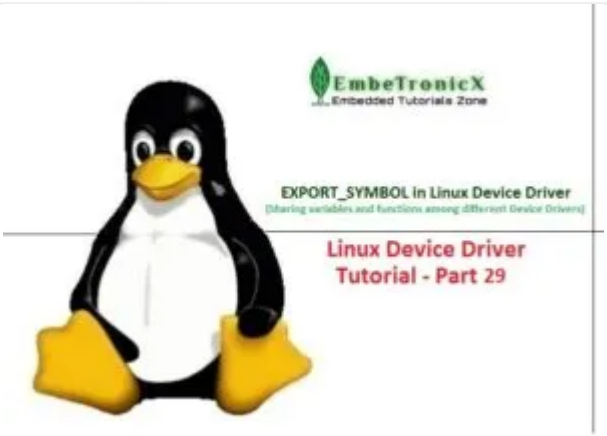
EmbeTronicX
Embedded Tutorials Zone

Sidebar

Home → Tutorials → Linux → Device Drivers → **Linux Device Driver Tutorial Part 29 – EXPORT_SYMBOL in Linux Device Driver**

Device Drivers



Linux Device Driver Tutorial Part 29 – EXPORT_SYMBOL in Linux Device Driver

This is the Series on Linux Device Driver. The aim of this series is to provide easy and practical examples that anyone can understand. This is the Linux Device Driver Tutorial Part 29 – EXPORT_SYMBOL in Linux Device Driver.

Note: In this article, I assume that you already know the basic routines for kernel module development.

When you are writing multiple drivers (modules) in the same device, you may want to use some of the functions form one module to another module. How will we do that? If we use only **`extern`** then it won't help you. We must have used some advanced things. So, We have to tell the kernel, that I want to share this function to other modules.

For example, take **`printk()`** function. This function will be defined in **`source/kernel/printk/printk.c`**. Then how can we able to access that **`printk()`** in our driver?

In this article we will see how to do it.

Post Contents [hide]

EXPORT_SYMBOL in Linux Device Driver

Introduction

In a programming language, a symbol is either a variable or a function. Or more generally, we can say, a symbol is a name representing space in the memory, which stores data (variable, for reading and writing) or instructions (function, for executing).

When you look at some kernel codes, you may find `EXPORT_SYMBOL()` very often. Have you wondered any time what the heck is that?

In the Linux Kernel 2.4, all the non-static symbols are exported to the kernel space automatically. But later, in Linux Kernel 2.6 instead of exporting all non-static symbols, they wanted to export the only symbols which are marked by `EXPORT_SYMBOL()` macro.

EXPORT_SYMBOL's role

When some symbols (variables or functions) are using `EXPORT_SYMBOL` macro (ex. `EXPORT_SYMBOL(func_name)`), those symbols are exposed to all the loadable kernel driver. You can call them directly

in your kernel module without modifying the kernel code. In other words, It tells the **kbuild** mechanism that the symbol referred to should be part of the global list of kernel symbols. That allows the kernel modules to access them.

Only the symbols that have been explicitly exported can be used by other modules.

Another macro is also available to export the symbols like **EXPORT_SYMBOL**. That is **EXPORT_SYMBOL_GPL()**.

EXPORT_SYMBOL exports the symbol to any loadable module.
EXPORT_SYMBOL_GPL exports the symbol only to GPL-licensed modules.

How to use EXPORT_SYMBOL?

- Declare and define the symbol (functions or variables) which you want to make it visible to other kernel modules. Then below the definition, use **EXPORT_SYMBOL(symbol name)**. Now it is visible to all loadable modules.
- Now take the kernel driver who is gonna use the above exported symbol. Declare the symbol using **extern**. Then use the symbol directly.
- Finally, load the module first, who has the definition of the export symbol. Then load the caller module using **insmod**".

Limitation

- That symbol should not be **static** or **inline.**
- Order of loading the driver is matter. ie. We should load the module which has the definition of the symbol, then only we can load the module who is using that symbol.

Driver Source Code – EXPORT_SYMBOL in Linux

First, I will explain to you the concept of driver code attached below.

In this tutorial, we have two drivers.

Driver 1 has one function called **etx shared func** and one global

variable called **etx_count.** This function and variable has been shared among with all the loadable modules using **EXPORT_SYMBOL.**

Driver 2 will be using that variable and function which are shared by ***Driver 1***. When we read this Driver 2, then it will call the shared function and we can read that variable also.

