



The ezoic logo, which consists of a small green circular icon with a white 'e' shape inside, followed by the word "ezoic" in a green, lowercase, sans-serif font.

[report this ad](#)[Sidebar ▾](#)

Device Drivers

Linux Device Driver Tutorial Part 2 - First Device Driver

This is the [Series on Linux Device Driver](#). The aim of this series is to provide the easy and practical examples that anyone can understand. Now we are going to see Linux Device Driver Tutorial Part 2 – First Device Driver. Before writing driver, we should give the module information. So First we will see about those module information.

Post Contents

[1 Linux Device Driver Tutorial Part 2 – First Device Driver](#)

[2 Module Information](#)

[2.1 License](#)

[2.2 Author](#)

[2.3 Module Description](#)

[2.4 Module Version](#)

[3 Simple Kernel Module Programming](#)

[3.1 Init function](#)

[3.2 Exit function](#)

[3.3 Printk\(\)](#)

[3.4 Difference between printf and printk](#)

[3.5 Simple Driver](#)

[4 Compiling our driver](#)

[5 Loading and Unloading the Device driver](#)

[5.1 Loading](#)

[5.2 Listing the Modules](#)

[5.3 Unloading](#)

[5.4 Getting Module Details](#)

[5.4.1 Share this:](#)

[5.4.2 Like this:](#)

[5.4.3 Related](#)

Linux Device Driver Tutorial Part 2 - First Device Driver

Module Information

- License
- Author
- Module Description
- Module Version

These all informations are present in the linux/module.h as a macros.

License

GPL, or the GNU General Public License, is an open source license meant for software. If your software is licensed under the terms of the GPL, it is free. However, “free” here does not essentially mean freeware—it can also be a paid software. Instead, “free” as per the GPL means freedom. As proponents of GPL proudly proclaim, *free as in freedom, not free beer.*

The following license idents are currently accepted as indicating free software modules.

"GPL" [GNU Public License v2 or later]

"GPL v2" [GNU Public License v2]

"GPL and additional rights" [GNU Public License v2 rights and more]

"Dual BSD/GPL" [GNU Public License v2 or BSD license choice]

"Dual MIT/GPL" [GNU Public License v2 or MIT license choice]

"Dual MPL/GPL" [GNU Public License v2 or Mozilla license choice]

The following other idents are available

"Proprietary" [Non free products]

There are dual licensed components, but when running with Linux it is the GPL that is relevant so this is a non issue. Similarly LGPL linked with GPL is a GPL combined work.

This exists for several reasons,

1. modinfo can show license info for users wanting to vet their setup is free
2. The community can ignore bug reports including proprietary modules
3. Vendors can do likewise based on their own policies

We can give the License for our driver (module) like below. For this you need to include the Linux/module.h header file.

```
1 MODULE_LICENSE("GPL");
2 MODULE_LICENSE("GPL v2");
3 MODULE_LICENSE("Dual BSD/GPL");
```

Note : It is not strictly necessary, but your module really should specify which license applies to its code.

Author

Using this Macro we can mention that who is wrote this driver or module. So modinfo can show author name for users wanting to know. We can give the Author name for our driver (module) like below. For this you

need to include the Linux/module.h header file.

```
1 MODULE_AUTHOR("Author");
```

Note: Use “Name <email>” or just “Name”, for multiple authors use multiple MODULE_AUTHOR() statements/lines.

Module Description

Using this Macro we can give the description of the module or driver. So modinfo can show module description for users wanting to know. We can give the description for our driver (module) like below. For this you need to include the linux/module.h header file.

```
1 MODULE_DESCRIPTION("A sample driver");
```

Module Version

Using this Macro we can give the version of the module or driver. So modinfo can show module version for users wanting to know.

Version of form [<epoch>:]<version>[-<extra-version>].

<epoch>: A (small) unsigned integer which allows you to start versions anew. If not mentioned, it's zero. eg. “2:1.0” is after “1:2.0”.

<version>: The <version> may contain only alphanumerics and the character `.'. Ordered by numeric sort for numeric parts, ascii sort for ascii parts (as per RPM or DEB algorithm).

<extraversion>: Like <version>, but inserted for local customizations, eg “rh3” or “rusty1”.

Example

```
1 MODULE_VERSION("2:1.0");
```

Simple Kernel Module Programming

So as of now we know the very basic things that needed for writing driver. Now we will move into programming. In every programming language, how we will start to write the code? Any ideas? Well, in all programming there would be a starting point and ending point. If you take C Language, starting point would be the main function, Isn't it? It will start from the starting of the main function and run through the functions which is calling from main function. Finally it exits at the main function closing point. But Here two separate functions used for that starting and ending.

1. Init function
2. Exit function

Kernel modules require a different set of header files than user programs require. And keep in mind, Module code should not invoke user space Libraries or API's or System calls.

Init function

This is the function which will executes first when the driver is loaded into the kernel. For example when we load the driver using insmod, this function will execute. Please see below to know the syntax of this function.

```
1 static int __init hello_world_init(void) /* Constructor */
2 {
3     return 0;
4 }
5 module_init(hello_world_init);
```

This function should register itself by using module_init() macro.

Exit function

This is the function which will executes last when the driver is unloaded from the kernel. For example when we unload the driver using rmmod, this function will execute. Please see below to know the syntax of this function.

```
1 void __exit hello_world_exit(void)
2 {
```

```
3
4 }
5 module_exit(hello_world_exit);
```

This function should register itself by using module_exit() macro.

Printk()

In C programming how we will print the values or whatever? Correct. Using printf() function. printf() is a user space function. So we cant use this here. So they created one another function for kernel which is printk().

One of the differences is that printk lets you classify messages according to their severity by associating different loglevels, or priorities, with the messages. You usually indicate the loglevel with a macro. I will explain about the macros now. There are several macros used for printk.

KERN_EMERG:

Used for emergency messages, usually those that precede a crash.

KERN_ALERT:

Situation requiring immediate action.

KERN_CRIT:

Critical conditions, often related to serious hardware or software

failures.

KERN_ERR:

Used to report error conditions; device drivers often use KERN_ERR to report hardware difficulties.

KERN_WARNING:

Warnings about problematic situations that do not, in themselves, create serious problems with the system.

KERN_NOTICE:

Situations that are normal, but still worthy of note. A number of security-related conditions are reported at this level.

KERN_INFO:

Informational messages. Many drivers print information about the hardware they find at startup time at this level.

KERN_DEBUG:

Used for debugging messages.

Example

```
1 printk(KERN_INFO "Welcome To EmbeTronicX");
```

Difference between printf and printk

- Printk() is a kernel level function, which has the ability to print out to different loglevels as defined in . We can see the prints using dmesg command.
- printf() will always print to a file descriptor - STD_OUT. We can see the prints in STD_OUT console.

Simple Driver

This is the complete code for our simple device driver

(hello_world_module.c). You can download the Project By clicking below link.

[Click Here For Code](#)

```

1 #include<linux/kernel.h>
2 #include<linux/init.h>
3 #include<linux/module.h>
4
5
6 static int __init hello_world_init(void)
7 {
8     printk(KERN_INFO "Welcome to EmbeTronicX\n");
9     printk(KERN_INFO "This is the Simple Module\n");
10    printk(KERN_INFO "Kernel Module Inserted Successfully...\n");
11    return 0;
12 }
13
14 void __exit hello_world_exit(void)
15 {
16     printk(KERN_INFO "Kernel Module Removed Successfully...\n");
17 }
18
19 module_init(hello_world_init);
20 module_exit(hello_world_exit);
21
22 MODULE_LICENSE("GPL");
23 MODULE_AUTHOR("EmbeTronicX <embedtronicx@gmail.com or admin@embedtronicx.com>");
24 MODULE_DESCRIPTION("A simple hello world driver");
25 MODULE_VERSION("2:1.0");

```

Compiling our driver

Once we have the C code, it is time to compile it and create the module file `hello_world_module.ko`. creating a Makefile for your module is straightforward.

```

1 obj-m += hello_world_module.o
2 KDIR = /lib/modules/$(shell uname -r)/build
3
4 all:
5     make -C $(KDIR) M=$(shell pwd) modules
6 clean:
7     make -C $(KDIR) M=$(shell pwd) clean

```

With the C code (`hello_world_module.c`) and Makefile ready, all we need to do is invoke `make` to build our first driver (`hello_world_module.ko`).

In Terminal you need to enter `sudo make` like below image.

[linux-device-driver-tutorial-make](#) Linux Device Driver Tutorial Part 2 -

First Device Driver

Now we got `hello_world_module .ko`. This is the kernel object which is loading into kernel.

Loading and Unloading the Device driver

A Kernel Module is a small file that may be loaded into the running Kernel, and unloaded.

Loading

To load a Kernel Module, use the `insmod` command with root privileges.

For example our module file name is `hello_world_module.ko`

```
sudo insmod hello_world_module.ko
```

[linux-device-driver-tutorial-insmod](#) Linux Device Driver Tutorial Part 2 - First Device Driver

`lsmod` used to see the modules were inserted. In below image, i've shown the prints in init function. Use `dmesg` to see the kernel prints.

[linux-device-driver-tutorial-insmod-dmesg-prints](#) Linux Device Driver Tutorial Part 2 - First Device Driver

So when i load the module, it executes the init function.

Listing the Modules

In order to see the list of currently loaded Modules, use the `lsmod` command. In above image you can see that i have used `lsmod` command.

Unloading

To un-load a Kernel Module, use the `rmmmod` command with root privileges.

In our case,

```
sudo rmmmod hello_world_module.ko or sudo rmmmod hello_world_module
```

linux-device-driver-tutorial-rmmod-dmesg Linux Device Driver Tutorial Part 2 - First Device Driver

So when i unload the module, it executes the exit function.

Getting Module Details

In order to get information about a Module (author, supported options), we may use the modinfo command.

For example

```
modinfo hello_world_module.ko
```

linux-device-driver-tutorial-modinfo Linux Device Driver Tutorial Part 2 - First Device Driver

So now we know where to start to write the Linux device driver. Thank you for Reading 😊 .

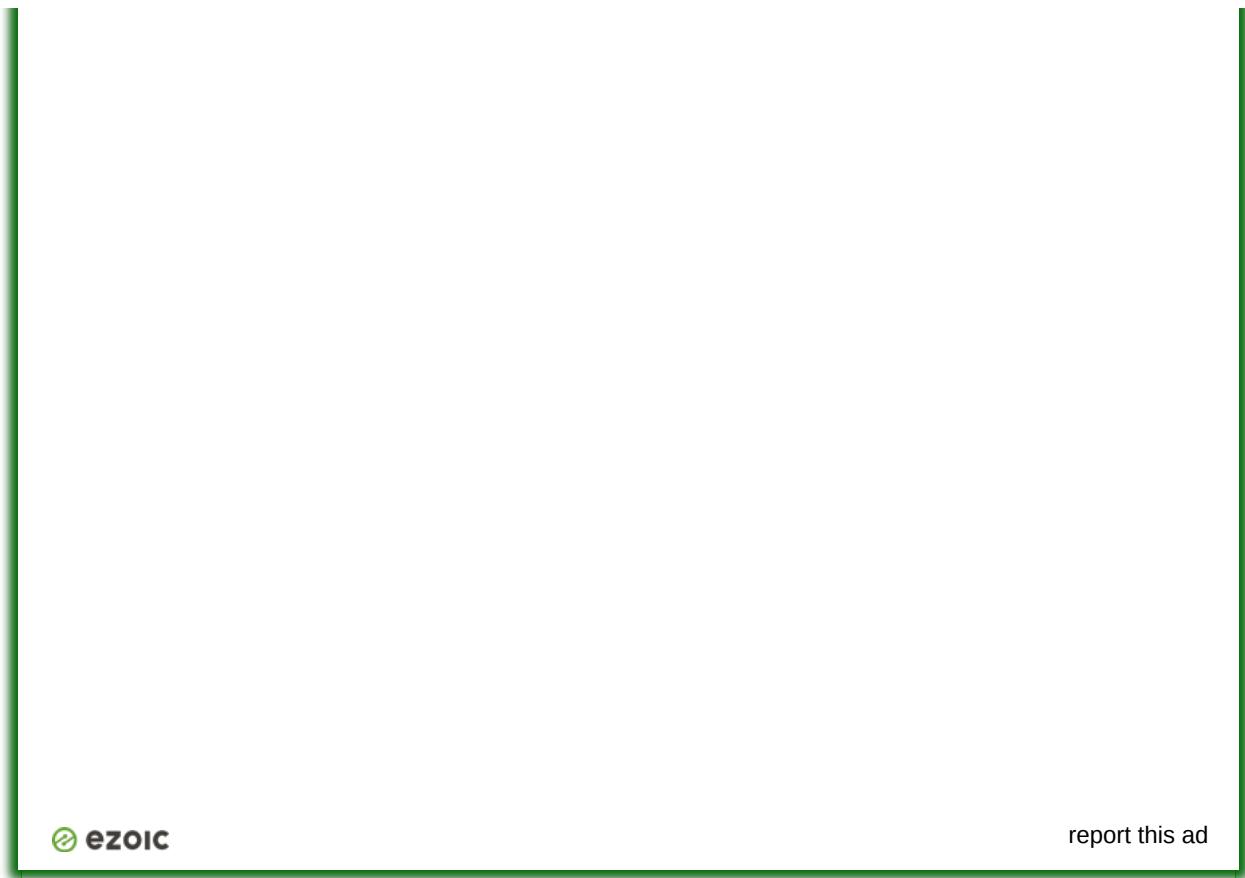
Share this:

[Share on Tumblr](#)[Tweet](#)[WhatsApp](#)[Pocket](#)[Telegram](#)

pinit_fg_en_rect_gray_20 Linux Device Driver Tutorial Part 2 - First Device Driver

Like this:

Loading...



 ezoic

[report this ad](#)



A small green square icon with a white 'e' symbol, followed by the word "ezoic" in a green, sans-serif font.

report this ad

Sidebar ▾

A blue folder icon with a white 'd' symbol, followed by the text "Device Drivers" in a blue, sans-serif font.

Linux Device Driver Tutorial Part 3 – Passing Arguments to Device Driver

Linux Device Driver Tutorial Part 3 – Module Parameter

This is the [Series on Linux Device Driver](#). The aim of this series is to provide the easy and practical examples of Linux Device Drivers that anyone can understand easily. Now we are going to see Linux Device Driver Tutorial Part 3 – Passing Arguments to Device Driver.

Post Contents

- [1 Linux Device Driver Tutorial Part 3 – Passing Arguments to Device Driver](#)
- [2 Module Parameters Macros
 - \[2.1 module_param\\(\\)\]\(#\)
 - \[2.2 module_param_array\\(\\);\]\(#\)
 - \[2.3 module_param_cb\\(\\)
 - \\[2.3.1 When we will need this notification?\\]\\(#\\)\]\(#\)](#)
- [3 Programming](#)
- [4 Compiling](#)
- [5 Loading the Driver](#)
- [6 Verify the parameters by using dmesg](#)
- [7 Unloading the Driver
 - \[7.0.1 Share this:\]\(#\)
 - \[7.0.2 Like this:\]\(#\)
 - \[7.0.3 Related\]\(#\)](#)

Linux Device Driver Tutorial Part 3 – Passing Arguments to Device Driver

We can pass the arguments to any other functions in same program. But Is it possible to pass any arguments to any program? I think Probably yes. Right? Well, we can. In C Programming we can pass the arguments to the program. For that we need to add argc and argv in main function definition. I hope everyone knows

that. Now come to our topic. Is it possible to pass the argument to the Device Driver? Fine. In this tutorial we are going to see that topic.

Module Parameters Macros

- module_param()
- module_param_array()
- module_param_cb()

Before discuss these macros we have to know about permissions of the variable.

There are several types of permissions:

- S_IWUSR
- S_IRUSR
- S_IXUSR
- S_IRGRP
- S_IWGRP
- S_IXGRP

In this S_I is common header.

R = read ,W =write ,X= Execute.

USR =user ,GRP =Group

Using OR ‘|’ (or operation) we can set multiple permissions at a time.

module_param()

This macro used to initialize the arguments. `module_param` takes three parameters: the name of the variable, its type, and a permissions mask to be used for an accompanying sysfs entry. The macro should be placed outside of any function and is typically found near the head of the source file. `module_param()` macro, defined in `linux/moduleparam.h`.

```
module_param(name, type, perm);
```

`module_param()` macro creates the sub-directory under `/sys/module`. For example

```
1 module_param(valueETX, int, S_IWUSR|S_IRUSR);
```

This will create the sysfs entry. (`/sys/module/hello_world_module/parameters/valueETX`)

Numerous types are supported for module parameters:

- `bool`
- `invbool`

A boolean (true or false) value (the associated variable should be of type `int`). The `invbool` type inverts the value, so that true values become false and vice versa.

- `charp`

A char pointer value. Memory is allocated for user-provided strings, and the pointer is set accordingly.

- `int`
- `long`
- `short`
- `uint`
- `ulong`
- `ushort`

Basic integer values of various lengths. The versions starting with `u` are for unsigned values.

`module_param_array()`

This macro is used to send the array as a argument. Array parameters, where the values are supplied as a comma-separated list, are also supported by the module loader. To declare an array parameter, use:

```
module_param_array(name, type, num, perm);
```

Where,

`name` is the name of your array (and of the parameter),

`type` is the type of the array elements,

`num` is an integer variable (optional) otherwise `NULL`, and

`perm` is the usual permissions value.

`module_param_cb()`

This macro used to register the callback whenever the argument (parameter) got changed. I think you don't understand. Let me explain properly.

For Example,

I have created one parameter by using `module_param()`.

```
1 module_param(valueETX, int, S_IWUSR|S_IRUSR);
```

This will create the sysfs entry. (`/sys/module/hello_world_module/parameters/valueETX`)

You can change the value of `valueETX` from command line by

```
echo 1 > /sys/module/hello_world_module/parameters/valueETX
```

This will update the `valueETX` variable. But there is no way to notify your module that “`valueETX`” has changed.

By using this `module_param_cb()` macro, we can get notification.

If you want to get notification whenever value got change. we need to register our handler function to its file operation structure.

```
1 struct kernel_param_ops {
2     int (*set)(const char *val, const struct kernel_param *kp);
3     int (*get)(char *buffer, const struct kernel_param *kp);
4     void (*free)(void *arg);
5 };
```

For further explanation, please refer below program.

When we will need this notification?

I will tell you the practical scenario. Whenever value is set to

1, you have to write something in to a hardware register. How can you do this if the change of value variable is not notified to you? Got it? I think you have understood. If you didn't understand, just see the explanation posted below.

Programming

In this example, i explained all (module_param, module_param_array, module_param_cb).

For module_param(), i have created two variables. One is integer (valueETX) and another one is string (nameETX).

For module_param_array(), i have created one integer array variable () .

For module_param_cb(), i have created one integer variable (cb_valueETX).

You can change the all variable using their sysfs entry which is under /sys/module/hello_world_module/parameters/

But you want get any notification when they got change except the variable which is created by module_param_cb() macro.

Download the code by clicking below link.

[\[Download Project Here\]](#)

```

1 #include<linux/kernel.h>
2 #include<linux/init.h>
3 #include<linux/module.h>
4 #include<linux/moduleparam.h>
5
6 int valueETX, arr_valueETX[4];
7 char *nameETX;
8 int cb_valueETX = 0;
9
10 module_param(valueETX, int, S_IRUSR|S_IWUSR); //integer value
11 module_param(nameETX, charp, S_IRUSR|S_IWUSR); //String
12 module_param_array(arr_valueETX, int, NULL, S_IRUSR|S_IWUSR); //Array of integers
13
14 /*-----Module_param_cb()-----*/
15 int notify_param(const char *val, const struct kernel_param *kp)
16 {
17     int res = param_set_int(val, kp); // Use helper for write variable
18     if(res==0) {
19         printk(KERN_INFO "Call back function called...\n");
20         printk(KERN_INFO "New value of cb_valueETX = %d\n", cb_valueETX);
21         return 0;
22     }
23     return -1;
24 }
25
26 const struct kernel_param_ops my_param_ops =
27 {
28     .set = &notify_param, // Use our setter ...
29     .get = &param_get_int, // .. and standard getter
30 };
31
32 module_param_cb(cb_valueETX, &my_param_ops, &cb_valueETX, S_IRUGO|S_IWUSR );
33 /*-----*/
34
35 static int __init hello_world_init(void)
36 {
37     int i;
38     printk(KERN_INFO "ValueETX = %d \n", valueETX);
39     printk(KERN_INFO "cb_valueETX = %d \n", cb_valueETX);
40     printk(KERN_INFO "NameETX = %s \n", nameETX);
41     for (i = 0; i < (sizeof arr_valueETX / sizeof (int)); i++) {
42         printk(KERN_INFO "Arr_value[%d] = %d\n", i, arr_valueETX[i]);
43     }
44     printk(KERN_INFO "Kernel Module Inserted Successfully...\n");
45     return 0;
46 }
47
48 void __exit hello_world_exit(void)
49 {
50     printk(KERN_INFO "Kernel Module Removed Successfully...\n");
51 }
52
53 module_init(hello_world_init);
54 module_exit(hello_world_exit);
55
56 MODULE_LICENSE("GPL");
57 MODULE_AUTHOR("EmbeTronicX <embedtronicx@gmail.com or admin@embedtronicx.com>");
58 MODULE_DESCRIPTION("A simple hello world driver");
59 MODULE_VERSION("1.0");

```

Compiling

This is the code of **Makefile**.

```

1 obj-m += hello_world_module.o
2
3 KDIR = /lib/modules/$(shell uname -r)/build
4
5 all:
6   make -C $(KDIR) M=$(shell pwd) modules
7
8 clean:
9   make -C $(KDIR) M=$(shell pwd) clean

```

In terminal enter **sudo make**

[linux-device-driver-tutorial-passing-arguments-to-device-driver-make](#) [Linux Device Driver Tutorial Part 3 – Module Parameter](#)

Loading the Driver

```

sudo      insmod      hello_world_module.ko      valueETX=14
nameETX="EmbeTronicX" arr_valueETX=100,102,104,106

```

[linux-device-driver-tutorial-passing-arguments-to-device-driver-insmod](#) [Linux Device Driver Tutorial Part 3 – Module Parameter](#)

Verify the parameters by using **dmesg**

Now our module got loaded. now check **dmesg**. In below picture, every value got passed to our device driver.

[linux-device-driver-tutorial-passing-arguments-to-device-driver-dmesg](#) [Linux Device Driver Tutorial Part 3 – Module Parameter](#)

Now i'm going to check **module_param_cb()** is weather calling that handler function or not. For that i need to change the variable in sysfs.

```
echo 13 > /sys/module/hello_world_module/parameters/cb_valueETX
```

[linux-device-driver-tutorial-passing-arguments-to-device-driver-echo](#) [Linux Device Driver Tutorial Part 3 – Module Parameter](#)

Now do **dmesg** and check.

[linux-device-driver-tutorial-passing-arguments-to-device-driver-echo-dmesg](#)

Linux Device Driver Tutorial Part 3 – Module Parameter

See the above result. So Our callback function got called. But if you change the value of other variables, you wont get notification.

Unloading the Driver

Finally unload the driver by using `sudo rmmod hello_world_module`.

I hope you understood. If you have any doubt, please comment below. ☺

Share this:

[Share on Tumblr](#)

[Tweet](#)

[Print](#)

[WhatsApp](#)

[Pocket](#)

[Telegram](#)

[pinit_fg_en_rect_gray_20](#) Linux Device Driver Tutorial Part 3 – Module Parameter

Like this:

Loading...



[report this ad](#)



Sidebar ▾

Device Drivers

Linux Device Driver Tutorial Part 4 - Character Device Driver

This is the [Series on Linux Device Driver](#). The aim of this series is to provide the easy and practical examples of Linux Device Drivers that anyone can understand easily. Now we are going to see Linux Device Driver Tutorial Part 4 - Character Device Driver Major Number and Minor Number.

Post Contents

- 1 Introduction
- 2 How Applications will communicate Hardware device?
- 3 Character Device Driver Major Number and Minor Number
- 4 Major Number and Minor Number
 - 4.1 Major number
 - 4.2 Minor Number
- 5 Allocating Major and Minor Number
 - 5.1 Statically allocating
 - 5.2 Dynamically Allocating
 - 5.3 Difference between static and dynamic method
- 6 Unregister the Major and Minor Number
- 7 Program for Statically Allocating Major Number
- 8 Program for Dynamically Allocating Major Number
 - 8.0.1 Share this:
 - 8.0.2 Like this:
 - 8.0.3 Related

Introduction

We already know what drivers are, and why we need them. What is so special about character drivers? If we write drivers for byte-oriented operations then we refer to them as character drivers. Since the majority of devices are byte-oriented, the majority of device drivers are character device drivers. Take, for example, serial drivers, audio drivers, video drivers, camera drivers, and basic I/O drivers. In fact, all device drivers that are neither storage nor network device drivers are some type of a character driver.

How Applications will communicate Hardware device?

This below diagram will show the full path of the communication.

[character-device-driver-communication Linux Device Driver Tutorial Part 4 – Character Device Driver](#)

- First Application will open the device file. This device file is created by device driver).
- Then This device file will find the correspond device driver using major and minor number.
- Then That Device driver will talk to Hardware device.

Character Device Driver Major Number and Minor Number

One of the basic features of the Linux kernel is that it abstracts the handling of devices. All hardware devices look like regular files; they can be opened, closed, read and written using the same, standard, system calls that are used to manipulate files. To Linux, everything is a file. To write to the hard disk, you write to a file. To read from the keyboard is to read from a file. To store backups on a tape device is to write to a file. Even to read from memory is to read from a file. If the file from which you are trying to read or to which you are trying to write is a “normal” file, the process is fairly easy to understand: the file is opened and you read or write data. So device driver also like file. Driver will create the special file for every hardware devices. We can communicate to the hardware using those special files (device file).

If you want to create the special file, we should know about the Major number and minor number in device driver. In this tutorial we will learn that major and minor number.

Major Number and Minor Number

The Linux kernel represents character and block devices as pairs of numbers <major>:<minor>.

Major number

Traditionally, the major number identifies the driver associated with the device. A major number can also be shared by multiple device drivers. See /proc/devices to find out how major numbers are assigned on a running Linux instance.

```
linux@embedtronix-VirtualBox:~/project/devicedriver/devicefile$  
cat /proc/devices
```

Character devices:

1 mem

4 /dev/vc/0

4 tty

Block devices:

1 ramdisk

259 blkext

These numbers are major numbers.

Minor Number

Major number is identify the corresponding driver. Many devices may use same major number. So we need to assign the number to each devices which is using same major number. So this is the minor number. In other words, The device driver uses the minor number <minor> to distinguish individual physical or logical devices.

Allocating Major and Minor Number

We can allocate the major and minor numbers by two ways.

1. Statically allocating
2. Dynamically Allocating

Statically allocating

If you want to set the particular major number to your driver, you can use this method. This method will allocate that major number if it is available. Otherwise it won't.

```
int register_chrdev_region(dev_t first, unsigned int count, char *name);
```

Here, *first* is the beginning device number of the range you would like to allocate.

count is the total number of contiguous device numbers you are requesting. Note that, if *count* is large, the range you request could spill over to the next major number; but everything will still work properly as long as the number range you request is available.

name is the name of the device that should be associated with this number range; it will appear in /proc/devices and sysfs.

The return value from *register_chrdev_region* will be 0 if the allocation was successfully performed. In case of error, a negative error code will be returned, and you will not have access to the requested region.

The *dev_t* type (defined in *<linux/types.h>*) is used to hold device numbers—both the major and minor parts. *dev_t* is a 32-bit quantity with 12 bits set aside for the major number and 20 for the minor number.

If you want to create the *dev_t* structure variable for your major and minor number, please use below function.

```
MKDEV(int major, int minor);
```

If you want to get your major number and minor number from dev_t, use below method.

```
MAJOR(dev_t dev);
```

```
MINOR(dev_t dev);
```

If you pass the dev_t structure to this MAJOR or MINOR function, it will return that major/minor number of your driver.

Example,

```
1 dev_t dev = MKDEV(235, 0);
2
3 register_chrdev_region(dev, 1, "EmbeTronicx_Dev");
```

Dynamically Allocating

If we don't want fixed major and minor number please use this method. This method will allocate the major number dynamically to your driver which is available.

```
int alloc_chrdev_region(dev_t *dev, unsigned int firstminor,
unsigned int count, char *name);
```

dev is an output-only parameter that will, on successful completion, hold the first number in your allocated range.

firstminor should be the requested first minor number to use; it is usually 0.

count is the total number of contiguous device numbers you are requesting.

name is the name of the device that should be associated with this number range; it will appear in */proc/devices* and *sysfs*.

Difference between static and dynamic method

Static method is only really useful if you know in advance which major number you want to start with. With Static method , you *tell* the kernel what device numbers you want (the start major/minor number and count) and it either gives them to you or not (depending on availability).

With Dynamic method, you tell the kernel how many device numbers you need (the starting minor number and count) and it will find a starting major number for you, if one is available, of course.

Partially to avoid conflict with other device drivers, it's considered preferable to use the Dynamic method function, which will dynamically allocate the device numbers for you.

The disadvantage of dynamic assignment is that you can't create the device nodes in advance, because the major number assigned to your module will vary. For normal use of the driver, this is hardly a problem, because once the number has been assigned, you can read it from */proc/devices*.

Unregister the Major and Minor Number

Regardless of how you allocate your device numbers, you should free them when they are no longer in use. Device numbers are freed with:

```
void unregister_chrdev_region(dev_t first, unsigned int count);
```

The usual place to call `unregister_chrdev_region` would be in your module's cleanup function (Exit Function).

Program for Statically Allocating Major Number

In this program, I'm assigning 235 as a major number.

```

1 #include<linux/kernel.h>
2 #include<linux/init.h>
3 #include<linux/module.h>
4 #include <linux/fs.h>
5
6 dev_t dev = MKDEV(235, 0);
7 static int __init hello_world_init(void)
8 {
9     register_chrdev_region(dev, 1, "EmbeTronicX_Dev");
10    printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
11    printk(KERN_INFO "Kernel Module Inserted Successfully...\n");
12    return 0;
13 }
14
15 void __exit hello_world_exit(void)
16 {
17     unregister_chrdev_region(dev, 1);
18     printk(KERN_INFO "Kernel Module Removed Successfully...\n");
19 }
20
21 module_init(hello_world_init);
22 module_exit(hello_world_exit);
23
24 MODULE_LICENSE("GPL");
25 MODULE_AUTHOR("EmbeTronicX <embedtronicx@gmail.com or admin@embedtronicx.com>");
26 MODULE_DESCRIPTION("A simple hello world driver");
27 MODULE_VERSION("1.0");
```

- Build the driver by using Makefile (`sudo make`)
- Load the driver using `sudo insmod`
- Check the major number using `cat /proc/devices`

```
linux@embedtronicx-VirtualBox:/home/driver$ cat
/proc/devices | grep "EmbeTronicX_Dev"
```

235 Embetronicx_Dev

- Unload the driver using `sudo rmmod`

Program for Dynamically Allocating Major Number

This program will allocate major number dynamically.

```

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
7 dev_t dev = 0;
8
9 static int __init hello_world_init(void)
10 {
11     /*Allocating Major number*/
12     if((alloc_chrdev_region(&dev, 0, 1, "Embetronicx_Dev")) <0){
13         printk(KERN_INFO "Cannot allocate major number for device 1\n");
14         return -1;
15     }
16     printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
17     printk(KERN_INFO "Kernel Module Inserted Successfully...\n");
18     return 0;
19 }
20
21 void __exit hello_world_exit(void)
22 {
23     unregister_chrdev_region(dev, 1);
24     printk(KERN_INFO "Kernel Module Removed Successfully...\n");
25 }
26
27 module_init(hello_world_init);
28 module_exit(hello_world_exit);
29
30 MODULE_LICENSE("GPL");
31 MODULE_AUTHOR("EmbeTronicX <embedtronix@gmail.com or admin@embedtronix.com>");
32 MODULE_DESCRIPTION("A simple hello world driver");
33 MODULE_VERSION("1.1");

```

- Build the driver by using Makefile (`sudo make`)
- Load the driver using `sudo insmod`
- Check the major number using `cat /proc/devices`

```
linux@embedtronix-VirtualBox::~/home/driver$ cat
/proc/devices | grep "Embetronicx_Dev"
```

243 Embetronicx_Dev

- Unload the driver using `sudo rmmod`

This function allocates major number of 243 for this driver.

Before unloading the driver just check the files in `/dev` directory using `ls /dev/`. You wont find our driver file. Because we haven't created yet. In our next tutorial we will see that device file.

I hope it helped you. If you have any doubt in this please comment below.

Share this:[Share on Tumblr](#)[Tweet](#)[WhatsApp](#)[Pocket](#)

[pinit_fg_en_rect_gray_20 Linux Device Driver Tutorial Part 4 – Character Device Driver](#)

Like this:

Loading...



[report this ad](#)

 ezoic[report this ad](#)[Sidebar ▾](#)

📁 Device Drivers

Device File Creation for Character Drivers

Linux Device Driver Tutorial Part 5 - Device File Creation

This article is a continuation of the [Series on Linux Device Driver](#), and carries on the discussion on character drivers and their implementation. In this tutorial we will discuss Device File Creation for Character Drivers.

Post Contents

- [1 Device File Creation for Character Drivers](#)
- [2 Device Files](#)
- [3 Creating Device File
 - \[3.1 Manually Creating Device File
 - \\[3.1.1 Advantages\\]\\(#\\)
 - \\[3.1.2 Programming\\]\\(#\\)\]\(#\)
 - \[3.2 Automatically Creating Device File
 - \\[3.2.1 Create the class\\]\\(#\\)
 - \\[3.2.2 Create Device\\]\\(#\\)
 - \\[3.2.3 Programming\\]\\(#\\)
 - \\[3.2.4 Share this:\\]\\(#\\)
 - \\[3.2.5 Like this:\\]\\(#\\)
 - \\[3.2.6 Related\\]\\(#\\)\]\(#\)](#)

Device File Creation for Character Drivers

In our Last [tutorial](#) we have seen how to assign major and minor number. But if you see there it will create major and minor number. But i wont create any device files in /dev/ directory. Device file is important to communicate to hardware. Let's start our tutorial.

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:

```
crw--w---- 1 root tty 4, 0 Aug 15 10:40 tty0
```

```
brw-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](#).

Creating Device File

We can create the device 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

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

Programming

I took this program from previous tutorial. I'm going to create device file manually for this driver.

```

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
7 dev_t dev = 0;
8
9 static int __init hello_world_init(void)
10 {
11     /*Allocating Major number*/
12     if((alloc_chrdev_region(&dev, 0, 1, "Embetronicx_Dev")) <0){
13         printk(KERN_INFO "Cannot allocate major number for device\n");
14         return -1;
15     }
16     printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
17     printk(KERN_INFO "Kernel Module Inserted Successfully...\n");
18     return 0;
19 }
20
21 void __exit hello_world_exit(void)
22 {
23     unregister_chrdev_region(dev, 1);
24     printk(KERN_INFO "Kernel Module Removed Successfully...\n");
25 }
26
27 module_init(hello_world_init);
28 module_exit(hello_world_exit);
29
30 MODULE_LICENSE("GPL");
31 MODULE_AUTHOR("EmbeTronicX <embetronicx@gmail.com or admin@embetronicx.com>");
32 MODULE_DESCRIPTION("A simple hello world driver");
33 MODULE_VERSION("1.1");

```

- Build the driver by using Makefile (*sudo make*)
- Load the driver using *sudo insmod*
- Check the device file using *ls -l /dev/*. By this time device file is not created for your driver.
- Create device file using *mknod* and then check using *ls -l /dev/*.

```
linux@embetronicx-VirtualBox:/home/driver$ sudo mknod -m 666
/dev/etx_device c 246 0
```

```
linux@embetronicx-VirtualBox::/home/driver$ ls -l /dev/ |
```

```
grep "etx_device"

crw-rw-rw- 1 root root 246, 0 Aug 15 13:53 etx_device
```

- Now our device file got created and registered with major number.
- Unload the driver using `sudo rmmod`

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);
```

Create Device

This function can be used by char device classes. A struct device will be created in sysfs, registered to the specified class.

```
struct device *device_create (struct *class, struct device  
*parent,  
  
dev_t dev, const char *fmt, ...);
```

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);
```

If you don't understand please refer the below program, Then you will understand.

Programming

```

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/device.h>
7
8 dev_t dev = 0;
9 static struct class *dev_class;
10
11
12 static int __init hello_world_init(void)
13 {
14     /*Allocating Major number*/
15     if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
16         printk(KERN_INFO "Cannot allocate major number for device\n");
17         return -1;
18     }
19     printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
20
21     /*Creating struct class*/
22     if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
23         printk(KERN_INFO "Cannot create the struct class for device\n");
24         goto r_class;
25     }
26
27     /*Creating device*/
28     if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
29         printk(KERN_INFO "Cannot create the Device\n");
30         goto r_device;
31     }
32     printk(KERN_INFO "Kernel Module Inserted Successfully...\n");
33 return 0;
34
35 r_device:
```

```
36         class_destroy(dev_class);
37 r_class:
38         unregister_chrdev_region(dev,1);
39         return -1;
40 }
41
42 void __exit hello_world_exit(void)
43 {
44         device_destroy(dev_class,dev);
45         class_destroy(dev_class);
46         unregister_chrdev_region(dev, 1);
47         printk(KERN_INFO "Kernel Module Removed Successfully...\n");
48 }
49
50 module_init(hello_world_init);
51 module_exit(hello_world_exit);
52
53 MODULE_LICENSE("GPL");
54 MODULE_AUTHOR("EmbeTronicX <embedtronicx@gmail.com or admin@embedtronicx.com>");
55 MODULE_DESCRIPTION("A simple hello world driver");
56 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"*

```
linux@embedtronicx-VirtualBox:/home/developer$ 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. If you have any doubt, please comment below.

Share this:[Share on Tumblr](#) [Tweet](#)[WhatsApp](#)[Pocket](#)

[pinit_fg_en_rect_gray_20 Linux Device Driver Tutorial Part 5 – Device File Creation](#)

Like this:

Loading...



[report this ad](#)

 ezoic[report this ad](#)[Sidebar ▾](#)

📁 Device Drivers

Cdev structure and File Operations of Character drivers

Linux Device Driver Tutorial Part 6 - Cdev structure and File Operations

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 the Part 6 of Linux device driver tutorial. In this tutorial we will discuss Cdev structure and File Operations of Character drivers.

Based on our previous tutorial, we know about major, minor number and device file. So as i said in [previous tutorial](#), we need to open, read, write and close the device file. So let's start...

Post Contents

- 1 Cdev structure and File Operations of Character drivers
 - 1.1 cdev structure
 - 1.2 File_Operations
 - 1.2.1 Example
- 2 Dummy Driver
 - 2.0.1 Share this:
 - 2.0.2 Like this:
 - 2.0.3 Related

Cdev structure and File Operations of Character drivers

If we want to open, read, write and close we need to register some structures to driver.

cdev structure

In linux kernel struct inode structure is used to represent files. Therefore, it is different from the file structure that represents an open file descriptor. There can be numerous file structures representing multiple open descriptors on a single file, but they all point to a single inode structure.

The inode structure contains a great deal of information about the file. As a general rule, cdev structure is useful for writing driver code:

struct cdev is one of the elements of the inode structure. As you probably may know already, an inode structure is used by the kernel internally to represent files. The struct cdev is the kernel's internal structure that represents char devices. This field contains a pointer to that structure when the inode refers to a char device file.

```
1 struct cdev {  
2     struct kobject kobj;  
3     struct module *owner;  
4     const struct file_operations *ops;  
5     struct list_head list;  
6     dev_t dev;  
7     unsigned int count;  
8 };
```

This is cdev structure. Here we need to fill two fields,

1. file_operation (This we will see after this cdev structure)
2. owner (This should be THIS_MODULE)

There are two ways of allocating and initializing one of these structures.

1. Runtime Allocation
2. Own allocation

If you wish to obtain a standalone cdev structure at runtime, you may do so with code such as:

```
struct cdev *my_cdev = cdev_alloc( );
my_cdev->ops = &my_fops;
```

Or else you can embed the cdev structure within a device-specific structure of your own by using below function.

```
void cdev_init(struct cdev *cdev, struct file_operations *fops);
```

Once the cdev structure is set up with file_operations and owner, the final step is to tell the kernel about it with a call to:

```
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
```

Here,

dev is the cdev structure,

num is the first device number to which this device responds, and

count is the number of device numbers that should be associated with the device. Often count is one, but there are situations where it makes sense to have more than one device number correspond to a specific device.

If this function returns negative error code, your device has not been added to the system. So check the return value of this function.

After a call to `cdev_add()`, your device is immediately alive. All functions you defined (through the `file_operations` structure) can be called.

To remove a char device from the system, call:

```
void cdev_del(struct cdev *dev);
```

Clearly, you should not access the `cdev` structure after passing it to `cdev_del`.

File Operations

The `file_operations` structure is how a char driver sets up this connection. The structure, defined in `<linux/fs.h>`, is a collection of function pointers. Each open file is associated with its own set of functions . The operations are mostly in charge of implementing the system calls and are therefore, named open, read, and so on.

A `file_operations` structure or a pointer to one is called fops. Each field in the structure must point to the function in the driver that implements a specific operation, or be left NULL for unsupported operations. The whole structure is mentioned below snippet.

```
1 struct file_operations {
2     struct module *owner;
3     loff_t (*llseek) (struct file *, loff_t, int);
4     ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
5     ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
6     ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
7     ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
```

```

8   int (*iterate) (struct file *, struct dir_context *);
9   int (*iterate_shared) (struct file *, struct dir_context *);
10  unsigned int (*poll) (struct file *, struct poll_table_struct *);
11  long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
12  long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
13  int (*mmap) (struct file *, struct vm_area_struct *);
14  int (*open) (struct inode *, struct file *);
15  int (*flush) (struct file *, fl_owner_t id);
16  int (*release) (struct inode *, struct file *);
17  int (*fsync) (struct file *, loff_t, loff_t, int datasync);
18  int (*fasync) (int, struct file *, int);
19  int (*lock) (struct file *, int, struct file_lock *);
20  ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
21  unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, uns
22  int (*check_flags)(int);
23  int (*flock) (struct file *, int, struct file_lock *);
24  ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t,
25  ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, u
26  int (*setlease)(struct file *, long, struct file_lock **, void **);
27  long (*fallocate)(struct file *file, int mode, loff_t offset,
28          loff_t len);
29  void (*show_fdinfo)(struct seq_file *m, struct file *f);
30 #ifndef CONFIG_MMU
31     unsigned (*mmap_capabilities)(struct file *);
32 #endif
33     ssize_t (*copy_file_range)(struct file *, loff_t, struct file *,
34         loff_t, size_t, unsigned int);
35     int (*clone_file_range)(struct file *, loff_t, struct file *, loff_t,
36         u64);
37     ssize_t (*dedupe_file_range)(struct file *, u64, u64, struct file *,
38         u64);
39 };

```

This `file_operations` structure contains many fields. But we will concentrate on very basic functions. Below we will see some fields explanation.

struct module *owner:

The first `file_operations` field is not an operation at all; it is a pointer to the module that “owns” the structure. This field is used to prevent the module from being unloaded while its operations are in use. Almost all the time, it is simply initialized to `THIS_MODULE`, a macro defined in `<linux/module.h>`.

`ssize_t (*read) (struct file *, char _user *, size_t, loff_t *);`

Used to retrieve data from the device. A null pointer in this position causes the read system call to fail with -EINVAL (“Invalid argument”). A non negative return value represents the number of bytes successfully read (the return value is a “signed size” type, usually the native integer type for the target platform).

ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);

Sends data to the device. If NULL, -EINVAL is returned to the program calling the write system call. The return value, if non negative, represents the number of bytes successfully written.

int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);

The ioctl system call offers a way to issue device-specific commands (such as formatting a track of a floppy disk, which is neither reading nor writing). Additionally, a few ioctl commands are recognized by the kernel without referring to the fops table. If the device doesn’t provide an ioctl method, the system call returns an error for any request that isn’t predefined (-ENOTTY, “No such ioctl for device”).

int (*open) (struct inode *, struct file *);

Though this is always the first operation performed on the device file, the driver is not required to declare a corresponding method. If this

entry is NULL, opening the device always succeeds, but your driver isn't notified.

int (*release) (struct inode *, struct file *);

This operation is invoked when the file structure is being released.
Like open, release can be NULL.

Example

```
1 static struct file_operations fops =
2 {
3     .owner          = THIS_MODULE,
4     .read           = etx_read,
5     .write          = etx_write,
6     .open           = etx_open,
7     .release        = etx_release,
8 }
```

If you want to understand the complete flow just have a look at our dummy driver.

Dummy Driver

Here i have added dummy driver snippet. In this driver code, we can do all open, read, write, close operations. Just go through the code.

