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

Post Contents [hide]

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:

```
#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:

```
static long etx_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
   if (cmd == REG_CURRENT_TASK) {
      printk(KERN_INFO "REG_CURRENT_TASK\n");
      task = get_current();
```

```
        signum = SIGETX;
    }
    return 0;
}
```

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:

```
//Interrupt handler for IRQ 11.

static irqreturn_t irq_handler(int irq,void *dev_id) {

    struct siginfo info;

    printk(KERN_INFO "Shared IRQ: Interrupt Occurred");


    //Sending signal to app

    memset(&info, 0, sizeof(struct siginfo));

    info.si_signo = SIGETX;

    info.si_code = SI_QUEUE;

    info.si_int = 1;


    if (task != NULL) {

        printk(KERN_INFO "Sending signal to app\n");

        if(send_sig_info(SIGETX, &info, task) < 0) {

            printk(KERN_INFO "Unable to send signal\n");

        }
    }

    return IRQ_HANDLED;

}
```

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:

```
static int etx_release(struct inode *inode, struct file *file)

{

    struct task_struct *ref_task = get_current();

    printk(KERN_INFO "Device File Closed...!!!\n");



    //delete the task

    if(ref_task == task) {

        task = NULL;

    }

    return 0;

}
```

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

```
#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/ioctl.h>

#include <linux/interrupt.h>

#include <asm/io.h>


#define SIGETX 44


#define REG_CURRENT_TASK _IOW('a','a',int32_t*)


#define IRQ_NO 11


/* Signaling to Application */

static struct task_struct *task = NULL;
static int signum = 0;


int32_t value = 0;


dev_t dev = 0;

static struct class *dev_class;

static struct cdev etx_cdev;


static int __init etx_driver_init(void);

static void __exit etx_driver_exit(void);

static int etx_open(struct inode *inode, struct file *file);

static int etx_release(struct inode *inode, struct file *file);

static ssize_t etx_read(struct file *filp, char __user *buf, size_t len,loff_t * off);

static ssize_t etx_write(struct file *filp, const char *buf, size_t len, loff_t * off);

static long etx_ioctl(struct file *file, unsigned int cmd, unsigned long arg);


static struct file_operations fops =

{

        .owner          = THIS_MODULE,

        .read           = etx_read,

        .write          = etx_write,

        .open           = etx_open,

        .unlocked_ioctl = etx_ioctl,
        .release        = etx_release,
};

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

    //Sending signal to app
    memset(&info, 0, sizeof(struct siginfo));
    info.si_signo = SIGETX;
    info.si_code = SI_QUEUE;
    info.si_int = 1;

    if (task != NULL) {
        printk(KERN_INFO "Sending signal to app\n");
        if(send_sig_info(SIGETX, &info, task) < 0) {
            printk(KERN_INFO "Unable to send signal\n");
        }
    }

    return IRQ_HANDLED;
}
```

```
static int etx_open(struct inode *inode, struct file *file)
{
    printk(KERN_INFO "Device File Opened...!!!\n");
    return 0;
}

static int etx_release(struct inode *inode, struct file *file)
{
    struct task_struct *ref_task = get_current();
    printk(KERN_INFO "Device File Closed...!!!\n");

    //delete the task
    if(ref_task == task) {
        task = NULL;
    }
    return 0;
}

static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t *off)
{
    printk(KERN_INFO "Read Function\n");
    asm("int $0x3B");  //Triggering Interrupt. Corresponding to irq 11
    return 0;
}
static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, loff_t *off)
{
    printk(KERN_INFO "Write function\n");
    return 0;
}

static long etx_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
    if (cmd == REG_CURRENT_TASK) {
        printk(KERN_INFO "REG_CURRENT_TASK\n");
        task = get_current();
        signum = SIGETX;
    }
    return 0;
}


static int __init etx_driver_init(void)
{
    /*Allocating Major number*/
    if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
        printk(KERN_INFO "Cannot allocate major number\n");
        return -1;
    }
    printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));

