> 作 者：道哥，10+年嵌入式开发老兵，专注于：C/C++、嵌入式、Linux。
>
> 关注下方公众号，回复【书籍】，获取 Linux、嵌入式领域经典书籍；回复【PDF】，获取所有原创文章(PDF 格式)。

目录

别人的经验，我们的阶梯！

大家好，我是道哥，今天我为大伙儿解说的技术知识点是：【中断程序如何发送信号给应用层】。

最近分享的几篇文章都比较基础，关于字符类设备的驱动程序，以及中断处理程序。

也许在现代的项目是用不到这样的技术，但是万丈高楼平地起。

只有明白了这些最基础的知识点之后，再去看那些进化出来的高级玩意，才会有一步一个脚印的获得感。

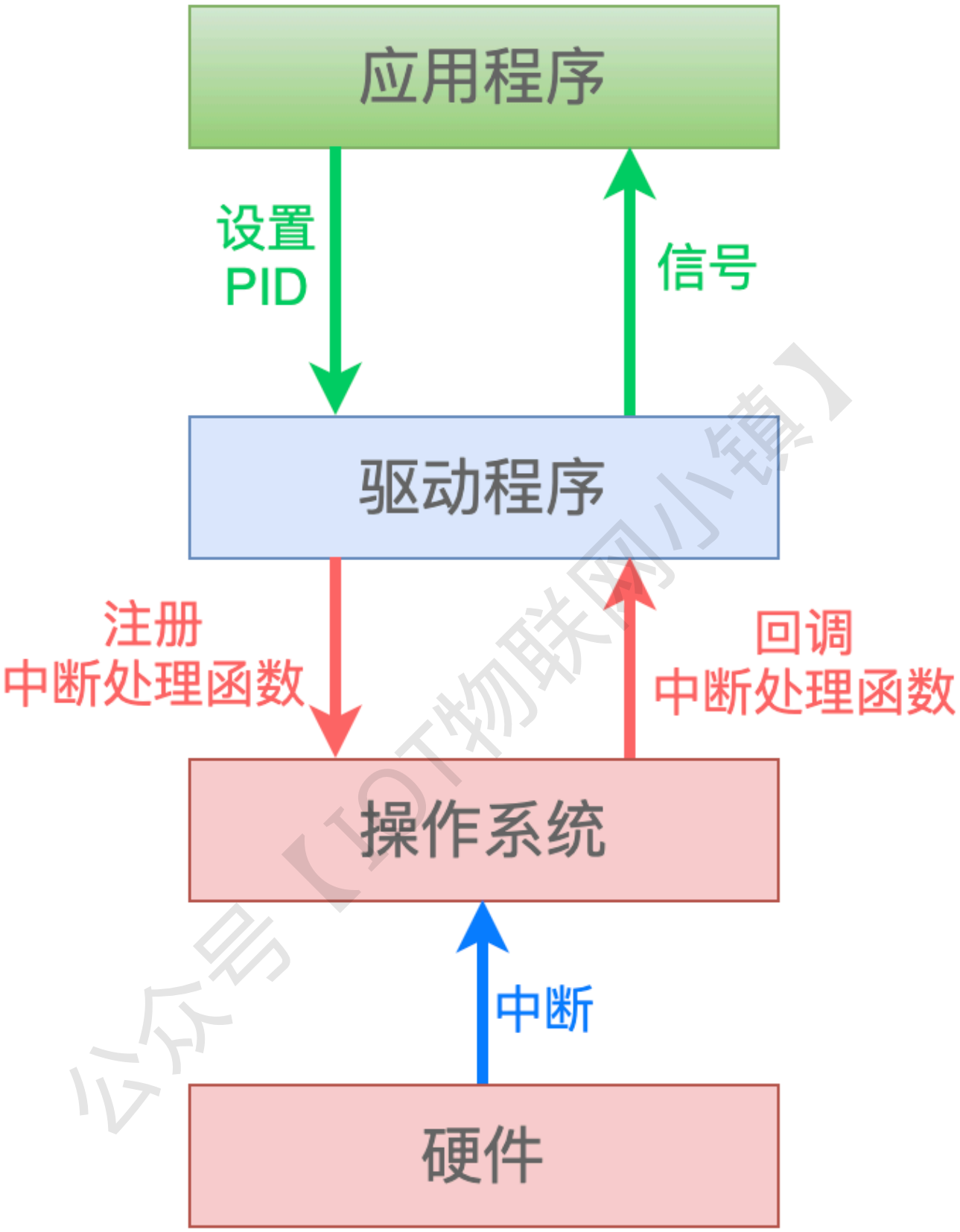
如果缺少了这些基础的环节，很多深层次的东西，学起来就有点空中楼阁的感觉。

就好比研究Linux内核，如果一上来就从Linux 4.x/5.x内核版本开始研究，可以看到很多“历史遗留”代码。

这些代码就见证着Linux一步一步的发展历史，甚至有些人还会专门去研究 Linux 0.11 版本的内核源码，因为很多基本思想都是一样的。

今天这篇文章，主要还是以代码实例为主，把之前的两个知识点结合起来：

在中断处理函数中，发送信号给应用层，以此来通知应用层处理响应的中断业务。



驱动程序

示例代码全貌

> 所有的操作都是在 ~/tmp/linux-4.15/drivers 目录下完成的。

首先创建驱动模块目录：

```
$ cd ~/tmp/linux-4.15/drivers
$ mkdir my_driver_interrupt_signal
$ touch my_driver_interrupt_signal.c
```

文件内容如下：

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/ctype.h>
#include <linux/device.h>
#include <linux/cdev.h>

#include <asm/siginfo.h>
#include <linux/pid.h>
#include <linux/uaccess.h>
#include <linux/sched/signal.h>
#include <linux/pid_namespace.h>
#include <linux/interrupt.h>

// 中断号
#define IRQ_NUM         1

// 定义驱动程序的 ID，在中断处理函数中用来判断是否需要处理
#define IRQ_DRIVER_ID   1234

// 设备名称
#define MYDEV_NAME      "mydev"

// 驱动程序数据结构
struct myirq
{
    int devid;
};

struct myirq mydev  ={ IRQ_DRIVER_ID };

#define KBD_DATA_REG        0x60