```
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 * off);
18 static ssize_t etx_write(struct file *filp, const char *buf, size_t len, loff_t * off)
19
20 static struct file_operations fops =
21 {
22     .owner          = THIS_MODULE,
23     .read           = etx_read,
24     .write          = etx_write,
25     .open            = etx_open,
26     .release        = etx_release,
27 };
28
29 static int etx_open(struct inode *inode, struct file *file)
30 {
31     printk(KERN_INFO "Driver Open Function Called...!!!\n");
32     return 0;
33 }
34
35 static int etx_release(struct inode *inode, struct file *file)
36 {
37     printk(KERN_INFO "Driver Release Function Called...!!!\n");
38     return 0;
39 }
40
41 static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t *off)
42 {
43     printk(KERN_INFO "Driver Read Function Called...!!!\n");
44     return 0;
45 }
46 static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, loff_t
47 {
48     printk(KERN_INFO "Driver Write Function Called...!!!\n");
49     return len;
50 }
51
52
53 static int __init etx_driver_init(void)
54 {
55     /*Allocating Major number*/
56     if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
57         printk(KERN_INFO "Cannot allocate major number\n");
58         return -1;
59     }
60     printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
61
62     /*Creating cdev structure*/
63     cdev_init(&etx_cdev,&fops);
64
65     /*Adding character device to the system*/
66     if((cdev_add(&etx_cdev,dev,1)) < 0){
67         printk(KERN_INFO "Cannot add the device to the system\n");
68         goto r_class;
69     }

```

```

70
71     /*Creating struct class*/
72     if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
73         printk(KERN_INFO "Cannot create the struct class\n");
74         goto r_class;
75     }
76
77     /*Creating device*/
78     if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
79         printk(KERN_INFO "Cannot create the Device 1\n");
80         goto r_device;
81     }
82     printk(KERN_INFO "Device Driver Insert...Done!!!\n");
83     return 0;
84
85 r_device:
86     class_destroy(dev_class);
87 r_class:
88     unregister_chrdev_region(dev,1);
89     return -1;
90 }
91
92 void __exit etx_driver_exit(void)
93 {
94     device_destroy(dev_class,dev);
95     class_destroy(dev_class);
96     cdev_del(&etx_cdev);
97     unregister_chrdev_region(dev, 1);
98     printk(KERN_INFO "Device Driver Remove...Done!!!\n");
99 }
100
101 module_init(etx_driver_init);
102 module_exit(etx_driver_exit);
103
104 MODULE_LICENSE("GPL");
105 MODULE_AUTHOR("EmbeTronicX <embedtronicx@gmail.com or admin@embedtronicx.com>");
106 MODULE_DESCRIPTION("A simple device driver");
107 MODULE_VERSION("1.3");

```

1. Build the driver by using Makefile (*sudo make*)
2. Load the driver using *sudo insmod*
3. Do *echo 1 > /dev/etx_device*

Echo will open the driver and write 1 into the driver and finally close the driver. So if i do echo to our driver, it should call the open, write and release functions. Just check.

```
linux@embedtronicx-VirtualBox:/home/driver/driver# echo 1 >
/dev/etx_device
```

4. Now Check using dmesg

```
linux@embedtronicx-VirtualBox:/home/driver/driver$ dmesg
```

```
[19721.611967] Major = 246 Minor = 0
[19721.618716] Device Driver Insert...Done!!!
[19763.176347] Driver Open Function Called....!!!
[19763.176363] Driver Write Function Called....!!!
[19763.176369] Driver Release Function Called....!!!
```

5. Do `cat > /dev/etx_device`

Cat command will open the driver, read the driver and close the driver. So if i do cat to our driver, it should call the open, read and release functions. Just check.

```
linux@embedtronicx-VirtualBox:/home/developer# cat
/dev/etx_device
```

6. Now Check using `dmesg`

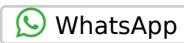
```
linux@embedtronicx-VirtualBox:/home/developer$ dmesg
[19763.176347] Driver Open Function Called....!!!
[19763.176363] Driver Read Function Called....!!!
[19763.176369] Driver Release Function Called....!!!
```

7. Unload the driver using `sudo rmmod`

Instead of doing echo and cat command in terminal you can also use `open()`, `read()`, `write()`, `close()` system calls from user space application.

I hope you understood this tutorial. This is just dummy driver tutorial. In our next tutorial we will see the some real time applications using file operations device driver. ☺

Share this:

[Share on Tumblr](#) [Tweet](#)[Pocket](#)

[pinit_fg_en_rect_gray_20 Linux Device Driver Tutorial Part 6 - Cdev structure and File Operations](#)

Like this:

Loading...



[report this ad](#)

 ezoic[report this ad](#)[Sidebar ▾](#)

📁 Device Drivers

Linux Device Driver Tutorial Part 7 - Linux Device Driver Tutorial Programming

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 the Part 7 of Linux device driver tutorial. In this tutorial we will discuss Linux Device Driver Tutorial Programming.

From our previous tutorials, we know about major, minor number, device file and file operations of device driver using dummy driver. But today we are going to write real driver without hardware.

Post Contents

- [1 Linux Device Driver Tutorial Programming](#)
- [2 Introduction](#)
- [3 Kernel Space Program \(Device Driver\)](#)
 - [3.1 Concept](#)
 - [3.2 Functions used in this driver](#)
 - [3.2.1 kmalloc\(\)](#)
 - [3.2.2 kfree\(\)](#)
 - [3.2.3 copy_from_user\(\)](#)
 - [3.2.4 copy_to_user\(\)](#)
 - [3.3 Open\(\)](#)
 - [3.4 write\(\)](#)
 - [3.5 read\(\)](#)
 - [3.6 close\(\)](#)
 - [3.7 Full Driver Code](#)
- [4 Building the Device Driver](#)
- [5 User Space Application](#)
- [6 Compile the User Space Application](#)

- [7 Execution \(Output\)](#)
 - [7.0.1 Share this:](#)
 - [7.0.2 Like this:](#)
 - [7.0.3 Related](#)

Linux Device Driver Tutorial Programming Introduction

We already know that in Linux everything is a File. So in this tutorial we are going to develop two applications.

1. User Space application (User program)
2. Kernel Space program (Driver)

User Program will communicate with the kernel space program using device file. Lets Start.

Kernel Space Program (Device Driver)

We already know about major, minor number, device file and file operations of device driver. If you don't know please visit our previous tutorials. Now we are going to discuss more about file operations in device driver. Basically there are four functions in device driver.

1. Open driver
2. Write Driver
3. Read Driver

4. Close Driver

Now we will see one by one of this functions. Before that i will explain the concept of this driver.

Concept

Using this driver we can send string or data to the kernel device driver using write function. It will store those string in kernel space. Then when i read the device file, it will send the data which is written by write by function.

Functions used in this driver

- kmalloc()
- kfree()
- copy_from_user()
- copy_to_user()

kmalloc()

We will see the memory allocation methods available in kernel, in future tutorial. But now we will use only kmalloc in this tutorial.

kmalloc function is used to allocate the memory in kernel space. This is like a malloc() function in user space. The function is fast (unless it blocks) and doesn't clear the memory it obtains. The allocated region still holds its previous content. The allocated region is also contiguous in physical memory.

```
#include <linux/slab.h>

void *kmalloc(size_t size, gfp_t flags);
```

Arguments

size_t size- how many bytes of memory are required.

gfp_t flags- the type of memory to allocate.

The *flags* argument may be one of:

GFP_USER – Allocate memory on behalf of user. May sleep.

GFP_KERNEL – Allocate normal kernel ram. May sleep.

GFP_ATOMIC – Allocation will not sleep. May use emergency pools. For example, use this inside interrupt handlers.

GFP_HIGHUSER – Allocate pages from high memory.

GFP_NOIO – Do not do any I/O at all while trying to get memory.

GFP_NOFS – Do not make any fs calls while trying to get memory.

GFP_NOWAIT – Allocation will not sleep.

__GFP_THISNODE – Allocate node-local memory only.

GFP_DMA – Allocation suitable for DMA. Should only be used for kmalloc caches. Otherwise, use a slab created with SLAB_DMA.

Also it is possible to set different flags by OR'ing in one or more of the following additional *flags*:

__GFP_COLD – Request cache-cold pages instead of trying to return cache-warm pages.

__GFP_HIGH – This allocation has high priority and may use emergency pools.

__GFP_NOFAIL – Indicate that this allocation is in no way allowed to fail (think twice before using).

__GFP_NORETRY – If memory is not immediately available, then give up at once.

__GFP_NOWARN – If allocation fails, don't issue any warnings.

__GFP_REPEAT – If allocation fails initially, try once more before failing.

There are other flags available as well, but these are not intended for general use, and so are not documented here. For a full list of potential flags, always refer to `linux/gfp.h`.

kfree()

This is like a `free()` function in user space. This is used to free the previously allocated memory.

```
void kfree(const void *objp)
```

Arguments

`*objp` – pointer returned by `kmalloc`

copy_from_user()

This function is used to Copy a block of data from user space (Copy data from user space to kernel space).

```
unsigned long copy_from_user(void *to, const void __user *from, unsigned long n);
```

Arguments

`to` – Destination address, in kernel space

`from` – Source address in user space

`n` – Number of bytes to copy

Returns number of bytes that could not be copied. On success, this will be zero.

copy_to_user()

This function is used to Copy a block of data into user space (Copy data from kernel space to user space).

```
unsigned long copy_to_user(const void __user *to, const void
```

```
*from, unsigned long n);
```

Arguments

to – Destination address, in user space

from – Source address in kernel space

n – Number of bytes to copy

Returns number of bytes that could not be copied. On success, this will be zero.

Open()

This function is called first, whenever we are opening the device file. In this function i am going to allocate the memory using kmalloc. In user space application you can use open() system call to open the device file.

```
1 static int etx_open(struct inode *inode, struct file *file)
2 {
3     /*Creating Physical memory*/
4     if((kernel_buffer = kmalloc(mem_size , GFP_KERNEL)) == 0){
5         printk(KERN_INFO "Cannot allocate memory in kernel\n");
6         return -1;
7     }
8     printk(KERN_INFO "Device File Opened...!!!\n");
9     return 0;
10 }
```

write()

When write the data to the device file it will call this write function. Here i will copy the data from user space to kernel space using copy_from_user() function. In user space application you can use write() system call to write any data the device file.

```
1 static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, loff_t *
2 {
3     copy_from_user(kernel_buffer, buf, len);
4     printk(KERN_INFO "Data Write : Done!\n");
5     return len;
6 }
```

read()

When we read the device file it will call this function. In this function i used `copy_to_user()`. This function is used to copy the data to user space application. In user space application you can use `read()` system call to read the data from the device file.

```
1 static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t *off)
2 {
3     copy_to_user(buf, kernel_buffer, mem_size);
4     printk(KERN_INFO "Data Read : Done!\n");
5     return mem_size;
6 }
```

close()

When we close the device file that will call this function. Here i will free the memory that is allocated by kmalloc using `kfree()`. In user space application you can use `close()` system call to close the device file.

```
1 static int etx_release(struct inode *inode, struct file *file)
2 {
3     kfree(kernel_buffer);
4     printk(KERN_INFO "Device File Closed...!!!\n");
5     return 0;
6 }
```

Full Driver Code

You can download the all codes [Here](#).

```
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
11
12 #define mem_size      1024
13
14 dev_t dev = 0;
15 static struct class *dev_class;
16 static struct cdev etx_cdev;
17 uint8_t *kernel_buffer;
18
19 static int __init etx_driver_init(void);
20 static void __exit etx_driver_exit(void);
21 static int etx_open(struct inode *inode, struct file *file);
22 static int etx_release(struct inode *inode, struct file *file);
23 static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t * off);
24 static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, loff_t * off)
25
26 static struct file_operations fops =
27 {
28     .owner        = THIS_MODULE,
29     .read         = etx_read,
30     .write        = etx_write,
31     .open         = etx_open,
32     .release      = etx_release,
33 };
34
35 static int etx_open(struct inode *inode, struct file *file)
36 {
37     /*Creating Physical memory*/
38     if((kernel_buffer = kmalloc(mem_size , GFP_KERNEL)) == 0){
39         printk(KERN_INFO "Cannot allocate memory in kernel\n");
40         return -1;
41     }
42     printk(KERN_INFO "Device File Opened...!!!\n");
43     return 0;
44 }
45
46 static int etx_release(struct inode *inode, struct file *file)
47 {
48     kfree(kernel_buffer);
49     printk(KERN_INFO "Device File Closed...!!!\n");
50     return 0;
51 }
52
53 static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t *off)
54 {
55     copy_to_user(buf, kernel_buffer, mem_size);
56     printk(KERN_INFO "Data Read : Done!\n");
57     return mem_size;
58 }
59 static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, loff_t
```

```
60  {
61      copy_from_user(kernel_buffer, buf, len);
62      printk(KERN_INFO "Data Write : Done!\n");
63      return len;
64  }
65
66 static int __init etx_driver_init(void)
67 {
68     /*Allocating Major number*/
69     if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
70         printk(KERN_INFO "Cannot allocate major number\n");
71         return -1;
72     }
73     printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
74
75     /*Creating cdev structure*/
76     cdev_init(&etx_cdev,&fops);
77
78     /*Adding character device to the system*/
79     if((cdev_add(&etx_cdev,dev,1)) < 0){
80         printk(KERN_INFO "Cannot add the device to the system\n");
81         goto r_class;
82     }
83
84     /*Creating struct class*/
85     if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
86         printk(KERN_INFO "Cannot create the struct class\n");
87         goto r_class;
88     }
89
90     /*Creating device*/
91     if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
92         printk(KERN_INFO "Cannot create the Device 1\n");
93         goto r_device;
94     }
95     printk(KERN_INFO "Device Driver Insert...Done!!!\n");
96     return 0;
97
98 r_device:
99     class_destroy(dev_class);
100 r_class:
101     unregister_chrdev_region(dev,1);
102     return -1;
103 }
104
105 void __exit etx_driver_exit(void)
106 {
107     device_destroy(dev_class,dev);
108     class_destroy(dev_class);
109     cdev_del(&etx_cdev);
110     unregister_chrdev_region(dev, 1);
111     printk(KERN_INFO "Device Driver Remove...Done!!!\n");
112 }
113
114 module_init(etx_driver_init);
115 module_exit(etx_driver_exit);
116
117 MODULE_LICENSE("GPL");
118 MODULE_AUTHOR("EmbeTronicX <embedtronicx@gmail.com or admin@embedtronicx.com>");
119 MODULE_DESCRIPTION("A simple device driver");
120 MODULE_VERSION("1.4");
```

Building the Device Driver

1. Build the driver by using Makefile (*sudo make*) You can download the Makefile [Here](#).

User Space Application

This application will communicate with the device driver. You can download the all codes (driver, Makefile and application) [Here](#).

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <sys/types.h>
5 #include <sys/stat.h>
6 #include <fcntl.h>
7 #include <unistd.h>
8
9 int8_t write_buf[1024];
10 int8_t read_buf[1024];
11 int main()
12 {
13     int fd;
14     char option;
15     printf("*****\n");
16     printf("*****WWW.EmbeTronicX.com*****\n");
17
18     fd = open("/dev/etx_device", O_RDWR);
19     if(fd < 0) {
20         printf("Cannot open device file...\n");
21         return 0;
22     }
23
24     while(1) {
25         printf("****Please Enter the Option****\n");
26         printf("        1. Write          \n");
27         printf("        2. Read           \n");
28         printf("        3. Exit            \n");
29         printf("*****\n");
30         scanf(" %c", &option);
31         printf("Your Option = %c\n", option);
32
33         switch(option) {
34             case '1':
35                 printf("Enter the string to write into driver :");
36                 scanf(" %[^\t\n]s", write_buf);
37                 printf("Data Writing ...");
38                 write(fd, write_buf, strlen(write_buf)+1);
39                 printf("Done!\n");
40                 break;
41             case '2':
42                 printf("Data Reading ...");
43                 read(fd, read_buf, 1024);
44                 printf("Done!\n\n");
45                 printf("Data = %s\n\n", read_buf);
46                 break;
47             case '3':
48                 close(fd);
```

```

49             exit(1);
50         break;
51     default:
52         printf("Enter Valid option = %c\n",option);
53         break;
54     }
55 }
56 close(fd);
57 }
```

Compile the User Space Application

Use below line in terminal to compile the user space application.

```
gcc -o test_app test_app.c
```

Execution (Output)

As of now, we have driver.ko and test_app. Now we will see the output.

- Load the driver using sudo insmod driver.ko
- Run the application (sudo ./test_app)

```
*****
*****www.EmbedTronicX.com*****
****Please Enter the Option*****
1. Write
2. Read
3. Exit
*****
```

- Select option 1 to write data to driver and write the string (In this case i'm going to write “embedtronicx” to driver.

```
1
Your Option = 1
Enter the string to write into driver :embedtronicx
Data Writing ...Done!
****Please Enter the Option*****
1. Write
2. Read
3. Exit
```

- That “embedtronicx” string got passed to the driver. And driver stored that string in the kernel space. That kernel space was allocated by kmalloc.
- Now select the option 2 to read the data from the device driver.

2

Your Option = 2

Data Reading . . . Done!

Data = embedtronicx

- See now, we got that string “embedtronicx”.

Just see the below image for your clarification.

[linux-device-driver-tutorial](#) [Linux Device Driver Tutorial Part 7 - Linux Device Driver Tutorial Programming](#)

Note : Instead of using user space application, you can use echo and cat command. But one condition. If you are going to use echo and cat command, please allocate the kernel space memory in init function instead of open() function. I wont say why. You have to find the reason. If you found the reason please comment below. You can use dmesg to see the kernel log. ☺

Share this:

[Share on Tumblr](#)[Print](#)[WhatsApp](#)[Pocket](#)[Telegram](#)

pinit_fg_en_rect_gray_20 Linux Device Driver Tutorial Part 7 - Linux Device Driver Tutorial Programming

Like this:

Loading...

[report this ad](#)

 ezoic[report this ad](#)[Sidebar ▾](#)

Device Drivers

IOCTL Tutorial in Linux

Linux Device Driver Tutorial Part 8 - I/O Control in Linux IOCTL()

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 the Part 8 of Linux device driver tutorial. Now we will discuss IOCTL Tutorial in Linux.

Post Contents

- [1 Introduction](#)
- [2 IOCTL Tutorial in Linux](#)
- [3 IOCTL](#)
- [4 Steps involved in IOCTL](#)
 - [4.1 Create IOCTL Command in Driver](#)
 - [4.2 Write IOCTL function in driver](#)
 - [4.3 Create IOCTL command in User space application](#)
 - [4.4 Use IOCTL system call in User space](#)
- [5 Device Driver Source Code](#)
- [6 Application Source Code](#)
- [7 Building Driver and Application](#)
- [8 Execution \(Output\)
 - \[8.0.1 Share this:\]\(#\)
 - \[8.0.2 Like this:\]\(#\)
 - \[8.0.3 Related\]\(#\)](#)

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 IOCTL.

IOCTL Tutorial in Linux

IOCTL

IOCTL is referred as Input and Output Control, which is used to talking to device drivers. This system call, available in most driver categories. The major use of this is in case of handling some specific operations of a device for which the kernel does not have a system call by default.

Some real time applications of ioctl is Ejecting the media from a “cd” drive, to change the Baud Rate of Serial port, Adjust the Volume, Reading or Writing device registers, etc. We already have write and read function in our device driver. But it is not enough for all cases.

Steps involved in IOCTL

There are some steps involved to use IOCTL.

- Create IOCTL command in driver
- Write IOCTL function in driver
- Create IOCTL command in User space application
- Use IOCTL system call in User space

Create IOCTL Command in Driver

To implement a new ioctl command we need to follow the following steps.

1. Define the ioctl code

```
#define      "ioctl      name"      __I0X("magic      number", "command
number", "argument type")
```

where *IOX* can be :

- “IO”: an ioctl with no parameters
- “IOW”: an ioctl with write parameters (*copy_from_user*)
- “IOR”: an ioctl with read parameters (*copy_to_user*)
- “IOWR”: an ioctl with both write and read parameters

- The Magic Number is a unique number or character that will differentiate our set of ioctl calls from the other ioctl calls. sometimes the major number for the device is used here.
- Command Number is the number that is assigned to the ioctl .This is used to differentiate the commands from one another.
- The last is the type of data.

2. Add the header file *linux/ioctl.h* to make use of the above mentioned calls.

Example:

```
1 #include <linux/ioctl.h>
2
3 #define WR_VALUE _IOW('a','a',int32_t*)
4 #define RD_VALUE _IOR('a','b',int32_t*)
```

Write IOCTL function in driver

The next step is to implement the ioctl call we defined in to the corresponding driver. We need to add the ioctl function which has the prototype.

Where

<file> : is the file pointer to the file that was passed by the application.

<cmd> : is the ioctl command that was called from the user space.

<arg> : are the arguments passed from the user space.

With in the function “ioctl” we need to implement all the commands that we defined above. We need to use the same commands in switch statement which is defined above.

Then need to inform the kernel that the ioctl calls are implemented in the function “etx_ioctl”. This is done by making the fops pointer “unlocked_ioctl” to point to “etx_ioctl” as shown below.

Example:

```
1 static long etx_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
2 {
3     switch(cmd) {
4         case WR_VALUE:
5             copy_from_user(&value , (int32_t*) arg, sizeof(value));
6             printk(KERN_INFO "Value = %d\n", value);
7             break;
8         case RD_VALUE:
9             copy_to_user((int32_t*) arg, &value, sizeof(value));
10            break;
```

```

11      }
12      return 0;
13  }
14
15
16 static struct file_operations fops =
17 {
18     .owner = THIS_MODULE,
19     .read = etx_read,
20     .write = etx_write,
21     .open = etx_open,
22     .unlocked_ioctl = etx_ioctl,
23     .release = etx_release,
24 };

```

Now we need to call the new ioctl command from a user application.

Create IOCTL command in User space application

Just define the ioctl command like how we defined in driver.

Example:

```

1 #define WR_VALUE _IOW('a','a',int32_t*)
2 #define RD_VALUE _IOR('a','b',int32_t*)

```

Use IOCTL system call in User space

Include the header file <sys/ioctl.h>. Now we need to call the new ioctl command from a user application.

```
long ioctl( "file descriptor","ioctl command","Arguments");
```

<file descriptor>: This the open file on which the ioctl command needs to be executed, which would generally be device files.

<ioctl command>: ioctl command which is implemented to achieve the desired functionality

<arguments>: The arguments that needs to be passed to the ioctl command.

Example:

```

1 ioctl(fd, WR_VALUE, (int32_t*) &number);
2
3 ioctl(fd, RD_VALUE, (int32_t*) &value);

```

Now we will see the complete driver and application.

Device Driver Source Code

In this example we only implemented IOCTL. In this driver, i defined one variable (int32_t value). Using ioctl command we can read or change the variable. So other functions like open, close, read, write, We simply left empty. Just go through the code below.

driver.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 #include<linux/slab.h>           //kmalloc()
9 #include<linux/uaccess.h>         //copy_to/from_user()
10 #include <linux/ioctl.h>
11
12
13 #define WR_VALUE _IOW('a','a',int32_t*)
14 #define RD_VALUE _IOR('a','b',int32_t*)
15
16 int32_t value = 0;
17
```

```
18 dev_t dev = 0;
19 static struct class *dev_class;
20 static struct cdev etx_cdev;
21
22 static int __init etx_driver_init(void);
23 static void __exit etx_driver_exit(void);
24 static int etx_open(struct inode *inode, struct file *file);
25 static int etx_release(struct inode *inode, struct file *file);
26 static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t * off);
27 static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, loff_t * off)
28 static long etx_ioctl(struct file *file, unsigned int cmd, unsigned long arg);
29
30 static struct file_operations fops =
31 {
32     .owner        = THIS_MODULE,
33     .read         = etx_read,
34     .write        = etx_write,
35     .open          = etx_open,
36     .unlocked_ioctl = etx_ioctl,
37     .release      = etx_release,
38 };
39
40 static int etx_open(struct inode *inode, struct file *file)
41 {
42     printk(KERN_INFO "Device File Opened...!!!\n");
43     return 0;
44 }
45
46 static int etx_release(struct inode *inode, struct file *file)
47 {
48     printk(KERN_INFO "Device File Closed...!!!\n");
49     return 0;
50 }
51
52 static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t *off)
53 {
54     printk(KERN_INFO "Read Function\n");
55     return 0;
56 }
57 static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, loff_t
58 {
59     printk(KERN_INFO "Write function\n");
60     return 0;
61 }
62
63 static long etx_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
64 {
65     switch(cmd) {
66         case WR_VALUE:
67             copy_from_user(&value ,(int32_t*) arg, sizeof(value));
68             printk(KERN_INFO "Value = %d\n", value);
69             break;
70         case RD_VALUE:
71             copy_to_user((int32_t*) arg, &value, sizeof(value));
72             break;
73     }
74     return 0;
75 }
76
77
78 static int __init etx_driver_init(void)
79 {
80     /*Allocating Major number*/
```

```

81         if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
82             printk(KERN_INFO "Cannot allocate major number\n");
83             return -1;
84         }
85         printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
86
87         /*Creating cdev structure*/
88         cdev_init(&etx_cdev,&fops);
89
90         /*Adding character device to the system*/
91         if((cdev_add(&etx_cdev,dev,1)) < 0){
92             printk(KERN_INFO "Cannot add the device to the system\n");
93             goto r_class;
94         }
95
96         /*Creating struct class*/
97         if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
98             printk(KERN_INFO "Cannot create the struct class\n");
99             goto r_class;
100        }
101
102         /*Creating device*/
103         if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
104             printk(KERN_INFO "Cannot create the Device 1\n");
105             goto r_device;
106         }
107         printk(KERN_INFO "Device Driver Insert...Done!!!\n");
108         return 0;
109
110 r_device:
111     class_destroy(dev_class);
112 r_class:
113     unregister_chrdev_region(dev,1);
114     return -1;
115 }
116
117 void __exit etx_driver_exit(void)
118 {
119     device_destroy(dev_class,dev);
120     class_destroy(dev_class);
121     cdev_del(&etx_cdev);
122     unregister_chrdev_region(dev, 1);
123     printk(KERN_INFO "Device Driver Remove...Done!!!\n");
124 }
125
126 module_init(etx_driver_init);
127 module_exit(etx_driver_exit);
128
129 MODULE_LICENSE("GPL");
130 MODULE_AUTHOR("EmbeTronicX <embedtronicx@gmail.com or admin@embedtronicx.com>");
131 MODULE_DESCRIPTION("A simple device driver");
132 MODULE_VERSION("1.5");

```

Makefile:

```

1 obj-m += driver.o
2
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

```

Application Source Code

This application is used to write the value to the driver. Then read the value again.

test_app.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <sys/types.h>
5 #include <sys/stat.h>
6 #include <fcntl.h>
7 #include <unistd.h>
8 #include<sys/ioctl.h>
9
10#define WR_VALUE _IOW('a','a',int32_t*)
11#define RD_VALUE _IOR('a','b',int32_t*)
12
13 int main()
14 {
15     int fd;
16     int32_t value, number;
17     printf("*****\n");
18     printf("*****WWW.EmbedTronicX.com*****\n");
19
20     printf("\nOpening Driver\n");
21     fd = open("/dev/etx_device", O_RDWR);
22     if(fd < 0) {
23         printf("Cannot open device file...\n");
24         return 0;
25     }
26
27     printf("Enter the Value to send\n");
28     scanf("%d",&number);
29     printf("Writing Value to Driver\n");
30     ioctl(fd, WR_VALUE, (int32_t*) &number);
31
32     printf("Reading Value from Driver\n");
33     ioctl(fd, RD_VALUE, (int32_t*) &value);
34     printf("Value is %d\n", value);
35
36     printf("Closing Driver\n");
37     close(fd);
38 }

```

Building Driver and Application

- Build the driver by using Makefile (*sudo make*)
- Use below line in terminal to compile the user space application.

```
gcc -o test_app test_app.c
```

Execution (Output)

As of now, we have driver.ko and test_app. Now we will see the output.

- Load the driver using sudo insmod driver.ko
- Run the application (sudo ./test_app)

```
*****
```

```
*****www.EmbedTronicX.com*****
```

Opening Driver

Enter the Value to send

- Enter the value to pass

23456

Writing Value to Driver

Reading Value from Driver

Value is 23456

Closing Driver

- Now check the value using dmesg

Device File Opened...!!!

Value = 23456

Device File Closed...!!!

- Our value 23456 was passed to the kernel and it was updated.

This is the simple example using ioctl in driver. If you want to send multiple arguments, put those variables into structure and pass the structure.

In our next tutorial we will see other userspace and kernel space communication methods.

Share this:[Share on Tumblr](#)[Tweet](#)[WhatsApp](#)[Pocket](#)

[pinit_fg_en_rect_gray_20 Linux Device Driver Tutorial Part 8 – I/O Control in Linux IOCTL\(\)](#)

Like this:

Loading...



[report this ad](#)

 ezoic[report this ad](#)[Sidebar ▾](#)

Device Drivers

Procfs in Linux

Linux Device Driver Tutorial Part 9 - Procfs in Linux

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 the Part 9 of Linux device driver tutorial. Now we will discuss ProcFS in Linux.

Post Contents

- [1 Introduction](#)
- [2 Procfs in Linux
 - \[2.1 Introduction\]\(#\)](#)
- [3 Creating Procfs Entry](#)
- [4 Procfs File System
 - \[4.1 Open and Release Function\]\(#\)
 - \[4.2 Write Function\]\(#\)
 - \[4.3 Read Function\]\(#\)](#)
- [5 Remove Proc Entry](#)
- [6 Complete Driver Code](#)
- [7 MakeFile](#)
- [8 Building and Testing Driver
 - \[8.0.1 Share this:\]\(#\)
 - \[8.0.2 Like this:\]\(#\)
 - \[8.0.3 Related\]\(#\)](#)

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 Procfs.

Procfs in Linux

Introduction

Many or most Linux users have at least heard of proc. Some of you may wonder why this folder is so important.

On the root, there is a folder titled “proc”. This folder is not really on /dev/sda1 or where ever you think the folder resides. This folder is a mount point for the procfs (Process Filesystem) which is a filesystem in memory. Many processes store information about themselves on this virtual filesystem. ProcFS also stores other system information.

It can act as a bridge connecting the user space and the kernel space. User space program can use proc files to read the information exported by kernel. Every entry in the proc file system provides some information from the kernel.

The entry “*meminfo*” gives the details of the memory being used in the system.

To read the data in this entry just run

```
cat /proc/meminfo
```

Similarly the “*modules*” entry gives details of all the modules that are currently a part of the kernel.

```
cat /proc/modules
```

It gives similar information as lsmod. Like this more proc entries are there.

- /proc/devices — registered character and block major numbers
- /proc/iomem — on-system physical RAM and bus device addresses
- /proc/ioports — on-system I/O port addresses (especially for x86 systems)
- /proc/interrupts — registered interrupt request numbers
- /proc/softirqs — registered soft IRQs
- /proc/swaps — currently active swaps
- /proc/kallsyms — running kernel symbols, including from loaded modules
- /proc/partitions — currently connected block devices and their partitions
- /proc/filesystems — currently active filesystem drivers
- /proc/cpuinfo — information about the CPU(s) on the system

Most proc files are read-only and only expose kernel information to user space programs.

proc files can also be used to control and modify kernel behavior on the fly. The proc files needs to be writable in this case.

For example, to enable IP forwarding of iptable, one can use the command below,

```
echo 1 > /proc/sys/net/ipv4/ip_forward
```

The proc file system is also very useful when we want to debug a kernel module. While debugging we might want to know the values of various variables in the module or may be the data that module is handling. In such situations we can create a proc entry for our selves and dump whatever data we want to look into in the entry.

We will be using the same example character driver that we created in the previous post to create the proc entry.

The proc entry can also be used to pass data to the kernel by writing into the kernel, so there can be two kinds of proc entries.

1. An entry that only reads data from the kernel space.
2. An entry that reads as well as writes data into and from kernel space.

Creating Procfs Entry

The creation of proc entries has undergone a considerable change in kernel version 3.10 and above. In this post we will see one of the methods we can use in linux kernel version 3.10 and above let us see how we can create proc entries in version 3.10 and above.

```
1 static inline struct proc_dir_entry *proc_create(const char *name, umode_t mode,  
2 struct proc_dir_entry *parent,
```

3

const struct file_operations *proc_fops)

The function is defined in proc_fs.h.

Where,

<name> : The name of the proc entry
<mode> : The access mode for proc entry
<parent> : The name of the parent directory under /proc. If NULL is passed as parent, the /proc directory will be set as parent.
<proc_fops> : The structure in which the file operations for the proc entry will be created.

For example to create a proc entry by the name “etx_proc” under /proc the above function will be defined as below,

```
1 proc_create("etx_proc", 0666, NULL, &proc_fops);
```

This proc entry should be created in Driver init function.

If you are using kernel version below 3.10, please use below functions to create proc entry.

```
create_proc_read_entry()  
create_proc_entry()
```

Both of these functions are defined in the file */linux/proc_fs.h*.

The `create_proc_entry` is a generic function that allows to create both read as well as write entries.

`create_proc_read_entry` is a function specific to create only read entries.

Its possible that most of the proc entries are created to read data from the kernel space that is why the kernel developers have provided a direct function to create a read proc entry.

Procfs File System

Now we need to create `file_operations` structure `proc_fops` in which we

can map the read and write functions for the proc entry.

```
1 static struct file_operations proc_fops = {  
2     .open = open_proc,  
3     .read = read_proc,  
4     .write = write_proc,  
5     .release = release_proc  
6 };
```

This is like a device driver file system. We need to register our proc entry filesystem. If you are using kernel version below 3.10, this will not be work. There is a different method.

Next we need to add the all functions to the driver.

Open and Release Function

This functions are optional.

```
1 static int open_proc(struct inode *inode, struct file *file)  
2 {  
3     printk(KERN_INFO "proc file opened.....\t");  
4     return 0;  
5 }  
6  
7 static int release_proc(struct inode *inode, struct file *file)  
8 {  
9     printk(KERN_INFO "proc file released.....\n");  
10    return 0;  
11 }
```

Write Function

The write function will receive data from the user space using the

function copy_from_user into a array "etx_array".

Thus the write function will look as below.

```
1 static ssize_t write_proc(struct file *filp, const char *buff, size_t len, loff_t * off)
2 {
3     printk(KERN_INFO "proc file write.....\t");
4     copy_from_user(etx_array,buff,len);
5     return len;
6 }
```

Read Function

Once data is written to the proc entry we can read from the proc entry using a read function, i.e transfer data to the user space using the function copy_to_user function.

The read function can be as below.

```
1 static ssize_t read_proc(struct file *filp, char __user *buffer, size_t length,loff_t *
2 {
3     printk(KERN_INFO "proc file read.....\n");
4     if(len)
5         len=0;
6     else{
7         len=1;
8         return 0;
9     }
10    copy_to_user(buffer,etx_array,20);
11
12    return length;;
13 }
```

Remove Proc Entry

Proc entry should be removed in Driver exit function using the below function.

```
void remove_proc_entry(const char *name, struct proc_dir_entry
*parent);
```

Example:

```
1 remove_proc_entry("etx_proc",NULL);
```

Complete Driver Code

This code will work for the kernel above 3.10 version. I just took the previous tutorial driver code and update with procfs.

```

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/ioctl.h>
11 #include<linux/proc_fs.h>
12
13 #define WR_VALUE _IOW('a','a',int32_t*)
14 #define RD_VALUE _IOR('a','b',int32_t*)
15
16 int32_t value = 0;
17 char etx_array[20]={"try_proc_array\n"};
18 static int len = 1;
19
20
21 dev_t dev = 0;
22 static struct class *dev_class;
23 static struct cdev etx_cdev;
24
25 static int __init etx_driver_init(void);
26 static void __exit etx_driver_exit(void);
27 /***** Driver Functions *****/
28 static int etx_open(struct inode *inode, struct file *file);
29 static int etx_release(struct inode *inode, struct file *file);
30 static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t * off);
31 static ssize_t etx_write(struct file *filp, const char *buf, size_t len, loff_t * off)
32 static long etx_ioctl(struct file *file, unsigned int cmd, unsigned long arg);
33
34 /***** Procfs Functions *****/
35 static int open_proc(struct inode *inode, struct file *file);
36 static int release_proc(struct inode *inode, struct file *file);
```

```
37 static ssize_t read_proc(struct file *filp, char __user *buffer, size_t length, loff_t
38 static ssize_t write_proc(struct file *filp, const char *buff, size_t len, loff_t * of
39
40 static struct file_operations fops =
41 {
42     .owner        = THIS_MODULE,
43     .read         = etx_read,
44     .write        = etx_write,
45     .open          = etx_open,
46     .unlocked_ioctl = etx_ioctl,
47     .release      = etx_release,
48 };
49
50 static struct file_operations proc_fops = {
51     .open = open_proc,
52     .read = read_proc,
53     .write = write_proc,
54     .release = release_proc
55 };
56
57 static int open_proc(struct inode *inode, struct file *file)
58 {
59     printk(KERN_INFO "proc file opened.....\t");
60     return 0;
61 }
62
63 static int release_proc(struct inode *inode, struct file *file)
64 {
65     printk(KERN_INFO "proc file released.....\n");
66     return 0;
67 }
68
69 static ssize_t read_proc(struct file *filp, char __user *buffer, size_t length, loff_t
70 {
71     printk(KERN_INFO "proc file read.....\n");
72     if(len)
73         len=0;
74     else{
75         len=1;
76         return 0;
77     }
78     copy_to_user(buffer,etx_array,20);
79
80     return length;;
81 }
82
83 static ssize_t write_proc(struct file *filp, const char *buff, size_t len, loff_t * of
84 {
85     printk(KERN_INFO "proc file wrote.....\n");
86     copy_from_user(etx_array,buff,len);
87     return len;
88 }
89
90 static int etx_open(struct inode *inode, struct file *file)
91 {
92     printk(KERN_INFO "Device File Opened...!!!\n");
93     return 0;
94 }
95
96 static int etx_release(struct inode *inode, struct file *file)
97 {
98     printk(KERN_INFO "Device File Closed...!!!\n");
99     return 0;
```

```
100 }
101
102 static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t *off)
103 {
104     printk(KERN_INFO "Readfunction\n");
105     return 0;
106 }
107 static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, loff_t
108 {
109     printk(KERN_INFO "Write Function\n");
110     return 0;
111 }
112
113 static long etx_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
114 {
115     switch(cmd) {
116         case WR_VALUE:
117             copy_from_user(&value ,(int32_t*) arg, sizeof(value));
118             printk(KERN_INFO "Value = %d\n", value);
119             break;
120         case RD_VALUE:
121             copy_to_user((int32_t*) arg, &value, sizeof(value));
122             break;
123     }
124     return 0;
125 }
126
127
128 static int __init etx_driver_init(void)
129 {
130     /*Allocating Major number*/
131     if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
132         printk(KERN_INFO "Cannot allocate major number\n");
133         return -1;
134     }
135     printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
136
137     /*Creating cdev structure*/
138     cdev_init(&etx_cdev,&fops);
139
140     /*Adding character device to the system*/
141     if((cdev_add(&etx_cdev,dev,1)) < 0){
142         printk(KERN_INFO "Cannot add the device to the system\n");
143         goto r_class;
144     }
145
146     /*Creating struct class*/
147     if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
148         printk(KERN_INFO "Cannot create the struct class\n");
149         goto r_class;
150     }
151
152     /*Creating device*/
153     if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
154         printk(KERN_INFO "Cannot create the Device 1\n");
155         goto r_device;
156     }
157
158     /*Creating Proc entry*/
159     proc_create("etx_proc",0666,NULL,&proc_fops);
160
161     printk(KERN_INFO "Device Driver Insert...Done!!!\n");
162     return 0;
```

```

163
164 r_device:
165     class_destroy(dev_class);
166 r_class:
167     unregister_chrdev_region(dev,1);
168     return -1;
169 }
170
171 void __exit etx_driver_exit(void)
172 {
173     remove_proc_entry("etx_proc",NULL);
174     device_destroy(dev_class,dev);
175     class_destroy(dev_class);
176     cdev_del(&etx_cdev);
177     unregister_chrdev_region(dev, 1);
178     printk(KERN_INFO "Device Driver Remove...Done!!!\n");
179 }
180
181 module_init(etx_driver_init);
182 module_exit(etx_driver_exit);
183
184 MODULE_LICENSE("GPL");
185 MODULE_AUTHOR("EmbeTronicX <embetronicx@gmail.com or admin@embetronicx.com>");
186 MODULE_DESCRIPTION("A simple device driver");
187 MODULE_VERSION("1.6");

```

MakeFile

```

1 obj-m += driver.o
2
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 our procfs entry using *ls* in procfs directory

linux@embetronicx-VirtualBox:ls /proc/

<i>filesystems</i>	<i>iomem</i>	<i>kallsyms</i>	<i>modules</i>	<i>partitions</i>
--------------------	--------------	-----------------	----------------	-------------------

- Now our procfs entry is there under /proc directory.
- Now you can read procfs variable using *cat*.

```
linux@embetronicx-VirtualBox: cat /proc/etx_proc
```

try_proc_array

- We initialized the etx_array with “try_proc_array”. That’s why we got “try_proc_array”.
- Now do proc write using echo command and check using cat.

```
linux@embetronicx-VirtualBox: echo "device driver proc" >
/proc/etx_proc
```

```
linux@embetronicx-VirtualBox: cat /proc/etx_proc
```

device driver proc

- We got the same string which was passed to the driver using procfs.

This is the simple example using procfs in device driver. This is just a basic. I hope this might helped you.

Share this:

[Share on Tumblr](#) [Tweet](#)



[Print](#) [WhatsApp](#)

[Pocket](#)



[pinit_fg_en_rect_gray_20](#) [Linux Device Driver Tutorial Part 9 - Procfs in Linux](#)

Like this:

Loading...



[report this ad](#)



 ezoic

[report this ad](#)

Sidebar▼

Device Drivers

Sysfs in Linux Tutorial

**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 the Part 11 of Linux device driver tutorial. In our previous tutorial we have seen the [Procfs](#). Now we will see SysFS in Linux kernel Tutorial.

Post Contents

- [1 Introduction](#)
- [2 SysFS in Linux Kernel Tutorial](#)
- [3 Introduction](#)
- [4 Kernel Objects](#)
- [5 SysFS in Linux
 - \[5.1 Create directory in /sys
 - \\[5.1.1 Example\\]\\(#\\)\]\(#\)
 - \[5.2 Create Sysfs file
 - \\[5.2.1 Create attribute\\]\\(#\\)
 - \\[5.2.2 Store and Show functions\\]\\(#\\)
 - \\[5.2.3 Create sysfs file\\]\\(#\\)\]\(#\)](#)
- [6 Complete Driver Code](#)
- [7 MakeFile](#)
- [8 Building and Testing Driver
 - \[8.0.1 Share this:\]\(#\)
 - \[8.0.2 Like this:\]\(#\)
 - \[8.0.3 Related\]\(#\)](#)

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

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 used 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,

```
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;
10    struct kobj_type *ktype;
11    struct dentry *dentry;
12 };
```

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.

```
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.

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

```

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 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,

```

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 *buf,
5 };

```

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 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, siz
```

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.

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`

```
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
17 /*Creating a directory in /sys/kernel/
18 kobj_ref = kobject_create_and_add("etx_sysfs",kernel_kobj);
19
20 /*Creating sysfs file for etx_value*/
21 if(sysfs_create_file(kobj_ref,&etx_attr.attr)){
```

```

22     printk(KERN_INFO "Cannot create sysfs file.....\n");
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 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.

```

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);
38
39 struct kobj_attribute etx_attr = __ATTR(etx_value, 0660, sysfs_show, sysfs_store);
40
41 static struct file_operations fops =
42 {
43     .owner      = THIS_MODULE,
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
100    /*Creating cdev structure*/
101    cdev_init(&etx_cdev,&fops);
102
103    /*Adding character device to the system*/
104    if((cdev_add(&etx_cdev,dev,1)) < 0){
105        printk(KERN_INFO "Cannot add the device to the system\n");
106        goto r_class;
107    }
```

```

108     /*Creating struct class*/
109     if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
110         printk(KERN_INFO "Cannot create the struct class\n");
111         goto r_class;
112     }
113
114     /*Creating device*/
115     if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
116         printk(KERN_INFO "Cannot create the Device 1\n");
117         goto r_device;
118     }
119
120     /*Creating a directory in /sys/kernel/ */
121     kobj_ref = kobject_create_and_add("etx_sysfs",kernel_kobj);
122
123     /*Creating sysfs file for etx_value*/
124     if(sysfs_create_file(kobj_ref,&etx_attr.attr)){
125         printk(KERN_INFO "Cannot create sysfs file.....\n");
126         goto r_sysfs;
127     }
128     printk(KERN_INFO "Device Driver Insert...Done!!!\n");
129     return 0;
130
131 r_sysfs:
132     kobject_put(kobj_ref);
133     sysfs_remove_file(kernel_kobj, &etx_attr.attr);
134
135 r_device:
136     class_destroy(dev_class);
137
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 <embedtronicx@gmail.com or admin@embedtronicx.com>");
160 MODULE_DESCRIPTION("A simple device driver - SysFs");
161 MODULE_VERSION("1.8");

```

MakeFile

```

1 obj-m += driver.o
2
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*

```
linux@embedtronicx-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--- 1 root root 4096 Dec 17 14:11 fscaps
drwxr-xr-x 2 root root 0 Dec 17 14:11 iommu_groups
-r--r--- 1 root root 4096 Dec 17 14:11 kexec_crash_loaded
-rw-r--- 1 root root 4096 Dec 17 14:11 kexec_crash_size
-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--- 1 root root 516 Dec 17 14:11 notes
-rw-r--- 1 root root 4096 Dec 17 14:11 profiling
-rw-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--- 1 root root 4096 Dec 17 12:19 uevent_helper
-r--r--- 1 root root 4096 Dec 17 12:19 uevent_seqnum
-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@embedtronicx-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@embedtronicx-VirtualBox#cat /sys/kernel/etx_sysfs/etx_value
0
```

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

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

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

```
linux@embedtronicx-VirtualBox#cat /sys/kernel/etx_sysfs/etx_value
123
```

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

This is the simple example using sysfs in device driver. This is just a basic. I hope this might helped you.

Share this:[Share on Tumblr](#)[WhatsApp](#)[Pocket](#)

[pinit_fg_en_rect_gray_20](#) [Linux Device Driver Tutorial Part 11 – Sysfs in Linux Kernel](#)

Like this:

Loading...



[report this ad](#)

 ezoic[report this ad](#)[Sidebar ▾](#)

📁 Device Drivers

WaitQueue in Linux

Linux Device Driver Tutorial Part 10 - WaitQueue in Linux

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 the Part 10 of Linux device driver tutorial. Now we will discuss WaitQueue in Linux.

