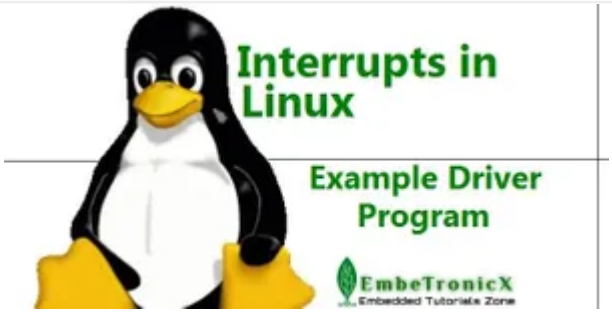
EmbeTronicX
Embedded Tutorials Zone

Sidebar

Home → Tutorials → Linux → Device Drivers → **Linux Device Driver Tutorial Part 13 – Interrupts Example Program in Linux Kernel**

Device Drivers



Linux Device Driver Tutorial Part 13 – Interrupts Example Program in Linux Kernel

13

This article is a continuation of the Series on Linux Device Driver and carries on the discussion on character drivers and their implementation. This is Part 13 of the 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 the Interrupt Example Program in Linux Kernel.

Post Contents [hide]

Interrupt Example Program in Linux Kernel

Before writing any interrupt program, you should keep these following points in 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 mute.
3. Interrupt handlers can not exchange data with the userspace.
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 **softirq** or **tasklet** or **workqueue**.
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 handler, 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 are useful for interrupts. This table explains the usage of all functions.

FUNCTION	DESCRIPTION
`request_irq ( unsigned int irq, irq_handler_t handler, unsigned long flags, const char *name, void *dev_id)`	Register an IRQ, the parameters are as follows: **irq**: IRQ number to allocate. **handler**: This is Interrupt handler function.This function will be invoked whenever the operating system receives the interrupt.The data type of return is irq_handler_t, if its return value is IRQ_HANDLED, it indicates that the processing is completed successfully, but if the return value is IRQ_NONE, the processing fails. **flags**: can be either zero or a bit mask of

FUNCTION	DESCRIPTION
	one or more of the flags defined in `linux/interrupt.h`. The most important of these flags are: IRQF_DISABLED IRQF_SAMPLE_RANDOM IRQF_SHARED IRQF_TIMER (Explained after this table) **name**: Used to identify the device name using this IRQ, for example, cat / proc / interrupts will list the IRQ number and device name. **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 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. **Return** returns zero on success and nonzero value indicates an error. *request_irq() cannot be called from interrupt context (other situations where code cannot block), because it can block.*
`void free_irq(unsigned int irq, void *dev_id)`	Release an IRQ registered by `request_irq()` with the following parameters: **irq**: IRQ number. **dev_id**: is the last parameter of request_irq. If the specified interrupt line is not shared, this function removes the handler and disables the line.

FUNCTION	DESCRIPTION
	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.
`enable_irq(unsigned int irq)`	Re-enable interrupt disabled by disable_irq or disable_irq_nosync.
`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.

13

- **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 is fed to the pool as entropy. Do not set this if your device issues interrupt 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 process interrupts 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

The 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 a pointer to the interrupt handling function. The CPU uses IDTR to point to IDT. The relation between those two can be depicted as below,

13

An interrupt can be programmatically raised using ‘`int`’ instruction. For example, the 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,

