<http://blog.csdn.net/alan445947767/article/details/40614105>

### 高级字符设备驱动之 POLL 机制

这一节里,我们在 irq\_key.c 的基础上添加 poll 机制来实现有数据的时候就去读,没数据的时候,自己规定一个时间,如果还没有数据,就表示超时时间。关于什么是 poll 机制,有兴趣的读者,可以参考《Linux 设备驱动第三版》第六章的第三小节。这些都是概念性的知识,Webee 就不抛砖引玉了。不懂的童子们,自觉点哟。

### 添加 poll 机制的按键驱动实例

添加 poll 机制的中断按键驱动源码在:

webee210\_drivers\ 5th\_advanced\_char\_drivers\1th\_poll\ poll\_key.c

与 irq\_key.c 相比,只是添加了 poll 函数。

static struct file\_operations irq\_key\_fops = {

.owner

= THIS\_MODULE,

.open

= irq\_key\_open,

.read

= irq\_key\_read,

.release = irq\_key\_close,

.poll

= irq\_key\_poll,

/\* 新添加的 \*/

};

当应用程序调用 poll 函数时,就会最终调用到 irq\_key\_poll 函数,主要做了二件事。

第一、使用 poll\_wait 函数将进程挂在 button\_waitq 队列上,而不是立即休眠。

第二、当有数据产生时(有按键动作时),将 poll 的状态记录下来,否则返回 0。

static unsigned irq\_key\_poll(struct file \*file, poll\_table \*wait)

{

unsigned int mask = 0;

/\* 该函数,只是将进程挂在 button\_waitq 队列上,而不是立即休眠 \*/

poll\_wait(file, &button\_waitq, wait);

/\* 当没有按键按下时,即不会进入按键中断处理函数,此时 ev\_press = 0

\* 当按键按下时,就会进入按键中断处理函数,此时 ev\_press 被设置为 1

\*/

if (ev\_press)

mask |= POLLIN | POLLRDNORM; /\* POLLIN 表示有数据可读 \*/

/\* 如果有按键按下时,mask |= POLLIN | POLLRDNORM,否则 mask = 0 \*/

return mask;

}

< driver / poll\_key.c >

/\*

\* Name:poll\_key.c

\* Copyright (C) 2014 Webee.JY (2483053468@qq.com)

\*

\* This program is free software; you can redistribute it and/or modify

\* it under the terms of the GNU General Public License version 2 as

\* published by the Free Software Foundation.

\*/

#include <linux/device.h>

#include <linux/interrupt.h>

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/fs.h>

#include <linux/init.h>

#include <linux/irq.h>

#include <asm/uaccess.h>

#include <asm/irq.h>

#include <asm/io.h>

#include <linux/gpio.h>

#include <linux/sched.h>

#include <linux/poll.h>

#define DEVICE\_NAME "IRQ\_KEY"

struct button\_irq\_desc {

int irq; /\* 中断号 \*/

int pin; /\* GPIO引脚 \*/

int key\_val; /\* 按键初始值 \*/

char \*name; /\* 名字 \*/

};

static struct button\_irq\_desc button\_irqs [] = {

{IRQ\_EINT(16), S5PV210\_GPH2(0), 0x01, "S1"}, /\* S1 \*/

{IRQ\_EINT(17), S5PV210\_GPH2(1), 0x02, "S2"}, /\* S2 \*/

{IRQ\_EINT(18), S5PV210\_GPH2(2), 0x03, "S3"}, /\* S3 \*/

{IRQ\_EINT(19), S5PV210\_GPH2(3), 0x04, "S4"}, /\* S4 \*/

{IRQ\_EINT(24), S5PV210\_GPH3(0), 0x05, "S5"}, /\* S5 \*/

{IRQ\_EINT(25), S5PV210\_GPH3(1), 0x06, "S6"}, /\* S6 \*/

{IRQ\_EINT(26), S5PV210\_GPH3(2), 0x07, "S7"}, /\* S7 \*/

{IRQ\_EINT(27), S5PV210\_GPH3(3), 0x08, "S8"}, /\* S8 \*/

};

static DECLARE\_WAIT\_QUEUE\_HEAD(button\_waitq);

static struct class \*irq\_key\_class;