#define KBD_STATUS_REG      0x64
#define KBD_SCANCODE_MASK   0x7f
#define KBD_STATUS_MASK     0x80

// 设备类
static struct class *my_class;

// 用来保存设备
```

```
struct cdev my_cdev;

// 用来保存设备号
int mydev_major = 0;
int mydev_minor = 0;

// 用来保存向谁发送信号，应用程序通过 ioctl 把自己的进程 ID 设置进来。
static int g_pid = 0;

// 用来发送信号给应用程序
static void send_signal(int sig_no)
{
    int ret;
    struct siginfo info;
    struct task_struct *my_task = NULL;
    if (0 == g_pid)
    {
        // 说明应用程序没有设置自己的 PID
        printk("pid[%d] is not valid! \n", g_pid);
        return;
    }

    printk("send signal %d to pid %d \n", sig_no, g_pid);

    // 构造信号结构体
    memset(&info, 0, sizeof(struct siginfo));
    info.si_signo = sig_no;
    info.si_errno = 100;
    info.si_code = 200;

    // 获取自己的任务信息，使用的是 RCU 锁
    rcu_read_lock();
    my_task = pid_task(find_vpid(g_pid), PIDTYPE_PID);
    rcu_read_unlock();

    if (my_task == NULL)
    {
        printk("get pid_task failed! \n");
        return;
    }

    // 发送信号
    ret = send_sig_info(sig_no, &info, my_task);
    if (ret < 0)
    {
           printk("send signal failed! \n");
    }
}

//中断处理函数
static irqreturn_t myirq_handler(int irq, void * dev)
{
```

```
    struct myirq mydev;
    unsigned char key_code;
    mydev = *(struct myirq*)dev;

