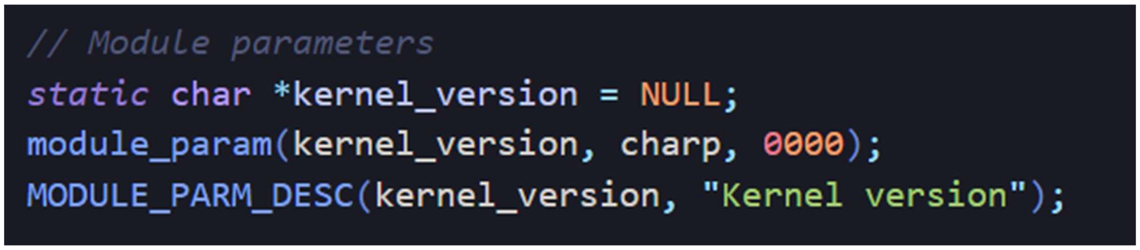
   Whenever the read and write functions of the driver are called appropriate messages should be printed in the kernel log.

2. Methodology

1. **Development Process**:
   - **Initialization**: The device driver is initialized by checking the kernel version and registering the character device.
   - **File Operations**: Implement read and write functions to handle data transfer between user space and kernel space.
   - **Device Registration**: Allocate major and minor numbers for the device and create the device node in /dev.
   - **Logging**: Print kernel log messages for debugging and verification.
2. **Code Structure**:
   - **Device Driver**: Contains the main logic for device initialization, read/write operations, and cleanup.
   - **User Program**: Interacts with the device driver to perform read and write operations based on user input.

3. Detailed Explanation – Source Code

Initialization of the module parameters:



```c
// Module parameters
static char *kernel_version = NULL;
module_param(kernel_version, charp, 0000);
MODULE_PARM_DESC(kernel_version, "Kernel version");
```

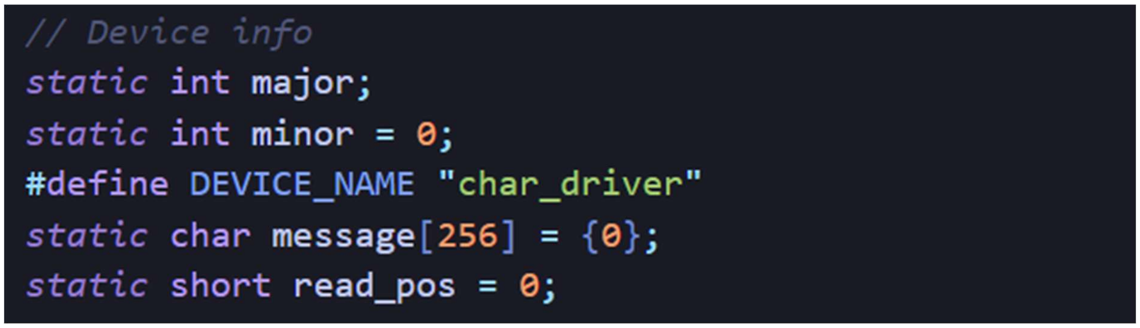
This section defines a module parameter and its description:

- static char *kernel_version = NULL;
  Declares a pointer to a character array (char *) and initializes it to NULL. This variable will store the kernel version passed as a parameter.
- module_param(kernel_version, charp, 0000);

  Defines kernel_version as a module parameter, allowing users to pass a string when loading the module. The charp indicates it's a character pointer, and 0000 sets the permission bits (e.g., for reading and writing).
- MODULE_PARM_DESC(kernel_version, "Kernel version");

  Provides a description for the kernel_version parameter.

Initialization of module variables and name:



```c
// Device info
static int major;
static int minor = 0;
#define DEVICE_NAME "char_driver"
static char message[256] = {0};
static short read_pos = 0;
```

This section initializes the device information and variables:

1. **Static Variables**:

   static int major;

   - This variable holds the major number for the device, which the kernel uses to identify the driver associated with the device file.

   static int minor = 0;

   - This variable holds the minor number for the device, which differentiates multiple devices of the same type. In this case, it's initialized to 0 since we are only using one character device right now. As we have more of the same driver we can give the new drivers different minor numbers.

2. **Macro Definition**:

   #define DEVICE_NAME "char_driver"

   - Defines a macro for the device name, simplifying references to the device name throughout the code.