Post Contents

- [1 WaitQueue in Linux](#)
- [2 Introduction](#)
- [3 Initializing Waitqueue
 - \[3.1 Static Method\]\(#\)
 - \[3.2 Dynamic Method\]\(#\)](#)
- [4 Queuing
 - \[4.1 wait_event\]\(#\)
 - \[4.2 wait_event_timeout\]\(#\)
 - \[4.3 wait_event_cmd\]\(#\)
 - \[4.4 wait_event_interruptible\]\(#\)
 - \[4.5 wait_event_interruptible_timeout\]\(#\)
 - \[4.6 wait_event_killable\]\(#\)](#)
- [5 Waking Up Queued Task
 - \[5.1 wake_up\]\(#\)
 - \[5.2 wake_up_all\]\(#\)
 - \[5.3 wake_up_interruptible\]\(#\)
 - \[5.4 wake_up_sync and wake_up_interruptible_sync\]\(#\)](#)
- [6 Driver Source Code - WaitQueue in Linux
 - \[6.1 Waitqueue created by Static Method\]\(#\)
 - \[6.2 Waitqueue created by Dynamic Method\]\(#\)](#)
- [7 MakeFile](#)

8 Building and Testing Driver

8.0.1 Share this:

8.0.2 Like this:

8.0.3 Related

WaitQueue in Linux

Introduction

When you write a Linux Driver or Module or Kernel Program, Some process should be wait or sleep for some event. There are several ways of handling sleeping and waking up in Linux, each suited to different needs. Waitqueue also one of the method to handle that case.

Whenever a process must wait for an event (such as the arrival of data or the termination of a process), it should go to sleep. Sleeping causes the process to suspend execution, freeing the processor for other uses. After some time, the process will be woken up and will continue with its job when the event which we are waiting will be occurred.

Wait queue is a mechanism provided in kernel to implement the wait. As the name itself suggests, wait queue is the list of processes waiting for an event. In other words, A wait queue is used to wait for someone to wake you up when a certain condition is true. They must be used carefully to ensure there is no race condition.

There are 3 important steps in Waitqueue.

1. Initializing Waitqueue
2. Queuing (Put the Task to sleep until the event comes)
3. Waking Up Queued Task

Initializing Waitqueue

Use this Header file for Waitqueue (`include/linux/wait.h`). There are two ways to initialize the Waitqueue.

1. Static method
2. Dynamic method

You can use any one of the method.

Static Method

```
1 DECLARE_WAIT_QUEUE_HEAD(wq);
```

Where the “wq” is the name of the queue on which task will be put to sleep.

Dynamic Method

```
1 wait_queue_head_t wq;
2 init_waitqueue_head (&wq);
```

You can create waitqueue using any one of the above method.

Queuing

Once the wait queue is declared and initialized, a process may use it to go to sleep. There are several macros are available for different uses. We will see one by one.

1. **wait_event**
2. **wait_event_timeout**
3. **wait_event_cmd**
4. **wait_event_interruptible**
5. **wait_event_interruptible_timeout**
6. **wait_event_killable**

Old kernel versions used the functions `sleep_on()` and `interruptible_sleep_on()`, but those two functions can introduce bad race conditions and should not be used.

Whenever we use the above one of the macro, it will add that task to the waitqueue which is created by us. Then it will wait for the event.

wait_event

sleep until a condition gets true.

```
wait_event(wq, condition);
```

wq – the waitqueue to wait on

condition – a C expression for the event to wait for

The process is put to sleep (`TASK_UNINTERRUPTIBLE`) until the *condition* evaluates to true. The *condition* is checked each time the waitqueue *wq* is woken up.

wait_event_timeout

sleep until a condition gets true or a timeout elapses

```
wait_event_timeout(wq, condition, timeout);
```

wq – the waitqueue to wait on

condition - a C expression for the event to wait for

timeout - timeout, in jiffies

The process is put to sleep (TASK_UNINTERRUPTIBLE) until the *condition* evaluates to true or *timeout* elapses. The *condition* is checked each time the waitqueue *wq* is woken up.

It **returns 0** if the *condition* evaluated to false after the *timeout* elapsed, **1** if the *condition* evaluated to true after the *timeout* elapsed, or the **remaining jiffies** (at least 1) if the *condition* evaluated to true before the *timeout* elapsed.

wait_event_cmd

sleep until a condition gets true

```
wait_event_cmd(wq, condition, cmd1, cmd2);
```

wq - the waitqueue to wait on

condtion - a C expression for the event to wait for

cmd1 - the command will be executed before sleep

cmd2 - the command will be executed after sleep

The process is put to sleep (TASK_UNINTERRUPTIBLE) until the *condition* evaluates to true. The *condition* is checked each time the waitqueue *wq* is woken up.

wait_event_interruptible

sleep until a condition gets true

```
wait_event_interruptible(wq, condition);
```

wq - the waitqueue to wait on

condition - a C expression for the event to wait for

The process is put to sleep (TASK_INTERRUPTIBLE) until the *condition* evaluates to true or a signal is received. The *condition* is checked each time the waitqueue *wq* is woken up.

The function will **return** -ERESTARTSYS if it was interrupted by a signal and 0 if *condition* evaluated to true.

wait_event_interruptible_timeout

sleep until a condition gets true or a timeout elapses

```
wait_event_interruptible_timeout(wq, condition, timeout);
```

wq - the waitqueue to wait on

condtion - a C expression for the event to wait for

timeout - timeout, in jiffies

The process is put to sleep (TASK_INTERRUPTIBLE) until the *condition* evaluates to true or a signal is received or timeout elapses. The *condition* is checked each time the waitqueue *wq* is woken up.

It **returns**, 0 if the *condition* evaluated to false after the *timeout* elapsed, 1 if the *condition* evaluated to true after the *timeout* elapsed, the **remaining jiffies** (at least 1) if the *condition* evaluated to true before the *timeout* elapsed, or -ERESTARTSYS if it was interrupted by a signal.

wait_event_killable

sleep until a condition gets true

```
wait_event_killable(wq, condition);
```

wq - the waitqueue to wait on

condition - a C expression for the event to wait for

The process is put to sleep (TASK_KILLABLE) until the *condition* evaluates to true or a signal is received. The *condition* is checked each time the waitqueue *wq* is woken up.

The function will **return** -ERESTARTSYS if it was interrupted by a signal and **0** if *condition* evaluated to true.

Waking Up Queued Task

When some Tasks are in sleep mode because of waitqueue, then we can use the below function to wake up those tasks.

1. **wake_up**
2. **wake_up_all**
3. **wake_up_interruptible**
4. **wake_up_sync and wake_up_interruptible_sync**

wake_up

wakes up only one process from the wait queue which is in non interruptible sleep.

```
wake_up(&wq);
```

wq - the waitqueue to wake up

wake_up_all

wakes up all the processes on the wait queue

```
wake_up_all(&wq);
```

wq – the waitqueue to wake up

wake_up_interruptible

wakes up only one process from the wait queue that is in interruptible sleep

```
wake_up_interruptible(&wq);
```

wq – the waitqueue to wake up

wake_up_sync and wake_up_interruptible_sync

```
wake_up_sync(&wq);
```

```
wake_up_interruptible_sync(&wq);
```

Normally, a *wake_up* call can cause an immediate reschedule to happen, meaning that other processes might run before *wake_up* returns. The “synchronous” variants instead make any awakened processes runnable, but do not reschedule the CPU. This is used to avoid rescheduling when the current process is known to be going to sleep, thus forcing a reschedule anyway. Note that awakened processes could run immediately on a different processor, so these functions should not be expected to provide mutual exclusion.

Driver Source Code - WaitQueue in Linux

First i will explain you the concept of driver code.

In this source code, two places we are sending wake up. One from read function and another one from driver exit function.

I've created one thread (*wait_function*) which has *while(1)*. That thread

will always wait for the event. It will be sleep until it gets wake up call. When it gets the wake up call, it will check the condition. If condition is 1 then the wakeup came from read function. If it is 2, then the wakeup came from exit function. If wake up came from read, it will print the read count and it will again wait. If its from exit function, it will exit from the thread.

Here I've added two versions of code.

1. Waitqueue created by static method
2. Waitqueue created by dynamic method

But operation wise both are same.

Waitqueue created by Static Method

```

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
11 #include <linux/kthread.h>
12 #include <linux/wait.h>          // Required for the wait queues
13
14
15 uint32_t read_count = 0;
16 static struct task_struct *wait_thread;
17
18 DECLARE_WAIT_QUEUE_HEAD(wait_queue_etx);
19
20 dev_t dev = 0;
21 static struct class *dev_class;
22 static struct cdev etx_cdev;
23 int wait_queue_flag = 0;
24
25 static int __init etx_driver_init(void);
26 static void __exit etx_driver_exit(void);
27
28 /***** Driver Fuctions *****/
29 static int etx_open(struct inode *inode, struct file *file);
30 static int etx_release(struct inode *inode, struct file *file);
31 static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t * off);
32 static ssize_t etx_write(struct file *filp, const char *buf, size_t len, loff_t * off)
33
34 static struct file_operations fops =
35 {
36     .owner      = THIS_MODULE,
37     .read       = etx_read,
38     .write      = etx_write,

```

```

39         .open          = etx_open,
40         .release       = etx_release,
41     };
42
43 static int wait_function(void *unused)
44 {
45
46     while(1) {
47         printk(KERN_INFO "Waiting For Event...\n");
48         wait_event_interruptible(wait_queue_etx, wait_queue_flag != 0 );
49         if(wait_queue_flag == 2) {
50             printk(KERN_INFO "Event Came From Exit Function\n");
51             return 0;
52         }
53         printk(KERN_INFO "Event Came From Read Function - %d\n", ++read_count)
54         wait_queue_flag = 0;
55     }
56     do_exit(0);
57     return 0;
58 }
59
60
61
62 static int etx_open(struct inode *inode, struct file *file)
63 {
64     printk(KERN_INFO "Device File Opened...!!!\n");
65     return 0;
66 }
67
68 static int etx_release(struct inode *inode, struct file *file)
69 {
70     printk(KERN_INFO "Device File Closed...!!!\n");
71     return 0;
72 }
73
74 static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t *off)
75 {
76     printk(KERN_INFO "Read Function\n");
77     wait_queue_flag = 1;
78     wake_up_interruptible(&wait_queue_etx);
79     return 0;
80 }
81 static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, loff_t
82 {
83     printk(KERN_INFO "Write function\n");
84     return 0;
85 }
86
87
88
89
90 static int __init etx_driver_init(void)
91 {
92     /*Allocating Major number*/
93     if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
94         printk(KERN_INFO "Cannot allocate major number\n");
95         return -1;
96     }
97     printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
98
99     /*Creating cdev structure*/
100    cdev_init(&etx_cdev,&fops);
101    etx_cdev.owner = THIS_MODULE;

```

```

102         etx_cdev.ops = &fops;
103
104     /*Adding character device to the system*/
105     if((cdev_add(&etx_cdev,dev,1)) < 0){
106         printk(KERN_INFO "Cannot add the device to the system\n");
107         goto r_class;
108     }
109
110     /*Creating struct class*/
111     if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
112         printk(KERN_INFO "Cannot create the struct class\n");
113         goto r_class;
114     }
115
116     /*Creating device*/
117     if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
118         printk(KERN_INFO "Cannot create the Device 1\n");
119         goto r_device;
120     }
121
122     //Initialize wait queue
123     init_waitqueue_head(&wait_queue_etx);
124
125     //Create the kernel thread with name 'mythread'
126     wait_thread = kthread_create(wait_function, NULL, "WaitThread");
127     if (wait_thread) {
128         printk("Thread Created successfully\n");
129         wake_up_process(wait_thread);
130     } else
131         printk(KERN_INFO "Thread creation failed\n");
132
133     printk(KERN_INFO "Device Driver Insert...Done!!!\n");
134     return 0;
135
136 r_device:
137     class_destroy(dev_class);
138 r_class:
139     unregister_chrdev_region(dev,1);
140     return -1;
141 }
142
143 void __exit etx_driver_exit(void)
144 {
145     wait_queue_flag = 2;
146     wake_up_interruptible(&wait_queue_etx);
147     device_destroy(dev_class,dev);
148     class_destroy(dev_class);
149     cdev_del(&etx_cdev);
150     unregister_chrdev_region(dev, 1);
151     printk(KERN_INFO "Device Driver Remove...Done!!!\n");
152 }
153
154 module_init(etx_driver_init);
155 module_exit(etx_driver_exit);
156
157 MODULE_LICENSE("GPL");
158 MODULE_AUTHOR("EmbeTronicX <embetronicx@gmail.com or admin@embetronicx.com>");
159 MODULE_DESCRIPTION("A simple device driver");
160 MODULE_VERSION("1.7");

```

Waitqueue created by Dynamic Method

```
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
11 #include <linux/kthread.h>
12 #include <linux/wait.h>          // Required for the wait queues
13
14
15 uint32_t read_count = 0;
16 static struct task_struct *wait_thread;
17
18 dev_t dev = 0;
19 static struct class *dev_class;
20 static struct cdev etx_cdev;
21 wait_queue_head_t wait_queue_etx;
22 int wait_queue_flag = 0;
23
24 static int __init etx_driver_init(void);
25 static void __exit etx_driver_exit(void);
26
27 /***** Driver Fuctions *****/
28 static int etx_open(struct inode *inode, struct file *file);
29 static int etx_release(struct inode *inode, struct file *file);
30 static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t * off);
31 static ssize_t etx_write(struct file *filp, const char *buf, size_t len, loff_t * off)
32
33
34 static struct file_operations fops =
35 {
36     .owner        = THIS_MODULE,
37     .read         = etx_read,
38     .write        = etx_write,
39     .open         = etx_open,
40     .release      = etx_release,
41 };
42
43
44 static int wait_function(void *unused)
45 {
46
47     while(1) {
48         printk(KERN_INFO "Waiting For Event...\n");
49         wait_event_interruptible(wait_queue_etx, wait_queue_flag != 0 );
50         if(wait_queue_flag == 2) {
51             printk(KERN_INFO "Event Came From Exit Function\n");
52             return 0;
53         }
54         printk(KERN_INFO "Event Came From Read Function - %d\n", ++read_count)
55         wait_queue_flag = 0;
56     }
57     return 0;
58 }
59
60
61
62 static int etx_open(struct inode *inode, struct file *file)
63 {
```

```

64         printk(KERN_INFO "Device File Opened...!!!\n");
65         return 0;
66     }
67
68     static int etx_release(struct inode *inode, struct file *file)
69     {
70         printk(KERN_INFO "Device File Closed...!!!\n");
71         return 0;
72     }
73
74     static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t *off)
75     {
76         printk(KERN_INFO "Read Function\n");
77         wait_queue_flag = 1;
78         wake_up_interruptible(&wait_queue_etx);
79         return 0;
80     }
81     static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, loff_t
82     {
83         printk(KERN_INFO "Write function\n");
84         return 0;
85     }
86
87
88
89
90     static int __init etx_driver_init(void)
91     {
92         /*Allocating Major number*/
93         if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
94             printk(KERN_INFO "Cannot allocate major number\n");
95             return -1;
96         }
97         printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
98
99         /*Creating cdev structure*/
100        cdev_init(&etx_cdev,&fops);
101
102        /*Adding character device to the system*/
103        if((cdev_add(&etx_cdev,dev,1)) < 0){
104            printk(KERN_INFO "Cannot add the device to the system\n");
105            goto r_class;
106        }
107
108        /*Creating struct class*/
109        if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
110            printk(KERN_INFO "Cannot create the struct class\n");
111            goto r_class;
112        }
113
114        /*Creating device*/
115        if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
116            printk(KERN_INFO "Cannot create the Device 1\n");
117            goto r_device;
118        }
119
120        //Initialize wait queue
121        init_waitqueue_head(&wait_queue_etx);
122
123        //Create the kernel thread with name 'mythread'
124        wait_thread = kthread_create(wait_function, NULL, "WaitThread");
125        if (wait_thread) {
126            printk("Thread Created successfully\n");

```

```
127         wake_up_process(wait_thread);
128     } else
129         printk(KERN_INFO "Thread creation failed\n");
130
131     printk(KERN_INFO "Device Driver Insert...Done!!!\n");
132     return 0;
133
134 r_device:
135     class_destroy(dev_class);
136 r_class:
137     unregister_chrdev_region(dev,1);
138     return -1;
139 }
140
141 void __exit etx_driver_exit(void)
142 {
143     wait_queue_flag = 2;
144     wake_up_interruptible(&wait_queue_etx);
145     device_destroy(dev_class,dev);
146     class_destroy(dev_class);
147     cdev_del(&etx_cdev);
148     unregister_chrdev_region(dev, 1);
149     printk(KERN_INFO "Device Driver Remove...Done!!!\n");
150 }
151
152 module_init(etx_driver_init);
153 module_exit(etx_driver_exit);
154
155 MODULE_LICENSE("GPL");
156 MODULE_AUTHOR("EmbeTronicX <embetronicx@gmail.com or admin@embetronicx.com>");
157 MODULE_DESCRIPTION("A simple device driver");
158 MODULE_VERSION("1.7");
```

MakeFile

```
1 obj-m += driver.o
2 KDIR = /lib/modules/$(shell uname -r)/build
3 all:
4     make -C $(KDIR) M=$(shell pwd) modules
5 clean:
6     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*
- Then Check the Dmesg

Major = 246 Minor = 0

Thread Created successfully

Device Driver Insert...Done!!!

Waiting For Event...

- So that thread is waiting for the event. Now we will send the event by reading the driver using *sudo cat /dev/etx_device*
- Now check the dmesg

Device File Opened...!!!

Read Function

Event Came From Read Function - 1

Waiting For Event...

Device File Closed...!!!

- We send the wake up from read function, So it will print the read count and then again it will sleep. Now send the event from exit function by *sudo rmmod driver*

Event Came From Exit Function

Device Driver Remove...Done!!!

- Now the condition was 2. So it will return from the thread and remove the driver.

Share this:

[Share on Tumblr](#) [Tweet](#)



[WhatsApp](#)

[Pocket](#)



pinit_fg_en_rect_gray_20 Linux Device Driver Tutorial Part 10 - WaitQueue in Linux

Like this:

Loading...



 ezoic[report this ad](#)[Sidebar ▾](#)

📁 Device Drivers

Interrupts in Linux kernel

Linux Device Driver Tutorial Part 12 - Interrupts 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 the Part 12 of Linux device driver tutorial. In our previous tutorial we have seen the [Sysfs](#). Now we will see about Interrupts in Linux kernel.

Post Contents

- [1 Interrupts in Linux kernel](#)
- [2 Interrupts](#)
- [3 Polling vs Interrupts](#)
- [4 What will happen when interrupt came?](#)
- [5 Interrupts and Exceptions](#)
 - [5.1 Interrupts](#)
 - [5.2 Exceptions](#)
- [6 Interrupt handler](#)
- [7 Process Context and Interrupt Context](#)
- [8 Top halves and Bottom halves](#)
 - [8.1 Top half](#)
 - [8.2 Bottom half](#)
 - [8.2.1 Share this:](#)
 - [8.2.2 Like this:](#)
 - [8.2.3 Related](#)

Interrupts in Linux kernel

This tutorial discusses interrupts and how the kernel responds to them, with special functions called interrupt handlers (ISR).

Interrupts

Here are some analogies to everyday life, suitable even for the computer-illiterate. Suppose you knew one or more guests could be arriving at the door. Polling would be like going to the door often to see if someone was there yet continuously.

That's what the doorbell is for. The guests are coming, but you have preparations to make, or maybe something unrelated that you need to do. You only go to the door when the doorbell rings. When doorbell rings, it's time to check the door again. You get more done, and they get quicker response when they ring the doorbell. This is interrupt mechanism.

[interrupts-in-linux-300x229](#) Linux Device Driver Tutorial Part 12 -
Interrupts in Linux Kernel

Another scenario is, Imagine that you are watching TV or doing something. Suddenly you heard someone's voice which is like your Crush's voice. What will happen next? That's it, you are interrupted!! You will be very happy. Then stop your work whatever you are doing now and go outside to see him/her.

Similar to us, Linux also stops his current work and distracted because of interrupts and then it will handle them.

In Linux, interrupt signals are the distraction which diverts processor to a

new activity outside from normal flow of execution. This new activity is called interrupt handler or interrupt service routine (ISR) and that distraction is Interrupts.

Polling vs Interrupts

POLLING	INTERRUPT
In polling the CPU keeps on checking all the hardwares of the availability of any request	In interrupt the CPU takes care of the hardware only when the hardware requests for some service
The polling method is like a salesperson. The salesman goes from door to door while requesting to buy a product or service. Similarly, the controller keeps monitoring the flags or signals one by one for all devices and provides service to whichever component that needs its service.	An interrupt is like a shopkeeper. If one needs a service or product, he goes to him and apprises him of his needs. In case of interrupts, when the flags or signals are received, they notify the controller that they need to be serviced.

What will happen when interrupt came?

An interrupt is produced by electronic signals from hardware devices and directed into input pins on an interrupt controller (a simple chip that multiplexes multiple interrupt lines into a single line to the processor). These are the process that will be done by kernel.

1. Upon receiving an interrupt, the interrupt controller sends a signal to the processor.
2. The processor detects this signal and interrupts its current execution to handle the interrupt.
3. The processor can then notify the operating system that an interrupt has occurred, and the operating system can handle the interrupt appropriately.

Different devices are associated with different interrupts using a unique value associated with each interrupt. This enables the operating system to differentiate between interrupts and to know which hardware device caused which interrupt. In turn, the operating system can service each interrupt with its corresponding handler.

Interrupt handling is amongst the most sensitive tasks performed by kernel and it must satisfy following:

1. Hardware devices generate interrupts asynchronously (with respect to the processor clock). That means interrupts can come anytime.
2. Because interrupts can come anytime, the kernel might be handling one of them while another one (of a different type) occurs.
3. Some critical regions exist inside the kernel code where interrupts must be disabled. Such critical regions must be limited as much as possible.

Interrupts and Exceptions

Exceptions are often discussed at the same time as interrupts. Unlike interrupts, exceptions occur synchronously with respect to the processor clock; they are often called synchronous interrupts. Exceptions are produced by the processor while executing instructions either in response to a programming error (e.g. divide by zero) or abnormal conditions that must be handled by the kernel (e.g. a page fault). Because many processor architectures handle exceptions in a similar manner to interrupts, the kernel infrastructure for handling the two is similar.

Simple definitions of the two:

Interrupts: asynchronous interrupts generated by hardware.

Exceptions: synchronous interrupts generated by the processor.

System calls (one type of exception) on the x86 architecture are implemented by the issuance of a software interrupt, which traps into the kernel and causes execution of a special system call handler. Interrupts work in a similar way, except hardware (not software) issues interrupts.

There is a further classification of interrupts and exceptions.

Interrupts

Maskable – All Interrupt Requests (IRQs) issued by I/O devices give rise to maskable interrupts. A maskable interrupt can be in two states: masked or unmasked; a masked interrupt is ignored by the control unit as long as it remains masked.

Nonmaskable – Only a few critical events (such as hardware failures) give rise to nonmaskable interrupts. Nonmaskable interrupts are always recognized by the CPU.

Exceptions

Faults – Like Divide by zero, Page Fault, Segmentation Fault.

Traps – Reported immediately following the execution of the trapping instruction. Like Breakpoints.

Aborts – Aborts are used to report severe errors, such as hardware failures and invalid or inconsistent values in system tables.

For a device's each interrupt, its device driver must register an interrupt handler.

Interrupt handler

An interrupt handler or interrupt service routine (ISR) is the function that the kernel runs in response to a specific interrupt:

1. Each device that generates interrupts has an associated interrupt handler.
2. The interrupt handler for a device is part of the device's driver (the kernel code that manages the device).

In Linux, interrupt handlers are normal C functions, which match a specific prototype and thus enables the kernel to pass the handler information in a standard way. What differentiates interrupt handlers from other kernel functions is that the kernel invokes them in response to interrupts and that they run in a special context called interrupt context. This special context is occasionally called atomic context because code executing in this context is unable to block.

Because an interrupt can occur at any time, an interrupt handler can be executed at any time. It is imperative that the handler runs quickly, to resume execution of the interrupted code as soon as possible. It is important that

1. To the hardware: the operating system services the interrupt without delay.
2. To the rest of the system: the interrupt handler executes in as short a period as possible.

At the very least, an interrupt handler's job is to acknowledge the interrupt's receipt to the hardware. However, interrupt handlers can often have a large amount of work to perform.

Process Context and Interrupt Context

The kernel accomplishes useful work using a combination of process contexts and interrupt contexts. Kernel code that services system calls issued by user applications runs on behalf of the corresponding application processes and is said to execute in process context. Interrupt handlers, on the other hand, run asynchronously in interrupt context. Processes contexts are not tied to any interrupt context and vice versa.

Kernel code running in process context is preemptible. An interrupt

context, however, always runs to completion and is not preemptible. Because of this, there are restrictions on what can be done from interrupt context. Code executing from interrupt context cannot do the following:

1. Go to sleep or relinquish the processor
2. Acquire a mutex
3. Perform time-consuming tasks
4. Access user space virtual memory

Based on our idea, ISR or Interrupt Handler should be execute very quickly and it should not run for more time (it should not perform time-consuming tasks). What if, I want to do huge amount of work upon receiving interrupt? So it is a problem right? If we do like that this will happen.

1. While ISR run, it doesn't let other interrupts to run (interrupts with higher priority will run).
2. Interrupts with same type will be missed.

To eliminate that problem, the processing of interrupts is split into two parts, or halves:

1. Top halves
2. Bottom halves

Top halves and Bottom halves

Top half

The interrupt handler is the top half. The top half will run immediately upon receipt of the interrupt and performs only the work that is time-critical, such as acknowledging receipt of the interrupt or resetting the hardware.

Bottom half

A bottom half is used to process data, letting the top half to deal with new incoming interrupts. Interrupts are enabled when a bottom half runs. Interrupt can be disabled if necessary, but generally this should be avoided as this goes against the basic purpose of having a bottom half - processing data while listening to new interrupts. The bottom half runs

in the future, at a more convenient time, with all interrupts enabled.

For example using the network card:

- When network cards receive packets from the network, the network cards immediately issue an interrupt. This optimizes network throughput and latency and avoids timeouts.
- The kernel responds by executing the network card's registered interrupt.
- The interrupt runs, acknowledges the hardware, copies the new networking packets into main memory, and readies the network card for more packets. These jobs are the important, time-critical, and hardware-specific work.
- The kernel generally needs to quickly copy the networking packet into main memory because the network data buffer on the networking card is fixed and minuscule in size, particularly compared to main memory. Delays in copying the packets can result in a buffer overrun, with incoming packets overwhelming the networking card's buffer and thus packets being dropped.
- After the networking data is safely in the main memory, the interrupt's job is done, and it can return control of the system to whatever code was interrupted when the interrupt was generated.
- The rest of the processing and handling of the packets occurs later, in the bottom half.

If the interrupt handler function could process and acknowledge interrupts within few microseconds consistently, then absolutely there is no need for top half/bottom half delegation.

There are 4 bottom half mechanisms are available in Linux:

1. Work-queue
2. Threaded IRQs
3. Softirqs
4. Tasklets

You can see the Bottom Half tutorials [Here](#). In our [Next tutorial](#), you can see the programming of Interrupts with ISR.

Share this:

[Share on Tumblr](#) [Tweet](#)



[Pocket](#)



[pinit_fg_en_rect_gray_20](#) [Linux Device Driver Tutorial Part 12 -](#)
[Interrupts in Linux Kernel](#)

Like this:

Loading...



[report this ad](#)



 ezoic

[report this ad](#)

Sidebar▼

Device Drivers

Linux Device Driver Tutorial Part 13 - Interrupts Example Program 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 the Part 13 of Linux device driver tutorial. In our [previous tutorial](#) we have seen the What is an Interrupt and How it works through theory. Now we will see Interrupt Example Program in Linux Kernel.

Post Contents

- [1 Interrupt Example Program in Linux Kernel](#)
- [2 Functions Related to Interrupt
 - \[2.1 Interrupts Flags\]\(#\)](#)
- [3 Registering an Interrupt Handler](#)
- [4 Freeing an Interrupt Handler](#)
- [5 Interrupt Handler](#)
- [6 Programming
 - \[6.1 Triggering Hardware Interrupt through Software\]\(#\)
 - \[6.2 Driver Source Code\]\(#\)
 - \[6.3 MakeFile\]\(#\)](#)
- [7 Building and Testing Driver
 - \[7.0.1 Share this:\]\(#\)
 - \[7.0.2 Like this:\]\(#\)
 - \[7.0.3 Related\]\(#\)](#)

Interrupt Example Program in Linux Kernel

Before writing any interrupt program, you should keep these following

points in your mind.

1. Interrupt handlers can not enter sleep, so to avoid calls to some functions which has sleep.
2. When the interrupt handler has part of the code to enter the critical section, use spinlocks lock, rather than mutexes. Because if it couldn't take mutex it will go to sleep until it takes the mutex.
3. Interrupt handlers can not exchange data with the user space.
4. The interrupt handlers must be executed as soon as possible. To ensure this, it is best to split the implementation into two parts, top half and bottom half. The top half of the handler will get the job done as soon as possible, and then work late on the bottom half, which can be done with softirqs or tasklets or workqueus.
5. Interrupt handlers can not be called repeatedly. When a handler is already executing, its corresponding IRQ must be disabled until the handler is done.

6. Interrupt handlers can be interrupted by higher authority handlers. If you want to avoid being interrupted by a highly qualified handlers, you can mark the interrupt handler as a fast handler. However, if too many are marked as fast handlers, the performance of the system will be degraded, because the interrupt latency will be longer.

Functions Related to Interrupt

Before programming we should know the basic functions which is useful for interrupts. This table explains the usage of all functions.

FUNCTION	DESCRIPTION
<pre>request_irq (unsigned int irq, irq_handler_t handler, unsigned long flags, const char *name, void *dev_id)</pre>	<p>Register an IRQ, the parameters are as follows:</p> <p>irq: IRQ number to allocate.</p> <p>handler: This is Interrupt handler function. This function will be invoked whenever the operating system receives the interrupt. The data type of return is <code>irq_handler_t</code>, if its return value is <code>IRQ_HANDLED</code>, it indicates that the processing is completed successfully, but if the return value is <code>IRQ_NONE</code>, the processing fails.</p> <p>flags: can be either zero or a bit mask of one or more of the flags defined in <code>linux/interrupt.h</code>. The most important of these flags are: <code>IRQF_DISABLED</code> <code>IRQF_SAMPLE_RANDOM</code> <code>IRQF_SHARED</code> <code>IRQF_TIMER</code> (Explained after this table)</p> <p>name: Used to identify the device name using this IRQ, for example, <code>cat /proc/interrupts</code> will list the IRQ number and device name.</p> <p>dev_id: IRQ shared by many devices. When an interrupt handler is freed, dev provides a unique cookie to enable the removal of only the desired interrupt handler from the interrupt line. Without</p>

FUNCTION	DESCRIPTION
	<p>this parameter, it would be impossible for the kernel to know which handler to remove on a given interrupt line. You can pass NULL here if the line is not shared, but you must pass a unique cookie if your interrupt line is shared. This pointer is also passed into the interrupt handler on each invocation. A common practice is to pass the driver's device structure. This pointer is unique and might be useful to have within the handlers.</p> <p>Return returns zero on success and nonzero value indicates an error.</p> <p><i>request_irq() cannot be called from interrupt context (other situations where code cannot block), because it can block.</i></p>
<pre>free_irq(unsigned int irq, void *dev_id)</pre>	<p>Release an IRQ registered by request_irq() with the following parameters:</p> <p>irq: IRQ number. dev_id: is the last parameter of request_irq.</p> <p>If the specified interrupt line is not shared, this function removes the handler and disables the line. If the interrupt line is shared, the handler identified via dev_id is removed, but the interrupt line is disabled only when the last handler is removed. With shared interrupt lines, a unique cookie is required to differentiate between the multiple handlers that can exist on a single line and enable free_irq() to remove only the correct handler. In either case (shared or unshared), if dev_id is non-NULL, it must match the desired handler. A call to free_irq() must be made from process context.</p>
<pre>enable_irq(unsigned irq)</pre>	<p>int</p> <p>Re-enable interrupt disabled by disable_irq or disable_irq_nosync.</p>

FUNCTION	DESCRIPTION
disable_irq(unsigned int irq)	Disable an IRQ from issuing an interrupt.
disable_irq_nosync(unsigned int irq)	Disable an IRQ from issuing an interrupt, but wait until there is an interrupt handler being executed.
in_irq()	returns true when in interrupt handler
in_interrupt()	returns true when in interrupt handler or bottom half

Interrupts Flags

These are the second parameter of the function. It has several flags. Here I explained important flags.

- IRQF_DISABLED.
 - When set, this flag instructs the kernel to disable all interrupts when executing this interrupt handler.
 - When unset, interrupt handlers run with all interrupts except their own enabled.

Most interrupt handlers do not set this flag, as disabling all interrupts is bad form. Its use is reserved for performance-sensitive interrupts that execute quickly. This flag is the current manifestation of the SA_INTERRUPT flag, which in the past distinguished between “fast” and “slow” interrupts.
- IRQF_SAMPLE_RANDOM. This flag specifies that interrupts generated by this device should contribute to the kernel [entropy pool](#). The kernel entropy pool provides truly random numbers derived from various random events. If this flag is specified, the timing of interrupts from this device are fed to the pool as entropy. Do not set this if your device issues interrupts at a predictable rate (e.g. the system timer) or can be influenced by external attackers (e.g. a networking device). On the other hand, most other hardware generates interrupts at non deterministic times and is therefore a good source of entropy.
- IRQF_TIMER. This flag specifies that this handler processes interrupts for the system timer.

- IRQF_SHARED. This flag specifies that the interrupt line can be shared among multiple interrupt handlers. Each handler registered on a given line must specify this flag; otherwise, only one handler can exist per line.

Registering an Interrupt Handler

```

1 #define IRQ_NO 11
2
3 if (request_irq(IRQ_NO, irq_handler, IRQF_SHARED, "etx_device", (void *)(irq_handler)))
4     printk(KERN_INFO "my_device: cannot register IRQ ");
5     goto irq;
6 }
```

Freeing an Interrupt Handler

```
1 free_irq(IRQ_NO,(void *)(irq_handler));
```

Interrupt Handler

```

1 static irqreturn_t irq_handler(int irq,void *dev_id) {
2     printk(KERN_INFO "Shared IRQ: Interrupt Occurred");
3     return IRQ_HANDLED;
4 }
```

Programming

Interrupt can be coming from anywhere (any hardware) and anytime. In our tutorial we are not going to use any hardware. Here instead of using hardware, we are going to trigger interrupt by simulating. If you have only PC (without hardware), but you want to play around Interrupts in Linux you can follow our method.

Triggering Hardware Interrupt through Software

Intel processors handle interrupt using IDT (Interrupt Descriptor Table). The IDT consists of 256 entries with each entry corresponding to a vector and of 8 bytes. All the entries are pointer to the interrupt handling function. The CPU uses IDTR to point to IDT. Relation between those two can be depicted as below,

[interrupt-handling-in-x86 Linux Device Driver Tutorial Part 13 – Interrupts](#)

Example Program in Linux Kernel

An interrupt can be programmatically raised using ‘int’ instruction.
For example , Linux system call was using int \$0x80.

In linux IRQ to vector mapping is done in arch/x86/include
/asm/irq_vectors.h.The used vector range is as follows ,

[idt-interrupt-vector Linux Device Driver Tutorial Part 13 - Interrupts](#)
[Example Program in Linux Kernel](#)

Refer [Here](#).

The IRQ0 is mapped to vector using the macro,

```
#define IRQ0_VECTOR (FIRST_EXTERNAL_VECTOR + 0x10)
```

where, FIRST_EXTERNAL_VECTOR = 0x20

So if we want to raise an interrupt IRQ11, programmatically we have to add 11 to vector of IRQ0.

$0x20 + 0x10 + 11 = 0x3B$ (59 in Decimal).

Hence executing “`asm("int $0x3B")`” will raise interrupt IRQ 11.

The instruction will be executed while reading device file of our driver (`/dev/etx_device`).

Driver Source Code

Here I took the old source code from the [sysfs tutorial](#). In that source I have just added interrupt code like `request_irq`, `free_irq` along with interrupt handler.

In this program interrupt will be triggered whenever we are reading device file of our driver (`/dev/etx_device`).

Whenever Interrupt triggers, it will prints the “**Shared IRQ: Interrupt**

Occurred" Text.

```
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 #include <linux/interrupt.h>
13 #include <asm/io.h>
14
15 #define IRQ_NO 11
16
17 //Interrupt handler for IRQ 11.
18
19 static irqreturn_t irq_handler(int irq,void *dev_id) {
20     printk(KERN_INFO "Shared IRQ: Interrupt Occurred");
21     return IRQ_HANDLED;
22 }
23
24
25 volatile int etx_value = 0;
26
27
28 dev_t dev = 0;
29 static struct class *dev_class;
30 static struct cdev etx_cdev;
31 struct kobject *kobj_ref;
32
33 static int __init etx_driver_init(void);
34 static void __exit etx_driver_exit(void);
35
36 /***** Driver Fuctions *****/
37 static int etx_open(struct inode *inode, struct file *file);
38 static int etx_release(struct inode *inode, struct file *file);
39 static ssize_t etx_read(struct file *filp,
40                         char __user *buf, size_t len, loff_t * off);
41 static ssize_t etx_write(struct file *filp,
42                         const char *buf, size_t len, loff_t * off);
43
44 /***** Sysfs Fuctions *****/
45 static ssize_t sysfs_show(struct kobject *kobj,
46                         struct kobj_attribute *attr, char *buf);
47 static ssize_t sysfs_store(struct kobject *kobj,
48                         struct kobj_attribute *attr,const char *buf, size_t count);
49
50 struct kobj_attribute etx_attr = __ATTR(etx_value, 0660, sysfs_show, sysfs_store);
51
52 static struct file_operations fops =
53 {
54     .owner      = THIS_MODULE,
55     .read       = etx_read,
56     .write      = etx_write,
57     .open       = etx_open,
58     .release    = etx_release,
59 };
```

```
60
61 static ssize_t sysfs_show(struct kobject *kobj,
62                         struct kobj_attribute *attr, char *buf)
63 {
64     printk(KERN_INFO "Sysfs - Read!!!\n");
65     return sprintf(buf, "%d", etx_value);
66 }
67
68 static ssize_t sysfs_store(struct kobject *kobj,
69                         struct kobj_attribute *attr,const char *buf, size_t count)
70 {
71     printk(KERN_INFO "Sysfs - Write!!!\n");
72     sscanf(buf,"%d",&etx_value);
73     return count;
74 }
75
76 static int etx_open(struct inode *inode, struct file *file)
77 {
78     printk(KERN_INFO "Device File Opened...!!!\n");
79     return 0;
80 }
81
82 static int etx_release(struct inode *inode, struct file *file)
83 {
84     printk(KERN_INFO "Device File Closed...!!!\n");
85     return 0;
86 }
87
88 static ssize_t etx_read(struct file *filp,
89                         char __user *buf, size_t len, loff_t *off)
90 {
91     printk(KERN_INFO "Read function\n");
92     asm("int $0x3B"); // Corresponding to irq 11
93     return 0;
94 }
95 static ssize_t etx_write(struct file *filp,
96                         const char __user *buf, size_t len, loff_t *off)
97 {
98     printk(KERN_INFO "Write Function\n");
99     return 0;
100 }
101
102
103 static int __init etx_driver_init(void)
104 {
105     /*Allocating Major number*/
106     if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
107         printk(KERN_INFO "Cannot allocate major number\n");
108         return -1;
109     }
110     printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
111
112     /*Creating cdev structure*/
113     cdev_init(&etx_cdev,&fops);
114
115     /*Adding character device to the system*/
116     if((cdev_add(&etx_cdev,dev,1)) < 0){
117         printk(KERN_INFO "Cannot add the device to the system\n");
118         goto r_class;
119     }
120
121     /*Creating struct class*/
122     if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
```

```

123         printk(KERN_INFO "Cannot create the struct class\n");
124         goto r_class;
125     }
126
127     /*Creating device*/
128     if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
129         printk(KERN_INFO "Cannot create the Device 1\n");
130         goto r_device;
131     }
132
133     /*Creating a directory in /sys/kernel/ */
134     kobj_ref = kobject_create_and_add("etx_sysfs",kernel_kobj);
135
136     /*Creating sysfs file for etx_value*/
137     if(sysfs_create_file(kobj_ref,&etx_attr.attr)){
138         printk(KERN_INFO "Cannot create sysfs file.....\n");
139         goto r_sysfs;
140     }
141     if (request_irq(IRQ_NO, irq_handler, IRQF_SHARED, "etx_device", (void *)(irq_h
142         printk(KERN_INFO "my_device: cannot register IRQ ");
143         goto irq;
144     }
145     printk(KERN_INFO "Device Driver Insert...Done!!!\n");
146     return 0;
147
148 irq:
149     free_irq(IRQ_NO,(void *)(irq_handler));
150
151 r_sysfs:
152     kobject_put(kobj_ref);
153     sysfs_remove_file(kernel_kobj, &etx_attr.attr);
154
155 r_device:
156     class_destroy(dev_class);
157 r_class:
158     unregister_chrdev_region(dev,1);
159     cdev_del(&etx_cdev);
160     return -1;
161 }
162
163 void __exit etx_driver_exit(void)
164 {
165     free_irq(IRQ_NO,(void *)(irq_handler));
166     kobject_put(kobj_ref);
167     sysfs_remove_file(kernel_kobj, &etx_attr.attr);
168     device_destroy(dev_class,dev);
169     class_destroy(dev_class);
170     cdev_del(&etx_cdev);
171     unregister_chrdev_region(dev, 1);
172     printk(KERN_INFO "Device Driver Remove...Done!!!\n");
173 }
174
175 module_init(etx_driver_init);
176 module_exit(etx_driver_exit);
177
178 MODULE_LICENSE("GPL");
179 MODULE_AUTHOR("EmbeTronicX <embedtronicx@gmail.com or admin@embedtronicx.com>");
180 MODULE_DESCRIPTION("A simple device driver - Interrupts");
181 MODULE_VERSION("1.9");

```

MakeFile

```
1 obj-m += driver.o
2
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`
- To trigger interrupt read device file (`sudo cat /dev/etx_device`)
- Now see the Dmesg (dmesg)

linux@embedtronicx-VirtualBox: dmesg

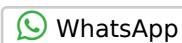
```
[19743.366386] Major = 246 Minor = 0
[19743.370707] Device Driver Insert...Done!!!
[19745.580487] Device File Opened...!!!
[19745.580507] Read function
[19745.580531] Shared IRQ: Interrupt Occurred
[19745.580540] Device File Closed....!!!
```

- We can able to see the print “**Shared IRQ: Interrupt Occurred**”
- Unload the module using `sudo rmmod driver`

This is the simple example using Interrupts in device driver. This is just a basic. You can also try using hardware. I hope this might helped you.

Share this:

[Share on Tumblr](#) [Tweet](#)



[Pocket](#)



[pinit_fg_en_rect_gray_20 Linux Device Driver Tutorial Part 13 – Interrupts Example Program in Linux Kernel](#)

Like this:

Loading...



[report this ad](#)



The ezoic logo, which consists of a small green square with a white 'e' symbol followed by the word "ezoic" in a lowercase sans-serif font.

[report this ad](#)[Sidebar ▾](#)

Device Drivers

Linux Device Driver Tutorial Part 14 – Workqueue in Linux Kernel Part 1

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 the Part 14 of Linux device driver tutorial. In our previous tutorial we have seen the [Example of Interrupt through Device Driver Programming](#). Now we will see one of the Bottomhalf which is Workqueue in Linux Kernel.

Post Contents

- [1 Bottom Half](#)
- [2 Workqueue in Linux Kernel](#)
- [3 Using Global Workqueue \(Global Worker Thread\)](#)
 - [3.1 Initialize work using Static Method](#)
 - [3.1.1 Example](#)
 - [3.2 Schedule work to the Workqueue](#)
 - [3.2.1 Schedule_work](#)
 - [3.2.2 Scheduled_delayed_work](#)
 - [3.2.3 Schedule_work_on](#)
 - [3.2.4 Scheduled_delayed_work_on](#)
 - [3.3 Delete work from workqueue](#)
 - [3.4 Cancel Work from wprkqueue](#)
 - [3.5 Check workqueue](#)
- [4 Programming](#)
 - [4.1 Driver Source Code](#)
 - [4.2 MakeFile](#)
- [5 Building and Testing Driver](#)
 - [5.0.1 Share this:](#)
 - [5.0.2 Like this:](#)

5.0.3 Related

Bottom Half

When Interrupt triggers, Interrupt Handler should be execute very quickly and it should not run for more time (it should not perform time-consuming tasks). If we have the interrupt handler which is doing more tasks then we need to divide into two halves.

1. Top Half
2. Bottom Half

Top Half is nothing but our interrupt handler. If our interrupt handler is doing less task, then top half is more than enough. No need of bottom half in that situation. But if our we have more work when interrupt hits, then we need bottom half. The bottom half runs in the future, at a more convenient time, with all interrupts enabled. So, The job of bottom halves is to perform any interrupt-related work not performed by the interrupt handler.

There are 4 bottom half mechanisms are available in Linux:

1. Work-queue
2. Threaded IRQs
3. Softirqs
4. Tasklets

In this tutorial, we will see Workqueue in Linux Kernel.

Workqueue in Linux Kernel

Work queues are added in linux kernel 2.6 version. Work queues are a different form of deferring work. Work queues defer work into a kernel thread; this bottom half always runs in process context. Because, Work queue is allowing users to create a kernel thread and bind work to the kernel thread. So, this will run in process context and the work queue can sleep.

- Code deferred to a work queue has all the usual benefits of process context.
- Most importantly, work queues are schedulable and can therefore sleep.

Normally, it is easy to decide between using work queues and softirqs/tasklets:

- If the deferred work needs to sleep, work queues are used.
- If the deferred work need not sleep, softirqs or tasklets are used.

There are two ways to implement Workqueue in Linux kernel.

1. Using global workqueue
2. Creating Own workqueue (We will see in next tutorial)

Using Global Workqueue (Global Worker Thread)

In this tutorial we will focus on this method.