/\* 键值: 按下时, 0x01, 0x02, 0x03, 0x04,0x05,0x06,0x07,0x08 \*/

/\* 键值: 松开时, 0x81, 0x82, 0x83, 0x84,0x85,0x86,0x87,0x88 \*/

static unsigned char key\_val;

/\* 中断事件标志, 中断服务程序将它置1，irq\_key\_read将它清0 \*/

static volatile int ev\_press = 0;

/\* 中断处理函数 \*/

static irqreturn\_t key\_interrupt(int irq, void \*dev\_id)

{

struct button\_irq\_desc \*button\_irqs = (struct button\_irq\_desc \*)dev\_id;

unsigned int pinval;

pinval = gpio\_get\_value(button\_irqs->pin);

if (pinval)

{

/\* 松开 \*/

key\_val = 0x80 | button\_irqs->key\_val;

}

else

{

/\* 按下 \*/

key\_val = button\_irqs->key\_val;

}

ev\_press = 1;

/\* 唤醒休眠的进程 \*/

wake\_up\_interruptible(&button\_waitq);

return IRQ\_RETVAL(IRQ\_HANDLED);

}

static int irq\_key\_open(struct inode \*inode, struct file \*file)

{

int i;

int err = 0;

/\* 使用request\_irq函数注册中断 \*/

for (i = 0; i < sizeof(button\_irqs)/sizeof(button\_irqs[0]); i++)

{

err = request\_irq(button\_irqs[i].irq, key\_interrupt, IRQF\_TRIGGER\_FALLING|IRQF\_TRIGGER\_RISING,

button\_irqs[i].name, (void \*)&button\_irqs[i]);

}

/\* 注册中断失败处理 \*/

if (err)

{

i--;

for (; i >= 0; i--)

{

disable\_irq(button\_irqs[i].irq);

free\_irq(button\_irqs[i].irq, (void \*)&button\_irqs[i]);

}

return -EBUSY;

}

return 0;

}

static ssize\_t irq\_key\_read(struct file \*file, char \_\_user \*buf, size\_t count, loff\_t \*ppos)

{

if (count != 1)

return -EINVAL;

/\* 如果没有按键动作, 休眠，即不会马上执行copy\_to\_user

\* ev\_press = 0时，进程会休眠，当有按键动作时，

\* 会进入按键中断处理函数，里面将ev\_press = 1，

\* 然后唤醒进程，然后马上执行copy\_to\_user，继续往下跑。

\*/

wait\_event\_interruptible(button\_waitq, ev\_press);

/\* 如果有按键动作, 返回键值给应用程序 \*/

copy\_to\_user(buf, &key\_val, 1);

ev\_press = 0;

return 1;

}

static int irq\_key\_close(struct inode \*inode, struct file \*file)

{

int i;

/\* 注销中断 \*/

for (i = 0; i < sizeof(button\_irqs)/sizeof(button\_irqs[0]); i++)

{

free\_irq(button\_irqs[i].irq, (void \*)&button\_irqs[i]);

}

return 0;

}

static unsigned irq\_key\_poll(struct file \*file, poll\_table \*wait)

{

unsigned int mask = 0;

/\* 该函数，只是将进程挂在button\_waitq队列上，而不是立即休眠 \*/

poll\_wait(file, &button\_waitq, wait);

/\* 当没有按键按下时，即不会进入按键中断处理函数，此时ev\_press = 0

\* 当按键按下时，就会进入按键中断处理函数，此时ev\_press被设置为1

\*/

if (ev\_press)

mask |= POLLIN | POLLRDNORM; /\* POLLIN表示有数据可读 \*/

/\* 如果有按键按下时，mask |= POLLIN | POLLRDNORM,否则mask = 0 \*/

return mask;

