# Part 5 – Device File Creation

[ <https://embetronicx.com/tutorials/linux/device-drivers/device-file-creation-for-character-drivers/> ]

In this tutorial we will discuss Device File Creation for Character Drivers.

# Device File Creation for Character Drivers

Device file is important to communicate to hardware. Let’s start our tutorial.

Device files are in /dev/ directory.

# Device Files

Device file allows transparent communication between user space applications and hardware.

They are not normal “files”, but look like files from the program’s point of view: you can read from them, write to them, *mmap()* onto them, and so forth. When you access such a device “file,” the kernel recognizes the I/O request and passes it a device driver, which performs some operation, such as reading data from a serial port, or sending data to a hardware.

Device files (although inappropriately named, we will continue to use this term) provide a convenient way to access system resources without requiring the applications programmer to know how the underlying device works. Under Linux, as with most Unix systems, device drivers themselves are part of the kernel.

All device files are stored in /dev directory. Use ls command to browse the directory:

$ ls -l /dev/

Each device on the system should have a corresponding entry in */dev*. For example, */dev/ttyS0*corresponds to the first serial port, known as COM1 under MS-DOS; */dev/hda2* corresponds to the second partition on the first IDE drive. In fact, there should be entries in */dev* for devices you do not have. The device files are generally created during system installation and include every possible device driver. They don’t necessarily correspond to the actual hardware on your system. There are a number of pseudo-devices in */dev* that don’t correspond to any actual peripheral. For example, */dev/null* acts as a byte sink; any write request to */dev/null* will succeed, but the data written will be ignored.

When using ls -l to list device files in */dev*, you’ll see something like the following:

**c**rw--w---- 1 root tty **4**, **0** Aug 15 10:40 tty0

**b**rw-rw---- 1 root disk **1**, **0** Aug 15 10:40 ram0

First of all, note that the first letter of the permissions field is denoted that driver type. Device files are denoted either by b, for block devices, or c, for character devices.

Also, note that the size field in the ls -l listing is replaced by two numbers, separated by a comma. The first value is the major device number and the second is the minor device number. This we have discussed in previous [tutorial](https://www.embetronicx.com/tutorials/linux/device-drivers/character-device-driver-major-number-and-minor-number/).

# Creating Device File

We can create the dive file in two ways.

1. Manually
2. Automatically

We will see one by one.

## Manually Creating Device File

We can create the device file manually by using mknod.

mknod -m <*permissions>* <*name>* <*device* *type>* <*major>* <*minor>*

*<name>* – your device file name that should have full path (/dev/name)

*<device type>* – Put **c** or **b**

c – Character Device

b – Block Device

*<major>* – major number of your driver

*<minor>* – minor number of your driver

-m <*permissions> –*optional argument that sets the permission bits of the new device file to *permissions*

**Example:**

$ sudo mknod -m 666 /dev/etx\_device c 246 0

If you don’t want to give permission, You can also use chmod to set the permissions for a device file after creation.

### Advantages of Manually creating Device Node

* Anyone can create the device file using this method.
* You can create the device file even before load the driver.

## Automatically Creating Device File

The automatic creation of device files can be handled with udev. Udev is the device manager for the Linux kernel that creates/removes device nodes in the /dev directory dynamically. Just follow the below steps.

1. Include the header file **linux/device.h** and **linux/kdev\_t.h**
2. Create the struct Class
3. Create Device with the class which is created by above step

### Create the class

This will create the struct class for our device driver. It will create structure under/sys/class/.

struct class \* class\_create (struct module \*owner, const char \*name);

*owner –*pointer to the module that is to “own” this struct class.

*name –*pointer to a string for the name of this class.

This is used to create a struct class pointer that can then be used in calls to class\_device\_create().

**Note,** the pointer created here is to be destroyed when finished by making a call to class\_destroy().

void class\_destroy (struct class \* cls);

*Where,*

*class –*pointer to the struct class that this device should be registered to

*parent –*pointer to the parent struct device of this new device, if any

*devt –*the dev\_t for the char device to be added

*fmt –*string for the device’s name

*… –*variable arguments

A “dev” file will be created, showing the dev\_t for the device, if the dev\_t is not 0,0. If a pointer to a parent struct device is passed in, the newly created struct device will be a child of that device in sysfs. The pointer to the struct device will be returned from the call. Any further sysfs files that might be required can be created using this pointer.

**Note,**you can destroy the device using device\_destroy().

void device\_destroy (struct class \* class, dev\_t devt);

### Programming

#include <linux/kernel.h>

#include <linux/init.h>

#include <linux/module.h>

#include <linux/kdev\_t.h>

#include <linux/fs.h>

#include <linux/device.h>

dev\_t dev = 0;

static struct class \*dev\_class;

static int \_\_init hello\_world\_init(void)

{

/\*Allocating Major number\*/

if((alloc\_chrdev\_region(&dev, 0, 1, "etx\_Dev")) <0){

printk(KERN\_INFO "Cannot allocate major number for device\n");

return -1;

}

printk(KERN\_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));

/\*Creating struct class\*/

if((dev\_class = class\_create(THIS\_MODULE,"etx\_class")) == NULL){

printk(KERN\_INFO "Cannot create the struct class for device\n");

goto r\_class;

}

/\*Creating device\*/

if((device\_create(dev\_class,NULL,dev,NULL,"etx\_device")) == NULL){

printk(KERN\_INFO "Cannot create the Device\n");

goto r\_device;

}

printk(KERN\_INFO "Kernel Module Inserted Successfully...\n");

return 0;

r\_device:

class\_destroy(dev\_class);

r\_class:

unregister\_chrdev\_region(dev,1);

return -1;

}

void \_\_exit hello\_world\_exit(void)

{

device\_destroy(dev\_class,dev);

class\_destroy(dev\_class);

unregister\_chrdev\_region(dev, 1);

printk(KERN\_INFO "Kernel Module Removed Successfully...\n");

}

module\_init(hello\_world\_init);

module\_exit(hello\_world\_exit);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("EmbeTronicX <embetronicx@gmail.com or admin@embetronicx.com>");

MODULE\_DESCRIPTION("A simple hello world driver");

MODULE\_VERSION("1.2");

* Build the driver by using Makefile (sudo make)
* Load the driver using *sudo insmod*
* Check the device file using ls -l /dev/ | grep "etx\_device"

$ ls -l /dev/ | grep "etx\_device"

crw------- 1  root  root  246, 0  Aug 15  13:36  etx\_device

* Unload the driver using *sudo rmmod*

By this time we have created the device file using those methods mentioned above. So Using this device file we can communicate the hardware. In our next tutorial we will show how to open that file, how to read that device file, how to write device file and how to close device file.