Writing our first pseudo character driver

- Before we start writing our pseudo character driver, let's understand some important kernel data structures.

The file_operations Structure

- In previous tutorials we have already reserved some device numbers for our use but we have hot connected any of our driver operations to those numbers.
- File operation structure is used to setup this connection. Structure defined in <linux/fs.h>, is a collection of function pointer
- A file_operations structure is called fops. Each field in the structure must point to the function in the driver that implements specific operations or have to left NULL for unsupported operations.

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
        ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
        int (*iopoll)(struct kiocb *kiocb, struct io_comp_batch *,
                      unsigned int flags);
        int (*iterate) (struct file *, struct dir_context *);
        int (*iterate_shared) (struct file *, struct dir_context *);
        __poll_t (*poll) (struct file *, struct poll_table_struct *);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        unsigned long mmap_supported_flags;
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, loff_t, loff_t, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
        int (*check_flags)(int);
        int (*flock) (struct file *, int, struct file_lock *);
        ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
        ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
        int (*setlease)(struct file *, long, struct file_lock **, void **);
        long (*fallocate)(struct file *file, int mode, loff_t offset,
                          loff_t len);
        void (*show_fdinfo)(struct seq_file *m, struct file *f);
#ifndef CONFIG_MMU
        unsigned (*mmap_capabilities)(struct file *);
#endif
        ssize_t (*copy_file_range)(struct file *, loff_t, struct file *,
                        loff_t, size_t, unsigned int);
        loff_t (*remap_file_range)(struct file *file_in, loff_t pos_in,
                                   struct file *file_out, loff_t pos_out,
                                   loff_t len, unsigned int remap_flags);
        int (*fadvise)(struct file *, loff_t, loff_t, int);
        int (*uring_cmd)(struct io_uring_cmd *ioucmd, unsigned int issue_flags);
        int (*uring_cmd_iopoll)(struct io_uring_cmd *, struct io_comp_batch *,
                                unsigned int poll_flags);
} __randomize_layout;
```

- file_operations structure contains many fields but we will concentrate on few basic functions.

struct module *owner;

- This field is used to prevent the module from being unloaded while its operations are in use.
- It is simply initialized to THIS_MODULE.

int (*open) (struct inode *, struct file *);

- This is always the first operation performed on the device file, the driver is not required to declare a corresponding method. If this entry is NULL, opening the device always succeeds, but your driver isn't notified.

ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);

- Used to retrieve data from the device
- A nonnegative return value represents the number of bytes successfully read

ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);

- Sends data to the device
- The return value, if nonnegative, represents the number of bytes successfully written.

int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);

- Offers a way to issue device-specific commands

int (*release) (struct inode *, struct file *);

- This operation is invoked when the file structure is being released.

loff_t (*llseek) (struct file*, loff_t, int)

- This method is used to change the current read/write position in a file, and the new position is returned as a (positive) return value

```
/* file_operations structure */
struct file_operations fops = {
    .owner = THIS_MODULE,
    .open = char_open,
    .release = char_release,
    .write = char_write,
    .read = char_read,
    .llseek = char_lseek,
};
```

The file Structure

- **Defined in <linux/fs.h>**
- It represents an open file. It is created by the kernel on open and is passed to any function that operates on the file, until the last close. After all instances of the file are closed, the kernel releases the data structure.

Initialize and register cdev structure

- Character device registration is important otherwise the user level system calls will not get connected to the file operation methods of the driver.
- **defined in <linux/cdev.h>.**
- For registration, we have to use **cdev_init()** and **cdev_add()**.
- **cdev_init()**
    - **syntax: void cdev_init(struct cdev* cdev, const struct file_operations* fops);**
    - It is a kernel API, which is implemented in char_dev.c. It is used to initialize the cdev structure.
    - In this tutorial, we are only handling one device, we should create a one variable of type struct cdev. But if we are managing more than one device(let's say 5) then we have to create 5 cdev variables.

- **cdev_add()**
    - **syntax: int cdev_add(struct cdev* char_cdev, dev_t dev, unsigned int count);**
        1. **where,**
            - **char_cdev:** cdev structure for the device
            - **dev:** first device number for which this device is responsible
            - **count:** number of consecutive minor numbers corresponding to this device.
    - It is used to register the cdev structure with kernel VFS.
    - After a call to cdev_add(), All functions you have defined in file_operations struct can be called.
    - To remove a character device from system, call **void cdev_del(struct cdev *dev);**

Steps that we need to follow while writing pseudo character driver

1. Dynamically Allocate device number using alloc_chrdev_region. (https://lnkd.in/gwY-n6ZH)
2. Initialize the cdev structure with fops (Explained above)
3. Register a cdev structure with VFS (Explained above)
4. Create device class under /sys/class/ (https://lnkd.in/gvxdBUGa)
5. Create device file (https://lnkd.in/gvxdBUGa)