3. **Message Buffer**:

   static char message[256] = {0};

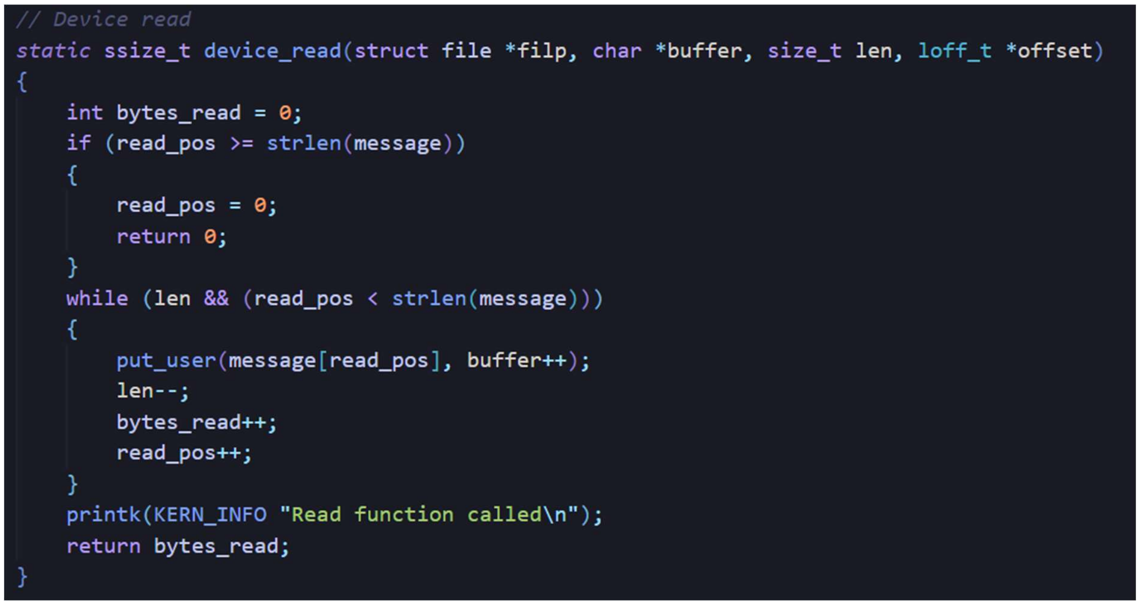
   - Declares a buffer to hold a message that can be read from or written to the device. It's initialized with zeros, providing a space of 256 characters for storage.

4. **Read Position Tracker**:

   static short read_pos = 0;

   - Keeps track of the current read position within the message buffer. This helps manage sequential reading operations, allowing the function to know where to continue reading from.

<u>Writing the device_read function:</u>



```c
// Device read
static ssize_t device_read(struct file *filp, char *buffer, size_t len, loff_t *offset)
{
    int bytes_read = 0;
    if (read_pos >= strlen(message))
    {
        read_pos = 0;
        return 0;
    }
    while (len && (read_pos < strlen(message)))
    {
        put_user(message[read_pos], buffer++);
        len--;
        bytes_read++;
        read_pos++;
    }
    printk(KERN_INFO "Read function called\n");
    return bytes_read;
}
```

1. **Initialize**:

   - bytes_read is initialized to 0 to keep track of the number of bytes read.

2. **Check End of Message**:

   - If read_pos is greater than or equal to the length of the message, it resets read_pos to 0 and returns 0, indicating the end of the message.

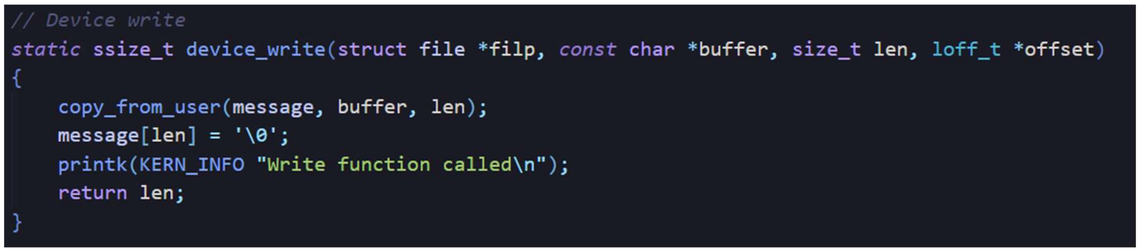
3. **Reading Loop**:

   - The function then enters a loop, copying characters from message to the user-provided buffer using **put_user().**

- It continues until the entire message is read or until len bytes have been read.