    // 检查设备 id，只有当相等的时候才需要处理
    if (IRQ_DRIVER_ID == mydev.devid)
    {
        // 读取键盘扫描码
        key_code = inb(KBD_DATA_REG);

        if (key_code == 0x01)
        {
            printk("EXC key is pressed! \n");
            send_signal(SIGUSR1);
        }
    }

    return IRQ_HANDLED;
}

// 驱动模块初始化函数
static void myirq_init(void)
{
    printk("myirq_init is called. \n");

    // 注册中断处理函数
    if(request_irq(IRQ_NUM, myirq_handler, IRQF_SHARED, MYDEV_NAME, &mydev)!=0)
    {
        printk("register irq[%d] handler failed. \n", IRQ_NUM);
        return -1;
    }

    printk("register irq[%d] handler success. \n", IRQ_NUM);
}

// 当应用程序打开设备的时候被调用
static int mydev_open(struct inode *inode, struct file *file)
{

    printk("mydev_open is called. \n");
    return 0;
}

static long mydev_ioctl(struct file* file, unsigned int cmd, unsigned long arg)
{
    void __user *pArg;
    printk("mydev_ioctl is called. cmd = %d \n", cmd);
    if (100 == cmd)
    {
        // 说明应用程序设置进程的 PID
        pArg = (void *)arg;
        if (!access_ok(VERIFY_READ, pArg, sizeof(int)))
```

```
        {
            printk("access failed! \n");
            return -EACCES;
        }

        // 把用户空间的数据复制到内核空间
        if (copy_from_user(&g_pid, pArg, sizeof(int)))
        {
            printk("copy_from_user failed! \n");
            return -EFAULT;
        }
    }

    return 0;
}

static const struct file_operations mydev_ops={
    .owner = THIS_MODULE,
    .open  = mydev_open,
    .unlocked_ioctl = mydev_ioctl
};

static int __init mydev_driver_init(void)
{
    int devno;
    dev_t num_dev;

    printk("mydev_driver_init is called. \n");

    // 注册中断处理函数
    if(request_irq(IRQ_NUM, myirq_handler, IRQF_SHARED, MYDEV_NAME, &mydev)!=0)
    {
        printk("register irq[%d] handler failed. \n", IRQ_NUM);
        return -1;
    }

    // 动态申请设备号(严谨点的话，应该检查函数返回值)
    alloc_chrdev_region(&num_dev, mydev_minor, 1, MYDEV_NAME);

    // 获取主设备号
    mydev_major = MAJOR(num_dev);
    printk("mydev_major = %d. \n", mydev_major);

    // 创建设备类
    my_class = class_create(THIS_MODULE, MYDEV_NAME);

    // 创建设备节点
    devno = MKDEV(mydev_major, mydev_minor);

    // 初始化cdev结构
    cdev_init(&my_cdev, &mydev_ops);
```

```
    // 注册字符设备
    cdev_add(&my_cdev, devno, 1);

    // 创建设备节点
    device_create(my_class, NULL, devno, NULL, MYDEV_NAME);

    return 0;
}

static void __exit mydev_driver_exit(void)
{
    printk("mydev_driver_exit is called. \n");

    // 删除设备节点
    cdev_del(&my_cdev);
    device_destroy(my_class, MKDEV(mydev_major, mydev_minor));

    // 释放设备类
    class_destroy(my_class);

    // 注销设备号
    unregister_chrdev_region(MKDEV(mydev_major, mydev_minor), 1);

    // 注销中断处理函数
    free_irq(IRQ_NUM, &mydev);
}

MODULE_LICENSE("GPL");
module_init(mydev_driver_init);
module_exit(mydev_driver_exit);
```

以上代码主要做了两件事情：

1. 注册中断号 1 的处理函数：myirq_handler();
2. 创建设备节点 /dev/mydev;

这里的中断号1，是键盘中断。

因为它是共享的中断，因此当键盘被按下的时候，操作系统就会依次调用所有的中断处理函数，当然就包括我们的驱动程序所注册的这个函数。

中断处理部分相关的几处关键代码如下：