In this method no need to create any workqueue or worker thread. So in this method we only need to initialize work. We can initialize the work using two methods.

- Static method
- Dynamic method (We will see in next tutorial)

Initialize work using Static Method

The below call creates a workqueue by the name and the function that we are passing in the second argument gets scheduled in the queue.

```
DECLARE_WORK(name, void (*func)(void *))
```

Where,

name: The name of the “work_struct” structure that has to be created.

func: The function to be scheduled in this workqueue.

Example

```
1 DECLARE_WORK(workqueue,workqueue_fn);
```

Schedule work to the Workqueue

These below functions used to allocate the work to the queue.

Schedule_work

This function puts a job in the kernel-global workqueue if it was not already queued and leaves it in the same position on the kernel-global workqueue otherwise.

```
int schedule_work( struct work_struct *work );
```

where,

work - job to be done

Returns zero if *work* was already on the kernel-global workqueue and non-zero otherwise.

Scheduled_delayed_work

After waiting for a given time this function puts a job in the kernel-global workqueue.

```
int scheduled_delayed_work( struct delayed_work *dwork, unsigned long delay );
```

where,

dwork - job to be done

delay - number of jiffies to wait or 0 for immediate execution

Schedule_work_on

This puts a job on a specific cpu.

```
int schedule_work_on( int cpu, struct work_struct *work );
```

where,

cpu- cpu to put the work task on

work- job to be done

Scheduled_delayed_work_on

After waiting for a given time this puts a job in the kernel-global workqueue on the specified CPU.

```
int scheduled_delayed_work_on(  
    int cpu, struct delayed_work *dwork, unsigned long delay );  
where,
```

cpu - cpu to put the work task on

dwork - job to be done

delay - number of jiffies to wait or 0 for immediate execution

Delete work from workqueue

There are also a number of helper functions that you can use to flush or cancel work on work queues. To flush a particular work item and block until the work is complete, you can make a call to `flush_work`. All work on a given work queue can be completed using a call to `. In both cases, the caller blocks until the operation is complete. To flush the kernel-global work queue, call flush_scheduled_work.`

```
int flush_work( struct work_struct *work );  
void flush_scheduled_work( void );
```

Cancel Work from wprkqueue

You can cancel work if it is not already executing in a handler. A call to `cancel_work_sync` will terminate the work in the queue or block until the callback has finished (if the work is already in progress in the handler). If the work is delayed, you can use a call to `cancel_delayed_work_sync`.

```
int cancel_work_sync( struct work_struct *work );  
int cancel_delayed_work_sync( struct delayed_work *dwork );
```

Check workqueue

Finally, you can find out whether a work item is pending (not yet executed by the handler) with a call to `work_pending` or `delayed_work_pending`.

```
work_pending( work );
```

```
delayed_work_pending( work );
```

Programming Driver Source Code

I took the source code from previous [interrupt example tutorial](#). In that source code, When we read the /dev/etx_device interrupt will hit (To understand interrupts in Linux go to [this tutorial](#)). Whenever interrupt hits, I'm scheduling the work to the workqueue. I'm not going to do any job in both interrupt handler and workqueue function, since it is a tutorial post. But in real workqueues, this function can be used to carry out any operations that need to be scheduled.

```

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 #include <linux/interrupt.h>
13 #include <asm/io.h>
14 #include <linux/workqueue.h>       // Required for workqueues
15
16
17 #define IRQ_NO 11
18
19
20 void workqueue_fn(struct work_struct *work);
21
22 /*Creating work by Static Method */
23 DECLARE_WORK(workqueue,workqueue_fn);
24
25 /*Workqueue Function*/
26 void workqueue_fn(struct work_struct *work)
27 {
28     printk(KERN_INFO "Executing Workqueue Function\n");
29 }
30
31
32 //Interrupt handler for IRQ 11.
33 static irqreturn_t irq_handler(int irq,void *dev_id) {
34     printk(KERN_INFO "Shared IRQ: Interrupt Occurred");
35     schedule_work(&workqueue);
36
37     return IRQ_HANDLED;
38 }
39
40
41 volatile int etx_value = 0;
```

```
43 dev_t dev = 0;
44 static struct class *dev_class;
45 static struct cdev etx_cdev;
46 struct kobject *kobj_ref;
47
48 static int __init etx_driver_init(void);
49 static void __exit etx_driver_exit(void);
50
51 /***** Driver Fuctions *****/
52 static int etx_open(struct inode *inode, struct file *file);
53 static int etx_release(struct inode *inode, struct file *file);
54 static ssize_t etx_read(struct file *filp,
55                         char __user *buf, size_t len, loff_t * off);
56 static ssize_t etx_write(struct file *filp,
57                         const char *buf, size_t len, loff_t * off);
58
59 /***** Sysfs Fuctions *****/
60 static ssize_t sysfs_show(struct kobject *kobj,
61                         struct kobj_attribute *attr, char *buf);
62 static ssize_t sysfs_store(struct kobject *kobj,
63                         struct kobj_attribute *attr,const char *buf, size_t count);
64
65 struct kobj_attribute etx_attr = __ATTR(etx_value, 0660, sysfs_show, sysfs_store);
66
67 static struct file_operations fops =
68 {
69     .owner        = THIS_MODULE,
70     .read         = etx_read,
71     .write        = etx_write,
72     .open          = etx_open,
73     .release      = etx_release,
74 };
75
76 static ssize_t sysfs_show(struct kobject *kobj,
77                         struct kobj_attribute *attr, char *buf)
78 {
79     printk(KERN_INFO "Sysfs - Read!!!\n");
80     return sprintf(buf, "%d", etx_value);
81 }
82
83 static ssize_t sysfs_store(struct kobject *kobj,
84                         struct kobj_attribute *attr,const char *buf, size_t count)
85 {
86     printk(KERN_INFO "Sysfs - Write!!!\n");
87     sscanf(buf,"%d",&etx_value);
88     return count;
89 }
90
91 static int etx_open(struct inode *inode, struct file *file)
92 {
93     printk(KERN_INFO "Device File Opened...!!!\n");
94     return 0;
95 }
96
97 static int etx_release(struct inode *inode, struct file *file)
98 {
99     printk(KERN_INFO "Device File Closed...!!!\n");
100    return 0;
101 }
102
103 static ssize_t etx_read(struct file *filp,
104                         char __user *buf, size_t len, loff_t *off)
```

```

106 {
107     printk(KERN_INFO "Read function\n");
108     asm("int $0x3B"); // Corresponding to irq 11
109     return 0;
110 }
111 static ssize_t etx_write(struct file *filp,
112                         const char __user *buf, size_t len, loff_t *off)
113 {
114     printk(KERN_INFO "Write Function\n");
115     return 0;
116 }
117
118
119 static int __init etx_driver_init(void)
120 {
121     /*Allocating Major number*/
122     if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
123         printk(KERN_INFO "Cannot allocate major number\n");
124         return -1;
125     }
126     printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
127
128     /*Creating cdev structure*/
129     cdev_init(&etx_cdev,&fops);
130
131     /*Adding character device to the system*/
132     if((cdev_add(&etx_cdev,dev,1)) < 0){
133         printk(KERN_INFO "Cannot add the device to the system\n");
134         goto r_class;
135     }
136
137     /*Creating struct class*/
138     if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
139         printk(KERN_INFO "Cannot create the struct class\n");
140         goto r_class;
141     }
142
143     /*Creating device*/
144     if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
145         printk(KERN_INFO "Cannot create the Device 1\n");
146         goto r_device;
147     }
148
149     /*Creating a directory in /sys/kernel/ */
150     kobj_ref = kobject_create_and_add("etx_sysfs",kernel_kobj);
151
152     /*Creating sysfs file for etx_value*/
153     if(sysfs_create_file(kobj_ref,&etx_attr.attr)){
154         printk(KERN_INFO "Cannot create sysfs file.....\n");
155         goto r_sysfs;
156     }
157     if (request_irq(IRQ_NO, irq_handler, IRQF_SHARED, "etx_device", (void *)(irq_h
158         printk(KERN_INFO "my_device: cannot register IRQ ");
159         goto irq;
160     }
161     printk(KERN_INFO "Device Driver Insert...Done!!!\n");
162     return 0;
163
164 irq:
165     free_irq(IRQ_NO,(void *)(irq_handler));
166
167 r_sysfs:
168     kobject_put(kobj_ref);

```

```

169         sysfs_remove_file(kernel_kobj, &etx_attr.attr);
170
171 r_device:
172     class_destroy(dev_class);
173 r_class:
174     unregister_chrdev_region(dev,1);
175     cdev_del(&etx_cdev);
176     return -1;
177 }
178
179 void __exit etx_driver_exit(void)
180 {
181     free_irq(IRQ_NO,(void *)(irq_handler));
182     kobject_put(kobj_ref);
183     sysfs_remove_file(kernel_kobj, &etx_attr.attr);
184     device_destroy(dev_class,dev);
185     class_destroy(dev_class);
186     cdev_del(&etx_cdev);
187     unregister_chrdev_region(dev, 1);
188     printk(KERN_INFO "Device Driver Remove...Done!!!\n");
189 }
190
191 module_init(etx_driver_init);
192 module_exit(etx_driver_exit);
193
194 MODULE_LICENSE("GPL");
195 MODULE_AUTHOR("EmbeTronicX <embedtronicx@gmail.com or admin@embedtronicx.com>");
196 MODULE_DESCRIPTION("A simple device driver - Workqueue part 1");
197 MODULE_VERSION("1.10");

```

MakeFile

```

1 obj-m += driver.o
2
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*
- To trigger interrupt read device file (*sudo cat /dev/etx_device*)
- Now see the Dmesg (*dmesg*)

linux@embedtronicx-VirtualBox: dmesg

[11213.943071] Major = 246 Minor = 0
[11213.945181] Device Driver Insert...Done!!!

```
[11217.255727] Device File Opened...!!!
[11217.255747] Read function
[11217.255783] Shared IRQ: Interrupt Occurred
[11217.255845] Executing Workqueue Function
[11217.255860] Device File Closed....!!!
```

- We can able to see the print “**Shared IRQ: Interrupt Occurred**” and “**Executing Workqueue Function**”
- Unload the module using sudo rmmod driver

In our [next tutorial](#) we will discuss Workqueue using Dynamic method.

Share this:[Share on Tumblr](#)[Tweet](#)[WhatsApp](#)[Pocket](#)

[pinit_fg_en_rect_gray_20](#) Linux Device Driver Tutorial Part 14 –
Workqueue in Linux Kernel Part 1

Like this:

Loading...



[report this ad](#)



Sidebar ▾

Device Drivers

Workqueue in Linux

Linux Device Driver Tutorial Part 15 – Workqueue in Linux Kernel Part 2

This is the [Series on Linux Device Driver](#). The aim of this series is to provide the easy and practical examples that anyone can understand. In our previous tutorial we have seen the [Workqueue using Static method](#) through Device Driver Programming. Now we are going to see Linux Device Driver Tutorial Part 15 – Workqueue in Linux (Dynamic Creation Method).

Post Contents

- 1 Initialize work using Static Method
- 2 Schedule work to the Workqueue
 - 2.1 Schedule_work
 - 2.2 Scheduled_delayed_work
 - 2.3 Schedule_work_on
 - 2.4 Scheduled_delayed_work_on
- 3 Delete work from workqueue
- 4 Cancel Work from workqueue
- 5 Check workqueue
- 6 Programming
 - 6.1 Driver Source Code
 - 6.2 MakeFile
- 7 Building and Testing Driver
 - 7.0.1 Share this:
 - 7.0.2 Like this:
 - 7.0.3 Related

Initialize work using Static Method

The below call creates a workqueue by the name work and the function that gets scheduled in the queue is work_fn.

INIT_WORK(*work*,*work_fn*)

Where,

name: The name of the “work_struct” structure that has to be created.

func: The function to be scheduled in this workqueue.

Schedule work to the Workqueue

These below functions used to allocate the work to the queue.

Schedule_work

This function puts a job in the kernel-global workqueue if it was not already queued and leaves it in the same position on the kernel-global workqueue otherwise.

```
int schedule_work( struct work_struct *work );
```

where,

work - job to be done

Returns zero if *work* was already on the kernel-global workqueue and non-zero otherwise.

Scheduled_delayed_work

After waiting for a given time this function puts a job in the kernel-global workqueue.

```
int scheduled_delayed_work( struct delayed_work *dwork, unsigned long delay );
```

where,

dwork – job to be done

delay – number of jiffies to wait or 0 for immediate execution

Schedule_work_on

This puts a job on a specific cpu.

```
int schedule_work_on( int cpu, struct work_struct *work );
```

where,

cpu – cpu to put the work task on

work – job to be done

Scheduled_delayed_work_on

After waiting for a given time this puts a job in the kernel-global workqueue on the specified CPU.

```
int scheduled_delayed_work_on( int cpu, struct delayed_work *dwork, unsigned long delay );
```

where,

cpu – cpu to put the work task on

dwork – job to be done

delay – number of jiffies to wait or 0 for immediate execution

Delete work from workqueue

There are also a number of helper functions that you can use to flush or cancel work on work queues. To flush a particular work item and block until the work is complete, you can make a call to `flush_work`. All work on a given work queue can be completed using a call to `. In both cases, the caller blocks until the operation is complete. To flush the kernel-global work queue, call flush_scheduled_work.`

```
int flush_work( struct work_struct *work );
void flush_scheduled_work( void );
```

Cancel Work from workqueue

You can cancel work if it is not already executing in a handler. A call to `cancel_work_sync` will terminate the work in the queue or block until the callback has finished (if the work is already in progress in the handler). If the work is delayed, you can use a call to `cancel_delayed_work_sync`.

```
int cancel_work_sync( struct work_struct *work );
int cancel_delayed_work_sync( struct delayed_work *dwork );
```

Check workqueue

Finally, you can find out whether a work item is pending (not yet executed by the handler) with a call to `work_pending` or `delayed_work_pending`.

```
work_pending( work );
delayed_work_pending( work );
```

Programming

Driver Source Code

In that source code, When we read the /dev/etx_device interrupt will hit (To understand interrupts in Linux go to [this tutorial](#)). Whenever interrupt hits, I'm scheduling the work to the workqueue. I'm not going to do any job in both interrupt handler and workqueue function, since it is a tutorial post. But in real workqueues, this function can be used to carry

out any operations that need to be scheduled.

```
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 #include <linux/interrupt.h>
13 #include <asm/io.h>
14 #include <linux/workqueue.h>       // Required for workqueues
15
16
17 #define IRQ_NO 11
18
19 /* Work structure */
20 static struct work_struct workqueue;
21
22 void workqueue_fn(struct work_struct *work);
23
24 /*Workqueue Function*/
25 void workqueue_fn(struct work_struct *work)
26 {
27     printk(KERN_INFO "Executing Workqueue Function\n");
28 }
29
30 //Interrupt handler for IRQ 11.
31 static irqreturn_t irq_handler(int irq,void *dev_id) {
32     printk(KERN_INFO "Shared IRQ: Interrupt Occurred");
33     /*Allocating work to queue*/
34     schedule_work(&workqueue);
35
36     return IRQ_HANDLED;
37 }
38
39 volatile int etx_value = 0;
40 dev_t dev = 0;
41 static struct class *dev_class;
42 static struct cdev etx_cdev;
43 struct kobject *kobj_ref;
44
45 static int __init etx_driver_init(void);
46 static void __exit etx_driver_exit(void);
47
48 /***** Driver Functions *****/
49 static int etx_open(struct inode *inode, struct file *file);
50 static int etx_release(struct inode *inode, struct file *file);
51 static ssize_t etx_read(struct file *filp,
52                         char __user *buf, size_t len, loff_t * off);
53 static ssize_t etx_write(struct file *filp,
54                         const char *buf, size_t len, loff_t * off);
55
56 /***** Sysfs Fuctions *****/
57 static ssize_t sysfs_show(struct kobject *kobj,
58                         struct kobj_attribute *attr, char *buf);
59 static ssize_t sysfs_store(struct kobject *kobj,
```

```

60             struct kobj_attribute *attr,const char *buf, size_t count);
61
62 struct kobj_attribute etx_attr = __ATTR(etx_value, 0660, sysfs_show, sysfs_store);
63
64 static struct file_operations fops =
65 {
66     .owner        = THIS_MODULE,
67     .read         = etx_read,
68     .write        = etx_write,
69     .open          = etx_open,
70     .release      = etx_release,
71 };
72
73 static ssize_t sysfs_show(struct kobject *kobj,
74                         struct kobj_attribute *attr, char *buf)
75 {
76     printk(KERN_INFO "Sysfs - Read!!!\n");
77     return sprintf(buf, "%d", etx_value);
78 }
79
80 static ssize_t sysfs_store(struct kobject *kobj,
81                         struct kobj_attribute *attr,const char *buf, size_t count)
82 {
83     printk(KERN_INFO "Sysfs - Write!!!\n");
84     sscanf(buf, "%d",&etx_value);
85     return count;
86 }
87
88 static int etx_open(struct inode *inode, struct file *file)
89 {
90     printk(KERN_INFO "Device File Opened...!!!\n");
91     return 0;
92 }
93
94 static int etx_release(struct inode *inode, struct file *file)
95 {
96     printk(KERN_INFO "Device File Closed...!!!\n");
97     return 0;
98 }
99
100 static ssize_t etx_read(struct file *filp,
101                         char __user *buf, size_t len, loff_t *off)
102 {
103     printk(KERN_INFO "Read function\n");
104     asm("int $0x3B"); // Corresponding to irq 11
105     return 0;
106 }
107 static ssize_t etx_write(struct file *filp,
108                         const char __user *buf, size_t len, loff_t *off)
109 {
110     printk(KERN_INFO "Write Function\n");
111     return 0;
112 }
113
114
115 static int __init etx_driver_init(void)
116 {
117     /*Allocating Major number*/
118     if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
119         printk(KERN_INFO "Cannot allocate major number\n");
120         return -1;
121     }
122     printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));

```

```

123     /*Creating cdev structure*/
124     cdev_init(&etx_cdev,&fops);
125
126     /*Adding character device to the system*/
127     if((cdev_add(&etx_cdev,dev,1)) < 0){
128         printk(KERN_INFO "Cannot add the device to the system\n");
129         goto r_class;
130     }
131
132     /*Creating struct class*/
133     if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
134         printk(KERN_INFO "Cannot create the struct class\n");
135         goto r_class;
136     }
137
138     /*Creating device*/
139     if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
140         printk(KERN_INFO "Cannot create the Device 1\n");
141         goto r_device;
142     }
143
144     /*Creating a directory in /sys/kernel/ */
145     kobj_ref = kobject_create_and_add("etx_sysfs",kernel_kobj);
146
147     /*Creating sysfs file for etx_value*/
148     if(sysfs_create_file(kobj_ref,&etx_attr.attr)){
149         printk(KERN_INFO"Cannot create sysfs file.....\n");
150         goto r_sysfs;
151     }
152     if (request_irq(IRQ_NO, irq_handler, IRQF_SHARED, "etx_device", (void *)(irq_h
153         printk(KERN_INFO "my_device: cannot register IRQ ");
154         goto irq;
155     }
156
157     /*Creating work by Dynamic Method */
158     INIT_WORK(&workqueue,workqueue_fn);
159
160     printk(KERN_INFO "Device Driver Insert...Done!!!\n");
161     return 0;
162
163
164 irq:
165     free_irq(IRQ_NO,(void *)(irq_handler));
166
167 r_sysfs:
168     kobject_put(kobj_ref);
169     sysfs_remove_file(kernel_kobj, &etx_attr.attr);
170
171 r_device:
172     class_destroy(dev_class);
173 r_class:
174     unregister_chrdev_region(dev,1);
175     cdev_del(&etx_cdev);
176     return -1;
177 }
178
179 void __exit etx_driver_exit(void)
180 {
181     free_irq(IRQ_NO,(void *)(irq_handler));
182     kobject_put(kobj_ref);
183     sysfs_remove_file(kernel_kobj, &etx_attr.attr);
184     device_destroy(dev_class,dev);
185     class_destroy(dev_class);

```

```

186         cdev_del(&etx_cdev);
187         unregister_chrdev_region(dev, 1);
188         printk(KERN_INFO "Device Driver Remove...Done!!!\n");
189     }
190
191 module_init(etx_driver_init);
192 module_exit(etx_driver_exit);
193
194 MODULE_LICENSE("GPL");
195 MODULE_AUTHOR("EmbeTronicX <embedtronicx@gmail.com or admin@embedtronicx.com>");
196 MODULE_DESCRIPTION("A simple device driver - Workqueue part 2");
197 MODULE_VERSION("1.11");

```

MakeFile

```

1 obj-m += driver.o
2
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*
- To trigger interrupt read device file (*sudo cat /dev/etx_device*)
- Now see the Dmesg (*dmesg*)

linux@embedtronicx-VirtualBox: dmesg

```

[11213.943071] Major = 246 Minor = 0
[11213.945181] Device Driver Insert...Done!!!
[11217.255727] Device File Opened...!!!
[11217.255747] Read function
[11217.255783] Shared IRQ: Interrupt Occurred
[11217.255845] Executing Workqueue Function
[11217.255860] Device File Closed...!!!

```

- We can able to see the print “**Shared IRQ: Interrupt Occurred**” and “**Executing Workqueue Function**”
- Unload the module using *sudo rmmod driver*

In our [next tutorial](#) we will discuss Workqueue using own worker thread.

Share this:[Share on Tumblr](#)[Tweet](#)[Print](#)[WhatsApp](#)[Pocket](#)[Telegram](#)

pinit_fg_en_rect_gray_20 Linux Device Driver Tutorial Part 15 –
Workqueue in Linux Kernel Part 2

Like this:

Loading...



[report this ad](#)

 ezoic[report this ad](#)[Sidebar ▾](#)

Device Drivers

Work queue in Linux

Linux Device Driver Tutorial Part 16 – Workqueue in Linux Kernel Part 3

This is the [Series on Linux Device Driver](#). The aim of this series is to provide the easy and practical examples that anyone can understand. In our previous [tutorials](#) we have used Global work queue. But in this tutorial we are going to use our own work queue in Linux device driver.

Post Contents

- [1 Work queue in Linux Device Driver](#)
- [2 Create and destroy work queue structure](#)
 - [2.1 alloc_workqueue](#)
 - [2.1.1 WQ_* flags](#)
- [3 Queuing Work to workqueue](#)
 - [3.1 queue_work](#)
 - [3.2 queue_work_on](#)
 - [3.3 queue_delayed_work](#)
 - [3.4 queue_delayed_work_on](#)
- [4 Programming](#)
 - [4.1 Driver Source Code](#)
 - [4.2 MakeFile](#)
- [5 Building and Testing Driver](#)
- [6 Difference between Schedule_work and queue_work](#)
 - [6.0.1 Share this:](#)
 - [6.0.2 Like this:](#)
 - [6.0.3 Related](#)

Work queue in Linux Device Driver

In our previous ([Part 1](#), [Part 2](#)) tutorials we haven't created any of the workqueue. We were just creating work and scheduling that work to the global workqueue. Now we are going to create our own workqueue. Let's get into the tutorial.

The core work queue is represented by structure struct workqueue_struct, which is the structure onto which work is placed. This work is added to queue in the top half (Interrupt context) and execution of this work happened in the bottom half (Kernel context).

The work is represented by structure struct work_struct, which identifies the work and the deferral function.

Create and destroy work queue structure

Work queues are created through a macro called create_workqueue, which returns a workqueue_struct reference. You can remove this work queue later (if needed) through a call to the destroy_workqueue function.

```
struct workqueue_struct *create_workqueue( name );  
  
void destroy_workqueue( struct workqueue_struct * );
```

You should use create_singlethread_workqueue() for create workqueue when you want to create only a single thread for all the processor..

Since create_workqueue and create_singlethread_workqueue() are macros. Both are using the alloc_workqueue function in background.

```
1 #define create_workqueue(name)  
2         alloc_workqueue("%s", WQ_MEM_RECLAIM, 1, (name))  
3 #define create_singlethread_workqueue(name)  
4         alloc_workqueue("%s", WQ_UNBOUND | WQ_MEM_RECLAIM, 1, (name))
```

alloc_workqueue

Allocate a workqueue with the specified parameters.

```
alloc_workqueue ( fmt, flags, max_active );
```

fmt - printf format for the name of the workqueue

flags - WQ_* flags

max_active - max in-flight work items, 0 for default

This will return Pointer to the allocated workqueue on success, NULL on failure.

WQ_* flags

This is the second argument of alloc_workqueue.

WQ_UNBOUND

Work items queued to an unbound wq are served by the special worker-pools which host workers which are not bound to any specific CPU. This makes the wq behave as a simple execution context provider without concurrency management. The unbound worker-pools try to start execution of work items as soon as possible. Unbound wq sacrifices locality but is useful for the following cases.

- Wide fluctuation in the concurrency level requirement is expected and using bound wq may end up creating large number of mostly unused workers across different CPUs as the issuer hops through different CPUs.
- Long running CPU intensive workloads which can be better managed by the system scheduler.

WQ_FREEZABLE

A freezable wq participates in the freeze phase of the system suspend operations. Work items on the wq are drained and no new work item starts execution until thawed.

WQ_MEM_RECLAIM

All wq which might be used in the memory reclaim paths **MUST** have this flag set. The wq is guaranteed to have at least one execution context regardless of memory pressure.

WQ_HIGHPRI

Work items of a highpri wq are queued to the highpri worker-pool of the target cpu. Highpri worker-pools are served by worker threads with elevated nice level.

Note that normal and highpri worker-pools don't interact with each other. Each maintain its separate pool of workers and implements concurrency management among its workers.

WQ_CPU_INTENSIVE

Work items of a CPU intensive wq do not contribute to the concurrency level. In other words, runnable CPU intensive work items will not prevent other work items in the same worker-pool from starting execution. This is useful for bound work items which are expected to hog CPU cycles so that their execution is regulated by the system scheduler.

Although CPU intensive work items don't contribute to the concurrency level, start of their executions is still regulated by the concurrency management and runnable non-CPU-intensive work items can delay execution of CPU intensive work items.

This flag is meaningless for unbound wq.

Queuing Work to workqueue

With the work structure initialized, the next step is enqueueing the work on a work queue. You can do this in a few ways.

queue_work

This will queue the work to the CPU on which it was submitted, but if the CPU dies it can be processed by another CPU.

```
int queue_work( struct workqueue_struct *wq, struct work_struct  
                *work );
```

Where,

wq – workqueue to use

work – work to queue

It returns false if *work* was already on a queue, true otherwise.

queue_work_on

This puts a work on a specific cpu.

```
int queue_work_on( int cpu, struct workqueue_struct *wq,  
                   struct work_struct *work );
```

Where,

cpu- cpu to put the work task on

wq – workqueue to use

work- job to be done

queue_delayed_work

After waiting for a given time this function puts a work in the workqueue.

```
int queue_delayed_work( struct workqueue_struct *wq,  
                        struct delayed_work *dwork, unsigned long delay );
```

Where,

wq – workqueue to use

dwork – work to queue

delay – number of jiffies to wait before queueing or 0 for immediate execution

queue_delayed_work_on

After waiting for a given time this puts a job in the workqueue on the specified CPU.

```
int queue_delayed_work_on( int cpu, struct workqueue_struct *wq,  
                           struct delayed_work *dwork, unsigned long delay );
```

Where,

cpu- cpu to put the work task on

wq – workqueue to use

dwork – work to queue

delay – number of jiffies to wait before queueing or 0 for immediate execution

Programming

Driver Source Code

In that source code, When we read the /dev/etx_device interrupt will hit (To understand interrupts in Linux go to [this tutorial](#)). Whenever interrupt hits, I'm scheduling the work to the workqueue. I'm not going to do any job in both interrupt handler and workqueue function, since it is a tutorial post. But in real workqueues, this function can be used to carry out any operations that need to be scheduled.

We have created workqueue “own_wq” in init function.

Let's go through the code.

```
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 #include <linux/interrupt.h>
13 #include <asm/io.h>
14 #include <linux/workqueue.h>       // Required for workqueues
15
16
17 #define IRQ_NO 11
18
19 static struct workqueue_struct *own_workqueue;
20
21 static void workqueue_fn(struct work_struct *work);
22
23 static DECLARE_WORK(work, workqueue_fn);
24
25
26 /*Workqueue Function*/
27 static void workqueue_fn(struct work_struct *work)
28 {
29     printk(KERN_INFO "Executing Workqueue Function\n");
30     return;
31 }
32
33
34
35 //Interrupt handler for IRQ 11.
36 static irqreturn_t irq_handler(int irq,void *dev_id) {
37     printk(KERN_INFO "Shared IRQ: Interrupt Occurred\n");
38     /*Allocating work to queue*/
39     queue_work(own_workqueue, &work);
40
41     return IRQ_HANDLED;
42 }
43
44
45 volatile int etx_value = 0;
46
47
48 dev_t dev = 0;
49 static struct class *dev_class;
50 static struct cdev etx_cdev;
51 struct kobject *kobj_ref;
52
53 static int __init etx_driver_init(void);
54 static void __exit etx_driver_exit(void);
55
56 /***** Driver Functions *****/
```

```

57 static int etx_open(struct inode *inode, struct file *file);
58 static int etx_release(struct inode *inode, struct file *file);
59 static ssize_t etx_read(struct file *filp,
60                         char __user *buf, size_t len, loff_t * off);
61 static ssize_t etx_write(struct file *filp,
62                         const char *buf, size_t len, loff_t * off);
63
64 /***** Sysfs Fuctions *****/
65 static ssize_t sysfs_show(struct kobject *kobj,
66                         struct kobj_attribute *attr, char *buf);
67 static ssize_t sysfs_store(struct kobject *kobj,
68                         struct kobj_attribute *attr,const char *buf, size_t count);
69
70 struct kobj_attribute etx_attr = __ATTR(etx_value, 0660, sysfs_show, sysfs_store);
71
72 static struct file_operations fops =
73 {
74     .owner        = THIS_MODULE,
75     .read         = etx_read,
76     .write        = etx_write,
77     .open          = etx_open,
78     .release      = etx_release,
79 };
80
81 static ssize_t sysfs_show(struct kobject *kobj,
82                         struct kobj_attribute *attr, char *buf)
83 {
84     printk(KERN_INFO "Sysfs - Read!!!\n");
85     return sprintf(buf, "%d", etx_value);
86 }
87
88 static ssize_t sysfs_store(struct kobject *kobj,
89                         struct kobj_attribute *attr,const char *buf, size_t count)
90 {
91     printk(KERN_INFO "Sysfs - Write!!!\n");
92     sscanf(buf,"%d",&etx_value);
93     return count;
94 }
95
96 static int etx_open(struct inode *inode, struct file *file)
97 {
98     printk(KERN_INFO "Device File Opened...!!!\n");
99     return 0;
100 }
101
102 static int etx_release(struct inode *inode, struct file *file)
103 {
104     printk(KERN_INFO "Device File Closed...!!!\n");
105     return 0;
106 }
107
108 static ssize_t etx_read(struct file *filp,
109                         char __user *buf, size_t len, loff_t *off)
110 {
111     printk(KERN_INFO "Read function\n");
112     asm("int $0x3B"); // Corresponding to irq 11
113     return 0;
114 }
115 static ssize_t etx_write(struct file *filp,
116                         const char __user *buf, size_t len, loff_t *off)
117 {
118     printk(KERN_INFO "Write Function\n");
119     return 0;

```

```

120 }
121
122
123 static int __init etx_driver_init(void)
124 {
125     /*Allocating Major number*/
126     if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
127         printk(KERN_INFO "Cannot allocate major number\n");
128         return -1;
129     }
130     printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
131
132     /*Creating cdev structure*/
133     cdev_init(&etx_cdev,&fops);
134
135     /*Adding character device to the system*/
136     if((cdev_add(&etx_cdev,dev,1)) < 0){
137         printk(KERN_INFO "Cannot add the device to the system\n");
138         goto r_class;
139     }
140
141     /*Creating struct class*/
142     if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
143         printk(KERN_INFO "Cannot create the struct class\n");
144         goto r_class;
145     }
146
147     /*Creating device*/
148     if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
149         printk(KERN_INFO "Cannot create the Device 1\n");
150         goto r_device;
151     }
152
153     /*Creating a directory in /sys/kernel/ */
154     kobj_ref = kobject_create_and_add("etx_sysfs",kernel_kobj);
155
156     /*Creating sysfs file for etx_value*/
157     if(sysfs_create_file(kobj_ref,&etx_attr.attr)){
158         printk(KERN_INFO "Cannot create sysfs file.....\n");
159         goto r_sysfs;
160     }
161     if (request_irq(IRQ_NO, irq_handler, IRQF_SHARED, "etx_device", (void *)(irq_h
162         printk(KERN_INFO "my_device: cannot register IRQ \n");
163         goto irq;
164     }
165
166     /*Creating workqueue */
167     own_workqueue = create_workqueue("own_wq");
168
169     printk(KERN_INFO "Device Driver Insert...Done!!!\n");
170     return 0;
171
172 irq:
173     free_irq(IRQ_NO,(void *)(irq_handler));
174
175 r_sysfs:
176     kobject_put(kobj_ref);
177     sysfs_remove_file(kernel_kobj, &etx_attr.attr);
178
179 r_device:
180     class_destroy(dev_class);
181 r_class:
182     unregister_chrdev_region(dev,1);

```

```

183         cdev_del(&etx_cdev);
184         return -1;
185     }
186
187 void __exit etx_driver_exit(void)
188 {
189     /* Delete workqueue */
190     destroy_workqueue(own_workqueue);
191     free_irq(IRQ_NO,(void*)(irq_handler));
192     kobject_put(kobj_ref);
193     sysfs_remove_file(kernel_kobj, &etx_attr.attr);
194     device_destroy(dev_class,dev);
195     class_destroy(dev_class);
196     cdev_del(&etx_cdev);
197     unregister_chrdev_region(dev, 1);
198     printk(KERN_INFO "Device Driver Remove...Done!!!\n");
199 }
200
201 module_init(etx_driver_init);
202 module_exit(etx_driver_exit);
203
204 MODULE_LICENSE("GPL");
205 MODULE_AUTHOR("EmbeTronicX <embedtronicx@gmail.com>");
206 MODULE_DESCRIPTION("A simple device driver - Workqueue part 3");
207 MODULE_VERSION("1.12");

```

MakeFile

```

1 obj-m += driver.o
2
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`
- To trigger interrupt read device file (`sudo cat /dev/etx_device`)
- Now see the Dmesg (dmesg)

[2562.609446] Major = 246 Minor = 0
[2562.649362] Device Driver Insert...Done!!!
[2565.133204] Device File Opened...!!!
[2565.133225] Read function
[2565.133248] Shared IRQ: Interrupt Occurred
[2565.133267] Executing Workqueue Function

[2565.140284] Device File Closed.....

- We can able to see the print “**Shared IRQ: Interrupt Occurred**” and “**Executing Workqueue Function**”
- Use “ps -aef” command to see our workqueue. You can able to see our workqueue which is “own_wq”

<i>UID</i>	<i>PID</i>	<i>PPID</i>	<i>C</i>	<i>STIME</i>	<i>TTY</i>	<i>TIME</i>	<i>CMD</i>
<i>root</i>	3516	2	0	21:35	?	00:00:00	<i>[own_wq]</i>

- Unload the module using sudo rmmod driver

Difference between Schedule_work and queue_work

- If you want to use your own dedicated workqueue you should create workqueue using create_workqueue. In that time you need to put work on your workqueue by using queue_work function.
- If you don't want to create any own workqueue, you can use kernel global workqueue. In that condition, you can use schedule_work function to put your work to global workqueue.

Share this:

[Share on Tumblr](#) [Tweet](#)[WhatsApp](#)[Pocket](#)

[pinit_fg_en_rect_gray_20](#) Linux Device Driver Tutorial Part 16 –
Workqueue in Linux Kernel Part 3

Like this:

Loading...



[report this ad](#)

 ezoic[report this ad](#)[Sidebar ▾](#)

📁 Device Drivers

tasklet tutorial

Linux Device Driver Tutorial Part 20 - Tasklet | Static Method

This is the [Series on Linux Device Driver](#). The aim of this series is to provide the easy and practical examples that anyone can understand. This is the Linux Device Driver Tutorial Part 20 - Tasklet Static Method Tutorial.

Post Contents

- [1 Prerequisites](#)
- [2 Bottom Half](#)
- [3 Tasklets in Linux Kernel
 - \[3.1 Points To Remember\]\(#\)](#)
- [4 Tasklet Structure](#)
- [5 Create Tasklet
 - \[5.1 DECLARE_TASKLET
 - \\[5.1.1 Example\\]\\(#\\)\]\(#\)
 - \[5.2 DECLARE_TASKLET_DISABLED\]\(#\)](#)
- [6 Enable and Disable Tasklet
 - \[6.1 tasklet_enable\]\(#\)
 - \[6.2 tasklet_disable\]\(#\)
 - \[6.3 tasklet_disable_nosync\]\(#\)](#)
- [7 Schedule Tasklet
 - \[7.1 tasklet_schedule\]\(#\)
 - \[7.1.1 Example\]\(#\)
 - \[7.2 tasklet_hi_schedule\]\(#\)
 - \[7.3 tasklet_hi_schedule_first\]\(#\)](#)
- [8 Kill Tasklet
 - \[8.1 tasklet_kill\]\(#\)
 - \[8.1.1 Example\]\(#\)](#)

- [8.2 tasklet_kill_immediate](#)
- [9 Programming](#)
 - [9.1 Driver Source Code](#)
 - [9.2 MakeFile](#)
- [10 Building and Testing Driver](#)
 - [10.0.1 Share this:](#)
 - [10.0.2 Like this:](#)
 - [10.0.3 Related](#)

Prerequisites

This is the continuation of Interrupts in Linux Kernel. So I'd suggest you to know some ideas about Linux Interrupts. You can find the some useful tutorials about Interrupts and Bottom Halves below.

- [1. Interrupts in Linux Kernel](#)
- [2. Interrupts Example Program](#)
- [3. Workqueue Example – Static Method](#)
- [4. Workqueue Example – Dynamic Method](#)
- [5. Workqueue Example – Own Workqueue](#)

Bottom Half

When Interrupt triggers, Interrupt Handler should be execute very quickly and it should not run for more time (it should not perform time-consuming tasks). If we have the interrupt handler which is doing more tasks then we need to divide into two halves.

1. Top Half
2. Bottom Half

Top Half is nothing but our interrupt handler. If our interrupt handler is doing less task, then top half is more than enough. No need of bottom half in that situation. But if we have more work when interrupt hits, then we need bottom half. The bottom half runs in the future, at a more convenient time, with all interrupts enabled. So, The job of bottom halves is to perform any interrupt-related work not performed by the interrupt handler.

There are 4 bottom half mechanisms available in Linux:

1. Work-queue
2. Threaded IRQs
3. Softirqs
4. **Tasklets**

In this tutorial, we will see Tasklets in Linux Kernel.

Tasklets in Linux Kernel

Tasklets are used to queue up work to be done at a later time. Tasklets can be run in parallel, but the same tasklet cannot be run on multiple CPUs at the same time. Also each tasklet will run only on the CPU that schedules it, to optimize cache usage. Since the thread that queued up the tasklet must complete before it can run the tasklet, race conditions are naturally avoided. However, this arrangement can be suboptimal, as other potentially idle CPUs cannot be used to run the tasklet. Therefore workqueues can, and should be used instead, and workqueues were already discussed [here](#).

In short, a **tasklet** is something like a very small thread that has neither stack, nor context of its own. Such “threads” work quickly and completely.

Points To Remember

Before using Tasklets, you should consider these below points.

- Tasklets are atomic, so we cannot use **sleep()** and such

synchronization primitives as **mutexes**, **semaphores**, etc. from them. But we can use **spinlock**.

- A tasklet only runs on the same core (CPU) that schedules it.
- Different tasklets can be running in parallel. But at the same time, a tasklet cannot be called concurrently with itself, as it runs on one CPU only.
- Tasklets are executed by the principle of non-preemptive scheduling, one by one, in turn. We can schedule them with two different priorities: **normal** and **high**.

We can create tasklet in Two ways.

1. Static Method

2. Dynamic Method

In this tutorial we will see static method.

Tasklet Structure

This is the important data structure for the tasklet.

```
1 struct tasklet_struct
2 {
3     struct tasklet_struct *next;
4     unsigned long state;
5     atomic_t count;
6     void (*func)(unsigned long);
7     unsigned long data;
8 };
```

Here,

next - The next tasklet in line for scheduling.

state - This state denotes Tasklet's State. **TASKLET_STATE_SCHED (Scheduled)** or **TASKLET_STATE_RUN (Running)**.

count - It holds a nonzero value if the tasklet is disabled and 0 if it is enabled.

func - This is the main function of the tasklet. Pointer to the function that needs to scheduled for execution at a later time.

data - Data to be passed to the function “func”.

Create Tasklet

The below macros used to create a tasklet.

DECLARE_TASKLET

This macro used to create the tasklet structure and assigns the parameters to that structure.

If we are using this macro then tasklet will be in enabled state.

```
DECLARE_TASKLET(name, func, data);
```

name - name of the structure to be create.

func - This is the main function of the tasklet. Pointer to the function that needs to scheduled for execution at a later time.

data - Data to be passed to the function “func”.

Example

```
1 DECLARE_TASKLET(tasklet,tasklet_fn, 1);
```

Now we will see how the macro is working. When I call the macro like above, first it creates tasklet structure with the name of tasklet. Then it assigns the parameter to that structure. It will be looks like below.

```
1 struct tasklet_struct tasklet = { NULL, 0, 0, tasklet_fn, 1 };
2
3             (or)
4
5 struct tasklet_struct tasklet;
6 tasklet.next = NULL;
7 tasklet.state = TASKLET_STATE_SCHED; //Tasklet state is scheduled
8 tasklet.count = 0;                  //tasklet enabled
9 tasklet.func = tasklet_fn;         //function
10 tasklet.data = 1;                 //data arg
```

DECLARE_TASKLET_DISABLED

The tasklet can be declared and set at disabled state, which means that tasklet can be scheduled, but will not run until the tasklet is specifically enabled. You need to use **tasklet_enable** to enable.

DECLARE_TASKLET_DISABLED(name, func, data);

name – name of the structure to be create.

func – This is the main function of the tasklet. Pointer to the function that needs to scheduled for execution at a later time.

data – Data to be passed to the function “func”.

Enable and Disable Tasklet

tasklet_enable

This used to enable the tasklet.

```
void tasklet_enable(struct);
```

t - pointer to the tasklet struct

tasklet_disable

This used to disable the tasklet wait for the completion of tasklet's operation.

```
void tasklet_disable(struct tasklet_struct *t);
```

t - pointer to the tasklet struct

tasklet_disable_nosync

This used to disables immediately.

```
void tasklet_disable_nosync(struct tasklet_struct *t);
```

t - pointer to the tasklet struct

NOTE : If the tasklet has been disabled, we can still add it to the queue for scheduling, but it will not be executed on the CPU until it is enabled again. Moreover, if the tasklet has been disabled several times, it should be enabled exactly the same number of times, there is the count field in the structure for this purpose.

Schedule Tasklet

When we schedule the tasklet, then that tasklet is placed into one queue out of two, depending on the priority. Queues are organized as singly-

linked lists. At that, each CPU has its own queues.

There are two priorities.

1. Normal Priority
2. High Priority

tasklet_schedule

Schedule a tasklet with normal priority. If a tasklet has previously been scheduled (but not yet run), the new schedule will be silently discarded.

```
void tasklet_schedule (struct tasklet_struct *t);
```

t – pointer to the tasklet struct

Example

```
1 /*Scheduling Task to Tasklet*/  
2 tasklet_schedule(&tasklet);
```

tasklet_hi_schedule

Schedule a tasklet with high priority. If a tasklet has previously been scheduled (but not yet run), the new schedule will be silently discarded.

```
void tasklet_hi_schedule (struct tasklet_struct *t);
```

t – pointer to the tasklet struct

tasklet_hi_schedule_first

This version avoids touching any other tasklets. Needed for kmemcheck in order not to take any page faults while enqueueing this tasklet. Consider VERY carefully whether you really need this or tasklet_hi_schedule().

```
void tasklet_hi_schedule_first(struct tasklet_struct *t);
```

t – pointer to the tasklet struct

Kill Tasklet

Finally, after a tasklet has been created, it's possible to delete a tasklet through these below functions.

tasklet_kill

This will wait for its completion, and then kill it.

```
void tasklet_kill( struct tasklet_struct *t );
```

t - pointer to the tasklet struct

Example

```
1 /*Kill the Tasklet */  
2 tasklet_kill(&tasklet);
```

tasklet_kill_immediate

This is used only when a given CPU is in the dead state.

```
void tasklet_kill_immediate( struct tasklet_struct *t, unsigned  
                           int cpu );
```

t - pointer to the tasklet struct

cpu - cpu num

Programming

Driver Source Code

In that source code, When we read the /dev/etx_device interrupt will hit (To understand interrupts in Linux go to [this tutorial](#)). Whenever interrupt hits, I'm scheduling the task to the tasklet. I'm not going to do any job in both interrupt handler and tasklet function, since it is a tutorial post. But in real tasklet, this function can be used to carry out any operations that need to be scheduled.

NOTE: In this source code many unwanted functions will be there (which is not related to the Tasklet). Because I'm just maintaining the source code throughout these Device driver series.