4. **Update Counters**:
    - Within the loop, it updates the counters: len is decremented, bytes_read is incremented, and read_pos is incremented.
5. **Log and Return**:
    - Logs the read operation with printk(KERN_INFO "Read function called\n");.
    - Returns the number of bytes read to the caller.

Writing the device_write function:



```c
// Device write
static ssize_t device_write(struct file *filp, const char *buffer, size_t len, loff_t *offset)
{
    copy_from_user(message, buffer, len);
    message[len] = '\0';
    printk(KERN_INFO "Write function called\n");
    return len;
}
```

1. **Copy Data**:

    copy_from_user(message, buffer, len);

    - This function copies len bytes of data from the user-provided buffer (in user space) to the message buffer (in kernel space).
2. **Null-Terminate the String**:

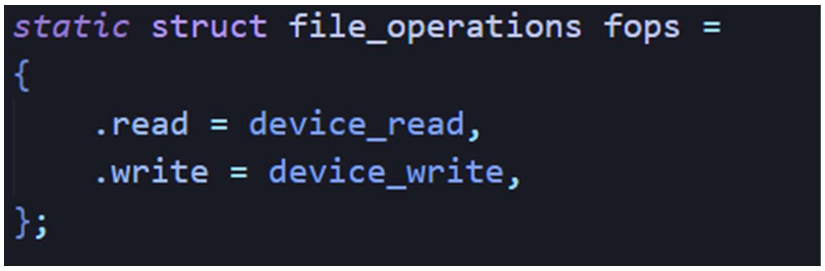
    message[len] = '\0';

    - Adds a null terminator at the end of the copied message, ensuring the buffer is treated as a proper null-terminated string.
3. **Log the Operation**:

    printk(KERN_INFO "Write function called\n");

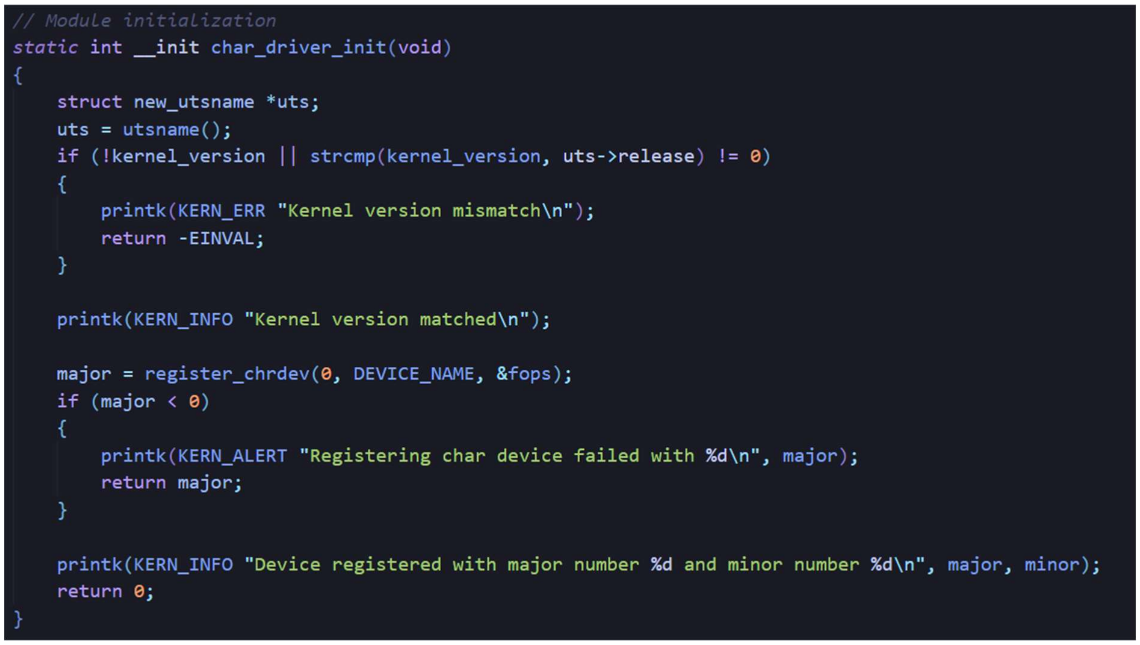
    - Logs a message to the kernel log, indicating that the write operation was called.
4. **Return the Number of Bytes Written**:

