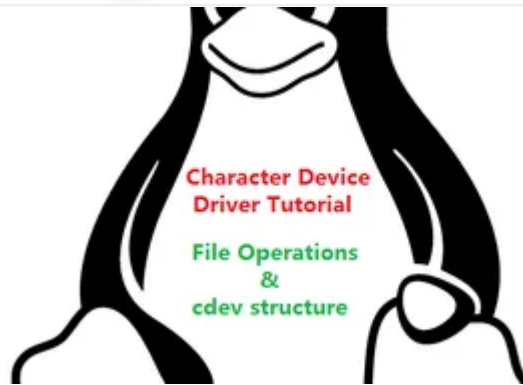


[Sidebar](#) ▼

[Home](#) → [Tutorials](#) → [Linux](#) → [Device Drivers](#) → **Linux Device Driver Tutorial Part 6 - Cdev structure and File Operations**

## 📁 [Device Drivers](#)



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

10

This article is a continuation of the [Series on Linux Device Driver](#) and carries on the discussion on character drivers and their implementation. This is Part 6 of the Linux device driver tutorial. In this tutorial, we will discuss Cdev structure and File Operations of Character drivers.

Apple Music Turns 5 as It Continues Rivalry With Spotify

Based on our previous tutorial, we know about the major, minor number, and device file. So as I said in the [previous tutorial](#), we need to open, read, write, and close the device file. So let's start...

#### Post Contents [\[hide\]](#)

### 1 Cdev structure and File Operations of Character drivers

#### 1.1 cdev structure

#### 1.2 File\_Operations

##### 1.2.1 read

##### 1.2.2 write

##### 1.2.3 ioctl

##### 1.2.4 Open

##### 1.2.5 release (close)

##### 1.2.6 Example

### 2 Dummy Driver

#### 2.0.1 Share this:

#### 2.0.2 Like this:

#### 10 2.0.3 Related

## Cdev structure and File Operations of Character drivers

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

structures to the driver.

## cdev structure

In Linux kernel **struct inode** 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, **cdev** structure is useful for writing driver code:

**struct cdev** 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 **struct cdev** 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.

```
1 struct cdev {  
2     struct kobject kobj;  
3     struct module *owner;  
4     const struct file_operations *ops;  
5     struct list_head list;  
6     dev_t dev;  
7     unsigned int count;  
8 };
```

This is **cdev** structure. Here we need to fill the 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:

```
struct cdev *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(struct cdev *cdev, struct file_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:

```
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
```

Where,

**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:

10

```
void cdev_del(struct cdev *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.

```

1  struct file_operations {
2      struct module *owner;
3      loff_t (*llseek) (struct file *, loff_t, int);
4      ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
5      ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
6      ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
7      ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
8      int (*iterate) (struct file *, struct dir_context *);
9      int (*iterate_shared) (struct file *, struct dir_context *);
10     unsigned int (*poll) (struct file *, struct poll_table_struct *);
11     long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
12     long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
13     int (*mmap) (struct file *, struct vm_area_struct *);
14     int (*open) (struct inode *, struct file *);
15     int (*flush) (struct file *, fl_owner_t id);
16     int (*release) (struct inode *, struct file *);
17     int (*fsync) (struct file *, loff_t, loff_t, int datasync);
18     int (*fasync) (int, struct file *, int);
19     int (*lock) (struct file *, int, struct file_lock *);
20     ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
21     unsigned long (*get_unmapped_area) (struct file *, unsigned long, unsigned long,
22     int (*check_flags) (int);
23     int (*flock) (struct file *, int, struct file_lock *);
24     ssize_t (*splice_write) (struct pipe_inode_info *, struct file *, loff_t *, size_t,
25     ssize_t (*splice_read) (struct file *, loff_t *, struct pipe_inode_info *, size_t,
26     int (*setlease) (struct file *, long, struct file_lock **, void **);
27     long (*fallocate) (struct file *file, int mode, loff_t offset,
28         loff_t len);
29     void (*show_fdinfo) (struct seq_file *m, struct file *f);
30     #ifndef CONFIG_MMU
31         unsigned (*mmap_capabilities) (struct file *);
32     #endif
33     ssize_t (*copy_file_range) (struct file *, loff_t, struct file *,
34         loff_t, size_t, unsigned int);
3510    int (*clone_file_range) (struct file *, loff_t, struct file *, loff_t,
36        u64);
37    ssize_t (*dedupe_file_range) (struct file *, u64, u64, struct file *,
38        u64);
39 };

```

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>**.

