# Part 11 – Sysfs in Linux Kernel

[ <https://embetronicx.com/tutorials/linux/device-drivers/sysfs-in-linux-kernel/> ]

Now we will see SysFS in Linux kernel Tutorial.

# Introduction

Operating system segregates virtual memory into kernel space and user space.  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 User space 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

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 get into the sysfs we should know about the Kernel Objects.

# Kernel Objects

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 are created with this name in sysfs.)  
|– **parent** (This iskobject’s parent. When we create a directory in sysfs for current kobject, it will create under this parent directory)  
|– **ktype** ( type associated with a kobject)  
|– **kset** (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 in creating and using sysfs.

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

## Create 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 created, NULL will be returned.

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 above function we will create 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 function that can be used to create the kobject attributes. They can be found in 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 \*buf, size\_t count);

};

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 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  2 | 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, size\_t count); |

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.

### Remove sysfs file

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 complete driver code. Try this code.

# Complete Driver Code

In this driver i have created one integer variable (etx\_value). 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
* Now our sysfs entry is there under /sys/kernel directory.
* Now check sysfs file in etx\_sysfs using ls -l /sys/kernel/etx\_sysfs

*$ 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

*#cat /sys/kernel/etx\_sysfs/etx\_value****0***

* So Value is 0 (initial value is 0). Now modify using echo command.

*#echo 123 > /sys/kernel/etx\_sysfs/etx\_value*

* Now again read that file using cat /sys/kernel/etx\_sysfs/etx\_value

*#cat /sys/kernel/etx\_sysfs/etx\_value*

***123***

* So our sysfs is working fine.
* Unload the module using sudo rmmod driver