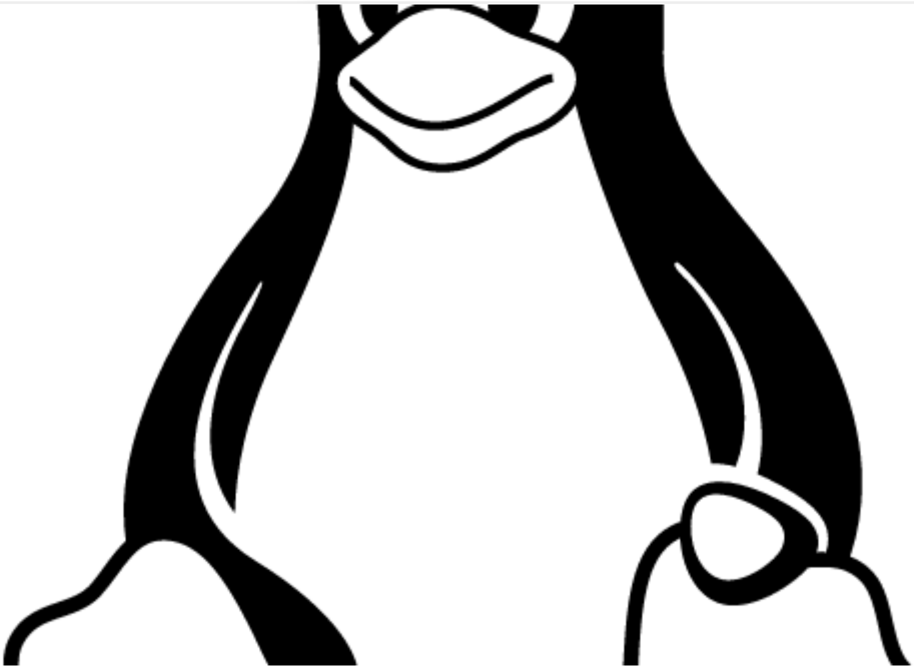
EmbeTronicX
Embedded Tutorials Zone

Sidebar▼

[Home](#) → [Tutorials](#) → [Linux](#) → [Device Drivers](#) → **Linux Device Driver Tutorial Part 11 – Sysfs in Linux Kernel**

Device Drivers



Linux Device Driver Tutorial Part 11 – Sysfs in Linux Kernel

11

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 11 of the Linux device driver tutorial. In our previous tutorial, we have seen the Procfs. Now we will see SysFS in Linux kernel Tutorial.

11

Post Contents [hide]

Introduction

The operating system segregates virtual memory into kernel space and userspace. Kernel space is strictly reserved for running the kernel, kernel extensions, and most device drivers. In contrast, user space is the memory area where all user-mode applications work, and this memory can be swapped out when necessary. There are many ways to Communicate between the Userspace and Kernel Space, they are:

- IOCTL
- Procfs
- Sysfs
- Configfs
- Debugfs
- Sysctl
- UDP Sockets
- Netlink Sockets

In this tutorial, we will see Sysfs.

SysFS in Linux Kernel Tutorial

Introduction

Sysfs is a virtual filesystem exported by the kernel, similar to `/proc`. The files in Sysfs contain information about devices and drivers. Some files in Sysfs are even writable, for configuration and control of devices attached to the system. Sysfs is always mounted on `/sys`.

The directories in Sysfs contain the hierarchy of devices, as they are attached to the computer.

Sysfs is the commonly used method to export system information from the kernel space to the user space for specific devices. The sysfs is tied to the device driver model of the kernel. The procfs is used to export the process-specific information and the debugfs is used to use for exporting the debug information by the developer.

Before getting into the sysfs we should know about the Kernel Objects.

Kernel Objects

The heart of the sysfs model is the **kobject**. **Kobject** is the glue that binds the sysfs and the kernel, which is represented by **`struct kobject`** and defined in **`<linux/kobject.h>`**. A **`struct kobject`** represents a kernel object, maybe a device or so, such as the things that show up as directory in the `sysfs` filesystem.

Kobjects are usually embedded in other structures.