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 a 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 print the "**Shared IRQ: Interrupt Occurred**" Text.

```
/***************************************************************************//**
*  \file       driver.c
*
```

```
 4  *  \details    Interrupt Example
 5  *
 6  *  \author     EmbeTronicX
 7  *
 8  *****************************************************************************/
 9  #include <linux/kernel.h>
10  #include <linux/init.h>
11  #include <linux/module.h>
12  #include <linux/kdev_t.h>
13  #include <linux/fs.h>
14  #include <linux/cdev.h>
15  #include <linux/device.h>
16  #include<linux/slab.h>                 //kmalloc()
17  #include<linux/uaccess.h>              //copy_to/from_user()
18  #include<linux/sysfs.h>
19  #include<linux/kobject.h>
20  #include <linux/interrupt.h>
21  #include <asm/io.h>
22
23  #define IRQ_NO 11
24
25  //Interrupt handler for IRQ 11.
26  static irqreturn_t irq_handler(int irq,void *dev_id) {
27    printk(KERN_INFO "Shared IRQ: Interrupt Occurred");
28    return IRQ_HANDLED;
29  }
30
31  volatile int etx_value = 0;
32
33  dev_t dev = 0;
34  static struct class *dev_class;
35  static struct cdev etx_cdev;
36  struct kobject *kobj_ref;
37
38  static int __init etx_driver_init(void);
39  static void __exit etx_driver_exit(void);
40
41  /*************** Driver Fuctions **********************/
42  static int etx_open(struct inode *inode, struct file *file);
43  static int etx_release(struct inode *inode, struct file *file);
44  static ssize_t etx_read(struct file *filp,
45                  char __user *buf, size_t len,loff_t * off);
46  static ssize_t etx_write(struct file *filp,
47                  const char *buf, size_t len, loff_t * off);
48
49  /*************** Sysfs Fuctions **********************/
50  static ssize_t sysfs_show(struct kobject *kobj,
51                  struct kobj_attribute *attr, char *buf);
52  static ssize_t sysfs_store(struct kobject *kobj,
53                  struct kobj_attribute *attr,const char *buf, size_t count);
54
55  struct kobj_attribute etx_attr = __ATTR(etx_value, 0660, sysfs_show, sysfs_store);
56
57  static struct file_operations fops =
58  {
59          .owner          = THIS_MODULE,
60          .read           = etx_read,
61          .write          = etx_write,
62          .open           = etx_open,
63          .release        = etx_release,
64  };
65
66  static ssize_t sysfs_show(struct kobject *kobj,
```

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

```
130         }
131 
132         /*Creating device*/
133         if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
134             printk(KERN_INFO "Cannot create the Device 1\n");
135             goto r_device;
136         }
137 
138         /*Creating a directory in /sys/kernel/ */
139         kobj_ref = kobject_create_and_add("etx_sysfs",kernel_kobj);
140 
141         /*Creating sysfs file for etx_value*/
142         if(sysfs_create_file(kobj_ref,&etx_attr.attr)){
143                 printk(KERN_INFO"Cannot create sysfs file......\n");
144                 goto r_sysfs;
145         }
146         if (request_irq(IRQ_NO, irq_handler, IRQF_SHARED, "etx_device", (void *)(ir
147             printk(KERN_INFO "my_device: cannot register IRQ ");
148                     goto irq;
149         }
150         printk(KERN_INFO "Device Driver Insert...Done!!!\n");
151     return 0;
152 
153 irq:
154         free_irq(IRQ_NO,(void *)(irq_handler));
155 
156 r_sysfs:
157         kobject_put(kobj_ref);
158         sysfs_remove_file(kernel_kobj, &etx_attr.attr);
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         free_irq(IRQ_NO,(void *)(irq_handler));
171         kobject_put(kobj_ref);
172         sysfs_remove_file(kernel_kobj, &etx_attr.attr);
173         device_destroy(dev_class,dev);
174         class_destroy(dev_class);
175         cdev_del(&etx_cdev);
176         unregister_chrdev_region(dev, 1);
177         printk(KERN_INFO "Device Driver Remove...Done!!!\n");
178 }
179 
180 module_init(etx_driver_init);
181 module_exit(etx_driver_exit);
182 
183 MODULE_LICENSE("GPL");
184 MODULE_AUTHOR("EmbeTronicX <embetronicx@gmail.com>");
185 MODULE_DESCRIPTION("A simple device driver - Interrupts");
186 MODULE_VERSION("1.9");
```

13

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 the interrupt read device file (**sudo cat /dev/etx_device**)
- Now see the Dmesg (**dmesg**)

```
linux@embetronicx-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**

A problem in New Linux kernel

If you are using the newer Linux kernel, then this may not work properly. You may get something like below.

do_IRQ: 1.59 No irq handler for vector

In order to solve that, you have to change the Linux kernel source code, Compile it, then install it.

Note: The complete process will take more than an hour and you need to download the Linux kernel source also.

13

Modify and Build the Linux Kernel

Step 1: Previously, I have used an old kernel. Now I am updating the kernel to 5.4.47 with ubuntu 18.04 (Virtualbox). First, you need to

download the Linux kernel source code using the below command.

