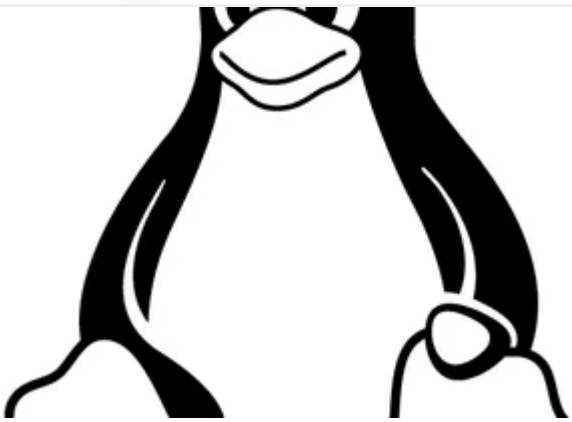
EmbeTronicX
Embedded Tutorials Zone

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

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,

X

Post Contents

Introduction

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`.

X

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,

```
1  #define KOBJ_NAME_LEN   20
2
3  struct kobject {
4          char                    *k_name;
5          char                    name[KOBJ_NAME_LEN];
6          struct kref             kref;
7          struct list_head        entry;
8          struct kobject          *parent;
9          struct kset             *kset;
```

X

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.

```
1 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.

X

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

```
1 struct kobject *kobj_ref;
2
3 /*Creating a directory in /sys/kernel/ */
4 kobj_ref = kobject_create_and_add("etx_sysfs",kernel_kobj); //sys/kernel/etx_sysfs
5
6 /*Freeing Kobj*/
7 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,

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

X

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.

```
1 ssize_t (*show)(struct kobject *kobj, struct kobj_attribute *attr, char *buf);
2 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**'.

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

Where,

kobj *– object we're creating for.*

attr – attribute descriptor.

X

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

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

Where,

kobj *– object we're creating for.*

attr – attribute descriptor.

Example

```
1  struct kobj_attribute etx_attr = __ATTR(etx_value, 0660, sysfs_show, sysfs_store);
2
3  static ssize_t sysfs_show(struct kobject *kobj,
4                  struct kobj_attribute *attr, char *buf)
5  {
6      return sprintf(buf, "%d", etx_value);
7  }
8
9  static ssize_t sysfs_store(struct kobject *kobj,
10                 struct kobj_attribute *attr,const char *buf, size_t count)
11 {
12         sscanf(buf,"%d",&etx_value);
13         return count;
14 }
15
16 //This Function will be called from Init function
```

X

```
23      goto r_sysfs;
24  }
25  //This should be called from exit function
26  kobject_put(kobj_ref);
27  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.

```
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   #include<linux/slab.h>                 //kmalloc()
9   #include<linux/uaccess.h>              //copy_to/from_user()
10  #include<linux/sysfs.h>
11  #include<linux/kobject.h>
12
13
14  volatile int etx_value = 0;
15
16
17  dev_t dev = 0;
18  static struct class *dev_class;
19  static struct cdev etx_cdev;
20  struct kobject *kobj_ref;
21
22  static int __init etx_driver_init(void);
23  static void __exit etx_driver_exit(void);
24
25  /*************** Driver Fuctions **********************/
26  static int etx_open(struct inode *inode, struct file *file);
27  static int etx_release(struct inode *inode, struct file *file);
28  static ssize_t etx_read(struct file *filp,
29                  char __user *buf, size_t len,loff_t * off);
30  static ssize_t etx_write(struct file *filp,
31                  const char *buf, size_t len, loff_t * off);
32
33  /*************** Sysfs Fuctions **********************/
34  static ssize_t sysfs_show(struct kobject *kobj,
35                  struct kobj_attribute *attr, char *buf);
36  static ssize_t sysfs_store(struct kobject *kobj,
37                  struct kobj_attribute *attr,const char *buf, size_t count);
```

X

```
44          .read           = etx_read,
45          .write          = etx_write,
46          .open           = etx_open,
47          .release        = etx_release,
48  };
49
50  static ssize_t sysfs_show(struct kobject *kobj,
51                  struct kobj_attribute *attr, char *buf)
52  {
53          printk(KERN_INFO "Sysfs - Read!!!\n");
54          return sprintf(buf, "%d", etx_value);
55  }
56
57  static ssize_t sysfs_store(struct kobject *kobj,
58                  struct kobj_attribute *attr,const char *buf, size_t count)
59  {
60          printk(KERN_INFO "Sysfs - Write!!!\n");
61          sscanf(buf,"%d",&etx_value);
62          return count;
63  }
64
65  static int etx_open(struct inode *inode, struct file *file)
66  {
67          printk(KERN_INFO "Device File Opened...!!!\n");
68          return 0;
69  }
70
71  static int etx_release(struct inode *inode, struct file *file)
72  {
73          printk(KERN_INFO "Device File Closed...!!!\n");
74          return 0;
75  }
76
77  static ssize_t etx_read(struct file *filp,
78                  char __user *buf, size_t len, loff_t *off)
79  {
80          printk(KERN_INFO "Read function\n");
81          return 0;
82  }
83  static ssize_t etx_write(struct file *filp,
84                  const char __user *buf, size_t len, loff_t *off)
85  {
86          printk(KERN_INFO "Write Function\n");
87          return 0;
88  }
89
90
91  static int __init etx_driver_init(void)
92  {
93          /*Allocating Major number*/
94          if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
95                  printk(KERN_INFO "Cannot allocate major number\n");
96                  return -1;
97          }
98          printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
99
```