}

static struct file\_operations irq\_key\_fops = {

.owner = THIS\_MODULE,

.open = irq\_key\_open,

.read = irq\_key\_read,

.release = irq\_key\_close,

.poll = irq\_key\_poll,

};

int major;

static int \_\_init Irq\_key\_init(void)

{

/\* 注册一个字符设备 \*/

major = register\_chrdev(0, "key\_drv", &irq\_key\_fops);

/\* 成功创建类后，可在/sys/class/目录下找到key\_drv类 \*/

irq\_key\_class = class\_create(THIS\_MODULE, "key\_drv");

/\* 在key\_drv类下创建/dev/IRQ\_KEY 设备，供应用程序打开设备\*/

device\_create(irq\_key\_class, NULL, MKDEV(major, 0), NULL, DEVICE\_NAME);

return 0;

}

static void Irq\_key\_exit(void)

{

unregister\_chrdev(major, "key\_drv");

device\_destroy(irq\_key\_class, MKDEV(major, 0));

class\_destroy(irq\_key\_class);

}

module\_init(Irq\_key\_init);

module\_exit(Irq\_key\_exit);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("webee");

MODULE\_DESCRIPTION("Character drivers for irq key");

< driver / Makefile >

ifneq ($(KERNELRELEASE),)

obj-m :=poll\_key.o

else

module-objs :=poll\_key.o

KERNELDIR :=/home/gec/linux\_kernel/linux2.6.35.7/

PWD :=$(shell pwd)

default:

$(MAKE) -C $(KERNELDIR) M=$(PWD) modules

endif

clean:

$(RM) \*.ko \*.mod.c \*.mod.o \*.o \*.order \*.symvers \*.cmd

### 高级字符设备驱动之异步通知

在上一节里,我们在中断的基础上添加 poll 机制来实现有数据的时候就去读,没数据的时候,自己规定一个时间,如果还没有数据,就表示超时时间。在此以前,我们都是让应用程序主动去读,那有没有一种情况,当驱动程序有数据时,主动去告诉应用程序,告诉它,有数据了,你赶紧来读吧。答案当然是有的,这种情况在 linux 里的专业术语就叫异步通知。关于什么是异步通知机制,有兴趣的读者,可以参考《Linux 设备驱动第三版》第六章的第四小节。想要在驱动里实现异步通知的效果,有哪些要素?有六个要素:

第一、应用程序要实现有:注册信号处理函数,使用 signal 函数。

第二、谁来发?驱动来发

第三、发给谁?发给应用程序,但应用程序必须告诉驱动 PID(什么是 PID,不会忘了吧?这么快就还给 Webee 啦?)

第四、怎么发?驱动程序使用 kill\_fasync 函数来发。

第五、应该在驱动的哪里调用 kill\_fasync 函数?kill\_fasync 函数的作用是,当有数据时去通知应用程序,理所当然的应该在中断处理函数里调用。

第六、file\_operations 需要添加什么函数指针成员吗?要的,需要添加 fasync函数指针,要实现这个函数指针,幸运的是,这个函数仅仅调用了fasync\_helper 函数,而且这个函数是内核帮我们实现好了,驱动工程师不用修改,fasync\_helper 函数的作用是初始化/释放 fasync\_struct 结构体。

### 添加异步通知的按键驱动实例

添 加 异 步 通 知 机 制 的 中 断 按 键 驱 动 源 码 在 : webee210\_drivers\5th\_advanced\_char\_drivers\2th\_fasync\ fasync\_key.c  
在这一节里,我们将在上一节的基础上修改驱动,将其修改为有异步通知功能的按键驱动,目标:按下按键时,驱动主动去通知应用程序。

static struct file\_operations irq\_key\_fops = {

.owner = THIS\_MODULE,

.open

= irq\_key\_open,

.read

= irq\_key\_read,

.release = irq\_key\_close,

.poll

= irq\_key\_poll,

.fasync = irq\_key\_fasync,

/\* 新添加的 \*/

};