## **read**

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

This is 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).

## **write**

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

It is used to send the 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.

## **ioctl**

```
int (*ioctl) (struct inode *, struct file *, unsigned int,
10            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"). You can find the IOCTL tutorial [here](#).

## Open

```
int (*open) (struct inode *, 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.

## release (close)

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

This operation is invoked when the file structure is being released. Like open, release can be NULL.

## Example

```
1 static struct file_operations fops =
2 {
3   .owner          = THIS_MODULE,
4   .read           = etx_read,
5   .write          = etx_write,
6   .open           = etx_open,
7   .release        = etx_release,
8 };
```

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.

```

1  #include <linux/kernel.h>
2  #include <linux/init.h>
3  #include <linux/module.h>
4  #include <linux/kdev_t.h>
5  #include <linux/fs.h>
6  #include <linux/cdev.h>
7  #include <linux/device.h>
8
9  dev_t dev = 0;
10 static struct class *dev_class;
11 static struct cdev etx_cdev;
12
13 static int __init etx_driver_init(void);
14 static void __exit etx_driver_exit(void);
15 static int etx_open(struct inode *inode, struct file *file);
16 static int etx_release(struct inode *inode, struct file *file);
17 static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t * of
18 static ssize_t etx_write(struct file *filp, const char *buf, size_t len, loff_t * o
19
20 static struct file_operations fops =
21 {
22     .owner          = THIS_MODULE,
23     .read           = etx_read,
24     .write          = etx_write,
25     .open           = etx_open,
26     .release        = etx_release,
27 };
28
29 static int etx_open(struct inode *inode, struct file *file)
30 {
31     printk(KERN_INFO "Driver Open Function Called...!!!\n");
32     return 0;
33 }
34
35 static int etx_release(struct inode *inode, struct file *file)
36 {
37     printk(KERN_INFO "Driver Release Function Called...!!!\n");
38     return 0;
39 }
40
41 static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t *of
42 {
43     printk(KERN_INFO "Driver Read Function Called...!!!\n");
44     return 0;
45 }
46 static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, lof
47 {
48     printk(KERN_INFO "Driver Write Function Called...!!!\n");
49     return len;
50 }
51
52
53 static int __init etx_driver_init(void)
54 {
55     /*Allocating Major number*/
56     if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) < 0){
57         printk(KERN_INFO "Cannot allocate major number\n");
58         return -1;
59     }
60     printk(KERN_INFO "Major = %d Minor = %d \n", MAJOR(dev), MINOR(dev));
61
62     /*Creating cdev structure*/
63     cdev_init(&etx_cdev, &fops);

```



```

64
65     /*Adding character device to the system*/
66     if((cdev_add(&etx_cdev,dev,1)) < 0){
67         printk(KERN_INFO "Cannot add the device to the system\n");
68         goto r_class;
69     }
70
71     /*Creating struct class*/
72     if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
73         printk(KERN_INFO "Cannot create the struct class\n");
74         goto r_class;
75     }
76
77     /*Creating device*/
78     if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
79         printk(KERN_INFO "Cannot create the Device 1\n");
80         goto r_device;
81     }
82     printk(KERN_INFO "Device Driver Insert...Done!!!\n");
83     return 0;
84
85 r_device:
86     class_destroy(dev_class);
87 r_class:
88     unregister_chrdev_region(dev,1);
89     return -1;
90 }
91
92 void __exit etx_driver_exit(void)
93 {
94     device_destroy(dev_class,dev);
95     class_destroy(dev_class);
96     cdev_del(&etx_cdev);
97     unregister_chrdev_region(dev, 1);
98     printk(KERN_INFO "Device Driver Remove...Done!!!\n");
99 }
100
101 module_init(etx_driver_init);
102 module_exit(etx_driver_exit);
103
104 MODULE_LICENSE("GPL");
105 MODULE_AUTHOR("EmbeTronicX <embetronicx@gmail.com or admin@embetronicx.com>");
106 MODULE_DESCRIPTION("A simple device driver");
107 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.