X

```
107         }
108 
109         /*Creating struct class*/
110         if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
111             printk(KERN_INFO "Cannot create the struct class\n");
112             goto r_class;
113         }
114 
115         /*Creating device*/
116         if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
117             printk(KERN_INFO "Cannot create the Device 1\n");
118             goto r_device;
119         }
120 
121         /*Creating a directory in /sys/kernel/ */
122         kobj_ref = kobject_create_and_add("etx_sysfs",kernel_kobj);
123 
124         /*Creating sysfs file for etx_value*/
125         if(sysfs_create_file(kobj_ref,&etx_attr.attr)){
126                 printk(KERN_INFO"Cannot create sysfs file......\n");
127                 goto r_sysfs;
128     }
129         printk(KERN_INFO "Device Driver Insert...Done!!!\n");
130     return 0;
131 
132 r_sysfs:
133         kobject_put(kobj_ref);
134         sysfs_remove_file(kernel_kobj, &etx_attr.attr);
135 
136 r_device:
137         class_destroy(dev_class);
138 r_class:
139         unregister_chrdev_region(dev,1);
140         cdev_del(&etx_cdev);
141         return -1;
142 }
143 
144 void __exit etx_driver_exit(void)
145 {
146         kobject_put(kobj_ref);
147         sysfs_remove_file(kernel_kobj, &etx_attr.attr);
148         device_destroy(dev_class,dev);
149         class_destroy(dev_class);
150         cdev_del(&etx_cdev);
151         unregister_chrdev_region(dev, 1);
152         printk(KERN_INFO "Device Driver Remove...Done!!!\n");
153 }
154 
155 module_init(etx_driver_init);
156 module_exit(etx_driver_exit);
157 
158 MODULE_LICENSE("GPL");
159 MODULE_AUTHOR("EmbeTronicX <embetronicx@gmail.com or admin@embetronicx.com>");
160 MODULE_DESCRIPTION("A simple device driver - SysFs");
161 MODULE_VERSION("1.8");
```

X

```
3  KDIR = /lib/modules/$(shell uname -r)/build
4
5
6  all:
7      make -C $(KDIR)  M=$(shell pwd) modules
8
9  clean:
10     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**

```
1  linux@embetronicx-VirtualBox: ls -l /sys/kernel/
2
3  drwxr-xr-x 2 root root 0 Dec 17 14:11 boot_params
4  drwx------ 26 root root 0 Dec 17 12:19 debug
5  drwxr-xr-x 2 root root 0 Dec 17 16:29 etx_sysfs
6  drwxr-xr-x 2 root root 0 Dec 17 14:11 fscache
7  -r--r--r-- 1 root root 4096 Dec 17 14:11 fscaps
8  drwxr-xr-x 2 root root 0 Dec 17 14:11 iommu_groups
9  -r--r--r-- 1 root root 4096 Dec 17 14:11 kexec_crash_loaded
10 -rw-r--r-- 1 root root 4096 Dec 17 14:11 kexec_crash_size
11 -r--r--r-- 1 root root 4096 Dec 17 14:11 kexec_loaded
12 drwxr-xr-x 2 root root 0 Dec 17 14:11 livepatch
13 drwxr-xr-x 6 root root 0 Dec 17 14:11 mm
14 -r--r--r-- 1 root root 516 Dec 17 14:11 notes
15 -rw-r--r-- 1 root root 4096 Dec 17 14:11 profiling
16 -rw-r--r-- 1 root root 4096 Dec 17 14:11 rcu_expedited
17 drwxr-xr-x 4 root root 0 Dec 17 12:19 security
18 drwxr-xr-x 117 root root 0 Dec 17 12:19 slab
19 dr-xr-xr-x 2 root root 0 Dec 17 14:11 tracing
20 -rw-r--r-- 1 root root 4096 Dec 17 12:19 uevent_helper
21 -r--r--r-- 1 root root 4096 Dec 17 12:19 uevent_seqnum
22 -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**

X

```
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`**

X

0

Article Rating

Share this:

Like this:

Loading...

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

X

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 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...

X

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 »

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…

X

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

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...

X

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

X