It is defined as,

```
#define KOBJ_NAME_LEN   20

struct kobject {
        char                    *k_name;
        char                    name[KOBJ_NAME_LEN];
        struct kref             kref;
        struct list_head        entry;
        struct kobject          *parent;
        struct kset             *kset;
        struct kobj_type        *ktype;
        struct dentry           *dentry;
};
```

Some of the important fields are:

`struct kobject`
|– **`name`** (Name of the kobject. Current kobject is created with this name in *sysfs.*)
|– **`parent`** (This is kobject's parent. When we create a directory in sysfs for current kobject, it will create under this parent directory)
|– **`ktype`** (the type associated with a kobject)
|– **`kset`** (a group of kobjects all of which are embedded in structures of the same type)
|– **`sd`** (points to a sysfs_dirent structure that represents this kobject in sysfs.)

|– **kref** (provides reference counting)

It is the glue that holds much of the device model and its sysfs interface together.

So Kobj is used to create kobject directory in **/sys**. This is enough. We will not go deep into the kobjects.

SysFS in Linux

There are several steps to creating and using sysfs.

1. Create a directory in **/sys**
2. Create Sysfs file

Create a directory in /sys

We can use this function (**kobject_create_and_add**) to create directory.

```
struct kobject * kobject_create_and_add ( const char * name, struct kobject * parent)
```

Where,

<**name**> – the name for the kobject

<**parent**> – the parent kobject of this kobject, if any.

If you pass **kernel_kobj** to the second argument, it will create the directory under /sys/kernel/. If you pass **firmware_kobj** to the second argument, it will create the directory under /sys/firmware/. If you pass **fs_kobj** to the second argument, it will create the directory under **/sys/fs/**. If you pass NULL to the second argument, it will create the directory under /sys/.

This function creates a kobject structure dynamically and registers it with sysfs. If the kobject was not able to be created, NULL will be returned.

When you are finished with this structure, call **kobject_put** and the structure will be dynamically freed when it is no longer being used.

Example

```
struct kobject *kobj_ref;

/*Creating a directory in /sys/kernel/ */
kobj_ref = kobject_create_and_add("etx_sysfs",kernel_kobj); //sys/kernel/etx_sysfs

/*Freeing Kobj*/
kobject_put(kobj_ref);
```

Create Sysfs file

Using the above function we will create a directory in **/sys**. Now we need to create `sysfs` file, which is used to interact user space with kernel space through sysfs. So we can create the sysfs file using sysfs attributes.

Attributes are represented as regular files in sysfs with one value per file. There are loads of helper functions that can be used to create the kobject attributes. They can be found in the header file `sysfs.h`

Create attribute

Kobj_attribute is defined as,

```
struct kobj_attribute {
    struct attribute attr;
    ssize_t (*show)(struct kobject *kobj, struct kobj_attribute *attr, char *buf);
    ssize_t (*store)(struct kobject *kobj, struct kobj_attribute *attr, const char *b
};
```

Where,

`attr` – the attribute representing the file to be created,

`show` – the pointer to the function that will be called when the file is read in *sysfs*,

`store` – the pointer to the function which will be called when the file is written in *sysfs.*

We can create an attribute using __ATTR macro.

`__ATTR(name, permission, show_ptr, store_ptr);`

Store and Show functions

Then we need to write show and store functions.

```
ssize_t (*show)(struct kobject *kobj, struct kobj_attribute *attr, char *buf);
ssize_t (*store)(struct kobject *kobj, struct kobj_attribute *attr, const char *buf,
```

Store function will be called whenever we are writing something to the sysfs attribute. See the example.

Show function will be called whenever we are reading sysfs attribute. See the example.

Create sysfs file

To create a single file attribute we are going to use '**sysfs_create_file**'.

```
int sysfs_create_file ( struct kobject *  kobj, const struct attribute * attr);
```

Where,

kobj *– object we're creating for.*

attr – attribute descriptor.

One can use another function ' **sysfs_create_group** ' to create a group of attributes.

Once you have done with sysfs file, you should delete this file using sysfs_remove_file

```
void sysfs_remove_file ( struct kobject *  kobj, const struct attribute * attr);
```

Where,

kobj *– object we're creating for.*

attr – attribute descriptor.