    /*Creating cdev structure*/
    cdev_init(&etx_cdev,&fops);

    /*Adding character device to the system*/
    if((cdev_add(&etx_cdev,dev,1)) < 0){
      printk(KERN_INFO "Cannot add the device to the system\n");
      goto r_class;
    }

    /*Creating struct class*/
    if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
      printk(KERN_INFO "Cannot create the struct class\n");
      goto r_class;
    }

    /*Creating device*/
    if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
      printk(KERN_INFO "Cannot create the Device 1\n");
```

```
        goto r_device;
    }

    if (request_irq(IRQ_NO, irq_handler, IRQF_SHARED, "etx_device", (void *)(irq_handler))) {
        printk(KERN_INFO "my_device: cannot register IRQ ");
        goto irq;
    }

    printk(KERN_INFO "Device Driver Insert...Done!!!\n");
    return 0;
irq:
    free_irq(IRQ_NO,(void *)(irq_handler));
r_device:
    class_destroy(dev_class);
r_class:
    unregister_chrdev_region(dev,1);
    return -1;
}

void __exit etx_driver_exit(void)
{
    free_irq(IRQ_NO,(void *)(irq_handler));
    device_destroy(dev_class,dev);
    class_destroy(dev_class);

    cdev_del(&etx_cdev);

    unregister_chrdev_region(dev, 1);

    printk(KERN_INFO "Device Driver Remove...Done!!!\n");

}


module_init(etx_driver_init);

module_exit(etx_driver_exit);


MODULE_LICENSE("GPL");

MODULE_AUTHOR("EmbeTronicX <embetronicx@gmail.com>");

MODULE_DESCRIPTION("A simple device driver - Signals");

MODULE_VERSION("1.20");
```

Makefile:

```
obj-m += driver.o

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

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

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

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

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <signal.h>

#define REG_CURRENT_TASK _IOW('a','a',int32_t*)

#define SIGETX 44

static int done = 0;
int check = 0;

void ctrl_c_handler(int n, siginfo_t *info, void *unused)
{
   if (n == SIGINT) {
      printf("\nrecieved ctrl-c\n");
      done = 1;
   }
}

void sig_event_handler(int n, siginfo_t *info, void *unused)
{
   if (n == SIGETX) {
      check = info->si_int;
      printf ("Received signal from kernel : Value =  %u\n", check);
   }
}

int main()
{
   int fd;
   int32_t value, number;
   struct sigaction act;

   printf("*********************************\n");
   printf("*******WWW.EmbeTronicX.com*******\n");
   printf("*********************************\n");
```

```
    /* install ctrl-c interrupt handler to cleanup at exit */
    sigemptyset (&act.sa_mask);
    act.sa_flags = (SA_SIGINFO | SA_RESETHAND);
    act.sa_sigaction = ctrl_c_handler;
    sigaction (SIGINT, &act, NULL);

    /* install custom signal handler */
    sigemptyset(&act.sa_mask);
    act.sa_flags = (SA_SIGINFO | SA_RESTART);
    act.sa_sigaction = sig_event_handler;
    sigaction(SIGETX, &act, NULL);

    printf("Installed signal handler for SIGETX = %d\n", SIGETX);

    printf("\nOpening Driver\n");
    fd = open("/dev/etx_device", O_RDWR);
    if(fd < 0) {
            printf("Cannot open device file...\n");
            return 0;
    }

    printf("Registering application ...");
    /* register this task with kernel for signal */
    if (ioctl(fd, REG_CURRENT_TASK,(int32_t*) &number)) {
        printf("Failed\n");
        close(fd);
        exit(1);
    }
    printf("Done!!!\n");

    while(!done) {
        printf("Waiting for signal...\n");

        //blocking check
        while (!done && !check);
        check = 0;
    }

    printf("Closing Driver\n");
    close(fd);
}
```

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