```
1 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.

```
1 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 the terminal you can also use **open()**, **read()**, **write()**, **close()** system calls from user-space applications.

I hope you understood this tutorial. This is just a dummy driver tutorial. In our [next tutorial](#), we will see some real-time applications using file operations device drivers. 😊

5

Article Rating

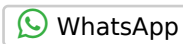
**Share this:**

Post

Tweet



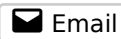
Print



WhatsApp

Share

2



Email



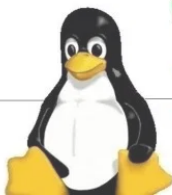
Telegram



More

**Like this:**

Loading...

**Related** EmbeTronicX  
Embedded Tutorials Zone

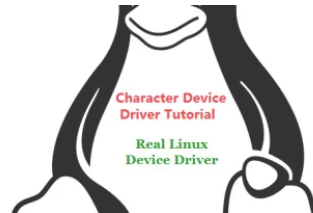
Misc Device Driver

Linux Device Driver  
Tutorial - Part 32[Linux Device Driver  
Tutorial Part 32 – Misc  
Device Driver](#)

In "Device Drivers"

[Linux Device Driver  
Tutorial Part 11 – Sysfs in  
Linux Kernel](#)

In "Device Drivers"

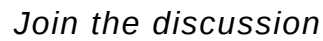
Character Device  
Driver Tutorial  
Real Linux  
Device Driver[Linux Device Driver  
Tutorial Part 7 - Linux  
Device Driver Tutorial  
Programming](#)

In "Device Drivers"

10

 <sup>10</sup>Subscribe ▼

Connect with | [Login](#)




## 10 COMMENTS



August 21, 2017 1:37 AM

Loading...



April 6, 2018 3:37 AM

Loading...



10



srishti verma

October 18, 2018 4:48 AM

After giving the command dmesg. I could see the message "module inserted" but I am not getting the message when driver got open, read and released. I had given the command echo 1 /dev/etx\_device before dmesg.

Loading...



0



Reply

EmbeTronicx India

 Reply to [srishti verma](#)

October 22, 2018 2:43 AM

Hi Srishti,

Have you changed the source code?

Thanks.

Loading...



0



Reply



ganesh sastry

April 1, 2019 10:19 PM

how to create a virtual device drive to copy the files from one vm to other vm

Loading...

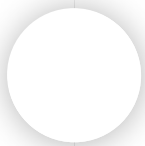


0



Reply

10





viyyapu Durga Prasad

July 16, 2019 1:04 PM

Hi EmbetroniX,

Your tutorials are very precise and easy to understand. I tried the dummy driver. When I write to the device, it is writing to driver but when check logs they are not printed. please help me on this

Thanks

Loading...



0



Reply



Martin Pålsson

August 26, 2019 1:24 PM

For everyone that has trouble with permission to write, please read the following post on stackoverflow:

<https://stackoverflow.com/questions/34522426/unable-to-write-on-dev-files>

TLDR:

sudo echo 1 > /dev/etx\_device does not do what you think it would.

rather, get the root prompt by

sudo su

and echo to file

echo 1 > /dev/etx\_device

Allt the best

Martin

10

Loading...



0



Reply



AtlaskD

November 24, 2019 12:00 PM

Hi EmbetronicX,  
why did you use **static** keyword for most of the functions and variables except for **etx\_driver\_exit()** and **dev**?  
Thanks

Loading...



0



Reply

EmbeTronicX

Reply to [AtlaskD](#)

November 24, 2019 12:51 PM

Hi AtlaskD,

Its your wish to use static keyword before any variable and functions. But u should know how static keyword works.

Static keyword limit the visibility to that file. So you cannot access that function or variable from outside of the file (i.e) you cannot access it from other files.

Here my intention to write a driver with multiple files. So all the files will have same init and exit function. Other function's scope is limited to only this file. I have to remove that static keyword from init function.

Hope this would answered your question.

Loading...



0



Reply

10







Divin

June 3, 2020 11:52 AM

Hi,  
Perfect attempt!

Could you please explain what are the use cases and advantages of using runtime allocation and own allocation?

Loading...



0



Reply

report this ad

10