Example

```
struct kobj_attribute etx_attr = __ATTR(etx_value, 0660, sysfs_show, sysfs_store);

static ssize_t sysfs_show(struct kobject *kobj,
                struct kobj_attribute *attr, char *buf)
{
    return sprintf(buf, "%d", etx_value);
}

static ssize_t sysfs_store(struct kobject *kobj,
                struct kobj_attribute *attr,const char *buf, size_t count)
{
        sscanf(buf,"%d",&etx_value);
        return count;
}

//This Function will be called from Init function
/*Creating a directory in /sys/kernel/ */
kobj_ref = kobject_create_and_add("etx_sysfs",kernel_kobj);

/*Creating sysfs file for etx_value*/
if(sysfs_create_file(kobj_ref,&etx_attr.attr)){
    printk(KERN_INFO"Cannot create sysfs file......\n");
    goto r_sysfs;
}
//This should be called from exit function
kobject_put(kobj_ref);
sysfs_remove_file(kernel_kobj, &etx_attr.attr);
```

Now we will see the complete driver code. Try this code.

Complete Driver Code

In this driver, I have created one integer variable (**etx_value**). The initial value of that variable is 0. Using sysfs, I can read and modify that variable.

```
#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>
#include<linux/slab.h>                 //kmalloc()
#include<linux/uaccess.h>              //copy_to/from_user()
#include<linux/sysfs.h>
#include<linux/kobject.h>


volatile int etx_value = 0;


dev_t dev = 0;
static struct class *dev_class;
static struct cdev etx_cdev;
struct kobject *kobj_ref;

static int __init etx_driver_init(void);
static void __exit etx_driver_exit(void);

/*************** Driver Fuctions **********************/
static int etx_open(struct inode *inode, struct file *file);
static int etx_release(struct inode *inode, struct file *file);
static ssize_t etx_read(struct file *filp,
                char __user *buf, size_t len,loff_t * off);
static ssize_t etx_write(struct file *filp,
                const char *buf, size_t len, loff_t * off);

/*************** Sysfs Fuctions **********************/
static ssize_t sysfs_show(struct kobject *kobj,
                struct kobj_attribute *attr, char *buf);
static ssize_t sysfs_store(struct kobject *kobj,
                struct kobj_attribute *attr,const char *buf, size_t count);

struct kobj_attribute etx_attr = __ATTR(etx_value, 0660, sysfs_show, sysfs_store);

static struct file_operations fops =
{
        .owner          = THIS_MODULE,
        .read           = etx_read,
        .write          = etx_write,
        .open           = etx_open,
        .release        = etx_release,
};

static ssize_t sysfs_show(struct kobject *kobj,
                struct kobj_attribute *attr, char *buf)
{
        printk(KERN_INFO "Sysfs - Read!!!\n");
        return sprintf(buf, "%d", etx_value);
}

static ssize_t sysfs_store(struct kobject *kobj,
                struct kobj_attribute *attr,const char *buf, size_t count)
{
        printk(KERN_INFO "Sysfs - Write!!!\n");
        sscanf(buf,"%d",&etx_value);
        return count;
}

static int etx_open(struct inode *inode, struct file *file)
{
        printk(KERN_INFO "Device File Opened...!!!\n");
        return 0;
}

static int etx_release(struct inode *inode, struct file *file)
{
        printk(KERN_INFO "Device File Closed...!!!\n");
        return 0;
}

static ssize_t etx_read(struct file *filp,
                char __user *buf, size_t len, loff_t *off)
{
        printk(KERN_INFO "Read function\n");
        return 0;
}
static ssize_t etx_write(struct file *filp,
                const char __user *buf, size_t len, loff_t *off)
{
        printk(KERN_INFO "Write Function\n");
        return 0;
}


static int __init etx_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));

        /*Creating cdev 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");
            goto r_class;
        }

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

        /*Creating device*/
        if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
            printk(KERN_INFO "Cannot create the Device 1\n");
            goto r_device;
        }

        /*Creating a directory in /sys/kernel/ */
        kobj_ref = kobject_create_and_add("etx_sysfs",kernel_kobj);

        /*Creating sysfs file for etx_value*/
        if(sysfs_create_file(kobj_ref,&etx_attr.attr)){
                printk(KERN_INFO"Cannot create sysfs file......\n");
                goto r_sysfs;
    }
        printk(KERN_INFO "Device Driver Insert...Done!!!\n");
    return 0;

r_sysfs:
        kobject_put(kobj_ref);
        sysfs_remove_file(kernel_kobj, &etx_attr.attr);

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

void __exit etx_driver_exit(void)
{
        kobject_put(kobj_ref);
        sysfs_remove_file(kernel_kobj, &etx_attr.attr);
        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 - SysFs");
MODULE_VERSION("1.8");
```