```
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 #include <linux/interrupt.h>
13 #include <asm/io.h>
14
15
16 #define IRQ_NO 11
17
18 void tasklet_fn(unsigned long);
19
20 /* Init the Tasklet by Static Method */
21 DECLARE_TASKLET(tasklet,tasklet_fn, 1);
22
23
24 /*Tasklet Function*/
25 void tasklet_fn(unsigned long arg)
26 {
27     printk(KERN_INFO "Executing Tasklet Function : arg = %ld\n", arg);
28 }
29
30
31 //Interrupt handler for IRQ 11.
32 static irqreturn_t irq_handler(int irq,void *dev_id) {
33     printk(KERN_INFO "Shared IRQ: Interrupt Occurred");
34     /*Scheduling Task to Tasklet*/
35     tasklet_schedule(&tasklet);
36
37     return IRQ_HANDLED;
38 }
39
40
41 volatile int etx_value = 0;
42
43
44 dev_t dev = 0;
45 static struct class *dev_class;
46 static struct cdev etx_cdev;
47 struct kobject *kobj_ref;
48
49 static int __init etx_driver_init(void);
50 static void __exit etx_driver_exit(void);
51
52 /***** Driver Fuctions *****/
53 static int etx_open(struct inode *inode, struct file *file);
54 static int etx_release(struct inode *inode, struct file *file);
55 static ssize_t etx_read(struct file *filp,
56                         char __user *buf, size_t len, loff_t * off);
```

```
57 static ssize_t etx_write(struct file *filp,
58                         const char *buf, size_t len, loff_t * off);
59
60 /***** Sysfs Fuctions *****/
61 static ssize_t sysfs_show(struct kobject *kobj,
62                         struct kobj_attribute *attr, char *buf);
63 static ssize_t sysfs_store(struct kobject *kobj,
64                         struct kobj_attribute *attr,const char *buf, size_t count);
65
66 struct kobj_attribute etx_attr = __ATTR(etx_value, 0660, sysfs_show, sysfs_store);
67
68 static struct file_operations fops =
69 {
70     .owner        = THIS_MODULE,
71     .read         = etx_read,
72     .write        = etx_write,
73     .open          = etx_open,
74     .release      = etx_release,
75 };
76
77 static ssize_t sysfs_show(struct kobject *kobj,
78                         struct kobj_attribute *attr, char *buf)
79 {
80     printk(KERN_INFO "Sysfs - Read!!!\n");
81     return sprintf(buf, "%d", etx_value);
82 }
83
84 static ssize_t sysfs_store(struct kobject *kobj,
85                         struct kobj_attribute *attr,const char *buf, size_t count)
86 {
87     printk(KERN_INFO "Sysfs - Write!!!\n");
88     sscanf(buf,"%d",&etx_value);
89     return count;
90 }
91
92 static int etx_open(struct inode *inode, struct file *file)
93 {
94     printk(KERN_INFO "Device File Opened...!!!\n");
95     return 0;
96 }
97
98 static int etx_release(struct inode *inode, struct file *file)
99 {
100    printk(KERN_INFO "Device File Closed...!!!\n");
101    return 0;
102 }
103
104 static ssize_t etx_read(struct file *filp,
105                         char __user *buf, size_t len, loff_t *off)
106 {
107     printk(KERN_INFO "Read function\n");
108     asm("int $0x3B"); // Corresponding to irq 11
109     return 0;
110 }
111 static ssize_t etx_write(struct file *filp,
112                         const char __user *buf, size_t len, loff_t *off)
113 {
114     printk(KERN_INFO "Write Function\n");
115     return 0;
116 }
117
118
119 static int __init etx_driver_init(void)
```

```

120 {
121     /*Allocating Major number*/
122     if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
123         printk(KERN_INFO "Cannot allocate major number\n");
124         return -1;
125     }
126     printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
127
128     /*Creating cdev structure*/
129     cdev_init(&etx_cdev,&fops);
130
131     /*Adding character device to the system*/
132     if((cdev_add(&etx_cdev,dev,1)) < 0){
133         printk(KERN_INFO "Cannot add the device to the system\n");
134         goto r_class;
135     }
136
137     /*Creating struct class*/
138     if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
139         printk(KERN_INFO "Cannot create the struct class\n");
140         goto r_class;
141     }
142
143     /*Creating device*/
144     if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
145         printk(KERN_INFO "Cannot create the Device I\n");
146         goto r_device;
147     }
148
149     /*Creating a directory in /sys/kernel/ */
150     kobj_ref = kobject_create_and_add("etx_sysfs",kernel_kobj);
151
152     /*Creating sysfs file for etx_value*/
153     if(sysfs_create_file(kobj_ref,&etx_attr.attr)){
154         printk(KERN_INFO "Cannot create sysfs file.....\n");
155         goto r_sysfs;
156     }
157     if (request_irq(IRQ_NO, irq_handler, IRQF_SHARED, "etx_device", (void *)(irq_h
158         printk(KERN_INFO "my_device: cannot register IRQ ");
159         goto irq;
160     }
161
162     printk(KERN_INFO "Device Driver Insert...Done!!!\n");
163     return 0;
164
165 irq:
166     free_irq(IRQ_NO,(void *)(irq_handler));
167
168 r_sysfs:
169     kobject_put(kobj_ref);
170     sysfs_remove_file(kernel_kobj, &etx_attr.attr);
171
172 r_device:
173     class_destroy(dev_class);
174 r_class:
175     unregister_chrdev_region(dev,1);
176     cdev_del(&etx_cdev);
177     return -1;
178 }
179
180 void __exit etx_driver_exit(void)
181 {
182     /*Kill the Tasklet */

```

```

183     tasklet_kill(&tasklet);
184     free_irq(IRQ_NO,(void *)(irq_handler));
185     kobject_put(kobj_ref);
186     sysfs_remove_file(kernel_kobj, &etx_attr.attr);
187     device_destroy(dev_class,dev);
188     class_destroy(dev_class);
189     cdev_del(&etx_cdev);
190     unregister_chrdev_region(dev, 1);
191     printk(KERN_INFO "Device Driver Remove...Done!!!\n");
192 }
193
194 module_init(etx_driver_init);
195 module_exit(etx_driver_exit);
196
197 MODULE_LICENSE("GPL");
198 MODULE_AUTHOR("EmbeTronicX <embedtronicx@gmail.com>");
199 MODULE_DESCRIPTION("A simple device driver - Tasklet part 1");
200 MODULE_VERSION("1.15");

```

MakeFile

```

1 obj-m += driver.o
2
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*
- To trigger interrupt read device file (*sudo cat /dev/etx_device*)
- Now see the Dmesg (*dmesg*)

linux@embedtronicx-VirtualBox: dmesg

```

[ 8592.698763] Major = 246 Minor = 0
[ 8592.703380] Device Driver Insert...Done!!!
[ 8601.716673] Device File Opened...!!!
[ 8601.716697] Read function
[ 8601.716727] Shared IRQ: Interrupt Occurred
[ 8601.716732] Executing Tasklet Function : arg = 1
[ 8601.716741] Device File Closed....!!!
[ 8603.916741] Device Driver Remove...Done!!!

```

- We can able to see the print “**Shared IRQ: Interrupt Occurred**” and “**Executing Tasklet Function : arg = 1**”
- Unload the module using sudo rmmod driver

In our [next tutorial](#) we will discuss Tasklet using Dynamic Method.

Share this:

[Share on Tumblr](#) [Tweet](#)



[Pocket](#)



[pinit_fg_en_rect_gray_20](#) Linux Device Driver Tutorial Part 20 – Tasklet | Static Method

Like this:

Loading...



report this ad



Sidebar ▾

Device Drivers

Tasklets in Linux Driver

Linux Device Driver Tutorial Part 21 – Tasklets | Dynamic Method

This is the [Series on Linux Device Driver](#). The aim of this series is to provide the easy and practical examples that anyone can understand. This is the Linux Device Driver Tutorial Part 21 – Tasklets Dynamic Method Tutorial.

Post Contents

1 Prerequisites
2 Tasklets in Linux Driver
3 Introduction
4 Dynamically Creation of Tasklet
4.1 tasklet_init
4.1.1 Example
5 Programming
5.1 Driver Source Code
5.2 MakeFile
6 Building and Testing Driver
6.0.1 Share this:
6.0.2 Like this:
6.0.3 Related

Prerequisites

This is the continuation of Interrupts in Linux Kernel. So I'd suggest you to know some ideas about Linux Interrupts. You can find the some useful tutorials about Interrupts and Bottom Halves below.

1. [Interrupts in Linux Kernel](#)
2. [Interrupts Example Program](#)
3. [Workqueue Example – Static Method](#)
4. [Workqueue Example – Dynamic Method](#)
5. [Workqueue Example – Own Workqueue](#)
6. [Tasklet Example – Static Method](#)

Tasklets in Linux Driver

Introduction

In our [Previous Tutorial](#) we have seen the Tasklet using Static Method. In that method we had initialized the tasklet statically. But in this tutorial we are going to initialize the tasklet using dynamically. **So except creation of the tasklet, everything will be same as Previous tutorial. Please refer previous tutorial for Scheduling, Enable, Disable, Kill the Tasklet.**

Dynamically Creation of Tasklet

tasklet_init

This function used to Initialize the tasklet in dynamically.

```
void tasklet_init ( struct tasklet_struct *t,
                    void(*)(unsigned long) func,
                    unsigned long data
)
```

Where,

t - tasklet struct that should be initialized

func - This is the main function of the tasklet. Pointer to the function that needs to scheduled for execution at a later time.

data - Data to be passed to the function “func”.

Example

```
1 /* Tasklet by Dynamic Method */
2 struct tasklet_struct *tasklet;
3
4 /* Init the tasklet bt Dynamic Method */
5 tasklet = kmalloc(sizeof(struct tasklet_struct), GFP_KERNEL);
6 if(tasklet == NULL) {
7     printk(KERN_INFO "etx_device: cannot allocate Memory");
8 }
9 tasklet_init(tasklet,tasklet_fn,0);
```

Now we will see how the function is working in background. When I call the function like above, it assigns the parameter to the passed tasklet structure. It will look like below.

```
1 tasklet->func = tasklet_fn;           //function
2 tasklet->data = 0;                   //data arg
3 tasklet->state = TASKLET_STATE_SCHED; //Tasklet state is scheduled
4 atomic_set(&tasklet->count, 0);       //tasklet enabled
```

NOTE : Please refer previous tutorial for rest of the function like Scheduling, Enable, Disable, Kill the Tasklet.

Programming Driver Source Code

In that source code, When we read the /dev/etx_device interrupt will hit (To understand interrupts in Linux go to [this tutorial](#)). Whenever interrupt hits, I'm scheduling the task to the tasklet. I'm not going to do any job in both interrupt handler and tasklet function (only print), since it is a tutorial post. But in real tasklet, this function can be used to carry out any operations that need to be scheduled.

NOTE: In this source code many unwanted functions will be there (which is not related to the Tasklet). Because I'm just maintaining the source code throughout these Device driver series.

```
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 #include <linux/interrupt.h>
13 #include <asm/io.h>
14
15
16 #define IRQ_NO 11
17
18 void tasklet_fn(unsigned long);
19
20 /* Tasklet by Dynamic Method */
21 struct tasklet_struct *tasklet;
22
23
24 /*Tasklet Function*/
25 void tasklet_fn(unsigned long arg)
26 {
27     printk(KERN_INFO "Executing Tasklet Function : arg = %ld\n", arg);
28 }
29
30
31 //Interrupt handler for IRQ 11.
32 static irqreturn_t irq_handler(int irq,void *dev_id) {
33     printk(KERN_INFO "Shared IRQ: Interrupt Occurred");
34     /*Scheduling Task to Tasklet*/
35     tasklet_schedule(tasklet);
36
37     return IRQ_HANDLED;
38 }
39
40
41 volatile int etx_value = 0;
42
43
44 dev_t dev = 0;
45 static struct class *dev_class;
46 static struct cdev etx_cdev;
47 struct kobject *kobj_ref;
48
49 static int __init etx_driver_init(void);
50 static void __exit etx_driver_exit(void);
51
52 /***** Driver Functions *****/
53 static int etx_open(struct inode *inode, struct file *file);
54 static int etx_release(struct inode *inode, struct file *file);
55 static ssize_t etx_read(struct file *filp,
56                         char __user *buf, size_t len, loff_t * off);
57 static ssize_t etx_write(struct file *filp,
58                         const char *buf, size_t len, loff_t * off);
59
60 /***** Sysfs Fuctions *****/
61 static ssize_t sysfs_show(struct kobject *kobj,
62                         struct kobj_attribute *attr, char *buf);
63 static ssize_t sysfs_store(struct kobject *kobj,
64                         struct kobj_attribute *attr,const char *buf, size_t count);
65
66 struct kobj_attribute etx_attr = __ATTR(etx_value, 0660, sysfs_show, sysfs_store);
```

```

67
68 static struct file_operations fops =
69 {
70     .owner          = THIS_MODULE,
71     .read           = etx_read,
72     .write          = etx_write,
73     .open            = etx_open,
74     .release        = etx_release,
75 };
76
77 static ssize_t sysfs_show(struct kobject *kobj,
78                           struct kobj_attribute *attr, char *buf)
79 {
80     printk(KERN_INFO "Sysfs - Read!!!\n");
81     return sprintf(buf, "%d", etx_value);
82 }
83
84 static ssize_t sysfs_store(struct kobject *kobj,
85                           struct kobj_attribute *attr, const char *buf, size_t count)
86 {
87     printk(KERN_INFO "Sysfs - Write!!!\n");
88     sscanf(buf, "%d", &etx_value);
89     return count;
90 }
91
92 static int etx_open(struct inode *inode, struct file *file)
93 {
94     printk(KERN_INFO "Device File Opened...!!!\n");
95     return 0;
96 }
97
98 static int etx_release(struct inode *inode, struct file *file)
99 {
100    printk(KERN_INFO "Device File Closed...!!!\n");
101    return 0;
102 }
103
104 static ssize_t etx_read(struct file *filp,
105                        char __user *buf, size_t len, loff_t *off)
106 {
107     printk(KERN_INFO "Read function\n");
108     asm("int $0x3B"); // Corresponding to irq 11
109     return 0;
110 }
111 static ssize_t etx_write(struct file *filp,
112                         const char __user *buf, size_t len, loff_t *off)
113 {
114     printk(KERN_INFO "Write Function\n");
115     return 0;
116 }
117
118
119 static int __init etx_driver_init(void)
120 {
121     /*Allocating Major number*/
122     if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
123         printk(KERN_INFO "Cannot allocate major number\n");
124         return -1;
125     }
126     printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
127
128     /*Creating cdev structure*/
129     cdev_init(&etx_cdev,&fops);

```

```

130
131     /*Adding character device to the system*/
132     if((cdev_add(&etx_cdev,dev,1)) < 0){
133         printk(KERN_INFO "Cannot add the device to the system\n");
134         goto r_class;
135     }
136
137     /*Creating struct class*/
138     if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
139         printk(KERN_INFO "Cannot create the struct class\n");
140         goto r_class;
141     }
142
143     /*Creating device*/
144     if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
145         printk(KERN_INFO "Cannot create the Device 1\n");
146         goto r_device;
147     }
148
149     /*Creating a directory in /sys/kernel/ */
150     kobj_ref = kobject_create_and_add("etx_sysfs",kernel_kobj);
151
152     /*Creating sysfs file for etx_value*/
153     if(sysfs_create_file(kobj_ref,&etx_attr.attr)){
154         printk(KERN_INFO "Cannot create sysfs file.....\n");
155         goto r_sysfs;
156     }
157     if (request_irq(IRQ_NO, irq_handler, IRQF_SHARED, "etx_device", (void *)(irq_h
158         printk(KERN_INFO "etx_device: cannot register IRQ ");
159         goto irq;
160     }
161
162     /* Init the tasklet bt Dynamic Method */
163     tasklet = kmalloc(sizeof(struct tasklet_struct),GFP_KERNEL);
164     if(tasklet == NULL) {
165         printk(KERN_INFO "etx_device: cannot allocate Memory");
166         goto irq;
167     }
168     tasklet_init(tasklet,tasklet_fn,0);
169
170     printk(KERN_INFO "Device Driver Insert...Done!!!\n");
171     return 0;
172
173
174 irq:
175     free_irq(IRQ_NO,(void *)(irq_handler));
176
177 r_sysfs:
178     kobject_put(kobj_ref);
179     sysfs_remove_file(kernel_kobj, &etx_attr.attr);
180
181 r_device:
182     class_destroy(dev_class);
183 r_class:
184     unregister_chrdev_region(dev,1);
185     cdev_del(&etx_cdev);
186     return -1;
187 }
188
189 void __exit etx_driver_exit(void)
190 {
191     /* Kill the Tasklet */
192     tasklet_kill(tasklet);

```

```

193     free_irq(IRQ_NO,(void *)(irq_handler));
194     kobject_put(kobj_ref);
195     sysfs_remove_file(kernel_kobj, &etx_attr.attr);
196     device_destroy(dev_class,dev);
197     class_destroy(dev_class);
198     cdev_del(&etx_cdev);
199     unregister_chrdev_region(dev, 1);
200     printk(KERN_INFO "Device Driver Remove...Done!!!\n");
201 }
202
203 module_init(etx_driver_init);
204 module_exit(etx_driver_exit);
205
206 MODULE_LICENSE("GPL");
207 MODULE_AUTHOR("EmbeTronicX <embedtronicx@gmail.com>");
208 MODULE_DESCRIPTION("A simple device driver - Tasklet part 2");
209 MODULE_VERSION("1.16");

```

MakeFile

```

1 obj-m += driver.o
2
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*
- To trigger interrupt read device file (*sudo cat /dev/etx_device*)
- Now see the Dmesg (*dmesg*)

linux@embedtronicx-VirtualBox: dmesg

```

[12372.451624] Major = 246 Minor = 0
[12372.456927] Device Driver Insert...Done!!!
[12375.112089] Device File Opened...!!!
[12375.112109] Read function
[12375.112134] Shared IRQ: Interrupt Occurred
[12375.112139] Executing Tasklet Function : arg = 0
[12375.112147] Device File Closed...!!!
[12377.954952] Device Driver Remove...Done!!!

```

- We can able to see the print “**Shared IRQ: Interrupt Occurred**” and “**Executing Tasklet Function : arg = 0**”
- Unload the module using sudo rmmod driver

Share this:[Share on Tumblr](#) [Tweet](#)[WhatsApp](#)[Pocket](#)

[pinit_fg_en_rect_gray_20](#) Linux Device Driver Tutorial Part 21 – Tasklets
| Dynamic Method

Like this:

Loading...



report this ad

 ezoic[report this ad](#)[Sidebar ▾](#)

📁 Device Drivers

Linux Device Driver Tutorial Part 17 - Linked List in Linux Kernel

**Linux Device Driver Tutorial Part 17 -
Linked List in Linux Kernel Part 1**

This is the [Series on Linux Device Driver](#). The aim of this series is to provide the easy and practical examples that anyone can understand. In our previous [tutorials](#) we have seen work queue. So this is the Linux Device Driver Tutorial Part 17 – Linked List in Linux Kernel Part 1.

Post Contents

- [1 Linux Device Driver Tutorial Part 17 – Linked List in Linux Kernel](#)
- [2 Introduction about Linked List
 - \[2.1 Advantages of Linked Lists\]\(#\)
 - \[2.2 Disadvantages of Linked Lists\]\(#\)
 - \[2.3 Applications of Linked Lists\]\(#\)](#)
- [3 Types of Linked Lists](#)
- [4 Linked List in Linux Kernel](#)
- [5 Initialize Linked List Head](#)
- [6 Create Node in Linked List](#)
- [7 Add Node to Linked List
 - \[7.1 Add after Head Node\]\(#\)
 - \[7.2 Add before Head Node\]\(#\)](#)
- [8 Delete Node from Linked List
 - \[8.1 list_del\]\(#\)
 - \[8.2 list_del_init\]\(#\)](#)
- [9 Replace Node in Linked List
 - \[9.1 list_replace\]\(#\)
 - \[9.2 list_replace_init\]\(#\)](#)
- [10 Moving Node in Linked List
 - \[10.1 list_move\]\(#\)
 - \[10.2 list_move_tail\]\(#\)](#)

- [11 Rotate Node in Linked List](#)
- [12 Test the Linked List Entry](#)
 - [12.1 list_is_last](#)
 - [12.2 list_empty](#)
 - [12.3 list_is_singular](#)
- [13 Split Linked List into two part](#)
- [14 Join Two Linked Lists](#)
- [15 Traverse Linked List](#)
 - [15.1 list_entry](#)
 - [15.2 list_for_each](#)
 - [15.3 list_for_each_entry](#)
 - [15.4 list_for_each_entry_safe](#)
 - [15.5 list_for_each_prev](#)
 - [15.6 list_for_each_entry_reverse](#)
 - [15.6.1 Share this:](#)
 - [15.6.2 Like this:](#)
 - [15.6.3 Related](#)

Linux Device Driver Tutorial Part 17 - Linked List in Linux Kernel

Introduction about Linked List

A linked list is a data structure that consists of sequence of nodes. Each node is composed of two fields: **data field** and **reference field** which is a [pointer](#) that points to the next node in the sequence.

linked-list-1 Linux Device Driver Tutorial Part 17 - Linked List in Linux Kernel Part 1

Each node in the list is also called an element. A **head** pointer is used to track the first element in the linked list, therefore, it always points to the first element.

The elements do not necessarily occupy contiguous regions in memory and thus need to be linked together (each element in the list contains a pointer to the *next* element).

Advantages of Linked Lists

They are a dynamic in nature which allocates the memory when required.

Insertion and deletion operations can be easily implemented.

Stacks and queues can be easily executed.

Linked List reduces the access time.

Disadvantages of Linked Lists

The memory is wasted as pointers require extra memory for storage.

No element can be accessed randomly; it has to access each node sequentially.

Reverse Traversing is difficult in linked list.

Applications of Linked Lists

Linked lists are used to implement stacks, queues, graphs, etc.

Unlike array, In Linked Lists we don't need to know the size in advance.

Types of Linked Lists

There are three types of linked lists.

- Singly Linked List
- Doubly Linked List
- Circular Linked List

I'm not going to discuss about its types. Let's get into the Linked List in Linux kernel.

Linked List in Linux Kernel

Linked list is a very important data structure which allows large number of storage with efficient manipulation on data. Most of the kernel code has been written with help of this data structure. So in Linux kernel no need to implement our own Linked List or no need to use 3rd party library. It has built in Linked List which is Doubly Linked List. It is defined in /lib/modules/\$(uname -r)/build/include/linux/list.h.

Normally we used to declared linked list as like below snippet.

```
1 struct my_list{  
2     int data,  
3     struct my_list *prev;  
4     struct my_list *next;  
5 };
```

But if want to Implement in Linux, then you could write like below snippet.

```
1 struct my_list{  
2     struct list_head list;      //linux kernel list implementation  
3     int data;  
4 };
```

Where struct list_head is declared in list.h.

```
1 struct list_head {  
2     struct list_head *next;  
3     struct list_head *prev;  
4 };
```

Initialize Linked List Head

Before creating any node in linked list, we should create linked list's head node first. So below macro is used to create a head node.

```
LIST_HEAD(linked_list);
```

This macro will create the head node structure in the name of “linked_list” and it will initialize that to its own address.

For example,

I'm going to create head node in the name of “etx_linked_list”.

```
LIST_HEAD(etx_linked_list);
```

Let's see how internally it handles this. The macro is defined like below in `list.h`.

```

1 #define LIST_HEAD_INIT(name) { &(name), &(name) }
2
3 #define LIST_HEAD(name) \
4     struct list_head name = LIST_HEAD_INIT(name)
5
6 struct list_head {
7     struct list_head *next;
8     struct list_head *prev;
9 };

```

So it will create like below.

```
1 struct list_head etx_linked_list = { &etx_linked_list , &etx_linked_list};
```

While creating head node, it initializes the prev and next pointer to its own address. Which means that prev and next pointer points to itself. The node is empty If the node's prev and next pointer points to itself.

Create Node in Linked List

You have to create your linked list node dynamically or statically. Your linked list node should have member defined in `struct list_head`. Using below inline function, we can initialize that `struct list_head`.

```
INIT_LIST_HEAD(struct list_head *list);
```

For Example, My node is like this.

```

1 struct my_list{
2     struct list_head list;      //linux kernel list implementation
3     int data;
4 };
5
6 struct my_list new_node;

```

So we have to initialize the `list_head` variable using `INIT_LIST_HEAD` inline function.

```

1 INIT_LIST_HEAD(&new_node.list);
2 new_node.data = 10;

```

Add Node to Linked List

Add after Head Node

After created that node, we need to add that node to the linked list. So we can use this inline function to do that.

```
inline void list_add(struct list_head *new, struct list_head  
                      *head);
```

Insert a new entry **after the specified head**. This is good for implementing stacks.

Where,

struct list_head * new - new entry to be added

struct list_head * head - list head to add it after

For Example,

```
1 list_add(&new_node.list, &etx_linked_list);
```

Add before Head Node

Insert a new entry before the specified head. This is useful for implementing queues.

```
inline void list_add_tail(struct list_head *new, struct  
                           list_head *head);
```

Where,

struct list_head * new - new entry to be added

struct list_head * head - list head to add before the head

For Example,

```
1 list_add_tail(&new_node.list, &etx_linked_list);
```

Delete Node from Linked List

list_del

It will delete the entry node from the list. This function removes the entry node from the linked list by disconnect prev and next pointers from the list, but it doesn't free any memory space allocated for entry node.

```
inline void list_del(struct list_head *entry);
```

Where,

struct list_head * entry- the element to delete from the list.

list_del_init

It will delete the entry node from the list and reinitialize it. This function removes the entry node from the linked list by disconnect prev and next pointers from the list, but it doesn't free any memory space allocated for entry node.

```
inline void list_del_init(struct list_head *entry);
```

Where,

struct list_head * entry- the element to delete from the list.

Replace Node in Linked List

list_replace

This function is used to replace the old node with new node.

```
inline void list_replace(struct list_head *old, struct list_head  
                        *new);
```

Where,

struct list_head * old- the element to be replaced

struct list_head * new- the new element to insert

If *old* was empty, it will be overwritten.

list_replace_init

This function is used to replace the old node with new node and reinitialize the old entry.

```
inline void list_replace_init(struct list_head *old, struct  
                                list_head *new);
```

Where,

struct list_head * old- the element to be replaced

struct list_head * new- the new element to insert

If *old* was empty, it will be overwritten.

Moving Node in Linked List

list_move

This will delete one list from the linked list and again adds to after the head node.

```
inline void list_move(struct list_head *list, struct list_head  
                      *head);
```

Where,

struct list_head * list - the entry to move

struct list_head * head- the head that will precede our entry

list_move_tail

This will delete one list from the linked list and again adds to before the head node.

```
inline void list_move_tail(struct list_head *list, struct  
                           list_head *head);
```

Where,

`struct list_head * list` - the entry to move

`struct list_head * head` - the head that will precede our entry

Rotate Node in Linked List

This will rotate the list to the left.

```
inline void list_rotate_left(struct list_head *head);
```

Where,

`head` - the head of the list

Test the Linked List Entry

list_is_last

This tests whether `list` is the last entry in list `head`.

```
inline int list_is_last(const struct list_head *list, const  
                      struct list_head *head);
```

Where,

`const struct list_head * list` - the entry to test

`const struct list_head * head` - the head of the list

It returns **1** if it is last entry otherwise **0**.

list_empty

It tests whether a list is empty or not.

```
inline int list_empty(const struct list_head *head);
```

Where,

const struct list_head * head - the head of the list

It returns **1** if it is empty otherwise **0**.

list_is_singular

This will tests whether a list has just one entry.

```
inline int list_is_singular(const struct list_head *head);
```

Where,

const struct list_head * head - the head of the list

It returns **1** if it has only one entry otherwise **0**.

Split Linked List into two part

This cut a list into two.

This helper moves the initial part of *head*, up to and including *entry*, from *head* to *list*. You should pass on *entry* an element you know is on *head*. *list* should be an empty list or a list you do not care about losing its data.

```
inline void list_cut_position(struct list_head *list, struct  
                                list_head *head, struct list_head *entry);
```

Where,

struct list_head * list - a new list to add all removed entries

struct list_head * head- a list with entries

struct list_head * entry- an entry within head, could be the head itself

and if so we won't cut the list

Join Two Linked Lists

This will join two lists, this is designed for stacks.

```
inline void list_splice(const struct list_head *list, struct  
                      list_head *head);
```

Where,

const struct list_head * list - the new list to add.

struct list_head * head - the place to add it in the first list.

Traverse Linked List

list_entry

This macro is used to get the struct for this entry.

```
list_entry(ptr, type, member);
```

ptr - the struct list_head pointer.

type - the type of the struct this is embedded in.

member - the name of the list_head within the struct.

list_for_each

This macro used to iterate over a list.

```
list_for_each(pos, head);
```

pos - the &struct list_head to use as a loop cursor.

head - the head for your list.

So using those above two macros, we can traverse the linked list. We will see the example in next tutorial. We can also use these below methods also.

list_for_each_entry

This is used to iterate over list of given type.

```
list_for_each_entry(pos, head, member);
```

pos - the type * to use as a loop cursor.

head - the head for your list.

member - the name of the list_head within the struct.

list_for_each_entry_safe

This will iterate over list of given type safe against removal of list entry.

```
list_for_each_entry_safe ( pos, n, head, member);
```

Where,

pos - the type * to use as a loop cursor.

n - another type * to use as temporary storage

head - the head for your list.

member - the name of the list_head within the struct.

We can also traverse the linked list in reverse side also using below macros.

list_for_each_prev

This will used to iterate over a list backwards.

```
list_for_each_prev(pos, head);
```

pos - the &struct list_head to use as a loop cursor.

head - the head for your list.

list_for_each_entry_reverse

This macro used to iterate backwards over list of given type.

```
list_for_each_entry_reverse(pos, head, member);
```

pos - the type * to use as a loop cursor.

head the head for your list.

member - the name of the list_head within the struct.

So, We have gone through all the functions which is useful for Kernel Linked List. Please go through the next tutorial ([Part 2](#)) for Linked List sample program .

Share this:

[Share on Tumblr](#)[Tweet](#)[Print](#)[WhatsApp](#)[Pocket](#)[Telegram](#)

[pinit_fg_en_rect_gray_20](#) [Linux Device Driver Tutorial Part 17 - Linked List in Linux Kernel Part 1](#)

Like this:

Loading...



[report this ad](#)

 ezoic[report this ad](#)[Sidebar ▾](#)

📁 Device Drivers

Example Linked List in Linux Kernel

**Linux Device Driver Tutorial Part 18 -
Linked List in Linux Kernel Part 2**

This is the [Series on Linux Device Driver](#). The aim of this series is to provide the easy and practical examples that anyone can understand. In our previous [tutorial](#) we have seen the functions used in Kernel Linked List. So this is the Linux Device Driver Tutorial Part 18 - Example Linked List in Linux Kernel which is continuation (Part 2) of [Previous Tutorial](#).

Post Contents

- [1 Linux Device Driver Tutorial Part 18 – Example Linked List in Linux Kernel](#)
- [2 Creating Head Node](#)
- [3 Creating Node and add that into Linked List](#)
- [4 Traversing Linked List](#)
- [5 Deleting Linked List](#)
- [6 Programming
 - \[6.1 Driver Source Code\]\(#\)
 - \[6.2 MakeFile\]\(#\)](#)
- [7 Building and Testing Driver
 - \[7.0.1 Share this:\]\(#\)
 - \[7.0.2 Like this:\]\(#\)
 - \[7.0.3 Related\]\(#\)](#)

Linux Device Driver Tutorial Part 18 – Example Linked List in Linux Kernel

If you don't know the functions used in linked list, please refer [this previous tutorial](#) for the detailed explanation about all linked list functions.

So now we can directly enter into the Linux Linked List Kernel programming. I took the source code form the previous tutorial. First i will explain how this code works.

1. When we write the value to our device file using echo value > /dev/etx_value, it will invoke the interrupt. Because we configured the interrupt by using software. If you don't know how it works, Please [refer this tutorial](#).
2. Interrupt will invoke the ISR function.
3. In ISR we are allocating work to the Workqueue.
4. Whenever Workqueue executing, we are creating Linked List Node and adding the Node to the Linked List.
5. When we are reading the driver using cat /dev/etx_device, printing all the nodes which is present in the Linked List using traverse.
6. When we are removing the driver using rmmod, it will removes all the nodes in Linked List and free the memory.

Note : We are not using the sysfs functions. So I kept empty sysfs functions.

Creating Head Node

```
1 /*Declare and init the head node of the linked list*/  
2 LIST_HEAD(Head_Node);
```

This will create the head node in the name of Head_Node and initialize that.

Creating Node and add that into Linked List

```

1      /*Creating Node*/
2      temp_node = kmalloc(sizeof(struct my_list), GFP_KERNEL);
3
4      /*Assgin the data that is received*/
5      temp_node->data = etx_value;
6
7      /*Init the list within the struct*/
8      INIT_LIST_HEAD(&temp_node->list);
9
10     /*Add Node to Linked List*/
11     list_add_tail(&temp_node->list, &Head_Node);

```

This will create the node, assign the data to its member. Then finally add that node to the Linked List using `list_add_tail`. (*This part will be present in the workqueue function*)

Traversing Linked List

```

1      struct my_list *temp;
2      int count = 0;
3      printk(KERN_INFO "Read function\n");
4
5      /*Traversing Linked List and Print its Members*/
6      list_for_each_entry(temp, &Head_Node, list) {
7          printk(KERN_INFO "Node %d data = %d\n", count++, temp->data);
8      }
9
10     printk(KERN_INFO "Total Nodes = %d\n", count);

```

Here, we are traversing each nodes using `list_for_each_entry` and print those values. (*This part will be present in the read function*)

Deleting Linked List

```

1      /* Go through the list and free the memory. */
2      struct my_list *cursor, *temp;
3      list_for_each_entry_safe(cursor, temp, &Head_Node, list) {
4          list_del(&cursor->list);
5          kfree(cursor);
6      }

```

This will traverse the each node using `list_for_each_entry_safe` and delete that using `list_del`. Finally we need to free the memory which is allocated using `kmalloc`.

Programming

Driver Source Code

```
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 #include <linux/interrupt.h>
13 #include <asm/io.h>
14 #include <linux/workqueue.h>       // Required for workqueues
15
16
17 #define IRQ_NO 11
18
19 volatile int etx_value = 0;
20
21 dev_t dev = 0;
22 static struct class *dev_class;
23 static struct cdev etx_cdev;
24 struct kobject *kobj_ref;
25
26 static int __init etx_driver_init(void);
27 static void __exit etx_driver_exit(void);
28
29 static struct workqueue_struct *own_workqueue;
30
31
32 static void workqueue_fn(struct work_struct *work);
33
34 static DECLARE_WORK(work, workqueue_fn);
35
36 /*Linked List Node*/
37 struct my_list{
38     struct list_head list;      //linux kernel list implementation
39     int data;
40 };
41
42 /*Declare and init the head node of the linked list*/
43 LIST_HEAD(Head_Node);
44
45 /***** Driver Fuctions *****/
46 static int etx_open(struct inode *inode, struct file *file);
47 static int etx_release(struct inode *inode, struct file *file);
48 static ssize_t etx_read(struct file *filp,
49                         char __user *buf, size_t len, loff_t * off);
50 static ssize_t etx_write(struct file *filp,
51                         const char *buf, size_t len, loff_t * off);
52
53 /***** Sysfs Fuctions *****/
54 static ssize_t sysfs_show(struct kobject *kobj,
55                         struct kobj_attribute *attr, char *buf);
56 static ssize_t sysfs_store(struct kobject *kobj,
```

```

57             struct kobj_attribute *attr,const char *buf, size_t count);
58
59     struct kobj_attribute etx_attr = __ATTR(etx_value, 0660, sysfs_show, sysfs_store);
60     /*************************************************************************/
61
62
63     /*Workqueue Function*/
64     static void workqueue_fn(struct work_struct *work)
65     {
66         struct my_list *temp_node = NULL;
67
68         printk(KERN_INFO "Executing Workqueue Function\n");
69
70         /*Creating Node*/
71         temp_node = kmalloc(sizeof(struct my_list), GFP_KERNEL);
72
73         /*Assgin the data that is received*/
74         temp_node->data = etx_value;
75
76         /*Init the list within the struct*/
77         INIT_LIST_HEAD(&temp_node->list);
78
79         /*Add Node to Linked List*/
80         list_add_tail(&temp_node->list, &Head_Node);
81     }
82
83
84     //Interrupt handler for IRQ 11.
85     static irqreturn_t irq_handler(int irq,void *dev_id) {
86         printk(KERN_INFO "Shared IRQ: Interrupt Occurred\n");
87         /*Allocating work to queue*/
88         queue_work(own_workqueue, &work);
89
90         return IRQ_HANDLED;
91     }
92
93     static struct file_operations fops =
94     {
95         .owner        = THIS_MODULE,
96         .read         = etx_read,
97         .write        = etx_write,
98         .open          = etx_open,
99         .release      = etx_release,
100    };
101
102    static ssize_t sysfs_show(struct kobject *kobj,
103                           struct kobj_attribute *attr, char *buf)
104    {
105        printk(KERN_INFO "Sysfs - Read!!!\n");
106        return sprintf(buf, "%d", etx_value);
107    }
108
109    static ssize_t sysfs_store(struct kobject *kobj,
110                           struct kobj_attribute *attr,const char *buf, size_t count)
111    {
112        printk(KERN_INFO "Sysfs - Write!!!\n");
113        return count;
114    }
115
116    static int etx_open(struct inode *inode, struct file *file)
117    {
118        printk(KERN_INFO "Device File Opened...!!!\n");
119        return 0;

```

```

120 }
121
122 static int etx_release(struct inode *inode, struct file *file)
123 {
124     printk(KERN_INFO "Device File Closed...!!!\n");
125     return 0;
126 }
127
128 static ssize_t etx_read(struct file *filp,
129                         char __user *buf, size_t len, loff_t *off)
130 {
131     struct my_list *temp;
132     int count = 0;
133     printk(KERN_INFO "Read function\n");
134
135     /*Traversing Linked List and Print its Members*/
136     list_for_each_entry(temp, &Head_Node, list) {
137         printk(KERN_INFO "Node %d data = %d\n", count++, temp->data);
138     }
139
140     printk(KERN_INFO "Total Nodes = %d\n", count);
141     return 0;
142 }
143 static ssize_t etx_write(struct file *filp,
144                         const char __user *buf, size_t len, loff_t *off)
145 {
146     printk(KERN_INFO "Write Function\n");
147     /*Copying data from user space*/
148     sscanf(buf, "%d", &etx_value);
149     /* Triggering Interrupt */
150     asm("int $0x3B"); // Corresponding to irq 11
151     return len;
152 }
153
154
155 static int __init etx_driver_init(void)
156 {
157     /*Allocating Major number*/
158     if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) < 0){
159         printk(KERN_INFO "Cannot allocate major number\n");
160         return -1;
161     }
162     printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
163
164     /*Creating cdev structure*/
165     cdev_init(&etx_cdev,&fops);
166
167     /*Adding character device to the system*/
168     if((cdev_add(&etx_cdev,dev,1)) < 0){
169         printk(KERN_INFO "Cannot add the device to the system\n");
170         goto r_class;
171     }
172
173     /*Creating struct class*/
174     if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
175         printk(KERN_INFO "Cannot create the struct class\n");
176         goto r_class;
177     }
178
179     /*Creating device*/
180     if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
181         printk(KERN_INFO "Cannot create the Device \n");
182         goto r_device;

```

```

183     }
184
185     /*Creating a directory in /sys/kernel/ */
186     kobj_ref = kobject_create_and_add("etx_sysfs",kernel_kobj);
187
188     /*Creating sysfs file*/
189     if(sysfs_create_file(kobj_ref,&etx_attr.attr)){
190         printk(KERN_INFO "Cannot create sysfs file.....\n");
191         goto r_sysfs;
192     }
193     if (request_irq(IRQ_NO, irq_handler, IRQF_SHARED, "etx_device", (void *)(irq_h
194         printk(KERN_INFO "my_device: cannot register IRQ \n");
195         goto irq;
196     }
197
198     /*Creating workqueue */
199     own_workqueue = create_workqueue("own_wq");
200
201     printk(KERN_INFO "Device Driver Insert...Done!!!\n");
202     return 0;
203
204 irq:
205     free_irq(IRQ_NO,(void *)(irq_handler));
206
207 r_sysfs:
208     kobject_put(kobj_ref);
209     sysfs_remove_file(kernel_kobj, &etx_attr.attr);
210
211 r_device:
212     class_destroy(dev_class);
213 r_class:
214     unregister_chrdev_region(dev,1);
215     cdev_del(&etx_cdev);
216     return -1;
217 }
218
219 void __exit etx_driver_exit(void)
220 {
221
222     /* Go through the list and free the memory. */
223     struct my_list *cursor, *temp;
224     list_for_each_entry_safe(cursor, temp, &Head_Node, list) {
225         list_del(&cursor->list);
226         kfree(cursor);
227     }
228
229     /* Delete workqueue */
230     destroy_workqueue(own_workqueue);
231     free_irq(IRQ_NO,(void *)(irq_handler));
232     kobject_put(kobj_ref);
233     sysfs_remove_file(kernel_kobj, &etx_attr.attr);
234     device_destroy(dev_class,dev);
235     class_destroy(dev_class);
236     cdev_del(&etx_cdev);
237     unregister_chrdev_region(dev, 1);
238     printk(KERN_INFO "Device Driver Remove...Done!!\n");
239 }
240
241 module_init(etx_driver_init);
242 module_exit(etx_driver_exit);
243
244 MODULE_LICENSE("GPL");
245 MODULE_AUTHOR("EmbeTronicX <embedtronicx@gmail.com or admin@embedtronicx.com>");
```

```
246 MODULE_DESCRIPTION("A simple device driver - Kernel Linked List");
247 MODULE_VERSION("1.13");
```

MakeFile

```
1 obj-m += driver.o
2
3 KDIR = /lib/modules/$(shell uname -r)/build
4
5 all:
6     make -C $(KDIR) M=$(shell pwd) modules
7
8 clean:
9     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*
- *sudo su*
- To trigger interrupt read device file (*cat /dev/etx_device*)
- Now see the Dmesg (*dmesg*)

```
[ 5310.125001] Major = 246 Minor = 0 n
[ 5310.133127] Device Driver Insert...Done!!!
[ 5346.839872] Device File Opened....!!!
```

```
[ 5346.839950] Read function  
[ 5346.839954] Total Nodes = 0  
[ 5346.839982] Device File Closed....!!!
```

- By this time there is no nodes available.
- So now write the value to driver using echo 10 > /dev/etx_device
- By this time, One node has been added to the linked list.
- To test that read the device file using cat /dev/etx_device
- Now see the Dmesg (dmesg)

```
[ 5346.839982] Device File Closed....!!!  
[ 5472.408239] Device File Opened....!!!  
[ 5472.408266] Write Function  
[ 5472.408293] Shared IRQ: Interrupt Occurred  
[ 5472.408309] Device File Closed....!!!  
[ 5472.409037] Executing Workqueue Function  
[ 5551.996018] Device File Opened....!!!  
[ 5551.996040] Read function  
[ 5551.996044] Node 0 data = 10  
[ 5551.996046] Total Nodes = 1  
[ 5551.996052] Device File Closed....!!!
```

- Our value has added to the list.
- You can also write many times to create and add the node to linked list
- Unload the module using rmmod driver

Share this:[Share on Tumblr](#)[Tweet](#)[WhatsApp](#)[Pocket](#)[Telegram](#)

[pinit_fg_en_rect_gray_20 Linux Device Driver Tutorial Part 18 - Linked List in Linux Kernel Part 2](#)

Like this:

Loading...



[report this ad](#)

 ezoic[report this ad](#)[Sidebar ▾](#)

📁 Device Drivers

kernel thread

Linux Device Driver Tutorial Part 19 – Kernel Thread

This is the [Series on Linux Device Driver](#). The aim of this series is to provide the easy and practical examples that anyone can understand. This is the Linux Device Driver Tutorial Part 19 – Kernel Thread.

Post Contents

- [1 Process](#)
- [2 Threads](#)
- [3 Thread Management](#)
- [4 Types of Thread](#)
 - [4.1 User Level Thread](#)
 - [4.2 Kernel Level Thread](#)
- [5 Kernel Thread Management Functions](#)
 - [5.1 Create Kernel Thread](#)
 - [5.1.1 kthread_create](#)
 - [5.2 Start Kernel Thread](#)
 - [5.2.1 wake_up_process](#)
 - [5.3 Stop Kernel Thread](#)
 - [5.3.1 kthread_stop](#)
 - [5.4 Other functions in Kernel Thread](#)
 - [5.4.1 kthread_should_stop](#)
 - [5.4.2 kthread_bind](#)
- [6 Implementation](#)
 - [6.1 Thread Function](#)
 - [6.2 Creating and Starting Kernel Thread](#)
 - [6.2.1 kthread_run](#)
 - [6.3 Stop Kernel Thread](#)
- [7 Driver Source Code – Kthread in Linux](#)

[8 MakeFile](#)[9 Building and Testing Driver](#)[9.0.1 Share this:](#)[9.0.2 Like this:](#)[9.0.3 Related](#)

Process

An executing instance of a program is called a process. Some operating systems use the term ‘task’ to refer to a program that is being executed. **Process** is a heavy weight process. Context switch between the process is time consuming.

Threads

A *thread* is an independent flow of control that operates within the same address space as other independent flows of control within a process.

[threads-vs-process-300x196](#) Linux Device Driver Tutorial Part 19 – Kernel Thread

One process can have multiple threads, with each thread executing different code concurrently, while sharing data and synchronizing much more easily than cooperating processes. Threads require fewer system resources than processes, and can start more quickly. Threads, also known as light weight processes.

Some of the advantages of the thread, is that since all the threads within the processes share the same address space, the communication between the threads is far easier and less time consuming as compared to processes. This approach has one disadvantage also. It leads to several concurrency issues and require the synchronization mechanisms to handle the same.

Thread Management

Whenever we are creating thread, it has to manage by someone. So that management follows like below.

- A thread is a sequence of instructions.
- CPU can handle one instruction at a time.
- To switch between instructions on parallel threads, execution state need to be saved.
- Execution state in its simplest form is a program counter and CPU registers.
- Program counter tells us what instruction to execute next.
- CPU registers hold execution arguments for example addition operands.
- This alternation between threads requires management.
- Management includes saving state, restoring state, deciding what thread to pick next.

Types of Thread

There are two types of thread.