当应用程序调用了 fcntl(fd, F\_SETFL, Oflags | FASYNC)时,就会最终调用到 irq\_key\_fasync 函数。当有按键动作发生时,就会进入中断处理函数,在里面调用 kill\_fasync 函数告诉应用程序,有数据可读了 。

static irqreturn\_t key\_interrupt(int irq, void \*dev\_id)

{

struct button\_irq\_desc \*button\_irqs =

(struct button\_irq\_desc \*)dev\_id;

unsigned int pinval;

pinval = gpio\_get\_value(button\_irqs->pin);

if (pinval)

{

/\* 松开 \*/

key\_val = 0x80 | button\_irqs->key\_val;

}

else

{

/\* 按下 \*/

key\_val = button\_irqs->key\_val;

}

ev\_press = 1;

/\* 唤醒休眠的进程 \*/

wake\_up\_interruptible(&button\_waitq);

/\* 用 kill\_fasync 函数告诉应用程序,有数据可读了

\* button\_fasync 结构体里包含了发给谁(PID 指定)

\* SIGIO : 表示要发送的信号类型

\* POLL\_IN: 表示发送的原因(有数据可读了)

\*/

kill\_fasync(&button\_fasync, SIGIO, POLL\_IN);

return IRQ\_RETVAL(IRQ\_HANDLED);

}

< driver / fasync\_key.c >

/\*

\* Name:fasync\_key.c

\* Copyright (C) 2014 Webee.JY (2483053468@qq.com)

\*

\* This program is free software; you can redistribute it and/or modify

\* it under the terms of the GNU General Public License version 2 as

\* published by the Free Software Foundation.

\*/

#include <linux/device.h>

#include <linux/interrupt.h>

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/fs.h>

#include <linux/init.h>

#include <linux/irq.h>

#include <asm/uaccess.h>

#include <asm/irq.h>

#include <asm/io.h>

#include <linux/gpio.h>

#include <linux/sched.h>

#include <linux/poll.h>

#include <linux/fcntl.h>

#define DEVICE\_NAME "IRQ\_KEY"

struct button\_irq\_desc {

int irq; /\* 中断号 \*/

int pin; /\* GPIO引脚 \*/

int key\_val; /\* 按键初始值 \*/

char \*name; /\* 名字 \*/

};

static struct button\_irq\_desc button\_irqs [] = {

{IRQ\_EINT(16), S5PV210\_GPH2(0), 0x01, "S1"}, /\* S1 \*/

{IRQ\_EINT(17), S5PV210\_GPH2(1), 0x02, "S2"}, /\* S2 \*/

{IRQ\_EINT(18), S5PV210\_GPH2(2), 0x03, "S3"}, /\* S3 \*/

{IRQ\_EINT(19), S5PV210\_GPH2(3), 0x04, "S4"}, /\* S4 \*/

{IRQ\_EINT(24), S5PV210\_GPH3(0), 0x05, "S5"}, /\* S5 \*/

{IRQ\_EINT(25), S5PV210\_GPH3(1), 0x06, "S6"}, /\* S6 \*/

{IRQ\_EINT(26), S5PV210\_GPH3(2), 0x07, "S7"}, /\* S7 \*/

{IRQ\_EINT(27), S5PV210\_GPH3(3), 0x08, "S8"}, /\* S8 \*/

};

static DECLARE\_WAIT\_QUEUE\_HEAD(button\_waitq);

static struct class \*irq\_key\_class;

/\* 键值: 按下时, 0x01, 0x02, 0x03, 0x04,0x05,0x06,0x07,0x08 \*/

/\* 键值: 松开时, 0x81, 0x82, 0x83, 0x84,0x85,0x86,0x87,0x88 \*/

static unsigned char key\_val;

/\* 中断事件标志, 中断服务程序将它置1，irq\_key\_read将它清0 \*/

static volatile int ev\_press = 0;

static struct fasync\_struct \*button\_fasync;

/\* 中断处理函数 \*/

static irqreturn\_t key\_interrupt(int irq, void \*dev\_id)

{

struct button\_irq\_desc \*button\_irqs = (struct button\_irq\_desc \*)dev\_id;

unsigned int pinval;

pinval = gpio\_get\_value(button\_irqs->pin);

if (pinval)