```
static int __init char_driver_init(void)
{
    /* Dynamically allocate a device number */
    if((alloc_chrdev_region(&dev_num,0,1,"pseudo_char_dev"))<0)
    {
        printk(KERN_INFO"%s: Cannot allocate Major number\n",__func__);
        return -1;
    }
    printk(KERN_INFO"%s: MAJOR = %d, MINOR = %d\n",__func__,MAJOR(dev_num),MINOR(dev_num));

    /* Initialize the cdev structure with fops */
    cdev_init(&char_cdev,&fops);
    char_cdev.owner = THIS_MODULE;

    /* Register cdev structre with VFS */
    if((cdev_add(&char_cdev,dev_num,1))<0)
    {
        printk(KERN_INFO"%s: Cannot register cdev structure with VFS\n",__func__);
        goto unreg_chrdev;
    }

    /* create device class under /sys/class/ */
    if(IS_ERR(char_class = class_create(THIS_MODULE,"char_class")))
    {
        printk(KERN_INFO"%s: Cannot create device class under /sys/class\n",__func__);
        goto cdev_del;
    }

    /* device file creation */
    if(IS_ERR(device_create(char_class,NULL,dev_num,NULL,"char_dev")))
    {
        printk(KERN_INFO"%s: device file creation failed\n",__func__);
        goto destroy_class;
    }

    printk(KERN_INFO"%s: Module init was successful",__func__);
    return 0;

destroy_class:
    class_destroy(char_class);
cdev_del:
    cdev_del(&char_cdev);
unreg_chrdev:
    unregister_chrdev_region(dev_num,1);
    return -1;
}
```

Open Method

- The open method is provided for a driver to do any initialization in preparation for later operations. In most drivers, open should perform the following tasks:
    - Check for device-specific errors
    - Initialize the device
    - Update the f_op pointer, if necessary
    - Allocate and fill any data structure to be put in filp->private_data.

```
int char_open(struct inode* inode, struct file* filp)
{
    printk(KERN_INFO"%s: Device file opened\n",__func__);
    return 0;
}
```

Release Method

- It should perform the following tasks:
  - Deallocate anything that open allocated in filp->private_data
  - Shut down the device on last close

```
int char_release(struct inode* inode, struct file* filp)
{
    printk(KERN_INFO"%s: Device file closed\n",__func__);
    return 0;
}
```

Read Method

- The user level process executes a read system call to read from a device file.
- Whenever user level program executes read system call on our device file, we should transfer data from device memory buffer to user space buffer.
- **Implementation:**
  - **ssize_t char_read(struct file* filp, char __user* buff, size_t count, loff_t *f_pos)**
    1. Check the user requested **count** value with the **MAX_SIZE** of the device
    2. Copy **count** number of bytes from device memory buffer to user space buffer using **copy_to_user().**
    3. Update the **f_pos.**
    4. Return number of bytes successfully read or error code

```
ssize_t char_read(struct file* filp, char __user* buff, size_t count, loff_t *f_pos)
{
    /* 1. Check the user requested count value with the MAX_SIZE */
    if((*f_pos + count) > MAX_SIZE)
        count = MAX_SIZE - *f_pos;

    /* 2. Copy count number of bytes from device memory buffer to user space buffer */
    if(copy_to_user(buff,&dev_buf[*f_pos],count))
    {
        return -EFAULT;
    }

    /* 3. Update the f_pos */
    *f_pos += count;

    printk(KERN_INFO"%s: Read bytes: %ld\n",__func__,count);
    printk(KERN_INFO"%s: Read buffer: %s\n",__func__,dev_buf);
    printk(KERN_INFO"%s: User buffer: %s\n",__func__,buff);
    /* 4. Return number of bytes successfully read */
    return count;
}
```

copy_to_user

- Defined in **<linux/uaccess.h>**
- **Syntax:**
  unsigned long copy_to_user(const void __user* to, const void* from, unsigned long n)
- Where,
  - to = Destination memory address in the process's address space

- from = Source pointer in kernel-space
- n = size in bytes of the data to copy

- It copies data from kernel space to user space
- Returns number of bytes that could not be copied. On success, it will return 0.

Write Method Implementation

- The user level process executes a write system call to write into a device file
- Whenever user level program executes write system call on our device file, we should transfer data from user space buffer to device memory buffer.
- **Implementation:**
  - **ssize_t char_write(struct file* filp, const char __user* buff, size_t count, loff_t* f_pos)**
    1. Check the user requested **count** value with the **MAX_SIZE** of the device
    2. Copy **count** number of bytes from userspace buffer to device memory buffer using **copy_from_user().**
    3. Update the f_pos
    4. Return number of bytes successfully written or error code

```
ssize_t char_write(struct file* filp, const char __user* buff, size_t count, loff_t* f_pos)
{
    /* 1. Check the user requested count value with the MAX_SIZE */
    if((*f_pos + count) > MAX_SIZE)
        count = MAX_SIZE - *f_pos;