1. User Level Thread
2. Kernel Level Thread

User Level Thread

In this type, kernel is not aware of these threads. Everything is maintained by the user thread library. That thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. So all will be in User Space.

Kernel Level Thread

Kernel level threads are managed by the OS, therefore, thread operations are implemented in the kernel code. There is no thread management code in the application area.

Anyhow each types of the threads have advantages and disadvantages too.

Now we will move into Kernel Thread Programming. First we will see the functions used in kernel thread.

Kernel Thread Management Functions

There are many functions used in Kernel Thread. We will see one by one. We can classify those functions based on functionalities.

- Create Kernel Thread
- Start Kernel Thread
- Stop Kernel Thread
- Other functions in Kernel Thread

For use the below functions you should include `linux/kthread.h` header file.

Create Kernel Thread

kthread_create

create a kthread.

```
struct task_struct * kthread_create (int (* threadfn(void  
*data),  
  
void *data, const char namefmt[], ...);
```

Where,

`threadfn` - the function to run until `signal_pending(current)`.

`data` - data ptr for `threadfn`.

namefmt[] - printf-style name for the thread.

... - variable arguments

This helper function creates and names a kernel thread. But we need to wake up that thread manually. When woken, the thread will run `threadfn()` with data as its argument.

`threadfn` can either call `do_exit` directly if it is a standalone thread for which no one will call `kthread_stop`, or return when '`kthread_should_stop`' is true (which means `kthread_stop` has been called). The return value should be zero or a negative error number; it will be passed to `kthread_stop`.

It Returns

Start Kernel Thread

wake_up_process

This is used to Wake up a specific process.

```
int wake_up_process (struct task_struct * p);
```

Where,

p - The process to be woken up.

Attempt to wake up the nominated process and move it to the set of runnable processes.

It **returns 1** if the process was woken up, **0** if it was already running.

It may be assumed that this function implies a write memory barrier before changing the task state if and only if any tasks are woken up.

Stop Kernel Thread

kthread_stop

It stops a thread created by `kthread_create`.

```
int kthread_stop ( struct task_struct *k);
```

Where,

k - thread created by kthread_create.

Sets kthread_should_stop for k to return true, wakes it, and waits for it to exit. Your threadfn must not call do_exit itself if you use this function! This can also be called after kthread_create instead of calling wake_up_process: the thread will exit without calling threadfn.

It **Returns** the result of threadfn, or -EINTR if wake_up_process was never called.

Other functions in Kernel Thread

kthread_should_stop

should this kthread return now?

```
int kthread_should_stop (void);
```

When someone calls kthread_stop on your kthread, it will be woken and this will return true. You should then return, and your return value will be passed through to kthread_stop.

kthread_bind

This is used to bind a just-created kthread to a cpu.

```
void kthread_bind (struct task_struct *k, unsigned int cpu);
```

Where,

k - thread created by kthread_create.

cpu - cpu (might not be online, must be possible) for k to run on.

Implementation

Thread Function

First we have to create our thread which has the argument of void * and should return int value. We should follow some conditions in our thread function. Its advisable.

- If that thread is a long run thread, we need to check kthread_should_stop() every time as because any function may call kthread_stop. If any function called kthread_stop, that time kthread_should_stop will return true. We have to exit our thread function if true value been returned by kthread_should_stop.
- But if your thread function is not running long, then let that thread finish its task and kill itself using do_exit.

In my thread function, lets print something every minute and it is continuous process. So lets check the kthread_should_stop every time. See the below snippet to understand.

```
1 int thread_function(void *pv)
2 {
3     int i=0;
4     while(!kthread_should_stop()) {
```

```

5      printk(KERN_INFO "In EmbeTronicX Thread Function %d\n", i++);
6      msleep(1000);
7  }
8  return 0;
9 }
```

Creating and Starting Kernel Thread

So as of now, we have our thread function to run. Now, we will create kernel thread using `kthread_create` and start the kernel thread using `wake_up_process`.

```

1 static struct task_struct *etx_thread;
2
3 etx_thread = kthread_create(thread_function,NULL,"eTx Thread");
4 if(etx_thread) {
5     wake_up_process(etx_thread);
6 } else {
7     printk(KERN_ERR "Cannot create kthread\n");
8 }
```

There is another function which does both process (create and start). That is `kthread_run()`. You can replace the both `kthread_create` and `wake_up_process` using this function.

kthread_run

This is used to create and wake a thread.

```
kthread_run (threadfn, data, namefmt, ...);
```

Where,

`threadfn` – the function to run until `signal_pending(current)`.

`data` – data ptr for `threadfn`.

`namefmt` – printf-style name for the thread.

`...` – variable arguments

Convenient wrapper for `kthread_create` followed by `wake_up_process`.

It **returns** the `kthread` or `ERR_PTR(-ENOMEM)`.

You can see the below snippet which is using kthread_run.

```

1 static struct task_struct *etx_thread;
2
3 etx_thread = kthread_run(thread_function,NULL,"eTx Thread");
4 if(etx_thread) {
5     printk(KERN_ERR "Kthread Created Successfully...\n");
6 } else {
7     printk(KERN_ERR "Cannot create kthread\n");
8 }
```

Stop Kernel Thread

You can stop the kernel thread using kthread_stop. Use the below snippet to stop.

```
1 kthread_stop(etx_thread);
```

Driver Source Code - Kthread in Linux

Kernel thread will start when we insert the kernel module. It will print something every second. When we remove the module that time it stops the kernel thread. Let's see the source code.

```

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/kthread.h>        //kernel threads
11 #include <linux/sched.h>          //task_struct
12 #include <linux/delay.h>
13
14
15 dev_t dev = 0;
16 static struct class *dev_class;
17 static struct cdev etx_cdev;
18
19 static int __init etx_driver_init(void);
20 static void __exit etx_driver_exit(void);
21
22 static struct task_struct *etx_thread;
23
24
25 //***** Driver Functions *****/
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
34     int thread_function(void *pv);
35
36     int thread_function(void *pv)
37     {
38         int i=0;
39         while(!kthread_should_stop()) {
40             printk(KERN_INFO "In EmbeTronicX Thread Function %d\n", i++);
41             msleep(1000);
42         }
43         return 0;
44     }
45
46     static struct file_operations fops =
47     {
48         .owner          = THIS_MODULE,
49         .read           = etx_read,
50         .write          = etx_write,
51         .open           = etx_open,
52         .release        = etx_release,
53     };
54
55     static int etx_open(struct inode *inode, struct file *file)
56     {
57         printk(KERN_INFO "Device File Opened...!!!\n");
58         return 0;
59     }
60
61     static int etx_release(struct inode *inode, struct file *file)
62     {
63         printk(KERN_INFO "Device File Closed...!!!\n");
64         return 0;
65     }
66
67     static ssize_t etx_read(struct file *filp,
68                            char __user *buf, size_t len, loff_t *off)
69     {
70         printk(KERN_INFO "Read function\n");
71
72         return 0;
73     }
74     static ssize_t etx_write(struct file *filp,
75                            const char __user *buf, size_t len, loff_t *off)
76     {
77         printk(KERN_INFO "Write Function\n");
78         return len;
79     }
80
81     static int __init etx_driver_init(void)
82     {
83         /*Allocating Major number*/
84         if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
85             printk(KERN_INFO "Cannot allocate major number\n");
86             return -1;
87         }
88         printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
89
90         /*Creating cdev structure*/
91         cdev_init(&etx_cdev,&fops);
92
93         /*Adding character device to the system*/
```

```
94         if((cdev_add(&etx_cdev,dev,1)) < 0){
95             printk(KERN_INFO "Cannot add the device to the system\n");
96             goto r_class;
97         }
98
99         /*Creating struct class*/
100        if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
101            printk(KERN_INFO "Cannot create the struct class\n");
102            goto r_class;
103        }
104
105        /*Creating device*/
106        if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
107            printk(KERN_INFO "Cannot create the Device \n");
108            goto r_device;
109        }
110
111        etx_thread = kthread_create(thread_function,NULL,"eTx Thread");
112        if(etx_thread) {
113            wake_up_process(etx_thread);
114        } else {
115            printk(KERN_ERR "Cannot create kthread\n");
116            goto r_device;
117        }
118 #if 0
119         /* You can use this method to create and run the thread */
120         etx_thread = kthread_run(thread_function,NULL,"eTx Thread");
121         if(etx_thread) {
122             printk(KERN_ERR "Kthread Created Successfully...\n");
123         } else {
124             printk(KERN_ERR "Cannot create kthread\n");
125             goto r_device;
126         }
127 #endif
128         printk(KERN_INFO "Device Driver Insert...Done!!!\n");
129         return 0;
130
131
132     r_device:
133         class_destroy(dev_class);
134     r_class:
135         unregister_chrdev_region(dev,1);
136         cdev_del(&etx_cdev);
137         return -1;
138 }
139
140 void __exit etx_driver_exit(void)
141 {
142     kthread_stop(etx_thread);
143     device_destroy(dev_class,dev);
144     class_destroy(dev_class);
145     cdev_del(&etx_cdev);
146     unregister_chrdev_region(dev, 1);
147     printk(KERN_INFO "Device Driver Remove...Done!!\n");
148 }
149
150 module_init(etx_driver_init);
151 module_exit(etx_driver_exit);
152
153 MODULE_LICENSE("GPL");
154 MODULE_AUTHOR("EmbeTronicX <embedtronicx@gmail.com or admin@embedtronicx.com>");
155 MODULE_DESCRIPTION("A simple device driver - Kernel Thread");
156 MODULE_VERSION("1.14");
```

MakeFile

```
1 obj-m += driver.o
2
3 KDIR = /lib/modules/$(shell uname -r)/build
4
5 all:
6     make -C $(KDIR) M=$(shell pwd) modules
7
8 clean:
9     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*
- Then Check the Dmesg

Major = 246 Minor = 0

Device Driver Insert...Done!!!

In EmbeTronicX Thread Function 0

In EmbeTronicX Thread Function 1

In EmbeTronicX Thread Function 2

In EmbeTronicX Thread Function 3

- So our thread is running now.
- Remove driver using *sudo rmmod driver* to stop the thread.

Share this:

[Share on Tumblr](#) [Tweet](#)



[WhatsApp](#)

[Pocket](#)



[pinit_fg_en_rect_gray_20](#) Linux Device Driver Tutorial Part 19 – Kernel Thread

Like this:

Loading...



[report this ad](#)

 ezoic[report this ad](#)[Sidebar ▾](#)

Device Drivers

Mutex in Linux Kernel

Linux Device Driver Tutorial Part 22 - Mutex in Linux Kernel

This is the [Series on Linux Device Driver](#). The aim of this series is to provide the easy and practical examples that anyone can understand. This is the Linux Device Driver Tutorial Part 22 – Mutex in Linux Kernel.

Post Contents

- [1 Prerequisites](#)
- [2 Introduction](#)
 - [2.1 Example Problems](#)
- [3 Race Condition](#)
- [4 Mutex](#)
- [5 Mutex in Linux Kernel](#)
 - [5.1 Initializing Mutex](#)
 - [5.1.1 Static Method](#)
 - [5.1.2 Dynamic Method](#)
 - [5.1.2.1 Example](#)
 - [5.2 Mutex Lock](#)
 - [5.2.1 mutex_lock](#)
 - [5.2.2 mutex_lock_interruptible](#)
 - [5.2.3 mutex_trylock](#)
 - [5.3 Mutex Unlock](#)
 - [5.4 Mutex Status](#)
- [6 Example Programming](#)
 - [6.1 Driver Source Code](#)
 - [6.2 MakeFile](#)
 - [6.2.1 Share this:](#)
 - [6.2.2 Like this:](#)
 - [6.2.3 Related](#)

Prerequisites

In the example section, I had used Kthread to explain Mutex. If you don't know what is Kthread and How to use it, then I would recommend you to explore that by using below link.

1. [Kthread Tutorial in Linux Kernel](#)

Introduction

Before getting to know about Mutex, let's take an analogy first.

Let us assume we have a car designed to accommodate only one person at any instance of time, while all the four doors of the car are open. But, if any more than one person tries to enter the car, a bomb will set off an explosion!(Quite a fancy car manufactured by EmbeTronicX!!) Now that four doors of the car are open, the car is vulnerable to explosion as anyone can enter through one of the four doors.

Now how we can solve this issue? Yes correct. We can provide a key for the car. So, the person who wants to enter the car must have access to the key. If they don't have key, they have to wait until that key is

available or they can do some other work instead of waiting for key.

Example Problems

Let's correlate the analogy above to what happens in our software. Let's explore situations like these through examples.

1. You have one SPI connection. What if one thread wants to write something into that SPI device and another thread wants to read from that SPI device at the same time?
2. You have one LED display. What if one thread is writing data at a different position of Display and another thread is writing different data at a different position of Display at the same time?
3. You have one Linked List. What if one thread wants to insert something into the list and another one wants to delete something in the same Linked List at the same time?

In all the scenarios above, the problem encountered is the same. At any given point two threads are accessing a single resource. Now we will relate the above scenarios to our car example.

1. In SPI example, CAR = SPI, Person = Threads, Blast = Software Crash/Software may get wrong data.
2. In LED display example, CAR = LED Display, Person = Threads, Blast = Display will show some unwanted junks.
3. In Linked List example, CAR = Linked List, Person = Threads, Blast = Software Crash/Software may get wrong data.

The cases above are termed as Race Condition.

Race Condition

A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, we don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data.

To avoid the race conditions, we have many ways like Semaphore,

Spinlock and Mutex. In this tutorial we will concentrate on Mutex.

Mutex

A *mutex* is a mutual exclusion lock. Only one thread can hold the lock.

Mutex can be used to prevent the simultaneous execution of a block of code by multiple threads that are running in a single or multiple processes.

Mutex is used as a synchronization primitive in situations where a resource has to be shared by multiple threads simultaneously.

Mutex has ownership. The thread which locks a Mutex must also unlock it.

So whenever you are accessing a shared resource that time first we lock mutex and then access the shared resource. When we are finished with that shared resource then we unlock the Mutex.

I hope you got some idea about Mutex. Now, let us look at Mutex in Linux Kernel.

Mutex in Linux Kernel

Today most major operating systems employ multitasking. Multitasking is where multiple threads can execute in parallel and thereby utilizing the CPU in optimum way. Even though, multitasking is useful, if not implemented cautiously can lead to concurrency issues (Race condition), which can be very difficult to handle.

The actual mutex type (minus debugging fields) is quite simple:

```
1 struct mutex {  
2     atomic_t count;
```

```
3     spinlock_t      wait_lock;
4     struct list_head    wait_list;
5 };
```

We will be using this structure for Mutex. Refer to `Linux/include/linux/mutex.h`

Initializing Mutex

We can initialize Mutex in two ways

1. Static Method
2. Dynamic Method

Static Method

This method will be useful while using global Mutex. This macro is defined below.

```
DEFINE_MUTEX(name);
```

This call *defines* and *initializes* a mutex. Refer to `Linux/include/linux/mutex.h`

Dynamic Method

This method will be useful for per-object mutexes, when mutex is just a field in a heap-allocated object. This macro is defined below.

```
mutex_init(struct mutex *lock);
```

Argument:

`struct mutex *lock` – the mutex to be initialized.

This call *initializes* already allocated mutex. Initialize the mutex to unlocked state.

It is not allowed to initialize an already locked mutex.

Example

```
1 struct mutex etx_mutex;
2 mutex_init(&etx_mutex);
```

Mutex Lock

Once a mutex has been initialized, it can be locked by any one of them explained below.

mutex_lock

This is used to lock/acquire the mutex exclusively for the current task. If the mutex is not available, the current task will sleep until it acquires the Mutex.

The mutex must later on be released by the same task that acquired it. Recursive locking is not allowed. The task may not exit without first unlocking the mutex. Also, kernel memory where the mutex resides must not be freed with the mutex still locked. The mutex must first be initialized (or statically defined) before it can be locked. memset-ing the mutex to 0 is not allowed.

```
void mutex_lock(struct mutex *lock);
```

Argument:

struct mutex *lock – the mutex to be acquired

mutex_lock_interruptible

Locks the mutex like `mutex_lock`, and returns 0 if the mutex has been acquired or sleep until the mutex becomes available. If a signal arrives while waiting for the lock then this function returns -EINTR.

```
int mutex_lock_interruptible(struct mutex *lock);
```

Argument:

`struct mutex *lock` – the mutex to be acquired

mutex_trylock

This will try to acquire the mutex, without waiting (will attempt to obtain the lock, but will not sleep). Returns 1 if the mutex has been acquired successfully, and 0 on contention.

```
int mutex_trylock(struct mutex *lock);
```

Argument:

`struct mutex *lock` – the mutex to be acquired

This function must not be used in interrupt context. The mutex must be released by the same task that acquired it.

Mutex Unlock

This is used to unlock/release a mutex that has been locked by a task previously.

This function must not be used in interrupt context. Unlocking of a not locked mutex is not allowed.

```
void mutex_unlock(struct mutex *lock);
```

Argument:

struct mutex *lock – the mutex to be released

Mutex Status

This function is used to check whether mutex has been locked or not.

```
int mutex_is_locked(struct mutex *lock);
```

Argument:

struct mutex *lock – the mutex to check the status.

Returns 1 if the mutex is locked, 0 if unlocked.

Example Programming

This code snippet explains how to create two threads that accesses a global variable (etx_gloabl_variable). So before accessing the variable, it should lock the mutex. After that it will release the mutex.

Driver Source Code

```
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/kthread.h>        //kernel threads
11 #include <linux/sched.h>          //task_struct
12 #include <linux/delay.h>
13 #include <linux/mutex.h>
14
15 struct mutex etx_mutex;
16 unsigned long etx_global_variable = 0;
```

```
18 dev_t dev = 0;
19 static struct class *dev_class;
20 static struct cdev etx_cdev;
21
22 static int __init etx_driver_init(void);
23 static void __exit etx_driver_exit(void);
24
25 static struct task_struct *etx_thread1;
26 static struct task_struct *etx_thread2;
27
28 /***** Driver Functions *****/
29 static int etx_open(struct inode *inode, struct file *file);
30 static int etx_release(struct inode *inode, struct file *file);
31 static ssize_t etx_read(struct file *filp,
32                      char __user *buf, size_t len, loff_t * off);
33 static ssize_t etx_write(struct file *filp,
34                      const char *buf, size_t len, loff_t * off);
35 /*****
36
37 int thread_function1(void *pv);
38 int thread_function2(void *pv);
39
40 int thread_function1(void *pv)
41 {
42
43     while(!kthread_should_stop()) {
44         mutex_lock(&etx_mutex);
45         etx_global_variable++;
46         printk(KERN_INFO "In EmbeTronicX Thread Function1 %lu\n", etx_global_variable)
47         mutex_unlock(&etx_mutex);
48         msleep(1000);
49     }
50     return 0;
51 }
52
53 int thread_function2(void *pv)
54 {
55     while(!kthread_should_stop()) {
56         mutex_lock(&etx_mutex);
57         etx_global_variable++;
58         printk(KERN_INFO "In EmbeTronicX Thread Function2 %lu\n", etx_global_variable)
59         mutex_unlock(&etx_mutex);
60         msleep(1000);
61     }
62     return 0;
63 }
64
65 static struct file_operations fops =
66 {
67     .owner        = THIS_MODULE,
68     .read         = etx_read,
69     .write        = etx_write,
70     .open         = etx_open,
71     .release      = etx_release,
72 };
73
74 static int etx_open(struct inode *inode, struct file *file)
75 {
76     printk(KERN_INFO "Device File Opened...!!!\n");
77     return 0;
78 }
79
80 static int etx_release(struct inode *inode, struct file *file)
```

```

81  {
82      printk(KERN_INFO "Device File Closed...!!!\n");
83      return 0;
84  }
85
86  static ssize_t etx_read(struct file *filp,
87                         char __user *buf, size_t len, loff_t *off)
88  {
89      printk(KERN_INFO "Read function\n");
90
91      return 0;
92  }
93  static ssize_t etx_write(struct file *filp,
94                         const char __user *buf, size_t len, loff_t *off)
95  {
96      printk(KERN_INFO "Write Function\n");
97      return len;
98  }
99
100 static int __init etx_driver_init(void)
101 {
102     /*Allocating Major number*/
103     if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
104         printk(KERN_INFO "Cannot allocate major number\n");
105         return -1;
106     }
107     printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
108
109     /*Creating cdev structure*/
110     cdev_init(&etx_cdev,&fops);
111
112     /*Adding character device to the system*/
113     if((cdev_add(&etx_cdev,dev,1)) < 0){
114         printk(KERN_INFO "Cannot add the device to the system\n");
115         goto r_class;
116     }
117
118     /*Creating struct class*/
119     if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
120         printk(KERN_INFO "Cannot create the struct class\n");
121         goto r_class;
122     }
123
124     /*Creating device*/
125     if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
126         printk(KERN_INFO "Cannot create the Device \n");
127         goto r_device;
128     }
129
130
131     /* Creating Thread 1 */
132     etx_thread1 = kthread_run(thread_function1,NULL,"eTx Thread1");
133     if(etx_thread1) {
134         printk(KERN_ERR "Kthread1 Created Successfully...\n");
135     } else {
136         printk(KERN_ERR "Cannot create kthread1\n");
137         goto r_device;
138     }
139
140     /* Creating Thread 2 */
141     etx_thread2 = kthread_run(thread_function2,NULL,"eTx Thread2");
142     if(etx_thread2) {
143         printk(KERN_ERR "Kthread2 Created Successfully...\n");

```

```

144         } else {
145             printk(KERN_ERR "Cannot create kthread2\n");
146             goto r_device;
147         }
148
149         mutex_init(&etx_mutex);
150
151         printk(KERN_INFO "Device Driver Insert...Done!!!\n");
152     return 0;
153
154
155 r_device:
156     class_destroy(dev_class);
157 r_class:
158     unregister_chrdev_region(dev,1);
159     cdev_del(&etx_cdev);
160     return -1;
161 }
162
163 void __exit etx_driver_exit(void)
164 {
165     kthread_stop(etx_thread1);
166     kthread_stop(etx_thread2);
167     device_destroy(dev_class,dev);
168     class_destroy(dev_class);
169     cdev_del(&etx_cdev);
170     unregister_chrdev_region(dev, 1);
171     printk(KERN_INFO "Device Driver Remove...Done!!\n");
172 }
173
174 module_init(etx_driver_init);
175 module_exit(etx_driver_exit);
176
177 MODULE_LICENSE("GPL");
178 MODULE_AUTHOR("EmbeTronicX <embedtronicx@gmail.com>");
179 MODULE_DESCRIPTION("A simple device driver - Mutex");
180 MODULE_VERSION("1.17");

```

MakeFile

```

1 obj-m += driver.o
2
3 KDIR = /lib/modules/$(shell uname -r)/build
4
5 all:
6     make -C $(KDIR) M=$(shell pwd) modules
7
8 clean:
9     make -C $(KDIR) M=$(shell pwd) clean

```

Share this:

[Share on Tumblr](#)[Print](#)[WhatsApp](#)[Pocket](#)[Telegram](#)

[pinit_fg_en_rect_gray_20](#) Linux Device Driver Tutorial Part 22 – Mutex in Linux Kernel

Like this:

Loading...



[report this ad](#)

 ezoic[report this ad](#)[Sidebar ▾](#)

Device Drivers

Spinlock in Linux Kernel

Linux Device Driver Tutorial Part 23 - Spinlock in Linux Kernel Part 1

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 23 - Spinlock in Linux Kernel Part 1.

Post Contents

1 Prerequisites

2 Introduction

3 SpinLock

4 SpinLock in Linux Kernel Device Driver

 4.1 Initialize

 4.1.1 Static Method

 4.1.2 Dynamic Method

 4.2 Approach 1 (Locking between User context)

 4.2.1 Example

 4.3 Approach 2 (Locking between Bottom Halves)

 4.4 Approach 3 (Locking between User context and Bottom Halves)

 4.4.1 Example

 4.5 Approach 4 (Locking between Hard IRQ and Bottom Halves)

 4.5.1 Example

 4.6 Approach 5 (Alternative way of Approach 4)

 4.7 Approach 6 (Locking between Hard IRQs)

5 Example Programming

 5.1 Driver Source Code

 5.2 MakeFile

 5.2.1 Share this:

 5.2.2 Like this:

 5.2.3 Related

Prerequisites

In the example section, I had used Kthread to explain Mutex. If you don't know what is Kthread and How to use it, then I would recommend you to explore that by using below link.

1. [Kthread Tutorial in Linux Kernel](#)
2. [Mutex Tutorial in Linux Kernel](#)

Introduction

In our [previous tutorial](#) we have understood the use of Mutex and its Implementation. If you have understood Mutex then Spinlock is also similar. Both are used to protect a shared resource from being modified by two or more processes simultaneously.

SpinLock

In Mutex concept, when thread is trying to lock or acquire the Mutex which is not available then that thread will go to sleep until that Mutex is available. Whereas in Spinlock it is different. The spinlock is a very simple single-holder lock. If a process attempts to acquire a spinlock and it is unavailable, the process will keep trying (spinning) until it can acquire the lock. This simplicity creates a small and fast lock.

Like Mutex, there are two possible states in Spinlock: **Locked** or **Unlocked**.

SpinLock in Linux Kernel Device Driver

If the kernel is running on a uniprocessor and CONFIG_SMP, CONFIG_PREEMPT aren't enabled while compiling the kernel then spinlock will not be available. Because there is no reason to have a lock, when no one else can run at the same time.

But if you have disabled CONFIG_SMP and enabled CONFIG_PREEMPT then spinlock will simply disable preemption, which is sufficient to prevent any races.

Initialize

We can initialize Spinlock in two ways.

1. Static Method

2. Dynamic Method

Static Method

You can statically initialize a Spinlock using the macro given below.

```
1 #define DEFINE_SPINLOCK(etx_spinlock);
```

The macro given above will create spinlock_t variable in the name of and initialize to **UNLOCKED STATE**. Take a look at the expansion of DEFINE_SPINLOCK below.

```
1 #define DEFINE_SPINLOCK(x)      spinlock_t x = __SPIN_LOCK_UNLOCKED(x)
```

Dynamic Method

If you want to initialize dynamically you can use the method as given below.

```
1 spinlock_t etx_spinlock;
2 spin_lock_init(&etx_spinlock);
```

You can use any one of the methods.

After initializing the spinlock, there are several ways to use spinlock to lock or unlock, based on where the spinlock is used; either in user context or interrupt context. Let's look at the approaches with these situations.

Approach 1 (Locking between User context)

If you share data with user context (between Kernel Threads), then you can use this approach.

Lock:

```
spin_lock(spinlock_t *lock)
```

This will take the lock if it is free, otherwise it'll spin until that lock is free (Keep trying).

Try Lock:

```
spin_trylock(spinlock_t *lock)
```

Locks the spinlock if it is not already locked. If unable to obtain the lock it exits with an error and do not spin. It **returns** non-zero if obtains the lock otherwise returns zero.

Unlock:

```
spin_unlock(spinlock_t *lock)
```

It does the reverse of lock. It will unlock which is locked by above call.

Checking Lock:

```
spin_is_locked(spinlock_t *lock)
```

This is used to check whether the lock is available or not. It **returns** non-zero if the lock is currently acquired. otherwise returns zero.

Example

```
1 //Thread 1
2 int thread_function1(void *pv)
3 {
4     while(!kthread_should_stop()) {
5         spin_lock(&etx_spinlock);
6         etx_global_variable++;
```

```

7      printk(KERN_INFO "In EmbeTronicX Thread Function1 %lu\n", etx_global_variable);
8      spin_unlock(&etx_spinlock);
9      msleep(1000);
10     }
11     return 0;
12 }
13
14 //Thread 2
15 int thread_function2(void *pv)
16 {
17     while(!kthread_should_stop()) {
18         spin_lock(&etx_spinlock);
19         etx_global_variable++;
20         printk(KERN_INFO "In EmbeTronicX Thread Function2 %lu\n", etx_global_variable);
21         spin_unlock(&etx_spinlock);
22         msleep(1000);
23     }
24     return 0;
25 }
```

Approach 2 (Locking between Bottom Halves)

If you want to share data between two different Bottom halves or same bottom halves, then you can use the [Approach 1](#).

Approach 3 (Locking between User context and Bottom Halves)

If you share data with a [bottom half](#) and user context (like Kernel Thread), then this approach will be useful.

Lock:

```
spin_lock_bh(spinlock_t *lock)
```

It disables soft interrupts on that CPU, then grabs the lock. This has the effect of preventing softirqs, tasklets, and bottom halves from running on the local CPU. Here the suffix ‘_bh’ refers to “**Bottom Halves**”.

Unlock:

```
spin_unlock_bh(spinlock_t *lock)
```

It will release the lock and re-enables the soft interrupts which is disabled by above call.

Example

```
1 //Thread
2 int thread_function(void *pv)
3 {
4     while(!kthread_should_stop()) {
5         spin_lock_bh(&etx_spinlock);
6         etx_global_variable++;
7         printk(KERN_INFO "In EmbeTronicX Thread Function %lu\n", etx_global_variable);
8         spin_unlock_bh(&etx_spinlock);
9         msleep(1000);
10    }
11    return 0;
12 }
13 /*Tasklet Function*/
14 void tasklet_fn(unsigned long arg)
15 {
16     spin_lock_bh(&etx_spinlock);
17     etx_global_variable++;
18     printk(KERN_INFO "Executing Tasklet Function : %lu\n", etx_global_variable);
19     spin_unlock_bh(&etx_spinlock);
20 }
```

Approach 4 (Locking between Hard IRQ and Bottom Halves)

If you share data between Hardware ISR and Bottom halves then you have to disable the IRQ before lock. Because, the bottom halves processing can be interrupted by a hardware interrupt. So this will be used in that scenario.

Lock:

```
spin_lock_irq(spinlock_t *lock)
```

This will disable interrupts on that cpu, then grab the lock.

Unlock:

```
spin_unlock_irq(spinlock_t *lock)
```

It will release the lock and re-enables the interrupts which is disabled by above call.

Example

```
1 /*Tasklet Function*/
2 void tasklet_fn(unsigned long arg)
3 {
4     spin_lock_irq(&etx_spinlock);
5     etx_global_variable++;
6     printk(KERN_INFO "Executing Tasklet Function : %lu\n", etx_global_variable);
7     spin_unlock_irq(&etx_spinlock);
8 }
9
10 //Interrupt handler for IRQ 11.
11 static irqreturn_t irq_handler(int irq,void *dev_id) {
12     spin_lock_irq(&etx_spinlock);
13     etx_global_variable++;
14     printk(KERN_INFO "Executing ISR Function : %lu\n", etx_global_variable);
15     spin_unlock_irq(&etx_spinlock);
16     /*Scheduling Task to Tasklet*/
17     tasklet_schedule(tasklet);
18     return IRQ_HANDLED;
19 }
```

Approach 5 (Alternative way of Approach 4)

If you want to use different variant rather than using `spin_lock_irq()` and `spin_unlock_irq()` then you can use this approach.

Lock:

```
spin_lock_irqsave( spinlock_t *lock, unsigned long flags );
```

This will save whether interrupts were on or off in a flags word and grab the lock.

Unlock:

```
spin_unlock_irqrestore( spinlock_t *lock, unsigned long flags );
```

This will releases the spinlock and restores the interrupts using the flags argument.

Approach 6 (Locking between Hard IRQs)

If you want to share data between two different IRQs, then you should use [Approach 5](#).

Example Programming

This code snippet explains how to create two threads that accesses a global variable (etx_gloabl_variable). So before accessing the variable, it should lock the spinlock. After that it will release the spinlock. This example is using [Approach 1](#).

Driver Source Code

```
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/kthread.h>        //kernel threads
11 #include <linux/sched.h>          //task_struct
12 #include <linux/delay.h>
13
14 DEFINE_SPINLOCK(etx_spinlock);
15 //spinlock_t etx_spinlock;
16 unsigned long etx_global_variable = 0;
17
18 dev_t dev = 0;
19 static struct class *dev_class;
20 static struct cdev etx_cdev;
21
22 static int __init etx_driver_init(void);
23 static void __exit etx_driver_exit(void);
24
25 static struct task_struct *etx_thread1;
26 static struct task_struct *etx_thread2;
27
28 /***** Driver Fuctions *****/
29 static int etx_open(struct inode *inode, struct file *file);
30 static int etx_release(struct inode *inode, struct file *file);
31 static ssize_t etx_read(struct file *filp,
32                        char __user *buf, size_t len, loff_t * off);
33 static ssize_t etx_write(struct file *filp,
34                        const char *buf, size_t len, loff_t * off);
35 /*****
```

```
37 int thread_function1(void *pv);
38 int thread_function2(void *pv);
39
40 int thread_function1(void *pv)
41 {
42
43     while(!kthread_should_stop()) {
44         if(!spin_is_locked(&etx_spinlock)) {
45             printk(KERN_INFO "Spinlock is not locked in Thread Function1\n");
46         }
47         spin_lock(&etx_spinlock);
48         if(spin_is_locked(&etx_spinlock)) {
49             printk(KERN_INFO "Spinlock is locked in Thread Function1\n");
50         }
51         etx_global_variable++;
52         printk(KERN_INFO "In EmbeTronicX Thread Function1 %lu\n", etx_global_variable)
53         spin_unlock(&etx_spinlock);
54         msleep(1000);
55     }
56     return 0;
57 }
58
59 int thread_function2(void *pv)
60 {
61     while(!kthread_should_stop()) {
62         spin_lock(&etx_spinlock);
63         etx_global_variable++;
64         printk(KERN_INFO "In EmbeTronicX Thread Function2 %lu\n", etx_global_variable)
65         spin_unlock(&etx_spinlock);
66         msleep(1000);
67     }
68     return 0;
69 }
70
71 static struct file_operations fops =
72 {
73     .owner          = THIS_MODULE,
74     .read           = etx_read,
75     .write          = etx_write,
76     .open            = etx_open,
77     .release        = etx_release,
78 };
79
80 static int etx_open(struct inode *inode, struct file *file)
81 {
82     printk(KERN_INFO "Device File Opened...!!!!\n");
83     return 0;
84 }
85
86 static int etx_release(struct inode *inode, struct file *file)
87 {
88     printk(KERN_INFO "Device File Closed...!!!!\n");
89     return 0;
90 }
91
92 static ssize_t etx_read(struct file *filp,
93                         char __user *buf, size_t len, loff_t *off)
94 {
95     printk(KERN_INFO "Read function\n");
96
97     return 0;
98 }
99 static ssize_t etx_write(struct file *filp,
```

```

100             const char __user *buf, size_t len, loff_t *off)
101 {
102     printk(KERN_INFO "Write Function\n");
103     return len;
104 }
105
106 static int __init etx_driver_init(void)
107 {
108     /*Allocating Major number*/
109     if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
110         printk(KERN_INFO "Cannot allocate major number\n");
111         return -1;
112     }
113     printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
114
115     /*Creating cdev structure*/
116     cdev_init(&etx_cdev,&fops);
117
118     /*Adding character device to the system*/
119     if((cdev_add(&etx_cdev,dev,1)) < 0){
120         printk(KERN_INFO "Cannot add the device to the system\n");
121         goto r_class;
122     }
123
124     /*Creating struct class*/
125     if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
126         printk(KERN_INFO "Cannot create the struct class\n");
127         goto r_class;
128     }
129
130     /*Creating device*/
131     if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
132         printk(KERN_INFO "Cannot create the Device \n");
133         goto r_device;
134     }
135
136
137     /* Creating Thread 1 */
138     etx_thread1 = kthread_run(thread_function1,NULL,"eTx Thread1");
139     if(etx_thread1) {
140         printk(KERN_ERR "Kthread1 Created Successfully...\n");
141     } else {
142         printk(KERN_ERR "Cannot create kthread1\n");
143         goto r_device;
144     }
145
146     /* Creating Thread 2 */
147     etx_thread2 = kthread_run(thread_function2,NULL, "eTx Thread2");
148     if(etx_thread2) {
149         printk(KERN_ERR "Kthread2 Created Successfully...\n");
150     } else {
151         printk(KERN_ERR "Cannot create kthread2\n");
152         goto r_device;
153     }
154     //spin_lock_init(&etx_spinlock);
155
156     printk(KERN_INFO "Device Driver Insert...Done!!!\n");
157     return 0;
158
159
160 r_device:
161     class_destroy(dev_class);
162 r_class:

```

```

163     unregister_chrdev_region(dev,1);
164     cdev_del(&etx_cdev);
165     return -1;
166 }
167
168 void __exit etx_driver_exit(void)
169 {
170     kthread_stop(etx_thread1);
171     kthread_stop(etx_thread2);
172     device_destroy(dev_class,dev);
173     class_destroy(dev_class);
174     cdev_del(&etx_cdev);
175     unregister_chrdev_region(dev, 1);
176     printk(KERN_INFO "Device Driver Remove...Done!!\n");
177 }
178
179 module_init(etx_driver_init);
180 module_exit(etx_driver_exit);
181
182 MODULE_LICENSE("GPL");
183 MODULE_AUTHOR("EmbeTronicX <embedtronicx@gmail.com>");
184 MODULE_DESCRIPTION("A simple device driver - Spinlock");
185 MODULE_VERSION("1.18");

```

MakeFile

```

1 obj-m += driver.o
2
3 KDIR = /lib/modules/$(shell uname -r)/build
4
5 all:
6     make -C $(KDIR) M=$(shell pwd) modules
7
8 clean:
9     make -C $(KDIR) M=$(shell pwd) clean

```

In our [next part of the tutorial](#), we will see the another spinlock (**Reader/writer spinlocks**).

Share this:

[Share on Tumblr](#)
[Tweet](#)

[WhatsApp](#)
[Pocket](#)


[pinit_fg_en_rect_gray_20](#) Linux Device Driver Tutorial Part 23 – Spinlock in Linux Kernel Part 1

Like this:

Loading...



[report this ad](#)



 ezoic

[report this ad](#)

Sidebar▼

Device Drivers

read write spinlock

**Linux Device Driver Tutorial Part 24 -
Read Write Spinlock in Linux Kernel
(Spinlock Part 2)**

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 24 – Read Write Spinlock in Linux Kernel (Spinlock Part 2).

Post Contents

[1 Prerequisites](#)

[2 Introduction](#)

[3 SpinLock](#)

[4 Read Write Spinlocks](#)

[4.1 Working of Read Write Spinlock](#)

[4.2 Where to use Read Write Spinlock?](#)

[5 Read Write SpinLock in Linux Kernel Device Driver](#)

[5.1 Initialize](#)

[5.1.1 Static Method](#)

[5.1.2 Dynamic Method](#)

[5.2 Approach 1 \(Locking between User context\)](#)

[5.2.1 Read Lock](#)

[5.2.2 Write Lock](#)

[5.2.3 Example](#)

[5.3 Approach 2 \(Locking between Bottom Halves\)](#)

[5.4 Approach 3 \(Locking between User context and Bottom Halves\)](#)

[5.4.1 Read Lock](#)

[5.4.2 Write Lock](#)

[5.4.3 Example](#)

[5.5 Approach 4 \(Locking between Hard IRQ and Bottom Halves\)](#)

[5.5.1 Read Lock](#)

5.5.2 Write Lock
5.5.3 Example
5.6 Approach 5 (Alternative way of Approach 4)
5.6.1 Read Lock
5.6.2 Write Lock
5.7 Approach 6 (Locking between Hard IRQs)
6 Example Programming
6.1 Driver Source Code
6.2 MakeFile
6.2.1 Share this:
6.2.2 Like this:
6.2.3 Related

Prerequisites

In the example section, I had used Kthread to explain Mutex. If you don't know what is Kthread and How to use it, then I would recommend you to explore that by using below link.

1. [Kthread Tutorial in Linux Kernel](#)
2. [Spinlock Tutorial in Linux Kernel – Part 1](#)

Introduction

In our [previous tutorial](#) we have understood the use of Common Spinlock and its Implementation. If you have understood Spinlock then this Read Write Spinlock is also similar except some difference. We will cover those

things below.

SpinLock

The spinlock is a very simple single-holder lock. If a process attempts to acquire a spinlock and it is unavailable, the process will keep trying (spinning) until it can acquire the lock. Read Write spinlock also does the same but it has separate locks for read and write operation.

Read Write Spinlocks

I told that spinlock and read write spinlock are similar in operation. That means spinlock is enough. Then why do we need Read Write Spinlock? Ridiculous right?

Okay now we will take one scenario. I have five threads. All those threads are accessing one global variable. So we can use spinlock over there. Am I right? Well, yes it's right. In those threads, thread-1's role is to write the data into that variable. Other four thread's roles are simply reading the data from that variable. This the case. I hope you guys understand the scenario. Now I can ask you guys an question. If you implement spinlock,

Question : Now we forgot the thread-1(writer thread). So we have 4 reader thread (thread-2 to thread-5). Now those all four threads are want to read the data from the variable at the same time. What will happen in this situation if you use spinlock? What about the processing speed and performance?

Let's assume thread-2 got the lock while other reading threads are trying hardly for the lock. That variable won't change even thread-2's access. Because no one is writing. But here only thread-2 is accessing. But other reading threads are simply wasting time to take lock since variable won't change. That means performance will be reduced. Isn't it? Yes you are correct.

In this case if you implement read and write lock, thread-2 will take the read lock and read the data. And other reading threads also will take the read lock without spinning (blocking) and read the data. Because no one

is writing. So what about the performance now? There is no waiting between reading operations. Right? Then in this case read write spinlock is useful. Isn't It?

So, If multiple threads require read access to the same data, there is no reason why they should not be able to execute simultaneously. Spinlocks don't differentiate between read and read/write access. Thus spinlocks do not exploit this potential parallelism. To do so, read-write locks are required.

Working of Read Write Spinlock

- When there is no thread in the critical section, any reader or writer thread can enter into critical section by taking respective read or write lock. But only one thread can enter into critical section.
- If reader thread is in critical section, the new reader thread can enter arbitrarily, but the writer thread cannot enter. Writer thread has to wait until all the reader thread finish their process.
- If writer thread is in critical section, no reader thread or writer thread can enter.
- If one or more reader threads are in critical section by taking it's lock, the writer thread can of course not enter the critical section, but the writer thread cannot prevent the entry of the subsequent read thread. He has to wait until the critical section has a reader thread. So this read write spinlock is giving importance to reader thread and not writer thread. If you want to give importance to writer thread than reader thread, then another lock is available in linux which is [seqlock](#).

Where to use Read Write Spinlock?

1. If you are only reading the data then you take read lock
2. If you are writing then go for write lock

Note : Many people can hold a read lock, but a writer must be sole holder. Read-Write locks are more useful in scenarios where the architecture is clearly divided into reader and writers, with more number of reads.

In Read Write spinlock multiple readers are permitted at same time but only one writer. (i.e) If a writer has the lock, no reader is allowed to enter the critical section. If only a reader has the lock, then multiple readers

are permitted in the critical section.

Read Write SpinLock in Linux Kernel Device Driver

Initialize

We can initialize Read Write Spinlock in two ways.

1. Static Method
2. Dynamic Method

Static Method

You can statically initialize a Read Write Spinlock using the macro given below.

```
1 DEFINE_RWLOCK(etx_rwlock);
```

The macro given above will create rwlock_t variable in the name of etx_rwlock.

Dynamic Method

If you want to initialize dynamically you can use the method as given below.

