# Linux Device Driver Tutorial Part 6 – Cdev structure and File Operations

Post Contents [hide]



## Cdev structure and File Operations of Character drivers

If we want to open, read, write and close we need to register some structures to driver.

### cdev structure

In linux kernel structinode structure is used to represent files. Therefore, it is different from the file structure that represents an open file descriptor. There can be numerous file structures representing multiple open descriptors on a single file, but they all point to a single inode structure.

The inode structure contains a great deal of information about the file. As a general rule, **cdevstructure** is useful for writing driver code:

structcdev is one of the elements of the inode structure. As you probably may know already, an inode structure is used by the kernel internally to represent files. The structcdev is the kernel's internal structure that represents char devices. This field contains a pointer to that structure when the inode refers to a char device file.

```
structcdev {
structkobjectkobj;
struct module *owner;
conststructfile_operations *ops;
structlist_head list;
dev_tdev;
unsignedint count;
};
```

This is **cdev** structure. Here we need to fill two fields,

1. **file_operation** (This we will see after this **cdev** structure)
2. **owner (This should be THIS_MODULE)**

There are two ways of allocating and initializing one of these structures.

1. Runtime Allocation
2. Own allocation

If you wish to obtain a standalone cdev structure at runtime, you may do so with code such as:

```
structcdev *my_cdev = cdev_alloc( );
my_cdev->ops = &my_fops;
```

Or else you can embed the cdev structure within a device-specific structure of your own by using below function.

```
Void cdev_init(structcdev *cdev, structfile_operations *fops);
```

Once the cdev structure is set up with file_operations and owner, the final step is to tell the kernel about it with a call to:

```
intcdev_add(structcdev *dev, dev_tnum, unsigned int count);
```

Here,

dev is the cdev structure,
num is the first device number to which this device responds, and
count is the number of device numbers that should be associated with the device. Often count is one, but there are situations where it makes sense to have more than one device number correspond to a specific device.
If this function returns negative error code, your device has not been added to the system. So check the return value of this function.

After a call to cdev_add(), your device is immediately alive. All functions you defined (through the file_operations structure) can be called.
To remove a char device from the system, call:

```
Void cdev_del(structcdev *dev);
```

Clearly, you should not access the cdev structure after passing it to cdev_del.

## File_Operations

The file_operations structure is how a char driver sets up this connection. The structure, defined in ***<linux/fs.h>,*** is a collection of function pointers. Each open file is associated with its own set of functions . The operations are mostly in charge of implementing the system calls and are therefore, named open, read, and so on.

A file_operations structure or a pointer to one is called fops. Each field in the structure must point to the function in the driver that implements a specific operation, or be left NULL for unsupported operations. The whole structure is mentioned below snippet.

```
structfile_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) (structkiocb *, structiov_iter *);
ssize_t (*write_iter) (structkiocb *, structiov_iter *);
int (*iterate) (struct file *, structdir_context *);
int (*iterate_shared) (struct file *, structdir_context *);
unsignedint (*poll) (struct file *, structpoll_table_struct *);
long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
int (*mmap) (struct file *, structvm_area_struct *);
int (*open) (structinode *, struct file *);
int (*flush) (struct file *, fl_owner_t id);
int (*release) (structinode *, struct file *);
int (*fsync) (struct file *, loff_t, loff_t, intdatasync);
int (*fasync) (int, struct file *, int);
int (*lock) (struct file *, int, structfile_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, structfile_lock *);
ssize_t (*splice_write)(structpipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
ssize_t (*splice_read)(struct file *, loff_t *, structpipe_inode_info *, size_t, unsigned int);
int (*setlease)(struct file *, long, structfile_lock **, void **);
long (*fallocate)(struct file *file, int mode, loff_t offset,
loff_tlen);
void (*show_fdinfo)(structseq_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);
int (*clone_file_range)(struct file *, loff_t, struct file *, loff_t,
        u64);
ssize_t (*dedupe_file_range)(struct file *, u64, u64, struct file *,
        u64);
};
```

This file_operations structure contains many fields. But we will concentrate on very basic functions. Below we will see some fields explanation.

## struct module *owner:

The first file_operations field is not an operation at all; it is a pointer to the module that "**owns**" the structure. This field is used to prevent the module from being unloaded while its operations are in use. Almost all the time, it is simply initialized to THIS_MODULE, a macro defined in *<linux/module.h>*.

### ssize_t (*read) (struct file *, char _ _user *, size_t, loff_t *);

Used to retrieve data from the device. A null pointer in this position causes the read system call to fail with -EINVAL ("Invalid argument"). A non negative return value represents the number of bytes successfully read (the return value is a "signed size" type, usually the native integer type for the target platform).

### ssize_t (*write) (struct file *, const char _ _user *, size_t, loff_t *);

Sends data to the device. If NULL, -EINVAL is returned to the program calling the write system call. The return value, if non negative, represents the number of bytes successfully written.