    /* If count is 0, then return ENOMEM */
    if(!count)
        return -ENOMEM;

    /* 2. Copy count number of bytes from userspace buffer to device memory buffer */
    if(copy_from_user(&dev_buf[*f_pos],buff,count))
    {
        return -EFAULT;
    }

    /* 3. Update the f_pos */
    *f_pos += count;

    printk(KERN_INFO"%s: Written bytes: %ld\n",__func__,count);
    printk(KERN_INFO"%s: buffer: %s\n",__func__,dev_buf);
    /* 4. Return number of bytes successfully written */
    return count;
}
```

copy_from_user

- Defined in **<linux/uaccess.h>**
- **Syntax:**
  unsigned long copy_from_user(void* to, const void __user* from, unsigned long n)
- Where,
  - to = Destination memory address in the kernel space
  - from = source pointer in user space
  - n = size in bytes of the data to copy
- It copies data from user space to kernel space.
- Returns number of bytes that could not be copied. On success, it will return 0.

llseek method

- It used to change the current read/write position in a file.
- file structure has f_pos field which gives current reading or writing position.
- **Implementation:**
  - **loff_t char_lseek(struct file* filp, loff_t off, int whence)**
    1. whence can be any of these 3 values:
       - **SEEK_SET, SEEK_CUR, SEEK_END**
    2. Use switch-case/ if-else method to know what exactly is whence.
    3. If whence is **SEEK_SET,** then set the current file position **(filp->f_pos)** to **off.**
    4. If whence is **SEEK_CUR,** then set the current file position **(filp->f_pos)** to **filp->f_pos + off.**
    5. If whence is **SEEK_END,** then set the current file position **(filp->f_pos)** to **MAX_SIZE + off.**

```
loff_t char_lseek(struct file* filp, loff_t off, int whence)
{
    loff_t temp;
    printk(KERN_INFO"%s: Current file position = %lld\n",__func__,filp->f_pos);

    switch(whence)
    {
        case SEEK_SET:
                        if((off > MAX_SIZE) || (off < 0) )
                            return -EINVAL;
                        filp->f_pos = off;
                        break;
        case SEEK_CUR:
                        temp = filp->f_pos + off;
                        if(temp > MAX_SIZE || temp < 0)
                            return -EINVAL;
                        filp->f_pos = temp;
                        break;
        case SEEK_END:
                        temp = MAX_SIZE + off;
                        if((temp > MAX_SIZE) || (temp < 0))
                            return -EINVAL;
                        filp->f_pos = temp;
                        break;
        default:
                return -EINVAL;
    }
    printk(KERN_INFO"%s: Updated position: %lld\n",__func__,filp->f_pos);
    return filp->f_pos;
}
```

Device Driver Code

- https://github.com/Jainam103/PseudoCharacterDriver.git

User space Application Code

```c
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>

#define MAX_SIZE 256

int main()
{
    char recv_buf[MAX_SIZE];
    char *buf = "Hello from user space";
    printf("Opening a device\n");
    int fd = open("/dev/char_dev",O_RDWR);
    if(fd < 0)
    {
        printf("Cannot open a device\n");
        return -1;
    }
    if(write(fd,buf,strlen(buf))<0)
    {
        printf("Write operation failed\n");
        return -2;
    }
    if(lseek(fd,0,SEEK_SET) < 0)
    {
        printf("Seek operation failed\n");
        return -3;
    }
    if(read(fd,recv_buf,MAX_SIZE)<0)
    {
        printf("Read operation failed\n");
        return -4;
    }
    printf("Received string: %s\n",recv_buf);
    return 0;
}
```

- Compile it using gcc. (**gcc -o user_space_app user_space_app.c**).

Execution

1. Load the driver using **sudo insmod**

```
[1298875.903598] char_driver_init: MAJOR = 235, MINOR = 0
```

2. Run the application (**sudo ./user_space_app**)
3. Check the output of the **user_space_app**

```
Opening a device
Received string: Hello from user space
```

4. Check the output of **dmesg**

```
[1298875.903598] char_driver_init: MAJOR = 235, MINOR = 0
[1298875.903681] char_driver_init: Module init was successful
[1298933.681625] char_open: Device file opened
[1298933.681630] char_write: Written bytes: 21
[1298933.681630] char_write: buffer: Hello from user space
[1298933.681632] char_lseek: Current file position = 21
[1298933.681633] char_lseek: Updated position: 0
[1298933.681635] char_read: Read bytes: 256
[1298933.681635] char_read: Read buffer: Hello from user space
[1298933.681636] char_read: User buffer: Hello from user space
[1298933.681700] char_release: Device file closed
```

5. Unload the module using **rmmod**

```
[1299067.720198] char_driver_cleanup: Module unloaded
```

References

- https://lwn.net/Kernel/LDD3/
- https://www.udemy.com/course/linux-device-driver-programming-using-beaglebone-black/