We create a structure for the file operations - read and write:



```
static struct file_operations fops =
{
    .read = device_read,
    .write = device_write,
};
```

Writing the init macro for the initialization of the module:



```
// Module initialization
static int __init char_driver_init(void)
{
    struct new_utsname *uts;
    uts = utsname();
    if (!kernel_version || strcmp(kernel_version, uts->release) != 0)
    {
        printk(KERN_ERR "Kernel version mismatch\n");
        return -EINVAL;
    }

    printk(KERN_INFO "Kernel version matched\n");

    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 "Device registered with major number %d and minor number %d\n", major, minor);
    return 0;
}
```

1. **Initialization Function Declaration**:

   static int __init char_driver_init(void)

   - Declares the initialization function for the module. The __init macro indicates that this function is only used during initialization and can be discarded after the module is loaded.

2. **Kernel Version Check**:

   ```
   struct new_utsname *uts;

   uts = utsname();

   if (!kernel_version || strcmp(kernel_version, uts->release) != 0) {

     printk(KERN_ERR "Kernel version mismatch\n");

     return -EINVAL;

   }
   ```

- Retrieves system information using utsname().
- Compares the provided kernel_version parameter with the current kernel version (uts->release).
- Logs an error message and returns

  -EINVAL (Invalid Argument) if the versions do not match.

3. **Log Kernel Version Match**:

```
printk(KERN_INFO "Kernel version matched\n");
```

- Logs a message indicating that the kernel versions match.

4. **Register Character Device**:

```
major = register_chrdev(0, DEVICE_NAME, &fops);
if (major < 0) {
   printk(KERN_ALERT "Registering char device failed with %d\n", major);
   return major;
}
```

- Registers the character device with the system, requesting a dynamically allocated major number (if we give 0).
- Logs an alert and returns the error code if the registration fails.
- Stores the allocated major number in the major variable.

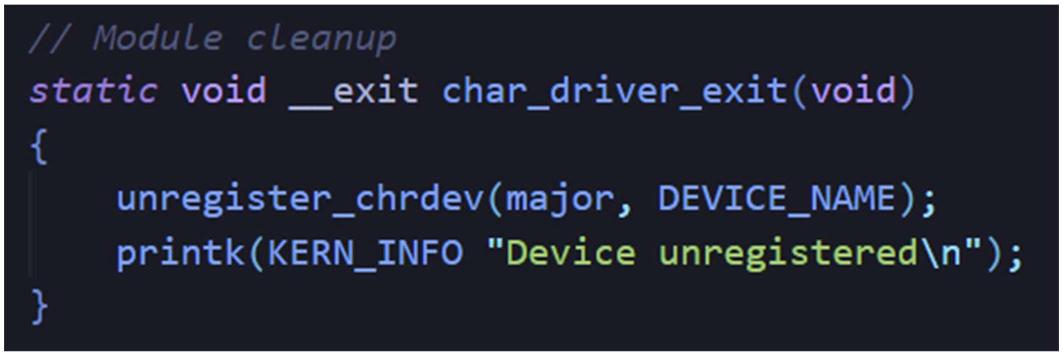
5. **Log Device Registration**:

```
printk(KERN_INFO "Device registered with major number %d and minor number %d\n",
major, minor);
```

- Logs a message indicating that the device has been registered, including the major and minor numbers.

6. **Return Success**

Writing the exit macro for the deregistration of the module:



```
// Module cleanup
static void __exit char_driver_exit(void)
{
    unregister_chrdev(major, DEVICE_NAME);
    printk(KERN_INFO "Device unregistered\n");
}
```

1. **Cleanup Function Declaration**:

   static void __exit char_driver_exit(void)

   - Declares the cleanup function for the module. The __exit macro indicates that the function is only used during module removal and can be discarded afterwards.

2. **Unregister Character Device**:

   unregister_chrdev(major, DEVICE_NAME);

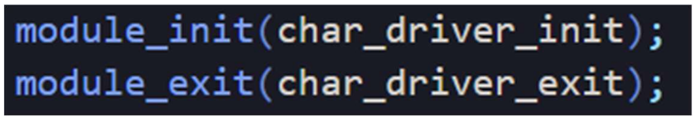
   - Unregisters the character device associated with the major number and device name. This effectively removes the device from the system, cleaning up the resources allocated during initialization.