```
wget https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.4.47.tar.xz
```

Step 2: Once you have downloaded the kernel source, then extract it using the below command.

```
sudo tar -xvf ../linux-5.4.47.tar
```

Step 3: Get into the directory and copy the config.

```
1 cd linux-5.4.47/
2 cp -v /boot/config-$(uname -r) .confi
```

Step 4: Let's install the required tools to compile the source code.

```
sudo apt install build-essential kernel-package fakeroot libncurses5-dev libssl-dev ccac
```

Step 5: Now we have the source code and tools that needed to compile. Let's do our modification. Add the below line in the file **arch/x86/kernel/irq.c**(just in downloaded source code) right after all the include lines.

```
13
EXPORT_SYMBOL (vector_irq);
```

Step 6: Now build the config using the below commands.

```
make oldconfig
make menuconfig
```

Step 7: Let's start compiling the kernel using the below command.

```
sudo make
```

If you want to speed up the compilation time, just use like below.

```
sudo make -j 4
```

You have to have more patience as a compilation takes more time. The build time depends upon your system's resources such as available CPU core and the current system load. For me, it took more than 2 hours as I am building on Virtualbox.

Install the modified kernel

Step 8: Enter into the admin mode and Install the kernel modules.

```
sudo su
make modules_install
```

Step 9: Install the modified Linux kernel using the below command.

```
sudo make install
```

Step 10: Update the grub config using the below commands.

```
sudo update-initramfs -c -k 5.4.47
sudo update-grub
```

Finally, Here we are. We have installed the new kernel. In order to reflect the changes, reboot it. Then check the kernel version.

```
reboot
uname -r
```

You should see the updated kernel version if there is no issues in compilation and installation like below.

```
owl@owl-VirtualBox:~/Desktop/LDD$ uname -r
```

```
5.4.47
```

If you have any doubts, please refer here.

Driver source code for a modified kernel

We have customized the kernel. Let’s take the below source code try it.

```
/***************************************************************************//**
*  \file       driver.c
*
*  \details    Interrupt Example
*
*  \author     EmbeTronicX
*
*  \Tested with kernel 5.4.47
*
*******************************************************************************/
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kdev_t.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include<linux/slab.h>                 //kmalloc()
#include<linux/uaccess.h>              //copy_to/from_user()
#include<linux/sysfs.h>
#include<linux/kobject.h>
#include <linux/interrupt.h>
#include <asm/io.h>
#include <asm/hw_irq.h>

#define IRQ_NO 11

//Interrupt handler for IRQ 11.
static irqreturn_t irq_handler(int irq,void *dev_id) {
  printk(KERN_INFO "Shared IRQ: Interrupt Occurred");
  return IRQ_HANDLED;
}


volatile int etx_value = 0;