```
//中断处理函数
static irqreturn_t myirq_handler(int irq, void * dev)
{
    ...
}

// 驱动模块初始化函数
static void myirq_init(void)
{
    ...
    request_irq(IRQ_NUM, myirq_handler, IRQF_SHARED, MYDEV_NAME, &mydev);
    ...
}
```

在中断处理函数中，目标是发送信号 SIGUSR1 到应用层，因此驱动程序需要知道应用程序的进程号(PID)。

根据之前的文章Linux驱动实践：驱动程序如何发送【信号】给应用程序？，应用程序必须主动把自己的 PID 告诉驱动模块才可以。这可以通过 write或者ioctl函数来实现，

驱动程序用来接收 PID 的相关代码是：

```
static long mydev_ioctl(struct file* file, unsigned int cmd, unsigned long arg)
{
    ...
    if (100 == cmd)
    {
        pArg = (void *)arg;
        ...
        copy_from_user(&g_pid, pArg, sizeof(int));
    }
}
```

知道了应用程序的 PID，驱动程序就可以在中断发生的时候(按下键盘ESC键)，发送信号出去了：

```
static void send_signal(int sig_no)
{
    struct siginfo info;
    ...
    send_sig_info(...);
}

static irqreturn_t myirq_handler(int irq, void * dev)
{
    ...
    send_signal(SIGUSR1);
}
```

Makefile 文件

```
ifneq ($(KERNELRELEASE),)
    obj-m := my_driver_interrupt_signal.o
else
    KERNELDIR ?= /lib/modules/$(shell uname -r)/build
    PWD := $(shell pwd)
default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
clean:
    rm -rf *.o *.ko *.mod.* modules.* Module.*
    $(MAKE) -C $(KERNEL_PATH) M=$(PWD) clean
endif
```

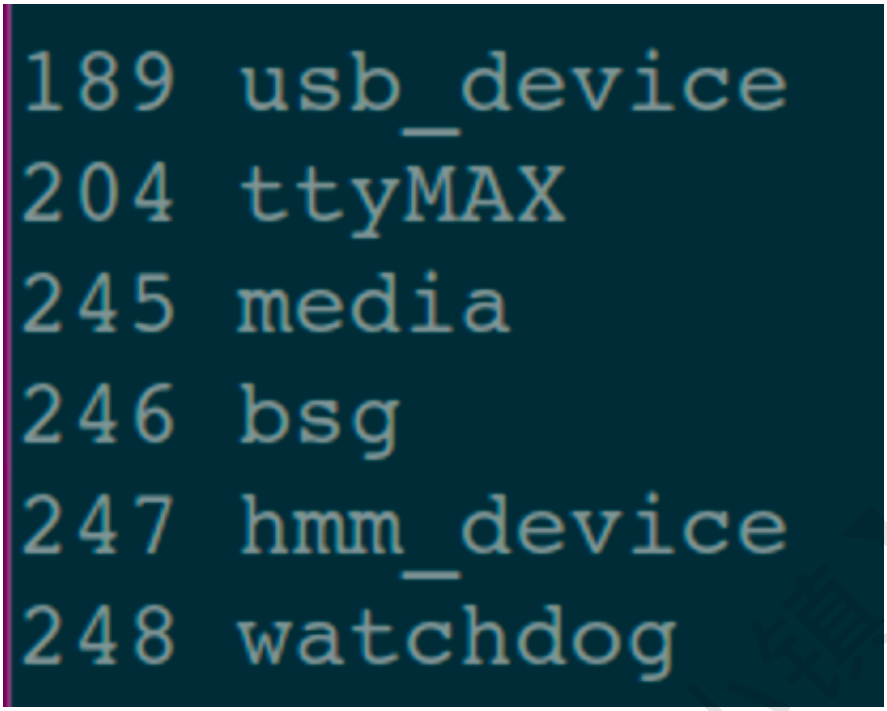
编译、测试

首先查看一下加载驱动模块之前，1号中断的所有驱动程序：