3. **Log Device Unregistration**:

   printk(KERN_INFO "Device unregistered\n");

   - Logs message to the kernel log, indicating that the device has been unregistered.

Initialization of the init and exit macros:



```
module_init(char_driver_init);
module_exit(char_driver_exit);
```

1. module_init(char_driver_init);

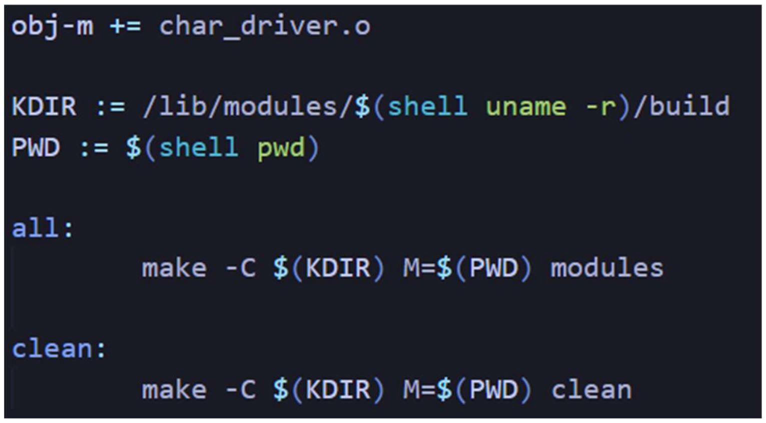
   - Specifies that char_driver_init is the initialization function that should be called when the module is loaded. This function sets up the device, checks kernel version compatibility and registers the device with the system.

2. module_exit(char_driver_exit);

   - Specifies that char_driver_exit is the cleanup function that should be called when the module is unloaded. This function unregisters the device and logs its removal.

<u>Compiling and inserting the module:</u>

Since the device's source code contains header files that are not present in the normal PATH of our machines, we have to use a Makefile (given below) to compile the code such that it can be inserted into the kernel.

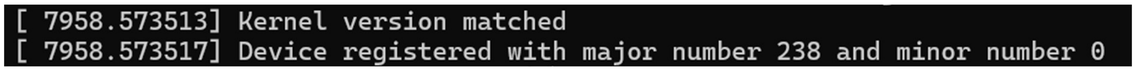


```
obj-m += char_driver.o

KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

all:
	make -C $(KDIR) M=$(PWD) modules

clean:
	make -C $(KDIR) M=$(PWD) clean
```

After making the module we get the **char_driver.ko** file amongst other files which we insert into the kernel using the command.

sudo insmod char_driver.ko kernel_version=$(uname -r)

and now check the kernel logs using **sudo dmesg** and we get the below output in the kernel log.



```
[ 7958.573513] Kernel version matched
[ 7958.573517] Device registered with major number 238 and minor number 0
```

We then create the device node by running the mknod command.

sudo mknod /dev/char_driver c <major_number> <minor_number>

since the major number here is 238 and minor number here is 0 we are going to use the command.

sudo mknod /dev/char_driver c 238 0

- /dev/char_driver: Specifies the path and name of the device node.
- c: Indicates it's a character device.
- 238: Major number assigned to the device by the driver.
- 0: Minor number assigned to the device.

After creating the node for the character device, we have to set permissions for it using the chmod command.

sudo chmod 666 /dev/char_driver

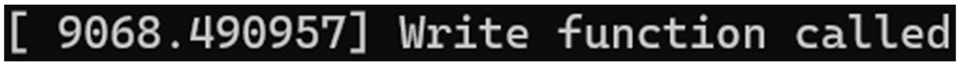
- chmod 666: Sets the read and write permissions for the owner, group, and others.

<u>Testing the Character Driver:</u>

We have now successfully inserted the character driver now we can write and read from it via the **echo** and **cat** command respectively as shown below.

echo PRAVAL_B220057CS > /dev/char_driver

we can check the kernel logs to see if we had a successful write.