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

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

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

```
52                 const char *buf, size_t len, loff_t * off);
53
54  /*************** Sysfs Fuctions **********************/
55  static ssize_t sysfs_show(struct kobject *kobj,
56                 struct kobj_attribute *attr, char *buf);
57  static ssize_t sysfs_store(struct kobject *kobj,
58                 struct kobj_attribute *attr,const char *buf, size_t count);
59
60  struct kobj_attribute etx_attr = __ATTR(etx_value, 0660, sysfs_show, sysfs_store);
61
62  static struct file_operations fops =
63  {
64          .owner          = THIS_MODULE,
65          .read           = etx_read,
66          .write          = etx_write,
67          .open           = etx_open,
68          .release        = etx_release,
69  };
70
71  static ssize_t sysfs_show(struct kobject *kobj,
72                 struct kobj_attribute *attr, char *buf)
73  {
74          printk(KERN_INFO "Sysfs - Read!!!\n");
75          return sprintf(buf, "%d", etx_value);
76  }
77
78  static ssize_t sysfs_store(struct kobject *kobj,
79                 struct kobj_attribute *attr,const char *buf, size_t count)
80  {
81          printk(KERN_INFO "Sysfs - Write!!!\n");
82          sscanf(buf,"%d",&etx_value);
83          return count;
84  }
85
86  static int etx_open(struct inode *inode, struct file *file)
87  {
88          printk(KERN_INFO "Device File Opened...!!!\n");
89          return 0;
90  }
91
92  static int etx_release(struct inode *inode, struct file *file)
93  {
94          printk(KERN_INFO "Device File Closed...!!!\n");
95          return 0;
96  }
97
98  static ssize_t etx_read(struct file *filp,
99                 char __user *buf, size_t len, loff_t *off)
100 {
101         struct irq_desc *desc;
102
103         printk(KERN_INFO "Read function\n");
104         desc = irq_to_desc(11);
105         if (!desc)
106         {
107             return -EINVAL;
108         }
109         __this_cpu_write(vector_irq[59], desc);
110         asm("int $0x3B");  // Corresponding to irq 11
111         return 0;
112 }
113
114 static ssize_t etx_write(struct file *filp,
```

```
115                 const char __user *buf, size_t len, loff_t *off)
116 {
117         printk(KERN_INFO "Write Function\n");
118         return 0;
119 }
120
121 static int __init etx_driver_init(void)
122 {
123         /*Allocating Major number*/
124         if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
125                 printk(KERN_INFO "Cannot allocate major number\n");
126                 return -1;
127         }
128         printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
129
130         /*Creating cdev structure*/
131         cdev_init(&etx_cdev,&fops);
132
133         /*Adding character device to the system*/
134         if((cdev_add(&etx_cdev,dev,1)) < 0){
135             printk(KERN_INFO "Cannot add the device to the system\n");
136             goto r_class;
137         }
138
139         /*Creating struct class*/
140         if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
141             printk(KERN_INFO "Cannot create the struct class\n");
142             goto r_class;
143         }
144
145         /*Creating device*/
146         if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
147             printk(KERN_INFO "Cannot create the Device 1\n");
148             goto r_device;
149         }
150
151         /*Creating a directory in /sys/kernel/ */
152         kobj_ref = kobject_create_and_add("etx_sysfs",kernel_kobj);
153
154         /*Creating sysfs file for etx_value*/
155         if(sysfs_create_file(kobj_ref,&etx_attr.attr)){
156                 printk(KERN_INFO"Cannot create sysfs file......\n");
157                 goto r_sysfs;
158         }
159         if (request_irq(IRQ_NO, irq_handler, IRQF_SHARED, "etx_device", (void *)(ir
160             printk(KERN_INFO "my_device: cannot register IRQ ");
161                     goto irq;
162         }
163         printk(KERN_INFO "Device Driver Insert...Done!!!\n");
164     return 0;
165
166 irq:
167         free_irq(IRQ_NO,(void *)(irq_handler));
168
169 r_sysfs:
170         kobject_put(kobj_ref);
171         sysfs_remove_file(kernel_kobj, &etx_attr.attr);
172
173 r_device:
174         class_destroy(dev_class);
175 r_class:
176         unregister_chrdev_region(dev,1);
177         cdev_del(&etx_cdev);
```

```
178         return -1;
179 }
180
181 void __exit etx_driver_exit(void)
182 {
183         free_irq(IRQ_NO,(void *)(irq_handler));
184         kobject_put(kobj_ref);
185         sysfs_remove_file(kernel_kobj, &etx_attr.attr);
186         device_destroy(dev_class,dev);
187         class_destroy(dev_class);
188         cdev_del(&etx_cdev);
189         unregister_chrdev_region(dev, 1);
190         printk(KERN_INFO "Device Driver Remove...Done!!!\n");
191 }
192
193 module_init(etx_driver_init);
194 module_exit(etx_driver_exit);
195
196 MODULE_LICENSE("GPL");
197 MODULE_AUTHOR("EmbeTronicX <embetronicx@gmail.com>");
198 MODULE_DESCRIPTION("A simple device driver - Interrupts");
199 MODULE_VERSION("1.9");
```

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

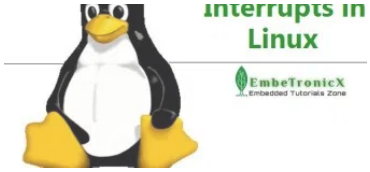
In our next tutorial, we will discuss one of the bottom half, which is workqueue.