Let's see the source code below.

driver1.c

```
#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>

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

static int __init etx_driver_init(void);
static void __exit etx_driver_exit(void);
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 * of
static ssize_t etx_write(struct file *filp, const char *buf, size_t len, loff_t * o

int etx_count = 0;
void etx_shared_func(void)
{
        printk(KERN_INFO "Shared function been called!!!\n");
    etx_count++;
}
//EXPORT_SYMBOL_GPL(etx_shared_func);
EXPORT_SYMBOL(etx_shared_func);
EXPORT_SYMBOL(etx_count);

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

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)
```

```
46  {
47          printk(KERN_INFO "Device File Closed...!!!\n");
48          return 0;
49  }
50
51  static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t *of
52  {
53          printk(KERN_INFO "Data Read : Done!\n");
54          return 1;
55  }
56  static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, lof
57  {
58          printk(KERN_INFO "Data Write : Done!\n");
59          return len;
60  }
61
62  static int __init etx_driver_init(void)
63  {
64          /*Allocating Major number*/
65          if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev1")) <0){
66                  printk(KERN_INFO "Cannot allocate major number\n");
67                  return -1;
68          }
69          printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
70
71          /*Creating cdev structure*/
72          cdev_init(&etx_cdev,&fops);
73
74          /*Adding character device to the system*/
75          if((cdev_add(&etx_cdev,dev,1)) < 0){
76              printk(KERN_INFO "Cannot add the device to the system\n");
77              goto r_class;
78          }
79
80          /*Creating struct class*/
81          if((dev_class = class_create(THIS_MODULE,"etx_class1")) == NULL){
82              printk(KERN_INFO "Cannot create the struct class\n");
83              goto r_class;
84          }
85
86          /*Creating device*/
87          if((device_create(dev_class,NULL,dev,NULL,"etx_device1")) == NULL){
88              printk(KERN_INFO "Cannot create the Device 1\n");
89              goto r_device;
90          }
91          printk(KERN_INFO "Device Driver 1 Insert...Done!!!\n");
92          return 0;
93
94  r_device:
95          class_destroy(dev_class);
96  r_class:
97          unregister_chrdev_region(dev,1);
98          return -1;
99  }
100
101 void __exit etx_driver_exit(void)
102 {
103         device_destroy(dev_class,dev);
104         class_destroy(dev_class);
105         cdev_del(&etx_cdev);
106         unregister_chrdev_region(dev, 1);
107         printk(KERN_INFO "Device Driver 1 Remove...Done!!!\n");
108 }
```

```
109
110 module_init(etx_driver_init);
111 module_exit(etx_driver_exit);
112
113 MODULE_LICENSE("GPL");
114 MODULE_AUTHOR("EmbeTronicX <embetronicx@gmail.com>");
115 MODULE_DESCRIPTION("EXPORT_SYMBOL Driver - 1");
116 MODULE_VERSION("1.25");
```