```
captain@ubuntu:my_driver_interrupt_signal$ head /proc/interrupts
           CPU0       CPU1
  0:          2          0   IO-APIC   2-edge      timer
  1:          0        548   IO-APIC   1-edge      i8042
  8:          1          0   IO-APIC   8-edge      rtc0
  9:          0       6596   IO-APIC   9-fasteoi   acpi
 12:        144      15539   IO-APIC  12-edge      i8042
 14:          0          0   IO-APIC  14-edge      ata_piix
 15:          0          0   IO-APIC  15-edge      ata_piix
 17:          0        828   IO-APIC  17-fasteoi   snd_intel8x0
 18:          0          0   IO-APIC  18-fasteoi   uhci_hcd:usb2
```

再看一下设备号：

```
$ cat /proc/devices
```



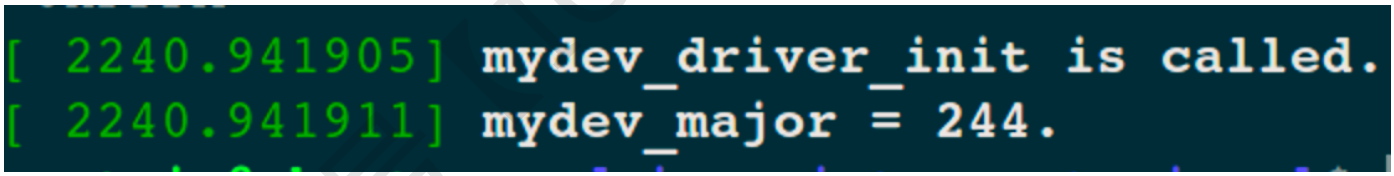
因为驱动注册在创建设备节点的时候，是动态请求系统分配的。

根据之前的几篇文章可以知道，系统一般会分配244这个主设备号给我们，此刻还不存在这个设备号。

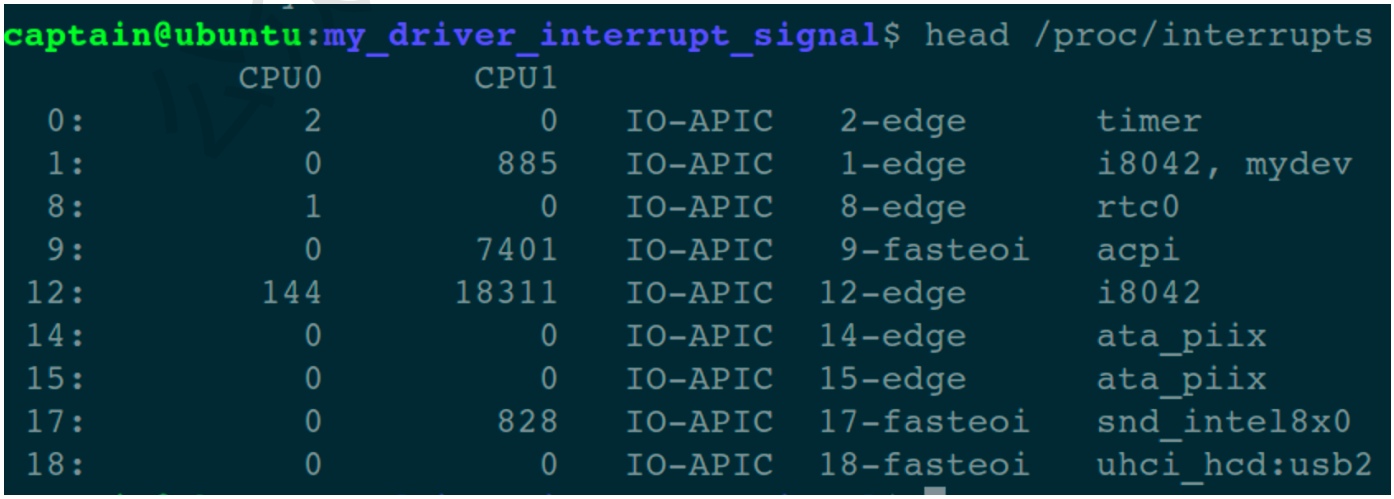
编译、加载驱动模块：