```
1 rwlock_t etx_rwlock;
2 rwlock_init(&etx_rwlock);
```

You can use any one of the methods.

After initializing the read/write spinlock, there are several ways to use read/write spinlock to lock or unlock, based on where the read/write spinlock is used; either in user context or interrupt context. Let's look at the approaches with these situations.

Approach 1 (Locking between User context)

If you share data with user context (between Kernel Threads), then you can use this approach.

Read Lock

Lock:

```
read_lock(rwlock_t *lock)
```

This will take the lock if it is free, otherwise it'll spin until that lock is free (Keep trying).

Unlock:

```
read_unlock(rwlock_t *lock)
```

It does the reverse of lock. It will unlock which is locked by above call.

Write Lock

Lock:

```
write_lock(rwlock_t *lock)
```

This will take the lock if it is free, otherwise it'll spin until that lock is free (Keep trying).

Unlock:

```
write_unlock(rwlock_t *lock)
```

It does the reverse of lock. It will unlock which is locked by above call.

Example

```
1 //Thread 1
2 int thread_function1(void *pv)
3 {
4     while(!kthread_should_stop()) {
5         write_lock(&etx_rwlock);
6         etx_global_variable++;
7         write_unlock(&etx_rwlock);
8         msleep(1000);
9     }
10    return 0;
11 }
12
13 //Thread 2
14 int thread_function2(void *pv)
15 {
16     while(!kthread_should_stop()) {
17         read_lock(&etx_rwlock);
18         printk(KERN_INFO "In EmbeTronicX Thread Function2 : Read value %lu\n", etx_glob
19         read_unlock(&etx_rwlock);
20         msleep(1000);
21     }
22    return 0;
23 }
```

Approach 2 (Locking between Bottom Halves)

If you want to share data between two different Bottom halves or same bottom halves, then you can use the [Approach 1](#).

Approach 3 (Locking between User context and Bottom Halves)

If you share data with a [bottom half](#) and user context (like Kernel

Thread), then this approach will be useful.

Read Lock

Lock:

```
read_lock_bh(rwlock_t *lock)
```

It disables soft interrupts on that CPU, then grabs the lock. This has the effect of preventing softirqs, tasklets, and bottom halves from running on the local CPU. Here the suffix ‘_bh’ refers to “**Bottom Halves**”.

Unlock:

```
read_unlock_bh(rwlock_t *lock)
```

It will release the lock and re-enables the soft interrupts which is disabled by above call.

Write Lock

Lock:

```
write_lock_bh(rwlock_t *lock)
```

It disables soft interrupts on that CPU, then grabs the lock. This has the effect of preventing softirqs, tasklets, and bottom halves from running on the local CPU. Here the suffix ‘_bh’ refers to “**Bottom Halves**”.

Unlock:

```
write_unlock_bh(rwlock_t *lock)
```

It will release the lock and re-enables the soft interrupts which is disabled by above call.

Example

```

1 //Thread
2 int thread_function(void *pv)
3 {
4     while(!kthread_should_stop()) {
5         write_lock_bh(&etx_rwlock);
6         etx_global_variable++;
7         write_unlock_bh(&etx_rwlock);
8         msleep(1000);
9     }
10    return 0;
11 }
12 /*Tasklet Function*/
13 void tasklet_fn(unsigned long arg)
14 {
15     read_lock_bh(&etx_rwlock);
16     printk(KERN_INFO "Executing Tasklet Function : %lu\n", etx_global_variable);
17     read_unlock_bh(&etx_rwlock);
18 }
```

Approach 4 (Locking between Hard IRQ and Bottom Halves)

If you share data between Hardware ISR and Bottom halves then you have to disable the IRQ before lock. Because, the bottom halves processing can be interrupted by a hardware interrupt. So this will be used in that scenario.

Read Lock

Lock:

```
read_lock_irq(rwlock_t *lock)
```

This will disable interrupts on that cpu, then grab the lock.

Unlock:

```
read_unlock_irq(rwlock_t *lock)
```

It will release the lock and re-enables the interrupts which is disabled by above call.

Write Lock

Lock:

```
write_lock_irq(rwlock_t *lock)
```

This will disable interrupts on that cpu, then grab the lock.

Unlock:

```
write_unlock_irq(rwlock_t *lock)
```

It will release the lock and re-enables the interrupts which is disabled by above call.

Example

```
1 /*Tasklet Function*/
2 void tasklet_fn(unsigned long arg)
3 {
4     write_lock_irq(&etx_rwlock);
5     etx_global_variable++;
6     write_unlock_irq(&etx_rwlock);
7 }
8
9 //Interrupt handler for IRQ 11.
10 static irqreturn_t irq_handler(int irq,void *dev_id) {
11     read_lock_irq(&etx_rwlock);
12     printk(KERN_INFO "Executing ISR Function : %lu\n", etx_global_variable);
13     read_unlock_irq(&etx_rwlock);
14     /*Scheduling Task to Tasklet*/
15     tasklet_schedule(tasklet);
16     return IRQ_HANDLED;
17 }
```

Approach 5 (Alternative way of Approach 4)

If you want to use different variant rather than using `read_lock_irq()`/`write_lock_irq()` and `read_unlock_irq()`/`write_unlock_irq()` then you can use this approach.

Read Lock

Lock:

```
read_lock_irqsave( rwlock_t *lock, unsigned long flags );
```

This will save whether interrupts were on or off in a flags word and grab the lock.

Unlock:

```
read_unlock_irqrestore( rwlock_t *lock, unsigned long flags );
```

This will releases the read/write spinlock and restores the interrupts using the flags argument.

Write Lock

Lock:

```
write_lock_irqsave( rwlock_t *lock, unsigned long flags );
```

This will save whether interrupts were on or off in a flags word and grab the lock.

Unlock:

```
write_unlock_irqrestore( rwlock_t *lock, unsigned long flags );
```

This will releases the read/write spinlock and restores the interrupts using the flags argument.

Approach 6 (Locking between Hard IRQs)

If you want to share data between two different IRQs, then you should use [Approach 5](#).

Example Programming

This code snippet explains how to create two threads that accesses a global variable (`etx_gloabl_variable`). So before accessing the variable, it should lock the read/write spinlock. After that it will release the read/write spinlock. This example is using [Approach 1](#).

Driver Source Code

```
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/kthread.h>        //kernel threads
11 #include <linux/sched.h>          //task_struct
12 #include <linux/delay.h>
13
14
15 //Static method to initialize the read write spinlock
16 static DEFINE_RWLOCK(etx_rwlock);
17
18 //Dynamic method to initialize the read write spinlock
19 //rwlock_t etx_rwlock;
20
21 unsigned long etx_global_variable = 0;
22 dev_t dev = 0;
23 static struct class *dev_class;
24 static struct cdev etx_cdev;
25
26 static int __init etx_driver_init(void);
27 static void __exit etx_driver_exit(void);
28
29 static struct task_struct *etx_thread1;
30 static struct task_struct *etx_thread2;
31
32 /***** Driver Fuctions *****/
33 static int etx_open(struct inode *inode, struct file *file);
34 static int etx_release(struct inode *inode, struct file *file);
35 static ssize_t etx_read(struct file *filp,
36                         char __user *buf, size_t len, loff_t * off);
37 static ssize_t etx_write(struct file *filp,
38                         const char *buf, size_t len, loff_t * off);
39 /*****
40
41 int thread_function1(void *pv);
42 int thread_function2(void *pv);
43
44 int thread_function1(void *pv)
```

```
45 {
46     while(!kthread_should_stop()) {
47         write_lock(&etx_rwlock);
48         etx_global_variable++;
49         write_unlock(&etx_rwlock);
50         msleep(1000);
51     }
52     return 0;
53 }
54
55 int thread_function2(void *pv)
56 {
57     while(!kthread_should_stop()) {
58         read_lock(&etx_rwlock);
59         printk(KERN_INFO "In EmbeTronicX Thread Function2 : Read value %lu\n", etx_glo
60         read_unlock(&etx_rwlock);
61         msleep(1000);
62     }
63     return 0;
64 }
65
66 static struct file_operations fops =
67 {
68     .owner        = THIS_MODULE,
69     .read         = etx_read,
70     .write        = etx_write,
71     .open          = etx_open,
72     .release      = etx_release,
73 };
74
75 static int etx_open(struct inode *inode, struct file *file)
76 {
77     printk(KERN_INFO "Device File Opened...!!!\n");
78     return 0;
79 }
80
81 static int etx_release(struct inode *inode, struct file *file)
82 {
83     printk(KERN_INFO "Device File Closed...!!!\n");
84     return 0;
85 }
86
87 static ssize_t etx_read(struct file *filp,
88                         char __user *buf, size_t len, loff_t *off)
89 {
90     printk(KERN_INFO "Read function\n");
91
92     return 0;
93 }
94 static ssize_t etx_write(struct file *filp,
95                         const char __user *buf, size_t len, loff_t *off)
96 {
97     printk(KERN_INFO "Write Function\n");
98     return len;
99 }
100
101 static int __init etx_driver_init(void)
102 {
103     /*Allocating Major number*/
104     if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
105         printk(KERN_INFO "Cannot allocate major number\n");
106         return -1;
107     }
```

```
108         printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
109
110     /*Creating cdev structure*/
111     cdev_init(&etx_cdev,&fops);
112
113     /*Adding character device to the system*/
114     if((cdev_add(&etx_cdev,dev,1)) < 0){
115         printk(KERN_INFO "Cannot add the device to the system\n");
116         goto r_class;
117     }
118
119     /*Creating struct class*/
120     if((dev_class = class_create(TTHIS_MODULE,"etx_class")) == NULL){
121         printk(KERN_INFO "Cannot create the struct class\n");
122         goto r_class;
123     }
124
125     /*Creating device*/
126     if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
127         printk(KERN_INFO "Cannot create the Device \n");
128         goto r_device;
129     }
130
131
132     /* Creating Thread 1 */
133     etx_thread1 = kthread_run(thread_function1,NULL,"eTx Thread1");
134     if(etx_thread1) {
135         printk(KERN_ERR "Kthread1 Created Successfully...\n");
136     } else {
137         printk(KERN_ERR "Cannot create kthread1\n");
138         goto r_device;
139     }
140
141     /* Creating Thread 2 */
142     etx_thread2 = kthread_run(thread_function2,NULL,"eTx Thread2");
143     if(etx_thread2) {
144         printk(KERN_ERR "Kthread2 Created Successfully...\n");
145     } else {
146         printk(KERN_ERR "Cannot create kthread2\n");
147         goto r_device;
148     }
149
150     //Dynamic method to initialize the read write spinlock
151     //rwlock_init(&etx_rwlock);
152
153     printk(KERN_INFO "Device Driver Insert...Done!!!\n");
154     return 0;
155
156
157 r_device:
158     class_destroy(dev_class);
159 r_class:
160     unregister_chrdev_region(dev,1);
161     cdev_del(&etx_cdev);
162     return -1;
163 }
164
165 void __exit etx_driver_exit(void)
166 {
167     kthread_stop(etx_thread1);
168     kthread_stop(etx_thread2);
169     device_destroy(dev_class,dev);
170     class_destroy(dev_class);
```

```
171     cdev_del(&etx_cdev);
172     unregister_chrdev_region(dev, 1);
173     printk(KERN_INFO "Device Driver Remove...Done!!\n");
174 }
175
176 module_init(etx_driver_init);
177 module_exit(etx_driver_exit);
178
179 MODULE_LICENSE("GPL");
180 MODULE_AUTHOR("EmbeTronicX <embedtronicx@gmail.com>");
181 MODULE_DESCRIPTION("A simple device driver - RW Spinlock");
182 MODULE_VERSION("1.19");
```

MakeFile

```
1 obj-m += driver.o
2
3 KDIR = /lib/modules/$(shell uname -r)/build
4
5 all:
6     make -C $(KDIR) M=$(shell pwd) modules
7
8 clean:
9     make -C $(KDIR) M=$(shell pwd) clean
```

Share this:

[Share on Tumblr](#)[Tweet](#)[WhatsApp](#)[Pocket](#)

[pinit_fg_en_rect_gray_20 Linux Device Driver Tutorial Part 24 – Read Write Spinlock in Linux Kernel \(Spinlock Part 2\)](#)

Like this:

Loading...



[report this ad](#)



 ezoic

[report this ad](#)

Sidebar ▾

Device Drivers

Sending Signal from Linux Device Driver to User Space

**Linux Device Driver Tutorial Part 25 -
Sending Signal from Linux Device Driver
to User Space**

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 25 - Sending Signal from Linux Device Driver to User Space.

Post Contents

- [1 Prerequisites](#)
- [2 Signals](#)
 - [2.1 Introduction](#)
- [3 Sending Signal from Linux Device Driver to User Space](#)
 - [3.1 Decide the signal that you want to send](#)
 - [3.2 Register the user space application with driver](#)
 - [3.3 Send signals to user space](#)
 - [3.4 Unregister the user space application](#)
- [4 Device Driver Source Code](#)
- [5 Application Source Code](#)
- [6 Building Driver and Application](#)
- [7 Execution \(Output\)](#)
 - [7.0.1 Share this:](#)
 - [7.0.2 Like this:](#)
 - [7.0.3 Related](#)

Prerequisites

In the example section, we explained signals using interrupt program. So I would recommend you to explore interrupts using below links before start this.

1. [Interrupts Concepts](#)
2. [Interrupts Examples Program](#)
3. [IOCTL Tutorial](#)

Signals

Introduction

Generally A **signal** is a action which is intended to send a particular message. It can be sound, gesture, event etc. Below are the normal signals which we are using day to day life.

- When we take money in ATM, we will get a message
- Calling someone by making sound or gesture
- Microwave oven making sound when it finishes its job
- etc.

What about Linux? **Signals** are a way of sending simple messages which is used to notify a process or thread of a particular event. In Linux, there are many process will be running at a time. We can send signal from one process to another process. Signals are one of the oldest inter-process communication methods. This signals are asynchronous. Like User space signals, can we send signal to user space from kernel space? Yes why not. We will see the complete Signals in upcoming tutorials. In this tutorial we will learn how to send signal from Linux Device Driver to User Space.

Sending Signal from Linux Device Driver to

User Space

Using the following steps easily we can send the signals.

1. Decide the signal that you want to send.
2. Register the user space application with driver.
3. Once something happened (in our example we used interrupts) send signals to user space.
4. Unregister the user space application when you done.

Decide the signal that you want to send

First select the signal number which you want to send. In our case we are going to send signal 44.

Example:

```
1 #define SIGETX 44
```

Register the user space application with driver

Before sending signal, your device driver should know to whom it needs to send the signal. For that we need to register the process to driver. So we need to send the PID to driver first. Then that driver will use the PID and sends the signal. You can register the application PID in any ways like IOCTL, Open/read/write call. In our example we are going to register using IOCTL.

Example:

```
1 static long etx_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
2 {
3     if (cmd == REG_CURRENT_TASK) {
4         printk(KERN_INFO "REG_CURRENT_TASK\n");
5         task = get_current();
6         signum = SIGETX;
7     }
}
```

```

8     return 0;
9 }
```

Send signals to user space

After registering the application to driver, then driver can able to send the signal when it requires. In our example, we will send the signal when we get the interrupt.

Example:

```

1 //Interrupt handler for IRQ 11.
2 static irqreturn_t irq_handler(int irq,void *dev_id) {
3     struct siginfo info;
4     printk(KERN_INFO "Shared IRQ: Interrupt Occurred");
5
6     //Sending signal to app
7     memset(&info, 0, sizeof(struct siginfo));
8     info.si_signo = SIGETX;
9     info.si_code = SI_QUEUE;
10    info.si_int = 1;
11
12    if (task != NULL) {
13        printk(KERN_INFO "Sending signal to app\n");
14        if(send_sig_info(SIGETX, &info, task) < 0) {
15            printk(KERN_INFO "Unable to send signal\n");
16        }
17    }
18    return IRQ_HANDLED;
19 }
```

Unregister the user space application

When you done with your task, you can unregister your application. Here we are unregistering when that application closes the driver.

Example:

```

1 static int etx_release(struct inode *inode, struct file *file)
2 {
3     struct task_struct *ref_task = get_current();
4     printk(KERN_INFO "Device File Closed...!!!\n");
5
6     //delete the task
7     if(ref_task == task) {
8         task = NULL;
9     }
10    return 0;
11 }
```

Device Driver Source Code

The complete device driver code is given below. In this source code, When we read the /dev/etx_device interrupt will hit (To understand interrupts in Linux go to [this tutorial](#)). Whenever interrupt hits, I'm sending signal to user space application who registered already. Since it is a tutorial post, I'm not going to do any job in interrupt handler except sending signal.

driver.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 #include<linux/slab.h>                                //kmalloc()
9 #include<linux/uaccess.h>                            //copy_to/from_user()
10 #include <linux/ioctl.h>
11 #include <linux/interrupt.h>
12 #include <asm/io.h>
13
14 #define SIGETX 44
15
16 #define REG_CURRENT_TASK _IOW('a','a',int32_t*)
17
18 #define IRQ_NO 11
19
20 /* Signaling to Application */
21 static struct task_struct *task = NULL;
22 static int signum = 0;
23
24 int32_t value = 0;
25
26 dev_t dev = 0;
27 static struct class *dev_class;
28 static struct cdev etx_cdev;
29
30 static int __init etx_driver_init(void);
31 static void __exit etx_driver_exit(void);
32 static int etx_open(struct inode *inode, struct file *file);
33 static int etx_release(struct inode *inode, struct file *file);
34 static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t * off);
35 static ssize_t etx_write(struct file *filp, const char *buf, size_t len, loff_t * off)
36 static long etx_ioctl(struct file *file, unsigned int cmd, unsigned long arg);
37
38 static struct file_operations fops =
39 {
40     .owner        = THIS_MODULE,
41     .read         = etx_read,
42     .write        = etx_write,
43     .open         = etx_open,
44     .unlocked_ioctl = etx_ioctl,
45     .release      = etx_release,
46 };
47
48 //Interrupt handler for IRQ 11.
```

```
49 static irqreturn_t irq_handler(int irq,void *dev_id) {
50     struct siginfo info;
51     printk(KERN_INFO "Shared IRQ: Interrupt Occurred");
52
53     //Sending signal to app
54     memset(&info, 0, sizeof(struct siginfo));
55     info.si_signo = SIGETX;
56     info.si_code = SI_QUEUE;
57     info.si_int = 1;
58
59     if (task != NULL) {
60         printk(KERN_INFO "Sending signal to app\n");
61         if(send_sig_info(SIGETX, &info, task) < 0) {
62             printk(KERN_INFO "Unable to send signal\n");
63         }
64     }
65
66     return IRQ_HANDLED;
67 }
68
69 static int etx_open(struct inode *inode, struct file *file)
70 {
71     printk(KERN_INFO "Device File Opened...!!!\n");
72     return 0;
73 }
74
75 static int etx_release(struct inode *inode, struct file *file)
76 {
77     struct task_struct *ref_task = get_current();
78     printk(KERN_INFO "Device File Closed...!!!\n");
79
80     //delete the task
81     if(ref_task == task) {
82         task = NULL;
83     }
84     return 0;
85 }
86
87 static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t *off)
88 {
89     printk(KERN_INFO "Read Function\n");
90     asm("int $0x3B"); //Triggering Interrupt. Corresponding to irq 11
91     return 0;
92 }
93 static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, loff_t
94 {
95     printk(KERN_INFO "Write function\n");
96     return 0;
97 }
98
99 static long etx_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
100 {
101     if (cmd == REG_CURRENT_TASK) {
102         printk(KERN_INFO "REG_CURRENT_TASK\n");
103         task = get_current();
104         signum = SIGETX;
105     }
106     return 0;
107 }
108
109
110 static int __init etx_driver_init(void)
111 {
```

```
112     /*Allocating Major number*/
113     if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
114         printk(KERN_INFO "Cannot allocate major number\n");
115         return -1;
116     }
117     printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
118
119     /*Creating cdev structure*/
120     cdev_init(&etx_cdev,&fops);
121
122     /*Adding character device to the system*/
123     if((cdev_add(&etx_cdev,dev,1)) < 0){
124         printk(KERN_INFO "Cannot add the device to the system\n");
125         goto r_class;
126     }
127
128     /*Creating struct class*/
129     if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
130         printk(KERN_INFO "Cannot create the struct class\n");
131         goto r_class;
132     }
133
134     /*Creating device*/
135     if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
136         printk(KERN_INFO "Cannot create the Device 1\n");
137         goto r_device;
138     }
139
140     if (request_irq(IRQ_NO, irq_handler, IRQF_SHARED, "etx_device", (void *)(irq_handl
141         printk(KERN_INFO "my_device: cannot register IRQ ");
142         goto irq;
143     }
144
145     printk(KERN_INFO "Device Driver Insert...Done!!!\n");
146     return 0;
147 irq:
148     free_irq(IRQ_NO,(void *)(irq_handler));
149 r_device:
150     class_destroy(dev_class);
151 r_class:
152     unregister_chrdev_region(dev,1);
153     return -1;
154 }
155
156 void __exit etx_driver_exit(void)
157 {
158     free_irq(IRQ_NO,(void *)(irq_handler));
159     device_destroy(dev_class,dev);
160     class_destroy(dev_class);
161     cdev_del(&etx_cdev);
162     unregister_chrdev_region(dev, 1);
163     printk(KERN_INFO "Device Driver Remove...Done!!!\n");
164 }
165
166 module_init(etx_driver_init);
167 module_exit(etx_driver_exit);
168
169 MODULE_LICENSE("GPL");
170 MODULE_AUTHOR("EmbeTronicX <embedtronicx@gmail.com>");
171 MODULE_DESCRIPTION("A simple device driver - Signals");
172 MODULE_VERSION("1.20");
```

Makefile:

```

1 obj-m += driver.o
2
3 KDIR = /lib/modules/$(shell uname -r)/build
4
5 all:
6     make -C $(KDIR) M=$(shell pwd) modules
7
8 clean:
9     make -C $(KDIR) M=$(shell pwd) clean

```

Application Source Code

This application register with the driver using IOCTL. Once it registered, it will be waiting for the signal from the driver. If we want to close this application we need to press CTRL+C. Because we it will run infinitely. We have installed CTRL+C signal handler.

test_app.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <sys/types.h>
5 #include <sys/stat.h>
6 #include <fcntl.h>
7 #include <unistd.h>
8 #include <sys/ioctl.h>
9 #include <signal.h>
10
11 #define REG_CURRENT_TASK _IOW('a','a',int32_t*)
12
13 #define SIGETX 44
14
15 static int done = 0;
16 int check = 0;
17
18 void ctrl_c_handler(int n, siginfo_t *info, void *unused)
19 {
20     if (n == SIGINT) {
21         printf("\nreceived ctrl-c\n");
22         done = 1;
23     }
24 }
25
26 void sig_event_handler(int n, siginfo_t *info, void *unused)
27 {
28     if (n == SIGETX) {
29         check = info->si_int;
30         printf ("Received signal from kernel : Value = %u\n", check);
31     }
32 }
33
34 int main()

```

```

35 {
36     int fd;
37     int32_t value, number;
38     struct sigaction act;
39
40     printf("*****\n");
41     printf("*****www.EmbedTronicX.com*****\n");
42     printf("*****\n");
43
44     /* install ctrl-c interrupt handler to cleanup at exit */
45     sigemptyset (&act.sa_mask);
46     act.sa_flags = (SA_SIGINFO | SA_RESETHAND);
47     act.sa_sigaction = ctrl_c_handler;
48     sigaction (SIGINT, &act, NULL);
49
50     /* install custom signal handler */
51     sigemptyset(&act.sa_mask);
52     act.sa_flags = (SA_SIGINFO | SA_RESTART);
53     act.sa_sigaction = sig_event_handler;
54     sigaction(SIGGETX, &act, NULL);
55
56     printf("Installed signal handler for SIGGETX = %d\n", SIGGETX);
57
58     printf("\nOpening Driver\n");
59     fd = open("/dev/etx_device", O_RDWR);
60     if(fd < 0) {
61         printf("Cannot open device file...\n");
62         return 0;
63     }
64
65     printf("Registering application ...");
66     /* register this task with kernel for signal */
67     if (ioctl(fd, REG_CURRENT_TASK,(int32_t*) &number)) {
68         printf("Failed\n");
69         close(fd);
70         exit(1);
71     }
72     printf("Done!!!\n");
73
74     while(!done) {
75         printf("Waiting for signal...\n");
76
77         //blocking check
78         while (!done && !check);
79         check = 0;
80     }
81
82     printf("Closing Driver\n");
83     close(fd);
84 }
```

Building Driver and Application

- Build the driver by using Makefile (`sudo make`)
- Use below line in terminal to compile the user space application.

`gcc -o test_app test_app.c`

Execution (Output)

As of now, we have driver.ko and test_app. Now we will see the output.

- Load the driver using `sudo insmod driver.ko`
- Run the application (`sudo ./test_app`)

```
*****
*****WWW.EmbeTronicX.com*****
*****
```

Installed signal handler for SIGETX = 44

*Opening Driver
Registering application ...Done!!!
Waiting for signal...*

- This application will be waiting for signal
- To send the signal from driver to app, we need to trigger the interrupt by reading the driver (`sudo cat /dev/etx_device`).
- Now see the Dmesg (dmesg)

Major = 246 Minor = 0

*Device Driver Insert...Done!!!
Device File Opened....!!!
REG_CURRENT_TASK
Device File Opened....!!!
Read Function
Shared IRQ: Interrupt Occurred
Sending signal to app
Device File Closed....!!!*

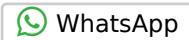
- As per the print, driver has send the signal. Now check the app.

Received signal from kernel : Value = 1

- So application also got the signal.
- Close the application by pressing CTRL+C
- Unload the module using sudo rmmod driver

Share this:

[Share on Tumblr](#) [Tweet](#)



[Pocket](#)



[pinit_fg_en_rect_gray_20](#) Linux Device Driver Tutorial Part 25 – Sending Signal from Linux Device Driver to User Space

Like this:

Loading...



[report this ad](#)

 ezoic[report this ad](#)[Sidebar ▾](#)

📁 Device Drivers

Using Kernel Timer In Linux Device Driver

**Linux Device Driver Tutorial Part 26 -
Using Kernel Timer 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 26 – Using Kernel Timer In Linux Device Driver.

Post Contents

[1 Timer](#)

[1.1 Introduction](#)

[2 Timer in Linux Kernel](#)

[3 Uses of Kernel Timers](#)

[4 Kernel Timer API](#)

[4.1 Initialize / Setup Kernel Timer](#)

[4.1.1 init_timer](#)

[4.1.2 setup_timer](#)

[4.1.3 DEFINE_TIMER](#)

[4.2 Start a Kernel Timer](#)

[4.2.1 add_timer](#)

[4.3 Modifying Kernel Timer's timeout](#)

[4.3.1 mod_timer](#)

[4.4 Stop a Kernel Timer](#)

[4.4.1 del_timer](#)

[4.4.2 del_timer_sync](#)

[4.5 Check Kernel Timer status](#)

[4.5.1 timer_pending](#)

[5 Device Driver Source Code](#)

[6 Building and Testing Driver](#)

7 Points to remember

- 7.0.1 Share this:
- 7.0.2 Like this:
- 7.0.3 Related

Timer

Introduction

What is a timer in general? According from [Wikipedia](#), A timer is a specialized type of clock used for measuring specific time intervals. Timers can be categorized into two main types. A timer which counts upwards from zero for measuring elapsed time is often called a stopwatch, while a device which counts down from a specified time interval is more usually called a timer.

Timer in Linux Kernel

In Linux, kernel keeps track of the flow of time by means of timer interrupts. This timer interrupts are generated at regular timer intervals by using system's timing hardware. Every time a timer interrupt occurs, the value of an internal kernel counter is incremented. The counter is initialized to 0 at system boot, so it represents the number of clock ticks since last boot.

Kernel timer offers less precision but is more efficient in situations where the timer will probably be canceled before it fires. There are many places in the kernel where timers are used to detect when a device or a

network peer has failed to respond within the expected time.

When you want to do some action after some time, then kernel timers are one of the option for you. These timers are used to schedule execution of a function at a particular time in the future, based on the clock tick, and can be used for a variety of tasks.

Uses of Kernel Timers

- Polling a device by checking its state at regular intervals when the hardware can't fire interrupts.
- User wants to send some message to other device at regular intervals.
- Send error when some action didn't happened in particular time period.
- Etc.

Kernel Timer API

Linux Kernel provides the driver to create timers which are not periodic by default, register the timers and delete the timers.

We need to include the `<linux/timer.h>` (`#include <linux/timer.h>`) in order to use kernel timers. Kernel timers are described by the `timer_list` structure, defined in `<linux/timer.h>`:

```
1 struct timer_list {  
2     /* ... */  
3     unsigned long expires;
```

```
4     void (*function)(unsigned long);
5     unsigned long data;
6 };
```

The **expires** field contains the expiration time of the timer (in jiffies). On expiration, **function()** will be called with the given **data** value.

Initialize / Setup Kernel Timer

There are multiple ways to Initialize / Setup Kernel Timer. We'll see one by one.

init_timer

```
void fastcall init_timer ( struct timer_list * timer);
```

This function is used to initialize the timer. **init_timer** must be done to a timer prior calling any of the other timer functions. If you are using this function to initialize the timer, then you need to set the callback function and data of the **timer_list** structure manually.

Argument:

timer – the timer to be initialized

setup_timer

```
void setup_timer(timer, function, data);
```

Instead of initializing timer manually by calling **init_timer**, you can use this function to set **data** and **function** of **timer_list** structure and initialize the timer. *This is recommended to use.*

Argument:

timer – the timer to be initialized

function – Callback function to be called when timer expires

data – data has to be given to the callback function

DEFINE_TIMER

```
DEFINE_TIMER(_name, _function, _expires, _data)
```

If we are using this method, then no need to create the **timer_list** structure in our side. Kernel will create the structure in the name of **_name** and initialize it.

Argument:

_name – name of the timer_list structure to be created

_function – Callback function to be called when timer expires

_expires – the expiration time of the timer (in jiffies)

_data – data has to be given to the callback function

Start a Kernel Timer

add_timer

```
void add_timer(struct timer_list *timer);
```

This will start a timer.

Argument:

timer – the timer needs to be start

Modifying Kernel Timer's timeout

mod_timer

```
int mod_timer (struct timer_list * timer, unsigned long expires);
```

This function is used to modify a timer's timeout. This is a more efficient way to update the expire field of an active timer (if the timer is inactive it will be activated).

`mod_timer(timer, expires)` is equivalent to:

```
del_timer(timer); timer->expires = expires; add_timer(timer);
```

Argument:

`timer` – the timer needs to be modified the timer period

`expires` – the updated expiration time of the timer (in jiffies)

Return:

The function returns whether it has modified a pending timer or not.

0 – `mod_timer` of an inactive timer

1 – `mod_timer` of an active timer

Stop a Kernel Timer

These below functions will be used to deactivate the kernel timers.

del_timer

```
int del_timer (struct timer_list * timer);
```

This will deactivate a timer. This works on both active and inactive timers.

Argument:

timer – the timer needs to be deactivate

Return:

The function returns whether it has deactivated a pending timer or not.

0 – **del_timer** of an inactive timer

1 – **del_timer** of an active timer

del_timer_sync

```
int del_timer_sync (struct timer_list * timer);
```

This will deactivate a timer and wait for the handler to finish. This works on both active and inactive timers.

Argument:

timer – the timer needs to be deactivate

Return:

The function returns whether it has deactivated a pending timer or not.

0 – **del_timer_sync** of an inactive timer

1 – **del_timer_sync** of an active timer

Note: *callers must prevent restarting of the timer, otherwise this function is meaningless. It must not be called from interrupt contexts. The caller must not hold locks which would prevent completion of the timer's handler. The timer's handler must not call add_timer_on. Upon exit the timer is not queued and the handler is not running on any CPU.*

Check Kernel Timer status

timer_pending

```
int timer_pending(const struct timer_list * timer);
```

This will tell whether a given timer is currently pending, or not. Callers must ensure serialization wrt. other operations done to this timer, eg. interrupt contexts, or other CPUs on SMP.

Argument:

timer – the timer needs to check status

Return:

The function returns whether timer is pending or not.

0 – timer is not pending

1 – timer is pending

Device Driver Source Code

In this example we took the basic driver source code from [this](#) tutorial. On top of that code we have added the timer. The steps are mentioned below.

1. Initialize the timer and set the time interval
2. After timeout, registered timer callback will be called.
3. In the timer callback function again we are re-enabling the timer. We have to do this step if we want periodic timer. Otherwise we can ignore this.
4. Once we done, we can disable the timer.

driver.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 #include <linux/timer.h>
9 #include <linux/jiffies.h>
10
11 //Timer Variable
12 #define TIMEOUT 5000    //milliseconds
13 static struct timer_list etx_timer;
14 static unsigned int count = 0;
15
16 dev_t dev = 0;
17 static struct class *dev_class;
18 static struct cdev etx_cdev;
19
20 static int __init etx_driver_init(void);
21 static void __exit etx_driver_exit(void);
22 static int etx_open(struct inode *inode, struct file *file);
23 static int etx_release(struct inode *inode, struct file *file);
24 static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t * off);
25 static ssize_t etx_write(struct file *filp, const char *buf, size_t len, loff_t * off)
26
27 static struct file_operations fops =
28 {
29     .owner        = THIS_MODULE,
30     .read         = etx_read,
31     .write        = etx_write,
32     .open         = etx_open,
33     .release      = etx_release,
34 };
35
36 //Timer Callback function. This will be called when timer expires
37 void timer_callback(unsigned long data)
38 {
39     /* do your timer stuff here */
40     printk(KERN_INFO "Timer Callback function Called [%d]\n",count++);
```

```
41     /*
42      * Re-enable timer. Because this function will be called only first time.
43      * If we re-enable this will work like periodic timer.
44      */
45     mod_timer(&etx_timer, jiffies + msecs_to_jiffies(TIMEOUT));
46 }
47
48
49 static int etx_open(struct inode *inode, struct file *file)
50 {
51     printk(KERN_INFO "Device File Opened...!!!\n");
52     return 0;
53 }
54
55 static int etx_release(struct inode *inode, struct file *file)
56 {
57     printk(KERN_INFO "Device File Closed...!!!\n");
58     return 0;
59 }
60
61 static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t *off)
62 {
63     printk(KERN_INFO "Read Function\n");
64     return 0;
65 }
66 static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, loff_t
67 {
68     printk(KERN_INFO "Write function\n");
69     return 0;
70 }
71
72 static int __init etx_driver_init(void)
73 {
74     /*Allocating Major number*/
75     if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
76         printk(KERN_INFO "Cannot allocate major number\n");
77         return -1;
78     }
79     printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
80
81     /*Creating cdev structure*/
82     cdev_init(&etx_cdev,&fops);
83
84     /*Adding character device to the system*/
85     if((cdev_add(&etx_cdev,dev,1)) < 0){
86         printk(KERN_INFO "Cannot add the device to the system\n");
87         goto r_class;
88     }
89
90     /*Creating struct class*/
91     if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
92         printk(KERN_INFO "Cannot create the struct class\n");
93         goto r_class;
94     }
95
96     /*Creating device*/
97     if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
98         printk(KERN_INFO "Cannot create the Device 1\n");
99         goto r_device;
100    }
101
102    /* setup your timer to call my_timer_callback */
103    setup_timer(&etx_timer, timer_callback, 0);
```

```

104     /* setup timer interval to based on TIMEOUT Macro */
105     mod_timer(&etx_timer, jiffies + msecs_to_jiffies(TIMEOUT));
106
107     printk(KERN_INFO "Device Driver Insert...Done!!!\n");
108     return 0;
109
110 r_device:
111     class_destroy(dev_class);
112 r_class:
113     unregister_chrdev_region(dev,1);
114     return -1;
115 }
116
117 void __exit etx_driver_exit(void)
118 {
119     /* remove kernel timer when unloading module */
120     del_timer(&etx_timer);
121
122     class_destroy(dev_class);
123     cdev_del(&etx_cdev);
124     unregister_chrdev_region(dev, 1);
125     printk(KERN_INFO "Device Driver Remove...Done!!!\n");
126 }
127
128 module_init(etx_driver_init);
129 module_exit(etx_driver_exit);
130
131 MODULE_LICENSE("GPL");
132 MODULE_AUTHOR("EmbeTronicX <embedtronicx@gmail.com>");
133 MODULE_DESCRIPTION("A simple device driver - Kernel Timer");
134 MODULE_VERSION("1.21");

```

Makefile:

```

1 obj-m += driver.o
2
3 KDIR = /lib/modules/$(shell uname -r)/build
4
5 all:
6     make -C $(KDIR) M=$(shell pwd) modules
7
8 clean:
9     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**
- Now see the Dmesg (**dmesg**)

linux@embedtronicx-VirtualBox: dmesg

[2253.635127] Device Driver Insert...Done!!!
[2258.642048] Timer Callback function Called [0]

```
[ 2263.647050] Timer Callback function Called [1]
[ 2268.652684] Timer Callback function Called [2]
[ 2273.658274] Timer Callback function Called [3]
[ 2278.663885] Timer Callback function Called [4]
[ 2283.668997] Timer Callback function Called [5]
[ 2288.675109] Timer Callback function Called [6]
[ 2293.680160] Timer Callback function Called [7]
[ 2298.685771] Timer Callback function Called [8]
[ 2303.691392] Timer Callback function Called [9]
[ 2308.697013] Timer Callback function Called [10]
[ 2313.702033] Timer Callback function Called [11]
[ 2318.707772] Timer Callback function Called [12]
```

- See the timestamp. That callback function is executing every 5 seconds.
- Unload the module using `sudo rmmod driver`

Points to remember

This timer callback function will be executed from interrupt context. If you want to check that, you can use function `in_interrupt()`, which takes no parameters and returns nonzero if the processor is currently running in interrupt context, either hardware interrupt or software interrupt. Since it is running in interrupt context, user cannot perform some actions inside the callback function mentioned below.

- Go to sleep or relinquish the processor
- Acquire a mutex
- Perform time-consuming tasks
- Access user space virtual memory

In our [next tutorial](#) we will see the High Resolution Timer (hrtimer).

Share this:

[Share on Tumblr](#)[Tweet](#)[WhatsApp](#)[Pocket](#)

[pinit_fg_en_rect_gray_20](#) Linux Device Driver Tutorial Part 26 – Using Kernel Timer In Linux Device Driver

Like this:

Loading...



[report this ad](#)

 ezoic[report this ad](#)[Sidebar ▾](#)

📁 Device Drivers

Using High Resolution Timer In Linux Device Driver

**Linux Device Driver Tutorial Part 27 -
Using High Resolution Timer 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 27 – Using High Resolution Timer In Linux Device Driver.

Post Contents

- [1 High Resolution Timer \(HRT/hrtimer\)](#)
- [2 Users of High Resolution Timer](#)
- [3 High Resolution timer API](#)
 - [3.1 ktime_set](#)
 - [3.2 Initialize High Resolution Timer](#)
 - [3.2.1 hrtimer_init](#)
 - [3.3 Start High Resolution Timer](#)
 - [3.3.1 hrtimer_start](#)
 - [3.4 Stop High Resolution Timer](#)
 - [3.4.1 hrtimer_cancel](#)
 - [3.4.2 hrtimer_try_to_cancel](#)
 - [3.5 Changing the High Resolution Timer's Timeout](#)
 - [3.5.1 hrtimer_forward](#)
 - [3.5.2 hrtimer_forward_now](#)
 - [3.6 Check High Resolution Timer's status](#)
 - [3.6.1 hrtimer_get_remaining](#)
 - [3.6.2 hrtimer_callback_running](#)
 - [3.6.3 hrtimer_cb_get_time](#)
- [4 Using High Resolution Timer In Linux Device Driver](#)
 - [4.1 Driver Source Code](#)
- [5 Building and Testing Driver](#)

6 Points to remember

- 6.0.1 Share this:
- 6.0.2 Like this:
- 6.0.3 Related

High Resolution Timer (HRT/hrtimer)

In our [last tutorial](#) we have seen kernel timer. Now we are taking about high resolution timer. Everyone might have some questions. Why the hell we need two timers? Why can they merge two timers into one? Can't able to integrate? Yes. They have tried to merge these two timers. But they have failed. Because **Cascading Timer Wheel (CTW)** is used in kernel timer. Cascading Timer Wheel (CTW) code is fundamentally not suitable for such an approach like merging these two timers. Because hrtimer is maintaining a time-ordered data structure of timers (timers are inserted in time order to minimize processing at activation time). The data structure used is a red-black tree, which is ideal for performance-focused applications (and happens to be available generically as a library within the kernel).

Kernel Timers are bound to **jiffies**. But this High Resolution Timer (HRT) is bound with 64-bit **nanoseconds** resolution.

With kernel version 2.6.21 onwards, high resolution timers (HRT) are available under Linux. For this, the kernel has to be compiled with the configuration parameter `CONFIG_HIGH_RES_TIMERS` enabled.

There are many ways to check whether high resolution timers are available,

- In the `/boot` directory, check the kernel config file. It should have a line like `CONFIG_HIGH_RES_TIMERS=y`.
- Check the contents of `/proc/timer_list`. For example, the `.resolution` entry showing 1 nanosecond and event_handler as `hrtimer_interrupt` in `/proc/timer_list` indicate that high resolution timers are available.
- Get the clock resolution using the `clock_getres` system call.

Users of High Resolution Timer

- The primary users of precision timers are user-space applications that utilize nanosleep, posix-timers and Interval Timer (itimer) interfaces.
- In-kernel users like drivers and subsystems which require precise timed events (e.g. multimedia).

High Resolution timer API

We need to include the `<linux/hrtimer.h>` (`#include <linux/hrtimer.h>`) in order to use kernel timers. Kernel timers are described by the `hrtimer` structure, defined in `<linux/hrtimer.h>`:

```

1 struct hrtimer {
2     struct rb_node node;
3     ktime_t expires;
4     int (*function) (struct hrtimer *); // Function pointer
5     struct hrtimer_base *base; // Base pointer
6 };

```

Where,

node – red black tree node for time ordered insertion

expires – the absolute expiry time in the hrtimers internal representation. The time is related to the clock on which the timer is based.

function – timer expiry callback function. This function has an integer return value, which should be either `HRTIMER_NORESTART` (for a one-shot timer which should not be started again) or `HRTIMER_RESTART`

for a recurring timer. In the restart case, the callback must set a new expiration time before returning.

base – pointer to the timer base (per cpu and per clock)

The **hrtimer** structure must be initialized by `init_hrtimer_#CLOCKTYPE`.

ktime_set

There is a new type, **ktime_t**, which is used to store a time value in nanoseconds. On 64-bit systems, a **ktime_t** is really just a 64-bit integer value in nanoseconds. On 32-bit machines, however, it is a two-field structure: one 32-bit value holds the number of seconds, and the other holds nanoseconds. The below function used to get the **ktime_t** from seconds and nanoseconds.