MakeFile

```
obj-m += driver.o

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


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

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

Building and Testing Driver

- Build the driver by using Makefile (***sudo make***)
- Load the driver using **`sudo insmod driver.ko`**
- Check the directory in /sys/kernel/ using **`ls -l /sys/kernel`**

```
linux@embetronicx-VirtualBox: ls -l /sys/kernel/

drwxr-xr-x 2 root root 0 Dec 17 14:11 boot_params
drwx------ 26 root root 0 Dec 17 12:19 debug
```

```
drwxr-xr-x 2 root root 0 Dec 17 16:29 etx_sysfs
drwxr-xr-x 2 root root 0 Dec 17 14:11 fscache
-r--r--r-- 1 root root 4096 Dec 17 14:11 fscaps
drwxr-xr-x 2 root root 0 Dec 17 14:11 iommu_groups
-r--r--r-- 1 root root 4096 Dec 17 14:11 kexec_crash_loaded
-rw-r--r-- 1 root root 4096 Dec 17 14:11 kexec_crash_size
-r--r--r-- 1 root root 4096 Dec 17 14:11 kexec_loaded
drwxr-xr-x 2 root root 0 Dec 17 14:11 livepatch
drwxr-xr-x 6 root root 0 Dec 17 14:11 mm
-r--r--r-- 1 root root 516 Dec 17 14:11 notes
-rw-r--r-- 1 root root 4096 Dec 17 14:11 profiling
-rw-r--r-- 1 root root 4096 Dec 17 14:11 rcu_expedited
drwxr-xr-x 4 root root 0 Dec 17 12:19 security
drwxr-xr-x 117 root root 0 Dec 17 12:19 slab
dr-xr-xr-x 2 root root 0 Dec 17 14:11 tracing
-rw-r--r-- 1 root root 4096 Dec 17 12:19 uevent_helper
-r--r--r-- 1 root root 4096 Dec 17 12:19 uevent_seqnum
-r--r--r-- 1 root root 4096 Dec 17 14:11 vmcoreinfo
```

- Now our sysfs entry is there under `/sys/kernel` directory.
- Now check sysfs file in etx_sysfs using `ls -l /sys/kernel/etx_sysfs`

```
linux@embetronicx-VirtualBox: ls -l /sys/kernel/etx_sysfs
-rw-rw----    1    root    root    4096    Dec 17 16:37    etx_value
```

- Our sysfs file also there. Now go under root permission using `sudo su`.
- Now read that file using `cat /sys/kernel/etx_sysfs/etx_value`

```
linux@embetronicx-VirtualBox#cat /sys/kernel/etx_sysfs/etx_value

0
```

- So Value is 0 (initial value is 0). Now, modify the value using the echo command.

```
linux@embetronicx-VirtualBox#echo 123 > /sys/kernel/etx_sysfs/etx_value
```

- Now again read that file using `cat /sys/kernel/etx_sysfs/etx_value`