driver2.c

```
 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 extern int etx_count;
21 void etx_shared_func(void); //Function declaration is by defalut extern
22
23 static struct file_operations fops =
24 {
25         .owner          = THIS_MODULE,
26         .read           = etx_read,
27         .write          = etx_write,
28         .open           = etx_open,
29         .release        = etx_release,
30 };
31
32 static int etx_open(struct inode *inode, struct file *file)
33 {
34         printk(KERN_INFO "Device File Opened...!!!\n");
35         return 0;
36 }
37
38 static int etx_release(struct inode *inode, struct file *file)
39 {
40         printk(KERN_INFO "Device File Closed...!!!\n");
41         return 0;
42 }
43
44 static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t *of
45 {
46     etx_shared_func();
47         printk(KERN_INFO "%d time(s) shared function called!\n", etx_count);
48         printk(KERN_INFO "Data Read : Done!\n");
49         return 0;
50 }
```

```
51  static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, lof
52  {
53          printk(KERN_INFO "Data Write : Done!\n");
54          return len;
55  }
56
57  static int __init etx_driver_init(void)
58  {
59          /*Allocating Major number*/
60          if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev2")) <0){
61                  printk(KERN_INFO "Cannot allocate major number\n");
62                  return -1;
63          }
64          printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
65
66          /*Creating cdev structure*/
67          cdev_init(&etx_cdev,&fops);
68
69          /*Adding character device to the system*/
70          if((cdev_add(&etx_cdev,dev,1)) < 0){
71              printk(KERN_INFO "Cannot add the device to the system\n");
72              goto r_class;
73          }
74
75          /*Creating struct class*/
76          if((dev_class = class_create(THIS_MODULE,"etx_class2")) == NULL){
77              printk(KERN_INFO "Cannot create the struct class\n");
78              goto r_class;
79          }
80
81          /*Creating device*/
82          if((device_create(dev_class,NULL,dev,NULL,"etx_device2")) == NULL){
83              printk(KERN_INFO "Cannot create the Device 1\n");
84              goto r_device;
85          }
86          printk(KERN_INFO "Device Driver 2 Insert...Done!!!\n");
87          return 0;
88
89  r_device:
90          class_destroy(dev_class);
91  r_class:
92          unregister_chrdev_region(dev,1);
93          return -1;
94  }
95
96  void __exit etx_driver_exit(void)
97  {
98          device_destroy(dev_class,dev);
99          class_destroy(dev_class);
100         cdev_del(&etx_cdev);
101         unregister_chrdev_region(dev, 1);
102         printk(KERN_INFO "Device Driver 2 Remove...Done!!!\n");
103 }
104
105 module_init(etx_driver_init);
106 module_exit(etx_driver_exit);
107
108 MODULE_LICENSE("GPL");
109 MODULE_AUTHOR("EmbeTronicX <embetronicx@gmail.com>");
110 MODULE_DESCRIPTION("EXPORT_SYMBOL Driver - 2");
111 MODULE_VERSION("1.26");
```

MakeFile

```
obj-m += driver1.o
obj-m += driver2.o

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

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

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

Compiling and Testing Driver

- Build the driver by using Makefile (***sudo make***)
- After compiling, you can able to see the file named as "***Module.symvers***". If you open that file, then our shared function and variable will be mentioned there.

```
0x1db7034a      etx_shared_func         /home/embetronicx/driver/driver1            EXP
0x6dcb135c      etx_count                   /home/embetronicx/driver/driver1
```

- Load the driver 1 using **sudo insmod driver1.ko**(Driver 1 should be loaded first. If you try to load the Driver 2 first, then you will get an error like "***insmod: ERROR: could not insert module driver2.ko: Unknown symbol in module***").
- Load the driver 1 using **sudo insmod driver2.ko**
- Now check the **dmesg**

```
[ 393.814900] Major = 246 Minor = 0
[ 393.818413] Device Driver 1 Insert...Done!!!
[ 397.620296] Major = 245 Minor = 0
[ 397.629002] Device Driver 2 Insert...Done!!!
```

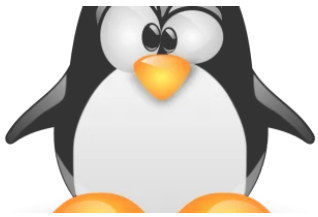
- Then do **cat /proc/kallsyms | grep etx_shared_func**or**cat /proc/kallsyms | grep etx_count**to check whether our shared function and variable become the part of kernel's symbol table or not.
- Now we can read the driver by using **sudo cat /dev/etx_device2**
- Now check the **dmesg**

```
[ 403.739998] Device File Opened...!!!
[ 403.740018] Shared function been called!!!
[ 403.740021] 1 time(s) shared function called!
[ 403.740023] Data Read : Done!
[ 403.740028] Device File Closed...!!!
```

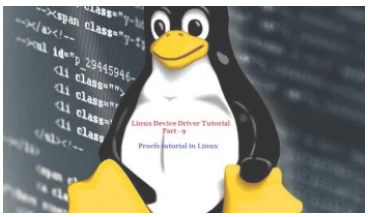
- Now we can see the print from shared function and variable count also.
- Unload the module 2 using `sudo rmmod driver2`(Driver 2 should be unloaded first. If you unload the Driver 1 first, then you will get error like "***rmmod: ERROR: Module driver1 is in use by: driver2***").
- Unload the module 1 using `sudo rmmod driver1`

In our next tutorial, we will discuss atomic variables in the Linux device driver.

Related



Linux Device Driver Tutorial Part 2 - First Linux Device Driver
In "Device Drivers"



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

Linux Device Driver Part 1 : Introduction
In "Device Drivers"


report this ad