```
ktime_set(long secs, long nanosecs);
```

Arguments:

secs – seconds to set

nsecs – nanoseconds to set

Return:

The `ktime_t` representation of the value.

Initialize High Resolution Timer

`hrtimer_init`

```
void hrtimer_init( struct hrtimer *timer, clockid_t clock_id, enum  
                    hrtimer_mode mode );
```

Arguments:

`timer` – the timer to be initialized

`clock_id` – the clock to be used

The clock to use is defined in `./include/linux/time.h` and represents the various clocks that the system supports (such as the real-time clock or a monotonic clock that simply represents time from a starting point, such as system boot).

`CLOCK_MONOTONIC`: a clock which is guaranteed always to move forward in time, but which does not reflect “wall clock time” in any specific way. In the current implementation, `CLOCK_MONOTONIC` resembles the jiffies tick count in that it starts at zero when the system boots and increases monotonically from there.

`CLOCK_REALTIME`: which matches the current real-world time.

`mode` – timer mode absolute (`HRTIMER_MODE_ABS`) or relative (`HRTIMER_MODE_REL`)

Start High Resolution Timer

Once a timer has been initialized, it can be started with the below mentioned function.

hrtimer_start

```
int hrtimer_start(struct hrtimer *timer, ktime_t time, const enum  
                    hrtimer_mode mode);
```

This call is used to (Re)start an hrtimer on the current CPU.

Arguments:

timer – the timer to be added

time – expiry time

mode – expiry mode: absolute (HRTIMER_MODE_ABS) or relative (HRTIMER_MODE_REL)

Returns:

0 on success 1 when the timer was active

Stop High Resolution Timer

Using below function, we can able to stop the High Resolution Timer.

hrtimer_cancel

```
int hrtimer_cancel (struct hrtimer * timer);
```

This will cancel a timer and wait for the handler to finish.

Arguments:

timer – the timer to be cancelled

Returns:

- 0 when the timer was not active

- 1 when the timer was active

hrtimer_try_to_cancel

```
int hrtimer_try_to_cancel (struct hrtimer * timer);
```

This will try to deactivate a timer.

Arguments:

timer – hrtimer to stop

Returns:

- 0 when the timer was not active
- 1 when the timer was active
- -1 when the timer is currently executing the callback function and cannot be stopped

Changing the High Resolution Timer's Timeout

If we are using this High Resolution Timer (hrtimer) as periodic timer, then the callback must set a new expiration time before returning.

Usually, restarting timers are used by kernel subsystems which need a callback at a regular interval.

hrtimer_forward

```
u64 hrtimer_forward (struct hrtimer * timer, ktime_t now, ktime_t  
interval);
```

This will forward the timer expiry so it will expire in the future by the given interval.

Arguments:

timer – hrtimer to forward

now – forward past this time

interval – the interval to forward

Returns:

Returns the number of overruns.

hrtimer_forward_now

```
u64 hrtimer_forward_now(struct hrtimer *timer, ktime_t interval);
```

This will forward the timer expiry so it will expire in the future from now by the given interval.

Arguments:

timer – hrtimer to forward

interval – the interval to forward

Returns:

Returns the number of overruns.

Check High Resolution Timer's status

The below explained functions are used to get the status and timings.

hrtimer_get_remaining

```
ktime_t hrtimer_get_remaining (const struct hrtimer * timer);
```

This is used to get remaining time for the timer.

Arguments:

timer – hrtimer to get the remaining time

Returns:

Returns the remaining time.

hrtimer_callback_running

```
int hrtimer_callback_running(struct hrtimer *timer);
```

This is the helper function to check, whether the timer is running the callback function.

Arguments:

timer – hrtimer to check

Returns:

- 0 when the timer's callback function is not running
- 1 when the timer's callback function is running

hrtimer_cb_get_time

```
ktime_t hrtimer_cb_get_time(struct hrtimer *timer);
```

This function used to get the current time of the given timer.

Arguments:

timer – hrtimer to get the time

Returns:

Returns the time.

Using High Resolution Timer In Linux Device Driver

In this example we took the basic driver source code from [this](#) tutorial. On top of that code we have added the high resolution timer. The steps are mentioned below.

1. Initialize and start the timer in init function
2. After timeout, registered timer callback will be called.
3. In the timer callback function again we are forwarding the time period and return **HRTIMER_RESTART**. We have to do this step if we want periodic timer. Otherwise we can ignore that time forwarding and return **HRTIMER_NORESTART**.
4. Once we are done, we can disable the timer.

Driver Source Code

driver.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 #include <linux/hrtimer.h>
9 #include <linux/ktime.h>
10
11 //Timer Variable
12 #define TIMEOUT 5000 * 1000000L //nano seconds
13 static struct hrtimer etx_hr_timer;
14 static unsigned int count = 0;
15
16 dev_t dev = 0;
17 static struct class *dev_class;
18 static struct cdev etx_cdev;
19
20 static int __init etx_driver_init(void);
21 static void __exit etx_driver_exit(void);
22 static int etx_open(struct inode *inode, struct file *file);
23 static int etx_release(struct inode *inode, struct file *file);
24 static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t * off);
```

```

25 static ssize_t etx_write(struct file *filp, const char *buf, size_t len, loff_t * off)
26
27 static struct file_operations fops =
28 {
29     .owner        = THIS_MODULE,
30     .read         = etx_read,
31     .write        = etx_write,
32     .open          = etx_open,
33     .release      = etx_release,
34 };
35
36 //Timer Callback function. This will be called when timer expires
37 enum hrtimer_restart timer_callback(struct hrtimer *timer)
38 {
39     /* do your timer stuff here */
40     printk(KERN_INFO "Timer Callback function Called [%d]\n",count++);
41     hrtimer_forward_now(timer,ktime_set(0,TIMEOUT));
42     return HRTIMER_RESTART;
43 }
44
45 static int etx_open(struct inode *inode, struct file *file)
46 {
47     printk(KERN_INFO "Device File Opened...!!!\n");
48     return 0;
49 }
50
51 static int etx_release(struct inode *inode, struct file *file)
52 {
53     printk(KERN_INFO "Device File Closed...!!!\n");
54     return 0;
55 }
56
57 static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t *off)
58 {
59     printk(KERN_INFO "Read Function\n");
60     return 0;
61 }
62 static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, loff_t
63 {
64     printk(KERN_INFO "Write function\n");
65     return 0;
66 }
67
68 static int __init etx_driver_init(void)
69 {
70     ktime_t ktime;
71
72     /*Allocating Major number*/
73     if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
74         printk(KERN_INFO "Cannot allocate major number\n");
75         return -1;
76     }
77     printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
78
79     /*Creating cdev structure*/
80     cdev_init(&etx_cdev,&fops);
81
82     /*Adding character device to the system*/
83     if((cdev_add(&etx_cdev,dev,1)) < 0){
84         printk(KERN_INFO "Cannot add the device to the system\n");
85         goto r_class;
86     }
87 }
```

```

88     /*Creating struct class*/
89     if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
90         printk(KERN_INFO "Cannot create the struct class\n");
91         goto r_class;
92     }
93
94     /*Creating device*/
95     if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
96         printk(KERN_INFO "Cannot create the Device 1\n");
97         goto r_device;
98     }
99
100    ktime = ktime_set(0, TIMEOUT);
101    hrtimer_init(&etx_hr_timer, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
102    etx_hr_timer.function = &timer_callback;
103    hrtimer_start( &etx_hr_timer, ktime, HRTIMER_MODE_REL);
104
105    printk(KERN_INFO "Device Driver Insert...Done!!!\n");
106    return 0;
107 r_device:
108     class_destroy(dev_class);
109 r_class:
110     unregister_chrdev_region(dev,1);
111     return -1;
112 }
113
114 void __exit etx_driver_exit(void)
115 {
116     //stop the timer
117     hrtimer_cancel(&etx_hr_timer);
118     device_destroy(dev_class,dev);
119     class_destroy(dev_class);
120     cdev_del(&etx_cdev);
121     unregister_chrdev_region(dev, 1);
122     printk(KERN_INFO "Device Driver Remove...Done!!!\n");
123 }
124
125 module_init(etx_driver_init);
126 module_exit(etx_driver_exit);
127
128 MODULE_LICENSE("GPL");
129 MODULE_AUTHOR("EmbeTronicX <embedtronicx@gmail.com>");
130 MODULE_DESCRIPTION("A simple device driver - High Resolution Timer");
131 MODULE_VERSION("1.22");

```

Makefile:

```

1 obj-m += driver.o
2
3 KDIR = /lib/modules/$(shell uname -r)/build
4
5 all:
6     make -C $(KDIR) M=$(shell pwd) modules
7
8 clean:
9     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`
- Now see the Dmesg (`dmesg`)

linux@embedtronicx-VirtualBox: dmesg

```
[ 2643.773119] Device Driver Insert...Done!!!
[ 2648.773546] Timer Callback function Called [0]
[ 2653.773609] Timer Callback function Called [1]
[ 2658.774170] Timer Callback function Called [2]
[ 2663.773271] Timer Callback function Called [3]
[ 2668.773388] Timer Callback function Called [4]
```

- See the timestamp. That callback function is executing every 5 seconds.
- Unload the module using `sudo rmmod driver`

Points to remember

This timer callback function will be executed from interrupt context. If you want to check that, you can use function `in_interrupt()`, which takes no parameters and returns nonzero if the processor is currently running in interrupt context, either hardware interrupt or software interrupt. Since it is running in interrupt context, user cannot perform some actions inside the callback function mentioned below.

- Go to sleep or relinquish the processor
- Acquire a mutex
- Perform time-consuming tasks
- Access user space virtual memory

Share this:

[Share on Tumblr](#) [Tweet](#)



[WhatsApp](#)

[Pocket](#)



[pinit_fg_en_rect_gray_20](#) [Linux Device Driver Tutorial Part 27 – Using High Resolution Timer In Linux Device Driver](#)

Like this:

Loading...



[report this ad](#)

 ezoic[report this ad](#)[Sidebar ▾](#)

Device Drivers

completion in linux

Linux Device Driver Tutorial Part 28 – Completion 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 28 – Completion in Linux Device Driver.

Post Contents

- [1 Prerequisites](#)
- [2 Completion](#)
- [3 Completion in Linux Device Driver](#)
 - [3.1 Initialize Completion](#)
 - [3.1.1 Static Method](#)
 - [3.1.2 Dynamic Method](#)
 - [3.2 Re-Initializing Completion](#)
 - [3.3 Waiting for completion](#)
 - [3.3.1 wait_for_completion](#)
 - [3.3.2 wait_for_completion_timeout](#)
 - [3.3.3 wait_for_completion_interruptible](#)
 - [3.3.4 wait_for_completion_interruptible_timeout](#)
 - [3.3.5 wait_for_completion_killable](#)
 - [3.3.6 wait_for_completion_killable_timeout](#)
 - [3.3.7 try_wait_for_completion](#)
 - [3.4 Waking Up Task](#)
 - [3.4.1 complete](#)
 - [3.4.2 complete_all](#)
 - [3.5 Check the status](#)
 - [3.5.1 completion_done](#)
- [4 Driver Source Code - Completion in Linux](#)

- [4.1 Completion created by static method](#)
- [4.2 Completion created by dynamic method](#)
- [4.3 MakeFile](#)
- [5 Building and Testing Driver](#)
 - [5.0.1 Share this:](#)
 - [5.0.2 Like this:](#)
 - [5.0.3 Related](#)

Prerequisites

In the example section, I had used kthread to explain this completion. If you don't know what is kthread and how to use it, then I would recommend you to explore that by using below link.

- [1. Kthread Tutorial in Linux Kernel](#)
- [2. Waitqueue Tutorial in Linux Kernel](#)

Completion

Completion, the name itself says. When we want to notify or wakeup some thread or something when we finished some work, then we can use completion. We'll take one situation. We want to wait one thread for something to run. Until that time that thread has to sleep. Once that process finished then we need to wake up that thread which is sleeping. We can do this by using completion without race conditions.

This completions are a synchronization mechanism which is good

method in the above situation mentioned rather than using improper locks/semaphores and busy-loops.

Completion in Linux Device Driver

In Linux kernel, Completions are developed by using [waitqueue](#).

The advantage of using completions is that they have a well defined, focused purpose which makes it very easy to see the intent of the code, but they also result in more efficient code as all threads can continue execution until the result is actually needed, and both the waiting and the signalling is highly efficient using low level scheduler sleep/wakeup facilities.

There are 5 important steps in Completions.

1. Initializing Completion
2. Re-Initializing Completion
3. Waiting for completion (The code is waiting and sleeping for something to finish)
4. Waking Up Task (Sending signal to sleeping part)
5. Check the status

Initialize Completion

We have to include `<linux/completion.h>` and creating a variable of type **struct completion**, which has only two fields:

```
1 struct completion {  
2     unsigned int done;  
3     wait_queue_head_t wait;  
4 };
```

Where, **wait** is the waitqueue to place tasks on for waiting (if any). **done** is the completion flag for indicating whether it's completed or not.

We can create the struct variable in two ways.

1. Static Method
2. Dynamic Method

You can use any one of the method.

Static Method

```
DECLARE_COMPLETION(data_read_done);
```

Where the “**data_read_done**” is the name of the struct which is going to create statically.

Dynamic Method

```
init_completion (struct completion * x);
```

Where, **x** - completion structure that is to be initialized

Example:

```
1 struct completion data_read_done;  
2  
3 init_completion(&data_read_done);
```

In this **init_completion** call we initialize the waitqueue and set **done** to 0, i.e. “not completed” or “not done”.

Re-Initializing Completion

```
reinit_completion (struct completion * x);
```

Where, x - completion structure that is to be reinitialized

Example:

```
1 reinit_completion(&data_read_done);
```

This function should be used to reinitialize a completion structure so it can be reused. This is especially important after **complete_all** is used. This simply resets the ->done field to 0 (“not done”), without touching the waitqueue. Callers of this function must make sure that there are no racy wait_for_completion() calls going on in parallel.

Waiting for completion

For a thread to wait for some concurrent activity to finish, it calls the any one of the function based on the use case.

wait_for_completion

This is used to make the function waits for completion of a task.

```
void wait_for_completion (struct completion * x);
```

Where, x - holds the state of this particular completion

This waits to be signaled for completion of a specific task. It is NOT interruptible and there is no timeout.

Example:

```
1 wait_for_completion (&data_read_done);
```

Note that **wait_for_completion()** is calling **spin_lock_irq()**/**spin_unlock_irq()**, so it can only be called safely when you know that interrupts are enabled. Calling it from IRQs-off atomic contexts will result in hard-to-detect spurious enabling of interrupts.

wait_for_completion_timeout

This is used to make the function waits for completion of a task with timeout. Timeouts are preferably calculated with `msecs_to_jiffies()` or `usecs_to_jiffies()`, to make the code largely HZ-invariant.

```
unsigned long wait_for_completion_timeout (struct completion
                                         * x, unsigned long timeout);
```

where, `x` - holds the state of this particular completion

`timeout` - timeout value in jiffies

This waits for either a completion of a specific task to be signaled or for a specified timeout to expire. The timeout is in jiffies. It is not interruptible.

It **returns 0** if timed out, and **positive** (at least 1, or number of jiffies left till timeout) if completed.

Example:

```
1 wait_for_completion_timeout (&data_read_done);
```

wait_for_completion_interruptible

This waits for completion of a specific task to be signaled. It is interruptible.

```
int wait_for_completion_interruptible (struct completion * x);
```

where, `x` - holds the state of this particular completion

It return **-ERESTARTSYS** if interrupted, **0** if completed.

wait_for_completion_interruptible_timeout

This waits for either a completion of a specific task to be signaled or for a specified timeout to expire. It is interruptible. The timeout is in jiffies. Timeouts are preferably calculated with msecs_to_jiffies() or usecs_to_jiffies(), to make the code largely HZ-invariant.

```
long wait_for_completion_interruptible_timeout (struct completion  
* x, unsigned long timeout);
```

where, **x** - holds the state of this particular completion

timeout - timeout value in jiffies

It return **-ERESTARTSYS** if interrupted, **0** if timed out, positive (at least 1, or number of jiffies left till timeout) if completed.

wait_for_completion_killable

This waits to be signaled for completion of a specific task. It can be interrupted by a kill signal.

```
int wait_for_completion_killable (struct completion * x);
```

where, **x** - holds the state of this particular completion

It return **-ERESTARTSYS** if interrupted, **0** if completed.

wait_for_completion_killable_timeout

This waits for either a completion of a specific task to be signaled or for a specified timeout to expire. It can be interrupted by a kill signal. The timeout is in jiffies. Timeouts are preferably calculated with `msecs_to_jiffies()` or `usecs_to_jiffies()`, to make the code largely HZ-invariant.

```
long wait_for_completion_killable_timeout (struct completion  
* x, unsigned long timeout);
```

where, **x** - holds the state of this particular completion

timeout - timeout value in jiffies

It return **-ERESTARTSYS** if interrupted, **0** if timed out, positive (at least 1, or number of jiffies left till timeout) if completed.

try_wait_for_completion

This function will not put the thread on the wait queue but rather returns false if it would need to enqueue (block) the thread, else it consumes one posted completion and returns true.

```
bool try_wait_for_completion (struct completion * x);
```

where, **x** - holds the state of this particular completion

It returns **0** if a completion is not available **1** if a got it succeeded.

This `try_wait_for_completion()` is safe to be called in IRQ or atomic context.

Waking Up Task

complete

This will wake up a single thread waiting on this completion. Threads will be awakened in the same order in which they were queued.

```
void complete (struct completion * x);
```

where, x - holds the state of this particular completion

Example:

```
1 complete(&data_read_done);
```

complete_all

This will wake up all threads waiting on this particular completion event.

```
void complete_all (struct completion * x);
```

where, x - holds the state of this particular completion

Check the status completion_done

This is the test to see if a completion has any waiters.

```
bool completion_done (struct completion * x);
```

where, x - holds the state of this particular completion

It returns 0 if there are waiters (wait_for_completion in progress) 1 if there are no waiters.

This `completion_done()` is safe to be called in IRQ or atomic context.

Driver Source Code - Completion in Linux

First i will explain you the concept of driver code.

In this source code, two places we are sending complete call. One from

read function and another one from driver exit function.

I've created one thread (**wait_function**) which has **while(1)**. That thread will always wait for the event to complete. It will be sleeping until it gets complete call. When it gets the complete call, it will check the condition. If condition is 1 then the complete came from read function. If it is 2, then the complete came from exit function. If complete came from read, it will print the read count and it will again wait. If its from exit function, it will exit from the thread.

Here I've added two versions of code.

1. Completion created by static method
2. Completion created by dynamic method

But operation wise, both are same.

You can also find the source code here.

Completion created by static method

```
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
11 #include <linux/kthread.h>
12 #include <linux/completion.h>      // Required for the completion
13
14
15 uint32_t read_count = 0;
16 static struct task_struct *wait_thread;
17
18 DECLARE_COMPLETION(data_read_done);
19
20 dev_t dev = 0;
21 static struct class *dev_class;
22 static struct cdev etx_cdev;
23 int completion_flag = 0;
24
25 static int __init etx_driver_init(void);
26 static void __exit etx_driver_exit(void);
27
28 /***** Driver Functions *****/
29 static int etx_open(struct inode *inode, struct file *file);
30 static int etx_release(struct inode *inode, struct file *file);
```

```
31 static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t * off);
32 static ssize_t etx_write(struct file *filp, const char *buf, size_t len, loff_t * off)
33 {
34     static struct file_operations fops =
35     {
36         .owner          = THIS_MODULE,
37         .read           = etx_read,
38         .write          = etx_write,
39         .open            = etx_open,
40         .release        = etx_release,
41     };
42
43     static int wait_function(void *unused)
44     {
45
46         while(1) {
47             printk(KERN_INFO "Waiting For Event...\n");
48             wait_for_completion (&data_read_done);
49             if(completion_flag == 2) {
50                 printk(KERN_INFO "Event Came From Exit Function\n");
51                 return 0;
52             }
53             printk(KERN_INFO "Event Came From Read Function - %d\n", ++read_count)
54             completion_flag = 0;
55         }
56         do_exit(0);
57         return 0;
58     }
59
60     static int etx_open(struct inode *inode, struct file *file)
61     {
62         printk(KERN_INFO "Device File Opened...!!!\n");
63         return 0;
64     }
65
66     static int etx_release(struct inode *inode, struct file *file)
67     {
68         printk(KERN_INFO "Device File Closed...!!!\n");
69         return 0;
70     }
71
72     static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t *off)
73     {
74         printk(KERN_INFO "Read Function\n");
75         completion_flag = 1;
76         if(!completion_done (&data_read_done)) {
77             complete (&data_read_done);
78         }
79         return 0;
80     }
81     static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, loff_t
82     {
83         printk(KERN_INFO "Write function\n");
84         return 0;
85     }
86
87     static int __init etx_driver_init(void)
88     {
89         /*Allocating Major number*/
90         if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
91             printk(KERN_INFO "Cannot allocate major number\n");
92             return -1;
93         }
```

```
94         printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
95
96     /*Creating cdev structure*/
97     cdev_init(&etx_cdev,&fops);
98     etx_cdev.owner = THIS_MODULE;
99     etx_cdev.ops = &fops;
100
101    /*Adding character device to the system*/
102    if((cdev_add(&etx_cdev,dev,1)) < 0){
103        printk(KERN_INFO "Cannot add the device to the system\n");
104        goto r_class;
105    }
106
107    /*Creating struct class*/
108    if((dev_class = class_create(TTHIS_MODULE,"etx_class")) == NULL){
109        printk(KERN_INFO "Cannot create the struct class\n");
110        goto r_class;
111    }
112
113    /*Creating device*/
114    if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
115        printk(KERN_INFO "Cannot create the Device 1\n");
116        goto r_device;
117    }
118
119    //Create the kernel thread with name 'mythread'
120    wait_thread = kthread_create(wait_function, NULL, "WaitThread");
121    if (wait_thread) {
122        printk("Thread Created successfully\n");
123        wake_up_process(wait_thread);
124    } else
125        printk(KERN_INFO "Thread creation failed\n");
126
127    printk(KERN_INFO "Device Driver Insert...Done!!!\n");
128    return 0;
129
130 r_device:
131     class_destroy(dev_class);
132 r_class:
133     unregister_chrdev_region(dev,1);
134     return -1;
135 }
136
137 void __exit etx_driver_exit(void)
138 {
139     completion_flag = 2;
140     if(!completion_done (&data_read_done)) {
141         complete (&data_read_done);
142     }
143     device_destroy(dev_class,dev);
144     class_destroy(dev_class);
145     cdev_del(&etx_cdev);
146     unregister_chrdev_region(dev, 1);
147     printk(KERN_INFO "Device Driver Remove...Done!!!\n");
148 }
149
150 module_init(etx_driver_init);
151 module_exit(etx_driver_exit);
152
153 MODULE_LICENSE("GPL");
154 MODULE_AUTHOR("EmbeTronicX <embedtronicx@gmail.com or admin@embedtronicx.com>");
155 MODULE_DESCRIPTION("A simple device driver - Completion (Static Method)");
156 MODULE_VERSION("1.23");
```

Completion created by dynamic method

```

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
11 #include <linux/kthread.h>
12 #include <linux/completion.h>      // Required for the completion
13
14
15 uint32_t read_count = 0;
16 static struct task_struct *wait_thread;
17
18 struct completion data_read_done;
19
20 dev_t dev = 0;
21 static struct class *dev_class;
22 static struct cdev etx_cdev;
23 int completion_flag = 0;
24
25 static int __init etx_driver_init(void);
26 static void __exit etx_driver_exit(void);
27
28 /***** Driver Functions *****/
29 static int etx_open(struct inode *inode, struct file *file);
30 static int etx_release(struct inode *inode, struct file *file);
31 static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t * off);
32 static ssize_t etx_write(struct file *filp, const char *buf, size_t len, loff_t * off)
33
34 static struct file_operations fops =
35 {
36     .owner        = THIS_MODULE,
37     .read         = etx_read,
38     .write        = etx_write,
39     .open          = etx_open,
40     .release      = etx_release,
41 };
42
43 static int wait_function(void *unused)
44 {
45
46     while(1) {
47         printk(KERN_INFO "Waiting For Event...\n");
48         wait_for_completion (&data_read_done);
49         if(completion_flag == 2) {
50             printk(KERN_INFO "Event Came From Exit Function\n");
51             return 0;
52         }
53         printk(KERN_INFO "Event Came From Read Function - %d\n", ++read_count)
54         completion_flag = 0;
55     }
56     do_exit(0);
57     return 0;
58 }
59

```

```

60 static int etx_open(struct inode *inode, struct file *file)
61 {
62     printk(KERN_INFO "Device File Opened...!!!\n");
63     return 0;
64 }
65
66 static int etx_release(struct inode *inode, struct file *file)
67 {
68     printk(KERN_INFO "Device File Closed...!!!\n");
69     return 0;
70 }
71
72 static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t *off)
73 {
74     printk(KERN_INFO "Read Function\n");
75     completion_flag = 1;
76     if(!completion_done (&data_read_done)) {
77         complete (&data_read_done);
78     }
79     return 0;
80 }
81 static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, loff_t
82 {
83     printk(KERN_INFO "Write function\n");
84     return 0;
85 }
86
87 static int __init etx_driver_init(void)
88 {
89     /*Allocating Major number*/
90     if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
91         printk(KERN_INFO "Cannot allocate major number\n");
92         return -1;
93     }
94     printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
95
96     /*Creating cdev structure*/
97     cdev_init(&etx_cdev,&fops);
98     etx_cdev.owner = THIS_MODULE;
99     etx_cdev.ops = &fops;
100
101    /*Adding character device to the system*/
102    if((cdev_add(&etx_cdev,dev,1)) < 0){
103        printk(KERN_INFO "Cannot add the device to the system\n");
104        goto r_class;
105    }
106
107    /*Creating struct class*/
108    if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
109        printk(KERN_INFO "Cannot create the struct class\n");
110        goto r_class;
111    }
112
113    /*Creating device*/
114    if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
115        printk(KERN_INFO "Cannot create the Device 1\n");
116        goto r_device;
117    }
118
119    //Create the kernel thread with name 'mythread'
120    wait_thread = kthread_create(wait_function, NULL, "WaitThread");
121    if (wait_thread) {
122        printk("Thread Created successfully\n");

```

```

123         wake_up_process(wait_thread);
124     } else
125         printk(KERN_INFO "Thread creation failed\n");
126
127     //Initializing Completion
128     init_completion(&data_read_done);
129
130     printk(KERN_INFO "Device Driver Insert...Done!!!\n");
131     return 0;
132
133 r_device:
134     class_destroy(dev_class);
135 r_class:
136     unregister_chrdev_region(dev,1);
137     return -1;
138 }
139
140 void __exit etx_driver_exit(void)
141 {
142     completion_flag = 2;
143     if(!completion_done (&data_read_done)) {
144         complete (&data_read_done);
145     }
146     device_destroy(dev_class,dev);
147     class_destroy(dev_class);
148     cdev_del(&etx_cdev);
149     unregister_chrdev_region(dev, 1);
150     printk(KERN_INFO "Device Driver Remove...Done!!!\n");
151 }
152
153 module_init(etx_driver_init);
154 module_exit(etx_driver_exit);
155
156 MODULE_LICENSE("GPL");
157 MODULE_AUTHOR("EmbeTronicX <embedtronicx@gmail.com or admin@embedtronicx.com>");
158 MODULE_DESCRIPTION("A simple device driver - Completion (Dynamic Method)");
159 MODULE_VERSION("1.24");

```

MakeFile

```

1 obj-m += driver.o
2 KDIR = /lib/modules/$(shell uname -r)/build
3 all:
4     make -C $(KDIR) M=$(shell pwd) modules
5 clean:
6     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***
- Then Check the Dmesg

*Major = 246 Minor = 0
Thread Created successfully*

*Device Driver Insert...Done!!!
Waiting For Event...*

- So that thread is waiting for the event. Now we will send the event by reading the driver using `sudo cat /dev/etx_device`
- Now check the dmesg

*Device File Opened...!!!
Read Function
Event Came From Read Function - 1
Waiting For Event...
Device File Closed...!!!*

- We send the complete from read function, So it will print the read count and then again it will sleep. Now send the event from exit function by `sudo rmmod driver`

*Event Came From Exit Function
Device Driver Remove...Done!!!*

- Now the condition was 2. So it will return from the thread and remove the driver.

Share this:

[Share on Tumblr](#) [Tweet](#)



[WhatsApp](#)

[Pocket](#)



[pinit_fg_en_rect_gray_20](#) Linux Device Driver Tutorial Part 28 – Completion in Linux Device Driver

Like this:

Loading...



[report this ad](#)

 ezoic[report this ad](#)[Sidebar ▾](#)

📁 Device Drivers

EXPORT_SYMBOL

**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 wanted 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 use some advance thing. So, We have to tell the kernel, that I want to share this function to other modules.

For example, take **printf()** 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

[1 EXPORT_SYMBOL in Linux Device Driver](#)

[1.1 Introduction](#)

[1.2 EXPORT_SYMBOL's role](#)

[1.3 How to use EXPORT_SYMBOL?](#)

[1.4 Limitation](#)

[2 Driver Source Code – EXPORT_SYMBOL in Linux](#)

[2.1 driver1.c](#)

[2.2 driver2.c](#)

[2.3 MakeFile](#)

[3 Compiling and Testing Driver](#)

[3.0.1 Share this:](#)

[3.0.2 Like this:](#)

[3.0.3 Related](#)

EXPORT_SYMBOL in Linux Device Driver

Introduction

In programming language, a symbol is either a variable or a function. Or more generally, we can say, a symbol is a name representing an 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 is 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 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

```
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 * off);
18 static ssize_t etx_write(struct file *filp, const char *buf, size_t len, loff_t * off)
19
20 int etx_count = 0;
21 void etx_shared_func(void)
22 {
23     printk(KERN_INFO "Shared function been called!!!\n");
24     etx_count++;
25 }
26 //EXPORT_SYMBOL_GPL(etx_shared_func);
27 EXPORT_SYMBOL(etx_shared_func);
28 EXPORT_SYMBOL(etx_count);
29
30 static struct file_operations fops =
31 {
32     .owner        = THIS_MODULE,
33     .read         = etx_read,
34     .write        = etx_write,
35     .open          = etx_open,
36     .release      = etx_release,
37 };
38
39 static int etx_open(struct inode *inode, struct file *file)
40 {
41     printk(KERN_INFO "Device File Opened...!!!\n");
42     return 0;
43 }
44
45 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 *off)
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, loff_t
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     /*Creating struct class*/
80     if((dev_class = class_create(THIS_MODULE,"etx_class1")) == NULL){
81         printk(KERN_INFO "Cannot create the struct class\n");
82         goto r_class;
83     }
84
85     /*Creating device*/
86     if((device_create(dev_class,NULL,dev,NULL,"etx_device1")) == NULL){
87         printk(KERN_INFO "Cannot create the Device 1\n");
88         goto r_device;
89     }
90     printk(KERN_INFO "Device Driver 1 Insert...Done!!!\n");
91     return 0;
92
93 r_device:
94     class_destroy(dev_class);
95 r_class:
96     unregister_chrdev_region(dev,1);
97     return -1;
98 }
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 <embedtronicx@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 * off);
18 static ssize_t etx_write(struct file *filp, const char *buf, size_t len, loff_t * off)
19
20 extern int etx_count;

```

```

21 void etx_shared_func(void); //Function declaration is by default 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 *off)
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, loff_t
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 <embedtronicx@gmail.com>");
110 MODULE_DESCRIPTION("EXPORT_SYMBOL Driver - 2");
111 MODULE_VERSION("1.26");

```

MakeFile

```

1 obj-m += driver1.o
2 obj-m += driver2.o
3
4 KDIR = /lib/modules/$(shell uname -r)/build
5
6 all:
7     make -C $(KDIR) M=$(shell pwd) modules
8
9 clean:
10    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.

<i>0x1db7034a</i>	<i>etx_shared_func</i>	<i>/home/embedtronicx/driver</i>
<i>/driver1</i>	<i>EXPORT_SYMBOL</i>	
<i>0x6dcb135c</i>	<i>etx_count</i>	<i>/home/embedtronicx/driver</i>
<i>/driver1</i>	<i>EXPORT_SYMBOL</i>	

- 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`

Share this:[Share on Tumblr](#) [Tweet](#)[WhatsApp](#)[Pocket](#)

[pinit_fg_en_rect_gray_20](#) Linux Device Driver Tutorial Part 29 –
EXPORT_SYMBOL in Linux Device Driver

Like this:

Loading...



[report this ad](#)

 ezoic[report this ad](#)[Sidebar ▾](#)

📁 Device Drivers

atomic variable

Linux Device Driver Tutorial Part 30 - Atomic variable 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 30 – atomic variables (atomic operations) in Linux Device Driver.

Post Contents

- [1 atomic variables in Linux Device Driver](#)
- [2 Prerequisites](#)
- [3 Introduction](#)
- [4 atomic variables](#)
- [5 Types of atomic variables](#)
 - [5.1 Atomic Integer Operations](#)
 - [5.1.1 Creating atomic variables](#)
 - [5.1.2 Reading atomic variables](#)
 - [5.1.2.1 atomic_read](#)
 - [5.1.3 Other operations on atomic variables](#)
 - [5.1.3.1 atomic_set](#)
 - [5.1.3.2 atomic_add](#)
 - [5.1.3.3 atomic_sub](#)
 - [5.1.3.4 atomic_inc](#)
 - [5.1.3.5 atomic_dec](#)
 - [5.1.3.6 atomic_sub_and_test](#)
 - [5.1.3.7 atomic_dec_and_test](#)
 - [5.1.3.8 atomic_inc_and_test](#)
 - [5.1.3.9 atomic_add_negative](#)
 - [5.1.3.10 atomic_add_return](#)
 - [5.1.3.11 atomic_add_unless](#)

- 5.2 Atomic Bitwise Operations
- 6 Example Programming
 - 6.1 Driver Source Code
 - 6.2 MakeFile
 - 6.3 Share this:
 - 6.4 Like this:
 - 6.5 Related

atomic variables in Linux Device Driver Prerequisites

In the below mentioned posts, we are using spinlock and mutex for synchronization. I would recommend you to explore that by using below link.

- Mutex Tutorial in Linux Device Driver
- Spinlock in Linux Device Driver - Part 1
- Spinlock in Linux Device Driver - Part 2

Introduction

Before looking into atomic variables, we will see one example.

I have one integer or long variable named **etx_global_variable** which is shared between two threads. Two threads are just incrementing the variable like below.

Thread 1:

```
1 etx_global_variable++; //Accessing the variable
```

Thread 2:

```
1 etx_global_variable++; //Accessing the variable
```

Now we will see how it is incrementing internally in each instruction when both threads are running concurrently. Assume the initial value of **etx_global_variable** is 0.

Thread 1	Thread 2
get the value of etx_global_variable from memory (0)	get the value of etx_global_variable from memory (0)
increment etx_global_variable (0 -> 1)	—
—	increment etx_global_variable (0 -> 1)
write back the etx_global_variable value to memory (1)	—
—	write back the etx_global_variable value to memory (1)

Now the value of **etx_global_variable** is 1 after the two threads are processed. Are we expecting the same value which is 1? Nope. We are expecting the value of **etx_global_variable** should be 2. Its not running as expected because global variable is sharing between two concurrent threads. So we need to implement synchronization, because both the threads are accessing (writing/reading) the variable. We can implement synchronization like below using locks.

Thread 1:

```
1 lock(); //spinlock or mutex
```

```
2 etx_global_variable++; //Accessing the variable  
3 unlock();
```

Thread 2:

```
1 lock(); //spinlock or mutex  
2 etx_global_variable++; //Accessing the variable  
3 unlock();
```

After this synchronization, we will see how it is incrementing internally when both threads are running concurrently. Assume the initial value of **etx_global_variable** is 0.

Thread 1	Thread 2
lock()	—
get the value of etx_global_variable from memory (0)	lock() (it will stuck here because the lock is already taken by thread 1)
increment etx_global_variable (0 -> 1)	—

write back the etx_global_variable value to memory (1)	—
unlock()	—
—	get the value of etx_global_variable from memory (1)
—	increment etx_global_variable (1 -> 2)
—	write back the etx_global_variable value to memory (2)
—	unlock()

This will be the one possibility to run. Another possibility is mentioned below.

Thread 1	Thread 2
—	lock()
lock() (it will stuck here because the lock is already taken by thread 2)	get the value of etx_global_variable from memory (0)
—	increment etx_global_variable (0 -> 1)
—	write back the etx_global_variable value to memory (1)
—	unlock()
get the value of etx_global_variable from memory (1)	—
increment etx_global_variable (1 -> 2)	—

write back the etx_global_variable value to memory (2)	—
unlock()	—

Great. That's all. Now we are getting 2 in the two methods mentioned above. But have anyone thought anytime that, why these things are required for single variable? Why don't we have alternate method for single variable? Yes, obviously we have alternate mechanism for integer and long variables. That is **atomic operation**. If you use mutex/spinlock for just single variable, it will add overhead. In this tutorial we gonna see that atomic variable, atomic operation and its usage.

atomic variables

The read, write and arithmetic operations on the atomic variables will be done in one instruction without interrupt.

So again we will take the same example mentioned above to explain the atomic variable operations. When we use atomic method, that will work like below.

Thread 1	Thread 2
get, increment and store etx_global_variable (0 -> 1)	—
—	get, increment and store etx_global_variable (1 -> 2)

and another possibility will be,

Thread 1	Thread 2
—	get, increment and store etx_global_variable (0 -> 1)
get, increment and store etx_global_variable (1 -> 2)	—

So extra locking mechanism is not required when we are using atomic variables, since operation is happening in one machine instruction.

An **atomic_t** holds an **int** value and **atomic64_t** holds the **long** value on all supported architectures.

In Linux Kernel Version 2.6, atomic variable has defined like below.

```
1 typedef struct {
2 volatile int counter;
3 } atomic_t;
4
5 #ifdef CONFIG_64BIT
6 typedef struct {
7 volatile long counter;
8 } atomic64_t;
9#endif
```

Then later, they have removed **volatile** and defined like below.

```
1 typedef struct {
2 int counter;
3 } atomic_t;
4
5 #ifdef CONFIG_64BIT
6 typedef struct {
7 long counter;
8 } atomic64_t;
9#endif
```

You can read [here](#) why they have removed volatile.

Types of atomic variables

Two different atomic variables are there.

- Atomic variables who operates on Integers
- Atomic variables who operates on Individual Bits

Atomic Integer Operations

When we are doing atomic operations, that variable should be created using **atomic_t** or **atomic64_t**. So we have separate special functions for reading, writing and arithmetic operations, and those are explained below.

The declarations are needed to use the atomic integer operations are in

<asm/atomic.h>. Some architectures provide additional methods that are unique to that architecture, but all architectures provide at least a minimum set of operations that are used throughout the kernel. When you write kernel code, you can ensure that these operations are correctly implemented on all architectures.

Creating atomic variables

```
1 atomic_t etx_global_variable; /* define etx_global_variable */
2
3 or
4
5 atomic_t etx_global_variable = ATOMIC_INIT(0); /* define etx_global_variable and initial
```

Reading atomic variables

atomic_read

This function atomically reads the value given atomic variable.

```
int atomic_read(atomic_t *v);
```

where,

v – pointer of type atomic_t

Return : It returns the integer value.

Other operations on atomic variables

atomic_set

This function atomically sets the value to atomic variable.

```
void atomic_set(atomic_t *v, int i);
```

where,

v – pointer of type atomic_t

i – the value to be set to v

atomic_add

This function atomically adds the value to atomic variable.

```
void atomic_add(int i, atomic_t *v);
```

where,

i – the value to be added to v

v – pointer of type atomic_t

atomic_sub

This function atomically subtract the value from atomic variable.

```
void atomic_sub(int i, atomic_t *v);
```

where,

i – the value to be subtract from v

v – pointer of type atomic_t

atomic_inc

This function atomically increments the value of the atomic variable by 1.

```
void atomic_inc (atomic_t *v);
```

where,

v – pointer of type atomic_t

atomic_dec

This function atomically decrements the value of the atomic variable by 1.

```
void atomic_dec (atomic_t *v);
```

where,

v – pointer of type atomic_t

atomic_sub_and_test

This function atomically subtract the value from atomic variable and test the result is zero or not.

```
void atomic_sub_and_test(int i, atomic_t *v);
```

where,

i – the value to be subtract from v

v – pointer of type atomic_t

Return : It returns true if the result is zero, or false for all other cases.

atomic_dec_and_test

This function atomically decrements the value of the atomic variable by 1 and test the result is zero or not.

```
void atomic_dec_and_test(atomic_t *v);
```

where,

v – pointer of type atomic_t

Return : It returns true if the result is zero, or false for all other cases.

atomic_inc_and_test

This function atomically increments the value of the atomic variable by 1

and test the result is zero or not.

```
void atomic_inc_and_test(atomic_t *v);
```

where,

v - pointer of type atomic_t

Return : It returns true if the result is zero, or false for all other cases.

atomic_add_negative

This function atomically adds the value to atomic variable and test the result is negative or not.

```
void atomic_add_negative(int i, atomic_t *v);
```

where,

i - the value to be added to v

v - pointer of type atomic_t

Return : It returns true if the result is negative, or false for all other cases.

atomic_add_return

This function atomically adds the value to atomic variable and return the value.

```
void atomic_add_return(int i, atomic_t *v);
```

where,

i - the value to be added to v

v - pointer of type atomic_t

Return : It returns true if the result the value (i + v).

Like this other functions also there. Those are,

Function	Description

<code>int atomic_sub_return(int i, atomic_t *v)</code>	Atomically subtract i from v and return the result
<code>int atomic_inc_return(int i, atomic_t *v)</code>	Atomically increments v by one and return the result
<code>int atomic_dec_return(int i, atomic_t *v)</code>	Atomically decrements v by one and return the result

atomic_add_unless

This function atomically adds the value to atomic variable unless the number is a given value.

```
atomic_add_unless (atomic_t *v, int a, int u);
```

where,

v – pointer of type atomic_t

a – the amount to add to v...

u – ...unless v is equal to u.

Return : It returns non-zero if v was not u, and zero otherwise.

There is 64 bit version also available. Unlike `atomic_t`, that will be operates on 64 bits. This 64 bit version also have similar function like above, the only change is we have to use 64.

Example

```
1 atomic64_t etx_global_variable = ATOMIC64_INIT(0);
2 long atomic64_read(atomic64_t *v);
3 void atomic64_set(atomic64_t *v, int i);
4 void atomic64_add(int i, atomic64_t *v);
5 void atomic64_sub(int i, atomic64_t *v);
6 void atomic64_inc(atomic64_t *v);
7 void atomic64_dec(atomic64_t *v);
8 int atomic64_sub_and_test(int i, atomic64_t *v);
9 int atomic64_add_negative(int i, atomic64_t *v);
10 long atomic64_add_return(int i, atomic64_t *v);
11 long atomic64_sub_return(int i, atomic64_t *v);
12 long atomic64_inc_return(int i, atomic64_t *v);
13 long atomic64_dec_return(int i, atomic64_t *v);
14 int atomic64_dec_and_test(atomic64_t *v);
```

```
15 int atomic64_inc_and_test(atomic64_t *v);
```

But all the operations are same as **atomic_t**.

Atomic Bitwise Operations

Atomic_t is good when we are working on integer arithmetic. But when it is comes to bitwise atomic operation, it doesn't work well. So kernel offers separate functions to achieve that. Atomic bit operations are very fast. The functions are architecture dependent and are declared in **<asm/bitops.h>**.

These bitwise functions operates on generic pointer. So **atomic_t** / **atomic64_t** is not required. So we can work with a pointer to whatever data we want.

The below functions are available for atomic bit operations.

Function	Description
void set_bit(int nr, void *addr)	Atomically set the nr -th bit starting from addr
void clear_bit(int nr, void *addr)	Atomically clear the nr -th bit starting from addr
void change_bit(int nr, void *addr)	Atomically flip the value of the nr -th bit starting from addr
int test_and_set_bit(int nr, void *addr)	Atomically set the nr -th bit starting from addr and return the previous value
int test_and_clear_bit(int nr, void *addr)	Atomically clear the nr -th bit starting from addr and return the previous value
int test_and_change_bit(int nr, void *addr)	Atomically flip the nr -th bit starting from addr and return the previous value

<code>int test_bit(int nr, void *addr)</code>	Atomically return the value of the nr -th bit starting from addr
<code>int find_first_zero_bit(unsigned long *addr, unsigned int size)</code>	Atomically returns the bit-number of the first zero bit, not the number of the byte containing a bit
<code>int find_first_bit(unsigned long *addr, unsigned int size)</code>	Atomically returns the bit-number of the first set bit, not the number of the byte containing a bit

And also non-atomic bit operations also available. What is the use of that when we have atomic bit operations? When we have code which is already locked by **mutex/spinlock** then we can go for this non-atomic version. This might be faster in that case. The below functions are available for non-atomic bit operations.

Function	Description
<code>void _set_bit(int nr, void *addr)</code>	Non-atomically set the nr -th bit starting from addr
<code>void _clear_bit(int nr, void *addr)</code>	Non-atomically clear the nr -th bit starting from addr
<code>void _change_bit(int nr, void *addr)</code>	Non-atomically flip the value of the nr -th bit starting from addr
<code>int _test_and_set_bit(int nr, void *addr)</code>	Non-atomically set the nr -th bit starting from addr and return the previous value
<code>int _test_and_clear_bit(int nr, void *addr)</code>	Non-atomically clear the nr -th bit starting from addr and return the previous value
<code>int _test_and_change_bit(int nr, void *addr)</code>	Non-atomically flip the nr -th bit starting from addr and return the previous value
<code>int _test_bit(int nr, void *addr)</code>	Non-atomically return the value of the nr -th bit starting from addr

Example Programming

In this program, we have two threads called **thread_function1** and **thread_function2**. Both will be accessing the atomic variables.

Driver Source Code

```

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/kthread.h>        //kernel threads
11 #include <linux/sched.h>          //task_struct
12 #include <linux/delay.h>
13
14 atomic_t etx_global_variable = ATOMIC_INIT(0);      //Atomic integer variable
15 unsigned int etc_bit_check = 0;
16
17 dev_t dev = 0;
18 static struct class *dev_class;
19 static struct cdev etx_cdev;
20
21 static int __init etx_driver_init(void);
22 static void __exit etx_driver_exit(void);
23
24 static struct task_struct *etx_thread1;
25 static struct task_struct *etx_thread2;
26
27 /***** Driver Functions *****/
28 static int etx_open(struct inode *inode, struct file *file);
29 static int etx_release(struct inode *inode, struct file *file);
30 static ssize_t etx_read(struct file *filp,
31                        char __user *buf, size_t len, loff_t * off);
32 static ssize_t etx_write(struct file *filp,
33                        const char *buf, size_t len, loff_t * off);
34 /*****
35
36 int thread_function1(void *pv);
37 int thread_function2(void *pv);
38
39 int thread_function1(void *pv)
40 {
41     unsigned int prev_value = 0;
42
43     while(!kthread_should_stop()) {
44         atomic_inc(&etx_global_variable);
45         prev_value = test_and_change_bit(1, (void*)&etc_bit_check);
46         printk(KERN_INFO "Function1 [value : %u] [bit:%u]\n", atomic_read(&etx_global_
47         msleep(1000));
48     }
49     return 0;
50 }
51
52 int thread_function2(void *pv)

```

```

53 {
54     unsigned int prev_value = 0;
55     while(!kthread_should_stop()) {
56         atomic_inc(&etx_global_variable);
57         prev_value = test_and_change_bit(1,(void*) &etc_bit_check);
58         printk(KERN_INFO "Function2 [value : %u] [bit:%u]\n", atomic_read(&etx_global_
59         msleep(1000);
60     }
61     return 0;
62 }
63
64 static struct file_operations fops =
65 {
66     .owner        = THIS_MODULE,
67     .read         = etx_read,
68     .write        = etx_write,
69     .open          = etx_open,
70     .release      = etx_release,
71 };
72
73 static int etx_open(struct inode *inode, struct file *file)
74 {
75     printk(KERN_INFO "Device File Opened...!!!\n");
76     return 0;
77 }
78
79 static int etx_release(struct inode *inode, struct file *file)
80 {
81     printk(KERN_INFO "Device File Closed...!!!\n");
82     return 0;
83 }
84
85 static ssize_t etx_read(struct file *filp,
86                         char __user *buf, size_t len, loff_t *off)
87 {
88     printk(KERN_INFO "Read function\n");
89
90     return 0;
91 }
92 static ssize_t etx_write(struct file *filp,
93                         const char __user *buf, size_t len, loff_t *off)
94 {
95     printk(KERN_INFO "Write Function\n");
96     return len;
97 }
98
99 static int __init etx_driver_init(void)
100 {
101     /*Allocating Major number*/
102     if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
103         printk(KERN_INFO "Cannot allocate major number\n");
104         return -1;
105     }
106     printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
107
108     /*Creating cdev structure*/
109     cdev_init(&etx_cdev,&fops);
110
111     /*Adding character device to the system*/
112     if((cdev_add(&etx_cdev,dev,1)) < 0){
113         printk(KERN_INFO "Cannot add the device to the system\n");
114         goto r_class;
115     }

```

```

116
117     /*Creating struct class*/
118     if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
119         printk(KERN_INFO "Cannot create the struct class\n");
120         goto r_class;
121     }
122
123     /*Creating device*/
124     if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
125         printk(KERN_INFO "Cannot create the Device \n");
126         goto r_device;
127     }
128
129
130     /* Creating Thread 1 */
131     etx_thread1 = kthread_run(thread_function1,NULL,"eTx Thread1");
132     if(etx_thread1) {
133         printk(KERN_ERR "Kthread1 Created Successfully...\n");
134     } else {
135         printk(KERN_ERR "Cannot create kthread1\n");
136         goto r_device;
137     }
138
139     /* Creating Thread 2 */
140     etx_thread2 = kthread_run(thread_function2,NULL,"eTx Thread2");
141     if(etx_thread2) {
142         printk(KERN_ERR "Kthread2 Created Successfully...\n");
143     } else {
144         printk(KERN_ERR "Cannot create kthread2\n");
145         goto r_device;
146     }
147
148     printk(KERN_INFO "Device Driver Insert...Done!!!\n");
149     return 0;
150
151
152 r_device:
153     class_destroy(dev_class);
154 r_class:
155     unregister_chrdev_region(dev,1);
156     cdev_del(&etx_cdev);
157     return -1;
158 }
159
160 void __exit etx_driver_exit(void)
161 {
162     kthread_stop(etx_thread1);
163     kthread_stop(etx_thread2);
164     device_destroy(dev_class,dev);
165     class_destroy(dev_class);
166     cdev_del(&etx_cdev);
167     unregister_chrdev_region(dev, 1);
168     printk(KERN_INFO "Device Driver Remove...Done!!\n");
169 }
170
171 module_init(etx_driver_init);
172 module_exit(etx_driver_exit);
173
174 MODULE_LICENSE("GPL");
175 MODULE_AUTHOR("EmbeTronicX <embedtronicx@gmail.com>");
176 MODULE_DESCRIPTION("A simple device driver - Atomic Variables");
177 MODULE_VERSION("1.27");

```

MakeFile

```
1 obj-m += driver.o
2
3 KDIR = /lib/modules/$(shell uname -r)/build
4
5 all:
6     make -C $(KDIR) M=$(shell pwd) modules
7
8 clean:
9     make -C $(KDIR) M=$(shell pwd) clean
```

Share this:[Share on Tumblr](#)[Tweet](#) [Print](#) [WhatsApp](#)[Pocket](#) [Telegram](#)

[pinit_fg_en_rect_gray_20](#) Linux Device Driver Tutorial Part 30 – Atomic variable in Linux Device Driver

Like this:

Loading...