```
linux@embetronicx-VirtualBox#cat /sys/kernel/etx_sysfs/etx_value

123
```

So our sysfs is working fine.

- Unload the module using `sudo rmmod driver`

This is a simple example using sysfs in the device drivers. This is just basic. I hope this might helped you. In our next tutorial, we will discuss interrupts in Linux.

5

Article Rating

★★★★★

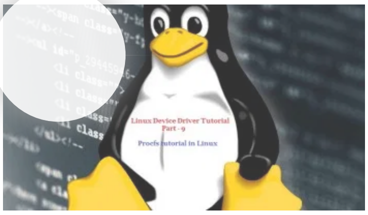
Share this:

Share 5 Post Tweet SHARE Print WhatsApp Share 5 Email Telegram More

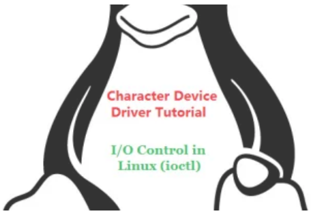
Like this:

Loading...

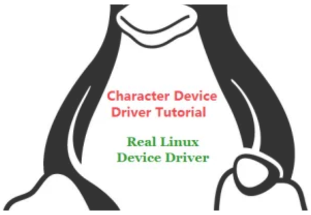
Related



Linux Device Driver


Character Device Driver Tutorial
I/O Control in Linux (ioctl)

Linux Device Driver



Linux Device Driver

Tutorial Part 9 - Procfs in Linux
In "Device Drivers"

Tutorial Part 8 – I/O Control in Linux IOCTL()
In "Device Drivers"

Tutorial Part 7 - Linux Device Driver Tutorial Programming
In "Device Drivers"

✉ Subscribe ▾

Connect with | Login

11

Join the discussion

B *I* U ~~S~~ {} [+]

This site uses Akismet to reduce spam. Learn how your comment data is processed.

11 COMMENTS

Oldest

Anonymous

February 1, 2018 4:51 PM

Can you guys pls do a post on commonly used data structures in linux kernel programming

Loading...

👍 0 👎 Reply

Murali Krishna

May 3, 2018 1:18 AM

Hi
Lot of useful Information,
I want to create a Directory in /sys/class instead of /sys/. Please can you guide me how to do this.

Loading...

👍 0 👎 Reply

Sana Srikar

August 27, 2018 8:47 PM

hi embetronicx ,
i have a doubt in this post.
Why using Kobject_create_and_add api isnt device_create() sufficient?
Please correct me if i am wrong. What i have seen in the code is that
device_create() is creating a struct device which will in have a kobject in it and the same api, device_create() will call kobject_add() which will carry out the next activities.
So calling kobject_create_and_add() ,doesnt it again create one

11

more kobject and call the api kobject_add() with newly created kobject which is not our intended one.?
in simple why device_create() alone is not sufficient ?

Loading...

👍 0 👎 Reply

EmbeTronicx India

Reply to Sana Srikar August 28, 2018 2:29 AM

Hi Sana,
device_create() function can be used by char device classes. A struct device will be created in sysfs, registered to the specified class.

So Now, device name will appear in /sys/devices/virtual// and /dev/

We want to create custom sysfs entry in /sys/kernel/*. For that we are using kobject_create_and_add().

Loading...

👍 0 👎 Reply

Sana Srikar

Reply to EmbeTronicx India August 29, 2018 9:51 AM

hey Hi embetronicx !!!! Thanks for replying. I am following you tutorials for learning Linux. Some nice tutorials embetronicx. To create a custom sysfs entry, yes ,we have to first create a kobject and then add it which will be done by kobject_create_and add() . But i think device_create() is misleading the article. I ,at first sight, thought that to create sysfs entries for a device we have to be doing like this after a week of search in kernel code i got to know that device_Create() will do this(creating a kobject adding it into the sysfs) for us… Read more »

11

👍 0 👎 Reply

EmbeTronicx India

Reply to Sana Srikar August 30, 2018 4:21 AM

Hi Sana,

Yes sysfs is separate topic. But this is device driver series. Beginners will follow from the Part 1. That's why we are taking the previous example and implementing the concept.

Loading...

👍 0 👎 Reply

Sana Srikar

Reply to EmbeTronicx India August 30, 2018 9:26 AM

my only concern is that they might understand it wrongly... but if they understand somehow its fine.

Loading...

👍 0 👎 Reply

Sana Srikar

August 29, 2018 9:57 AM

i also request you guys to keep on adding tutorials like this .
Thank you
Sana Srikar

Loading...

👍 0 👎 Reply

11

eliaskousk

September 1, 2018 7:01 PM

The tutorials here are very good, thank you for writing them. I just want to add a correction to the above example code. In order for it to compile you need to make the buf pointer (char * buf) a const (const char * buf) in the sysfs_store function – it's omitted in both the prototype and actual body of the function. Thanks again for your effort.

Loading...

0 Reply

EmbeTronicx India

Reply to *eliaskousk* October 14, 2018 6:19 AM

Hi Eliaskousk,

Thanks for your input. We are appreciating you. Please support us.

Thank you.

Loading...

0 Reply

Cretingame

September 11, 2019 7:02 AM

Thanks a lot, excellent tutorial.

But the website autorefresh is so annoying I had to copy the page. It’s annoying to have to rescroll every 2 minutes !!!

Loading...

0 Reply

11

11