5

Article Rating

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

In "Device Drivers"



Linux Device Driver Tutorial Part 12 - Interrupts in Linux Kernel

In "Device Drivers"

Linux Device Driver Tutorial Part 25 – Sending Signal from Linux Device Driver to User Space

In "Device Drivers"

13

Subscribe Connect with | Login

Join the discussion

{} [+]

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

13 COMMENTS Oldest

manoj

February 27, 2018 2:52 PM

Also please add a tutorial on different ways to handle situations when critial region code is being accessed by cpu (such as spinlocks, mutexes, semaphones etc implementation in kernel space)

Loading...

0 Reply

arun

August 30, 2018 1:24 AM

do_IRQ: 2.59 No irq handler for vector

13

im getting this error plz help me out.

Loading...

0 Reply

EmbeTronicx India

Reply to arun August 31, 2018 3:48 AM

Hi Arun,

Its looks like your hardware issue. Did you update the kernel recently? Please go through this below link. https://unix.stackexchange.com/questions/367503/do-irq-0-163-no-irq-handler-for-vector-irq-1

Loading...

0 Reply

EmbeTronicX

Reply to arun April 1, 2020 5:50 AM

Please refer this link.

Loading...

0 Reply

nguyendang

April 30, 2019 2:23 AM

do_IRQ: 2.59 No irq handler for vector

I have the same trouble as Arun but I cannot fix it so far, although following your instruction you replied to Arun.

Please help me, thank you!

Loading...

13 0 Reply

EmbeTronicX

Reply to nguyendang April 1, 2020 5:46 AM

Hi,

Please refer this Link to fix.

Loading...

0 Reply

Kunapareddy Jeevan

Reply to EmbeTronicX June 16, 2020 9:36 PM

I am unable to find the file in arch/x86/kernel/irq.c
I am using Virtualbox. does this giving me a problem to enable an hardware interrupt?
please help me

Loading...

0 Reply

owl Author

Reply to Kunapareddy Jeevan June 16, 2020 9:54 PM

Can you tell me, what is the issue you are facing?
Which version of the kernel you are using?

Loading...

0 Reply

kunapareddy Jeevan

Reply to owl June 17, 2020 8:32 PM

my issue is same as nguyendang and arun

do_IRQ: 2.59 No irq handler for vector

Loading...

0 Reply

13

owl Author

Reply to *kunapareddy Jeevan* June 17, 2020 8:39 PM

If you are using new kernel, then it is not available I guess. Anyway please try the above link i have provided.

Loading...

0 Reply

Kunapareddy Jeevan

Reply to owl June 17, 2020 11:11 PM

In the link, it given to add a line EXPORT_SYMBOL(vector_irq); in file arch/x86 /kernel/irq.c problem is that im unable to find that file in my kernel folder. when i add these lines in read function static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t *off) { struct irq_desc *desc; printk(KERN_INFO “Read function\n”); desc = irq_to_desc(11); if (!desc) return -EINVAL; __this_cpu_write(vector_irq[59], desc); asm(“int $0x3B”); // Corresponding to irq 11 return 0; } im getting error at vector_irq[59] because of not able to use export_symbol. i checked /proc/interrupts also.there i can see the interrupt.but when i check dmesg… Read more »

0 Reply

13

owl Author

Reply to *Kunapareddy Jeevan* June 20, 2020 9:23 AM

Hello,
We have updated the tutorial with step by step. Please see the updated tutorial. This may help you.

Loading...

0 Reply

Kalpesh

August 22, 2019 12:59 AM

What does IRQ 11 stands for(For which process is this IRQ number)

Loading...

0 Reply

13

13