{

/\* 松开 \*/

key\_val = 0x80 | button\_irqs->key\_val;

}

else

{

/\* 按下 \*/

key\_val = button\_irqs->key\_val;

}

ev\_press = 1;

/\* 唤醒休眠的进程 \*/

wake\_up\_interruptible(&button\_waitq);

/\* 用kill\_fasync函数告诉应用程序，有数据可读了

\* button\_fasync结构体里包含了发给谁(PID指定)

\* SIGIO : 表示要发送的信号类型

\* POLL\_IN: 表示发送的原因(有数据可读了)

\*/

kill\_fasync(&button\_fasync, SIGIO, POLL\_IN);

return IRQ\_RETVAL(IRQ\_HANDLED);

}

static int irq\_key\_open(struct inode \*inode, struct file \*file)

{

int i;

int err = 0;

/\* 使用request\_irq函数注册中断 \*/

for (i = 0; i < sizeof(button\_irqs)/sizeof(button\_irqs[0]); i++)

{

err = request\_irq(button\_irqs[i].irq, key\_interrupt, IRQF\_TRIGGER\_FALLING|IRQF\_TRIGGER\_RISING,

button\_irqs[i].name, (void \*)&button\_irqs[i]);

}

/\* 注册中断失败处理 \*/

if (err)

{

i--;

for (; i >= 0; i--)

{

disable\_irq(button\_irqs[i].irq);

free\_irq(button\_irqs[i].irq, (void \*)&button\_irqs[i]);

}

return -EBUSY;

}

return 0;

}

static ssize\_t irq\_key\_read(struct file \*file, char \_\_user \*buf, size\_t count, loff\_t \*ppos)

{

if (count != 1)

return -EINVAL;

/\* 如果没有按键动作, 休眠，即不会马上执行copy\_to\_user

\* ev\_press = 0时，进程会休眠，当有按键动作时，

\* 会进入按键中断处理函数，里面将ev\_press = 1，

\* 然后唤醒进程，然后马上执行copy\_to\_user，继续往下跑。

\*/

wait\_event\_interruptible(button\_waitq, ev\_press);

/\* 如果有按键动作, 返回键值给应用程序 \*/

copy\_to\_user(buf, &key\_val, 1);

ev\_press = 0;

return 1;

}

static int irq\_key\_close(struct inode \*inode, struct file \*file)

{

int i;

/\* 注销中断 \*/

for (i = 0; i < sizeof(button\_irqs)/sizeof(button\_irqs[0]); i++)

{

free\_irq(button\_irqs[i].irq, (void \*)&button\_irqs[i]);

}

return 0;

}

static unsigned irq\_key\_poll(struct file \*file, poll\_table \*wait)

{

unsigned int mask = 0;

/\* 该函数，只是将进程挂在button\_waitq队列上，而不是立即休眠 \*/

poll\_wait(file, &button\_waitq, wait);

/\* 当没有按键按下时，即不会进入按键中断处理函数，此时ev\_press = 0

\* 当按键按下时，就会进入按键中断处理函数，此时ev\_press被设置为1

\*/

if (ev\_press)

mask |= POLLIN | POLLRDNORM; /\* POLLIN表示有数据可读 \*/

/\* 如果有按键按下时，mask |= POLLIN | POLLRDNORM,否则mask = 0 \*/

return mask;

}

/\* 当应用程序调用了fcntl(fd, F\_SETFL, Oflags | FASYNC);

\* 则最终会调用驱动的fasync函数，在这里则是irq\_key\_fasync

\* irq\_key\_fasync最终又会调用到驱动的fasync\_helper函数

\* fasync\_helper函数的作用是初始化/释放fasync\_struct结构体

\*/

static int irq\_key\_fasync(int fd, struct file \*filp, int on)

{

return fasync\_helper(fd, filp, on, &button\_fasync);