### int (*ioctl) (structinode *, struct file *, unsigned int, unsigned long);

The ioctl system call offers a way to issue device-specific commands (such as formatting a track of a floppy disk, which is neither reading nor writing). Additionally, a few ioctl commands are recognized by the kernel without referring to the fops table. If the device doesn't provide an ioctl method, the system call returns an error for any request that isn't predefined (-ENOTTY, "No such ioctl for device").

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

Though 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.

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

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

Like open, release can be NULL.

## Example

```
staticstructfile_operations fops =
{
.owner        = THIS_MODULE,
.read         = etx_read,
.write        = etx_write,
.open         = etx_open,
.release      = etx_release,
};
```

If you want to understand the complete flow just have a look at our dummy driver.

## Dummy Driver

Here i have added dummy driver snippet. In this driver code, we can do all open, read, write, close operations. Just go through the code.

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kdev_t.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>

dev_tdev = 0;
staticstruct class *dev_class;
staticstructcdevetx_cdev;

staticint __initetx_driver_init(void);
static void __exit etx_driver_exit(void);
staticintetx_open(structinode *inode, struct file *file);
staticintetx_release(structinode *inode, struct file *file);
staticssize_tetx_read(struct file *filp, char __user *buf, size_tlen,loff_t * off);
staticssize_tetx_write(struct file *filp, const char *buf, size_tlen, loff_t * off);
```

```
staticstructfile_operations fops =
{
.owner          = THIS_MODULE,
.read           = etx_read,
.write          = etx_write,
.open           = etx_open,
.release        = etx_release,
};

staticintetx_open(structinode *inode, struct file *file)
{
printk(KERN_INFO "Driver Open Function Called...!!!\n");
return 0;
}

staticintetx_release(structinode *inode, struct file *file)
{
printk(KERN_INFO "Driver Release Function Called...!!!\n");
return 0;
}

staticssize_tetx_read(struct file *filp, char __user *buf, size_tlen, loff_t *off)
{
printk(KERN_INFO "Driver Read Function Called...!!!\n");
return 0;
}
staticssize_tetx_write(struct file *filp, const char __user *buf, size_tlen, loff_t *off)
{
printk(KERN_INFO "Driver Write Function Called...!!!\n");
returnlen;
}


staticint __initetx_driver_init(void)
{
      /*Allocating Major number*/
if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
printk(KERN_INFO "Cannot allocate major number\n");
return -1;
      }
printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));

      /*Creatingcdev structure*/
cdev_init(&etx_cdev,&fops);

      /*Adding character device to the system*/
if((cdev_add(&etx_cdev,dev,1)) < 0){
printk(KERN_INFO "Cannot add the device to the system\n");
```

```
gotor_class;
	}

	/*Creatingstruct class*/
if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
printk(KERN_INFO "Cannot create the struct class\n");
gotor_class;
	}

	/*Creating device*/
if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
printk(KERN_INFO "Cannot create the Device 1\n");
gotor_device;
	}
printk(KERN_INFO "Device Driver Insert...Done!!!\n");
return 0;

r_device:
class_destroy(dev_class);
r_class:
unregister_chrdev_region(dev,1);
return -1;
}

void __exit etx_driver_exit(void)
{
device_destroy(dev_class,dev);
class_destroy(dev_class);
cdev_del(&etx_cdev);
unregister_chrdev_region(dev, 1);
printk(KERN_INFO "Device Driver Remove...Done!!!\n");
}

module_init(etx_driver_init);
module_exit(etx_driver_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("EmbeTronicX<embetronicx@gmail.com or admin@embetronicx.com>");
MODULE_DESCRIPTION("A simple device driver");
MODULE_VERSION("1.3");
```

1. Build the driver by using Makefile (*sudo make*)
2. Load the driver using *sudo insmod*
3. Do echo 1 > /dev/etx_device

Echo will open the driver and write 1 into the driver and finally close the driver. So if i do echo to our driver, it should call the open, write and release functions. Just check.

linux@embetronicx-VirtualBox:/home/driver/driver# **echo 1 > /dev/etx_device**

4. Now Check using dmesg

linux@embetronicx-VirtualBox:*/home/driver/driver$* ***dmesg***

*[19721.611967] Major = 246 Minor = 0*
*[19721.618716] Device Driver Insert...Done!!!*
*[19763.176347] Driver Open Function Called...!!!*
*[19763.176363] Driver Write Function Called...!!!*
*[19763.176369] Driver Release Function Called...!!!*

5. Do **cat > /dev/etx_device**

Cat command will open the driver, read the driver and close the driver. So if i do cat to our driver, it should call the open, read and release functions. Just check.

linux@embetronicx-VirtualBox:/home/driver/driver# **cat /dev/etx_device**

6. Now Check using dmesg

linux@embetronicx-VirtualBox:*/home/driver/driver$* ***dmesg***

*[19763.176347] Driver Open Function Called...!!!*
*[19763.176363] Driver Read Function Called...!!!*
*[19763.176369] Driver Release Function Called...!!!*

7. Unload the driver using *sudo rmmod*

Instead of doing echo and cat command in terminal you can also use open(), read(), write(), close() system calls from user space application.