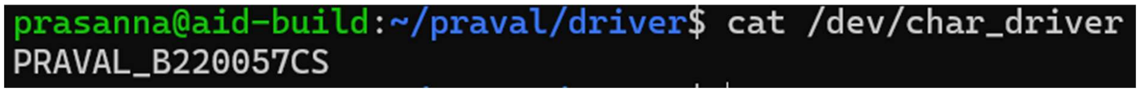


```
[ 9068.490957] Write function called
```

And to perform a read operation on the same char_driver we can use the command

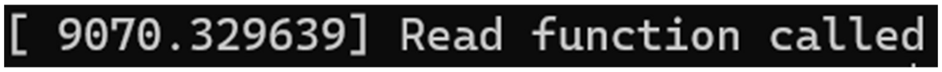
cat /dev/char_driver

which outputs the contents of the char_driver and also logs that the read function has been called



```
prasanna@aid-build:~/praval/driver$ cat /dev/char_driver
PRAVAL_B220057CS
```

And the kernel logs show that the read function has been called



```
[ 9070.329639] Read function called
```

We can also Read and Write from and to the device via a user program by using **open()** system call to open the device and the **read()** and **write()** system calls respectively as shown below.

To open the device:

fd = open(DEVICE, O_RDWR);

- O_RDWR: This flag indicates that the file should be opened for both reading and writing.

To write to the device:

write(fd, write_buff, strlen(write_buff));

To read from the device:

read(fd, read_buff, sizeof(read_buff));

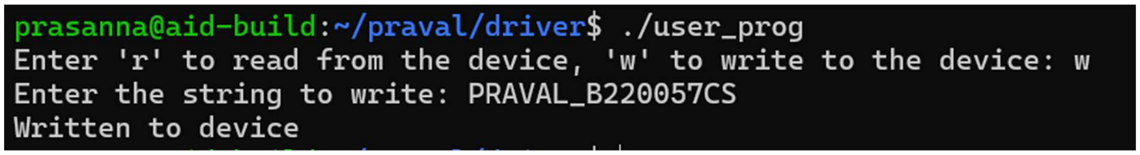
And after we have performed our operations on the character device we can close it using the **close()** system call as show below.

close(fd);

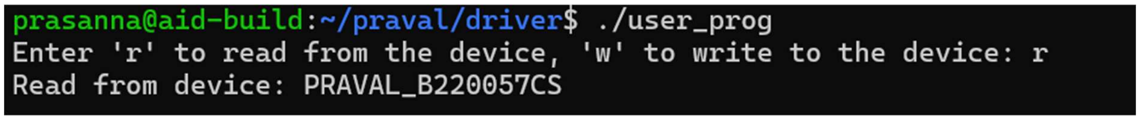
where fd is the file descriptor, write_buff is the write buffer and read_buff is the read buffer.

This user program can be written in any high-level language like C etc.

Quickly writing and testing a user program gives us the following.

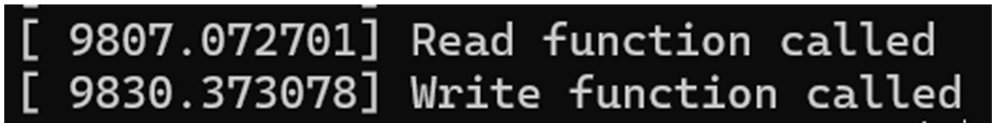


```
prasanna@aid-build:~/praval/driver$ ./user_prog
Enter 'r' to read from the device, 'w' to write to the device: w
Enter the string to write: PRAVAL_B220057CS
Written to device
```



```
prasanna@aid-build:~/praval/driver$ ./user_prog
Enter 'r' to read from the device, 'w' to write to the device: r
Read from device: PRAVAL_B220057CS
```

We can check the kernel logs for the respective log messages.



```
[ 9807.072701] Read function called
[ 9830.373078] Write function called
```

As we can see, we can write and read to and from the character driver via the terminal (using commands) and also via user written programs

<u>Removing the Device:</u>

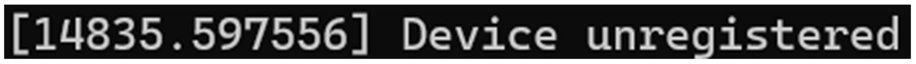
We can finally remove the driver using the following commands.

sudo rm /dev/char_driver

sudo rmmod

Which will run the exit macro that we have written above and unregister the character driver from the kernel and remove the driver file from the Linux file system.

Kernel logging that the device is unregistered.



```
[14835.597556] Device unregistered
```