```
$ make
$ sudo insmod my_driver_interrupt_signal.ko
```

首先看一下 dmesg 的输出信息：



然后看一下中断驱动程序：



可以看到我们的驱动程序(mydev)已经登记在1号中断的最右面。

最后看一下设备节点情况：

```
captain@ubuntu:my_driver_interrupt_signal$ ll /dev/mydev
crw------- 1 root root 244, 0 Dec 19 19:17 /dev/mydev
```

驱动模块已经准备妥当，下面就是应用程序了。

应用程序

应用程序的主要功能就是两部分：

1. 通过 ioctl 函数把自己的 PID 告诉驱动程序;
2. 注册信号 SIGUSR1 的处理函数;

示例代码全貌

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <signal.h>


char *dev_name = "/dev/mydev";

// 信号处理函数
static void signal_handler(int signum, siginfo_t *info, void *context)
{
    // 打印接收到的信号值
    printf("signal_handler: signum = %d \n", signum);
    printf("signo = %d, code = %d, errno = %d \n",
            info->si_signo,
            info->si_code,
            info->si_errno);
}

int main(int argc, char *argv[])
{
    int fd, count = 0;
    int pid = getpid();

    // 打开GPIO
    if((fd = open(dev_name, O_RDWR | O_NDELAY)) < 0){
        printf("open dev failed! \n");
        return -1;
    }
```

```
    printf("open dev success! \n");

    // 注册信号处理函数
    struct sigaction sa;
    sigemptyset(&sa.sa_mask);
    sa.sa_sigaction = &signal_handler;
    sa.sa_flags = SA_SIGINFO;

    sigaction(SIGUSR1, &sa, NULL);

    // set PID
    printf("call ioctl. pid = %d \n", pid);
    ioctl(fd, 100, &pid);

    // 死循环，等待接收信号
    while (1)
        sleep(1);

    // 关闭设备
    close(fd);
}
```

在应用程序的最后，是一个 while(1) 死循环。因为只有在按下键盘上的ESC按键时，驱动程序才会发送信号上来，因此应用程序需要一直存活着。

编译、测试

新开一个中断窗口，编译、执行应用程序：

```
$ gcc my_interrupt_singal.c -o my_interrupt_singal
$ sudo ./my_interrupt_singal
open dev success!
call ioctl. pid = 12907

// 这里进入 while 循环
```

由于应用程序调用了 open 和 ioctl 这两个函数，因此，驱动程序中两个对应的函数就会被执行。

这可以通过 dmesg 命令的输出信息看出来：

```
[ 2829.458649] mydev_open is called.
[ 2829.458713] mydev_ioctl is called. cmd = 100
```

这个时候，按下键盘上的 ESC 键，此时驱动程序中打印如下信息：

```
[ 3051.891265] EXC key is pressed!
[ 3051.891269] send signal 10 to pid 12907
```

说明：驱动程序捕获到了键盘上的 ESC 键，并且发送信号给应用程序了。

在执行应用程序的终端窗口中，可以看到如下输出信息：

```
signal_handler: signum = 10
signo = 10, code = 200, errno = 100
```

说明：应用程序接收到了驱动程序发来的信号！

------ End ------

文中的测试代码和相关文档，已经放在网盘了。

在公众号【IOT物联网小镇】后台回复关键字：1220，即可获取下载地址。

谢谢！

推荐阅读

【1】《Linux 从头学》系列文章

【2】C语言指针-从底层原理到花式技巧，用图文和代码帮你讲解透彻

【3】原来gdb的底层调试原理这么简单

【4】内联汇编很可怕吗？看完这篇文章，终结它！

其他系列专辑：精选文章、应用程序设计、物联网、C语言。

星标公众号，第一时间看文章！

C/C++、物联网、嵌入式、Lua语言

Linux 操作系统、应用程序开发设计

扫码关注公众号

道哥 个人微信

喜欢请分享，满意点个赞，最后点在看。