}

static struct file\_operations irq\_key\_fops = {

.owner = THIS\_MODULE,

.open = irq\_key\_open,

.read = irq\_key\_read,

.release = irq\_key\_close,

.poll = irq\_key\_poll,

.fasync = irq\_key\_fasync,

};

int major;

static int \_\_init Irq\_key\_init(void)

{

/\* 注册一个字符设备 \*/

major = register\_chrdev(0, "key\_drv", &irq\_key\_fops);

/\* 成功创建类后，可在/sys/class/目录下找到key\_drv类 \*/

irq\_key\_class = class\_create(THIS\_MODULE, "key\_drv");

/\* 在key\_drv类下创建/dev/IRQ\_KEY 设备，供应用程序打开设备\*/

device\_create(irq\_key\_class, NULL, MKDEV(major, 0), NULL, DEVICE\_NAME);

return 0;

}

static void Irq\_key\_exit(void)

{

unregister\_chrdev(major, "key\_drv");

device\_destroy(irq\_key\_class, MKDEV(major, 0));

class\_destroy(irq\_key\_class);

}

module\_init(Irq\_key\_init);

module\_exit(Irq\_key\_exit);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("webee");

MODULE\_DESCRIPTION("Character drivers for irq key");

< driver / Makefile >

ifneq ($(KERNELRELEASE),)

obj-m :=fasync\_key.o

else

module-objs :=fasync\_key.o

KERNELDIR :=/home/gec/linux\_kernel/linux2.6.35.7/

PWD :=$(shell pwd)

default:

$(MAKE) -C $(KERNELDIR) M=$(PWD) modules

endif

clean:

$(RM) \*.ko \*.mod.c \*.mod.o \*.o \*.order \*.symvers \*.cmd

### 高级字符设备驱动之定时器去抖动

前面的驱动测试实验,我们说过,有 BUG。什么 BUG 呢?就是按键并不是以按下/松开成对出现的。换句话说,当你只按下/松开一次 S2 时,可能打印出多次按下 S2 和多次松开。这就是传说中的,有抖动现象。相信大家在写单片机的按键程序时,也必将会涉及一点,就去按键去抖动。按键去抖动的方法无非有二种,一种是硬件 电路去抖动,这种在要求不是特别高的情况下是不会被采用的;另一种就是延时去抖动了。而延时又一般分为二种,一种是 for 循环死等待,一种是定时延时,我就是要使用内核的定时器实现去抖动。想要使用 Linux 内核定时器,需要设置哪些要素?  
第一、设置超时时间  
第二、设置处理函数  
Linux 内核里的定时器结构是怎样的?

struct timer\_list {

struct list\_head entry;

unsigned long expires;

void (\*function)(unsigned long);

unsigned long data;

struct tvec\_base \*base;

.....

};

问题一:void (\*function)(unsigned long data)里面的参数是谁传给它的?

答:是 timer\_list.data 传给它的,如果需要向 function 传递参数时,则应该

设置 timer\_list.data,否则可以不设置。

问题二:与定时器相关的操作函数有哪些?

(1) 使用 init\_timer 函数初始化定时器

(2) 设置 timer\_list.function,并实现这个函数指针

(3) 使用 add\_timer 函数向内核注册一个定时器

(4) 使用 mod\_timer 修改定时器时间,并启动定时器

问题三:int mod\_timer(struct timer\_list \*timer, unsigned long expires)的第二个参数为超时时间,怎么设置超时时间,如果定时为 10ms?

答:一般的形式为: jiffies + (HZ /100),HZ 表示 100 个 jiffies,jiffies为 10ms,即 HZ = 100\*10ms = 1s

### 添加定时器的按键驱动实例

添 加 异 步 通 知 机 制 的 中 断 按 键 驱 动 源 码 在 : webee210\_drivers\5th\_advanced\_char\_drivers\3th\_timer\timer\_key.c

在这一节里,我们将在上一节的基础上修改驱动,将其修改为有定时器的按键驱动,以达到软件去抖动的目的。在驱动入口函数定义定时器......

/\* 定义一个定时器结构体 \*/

static struct timer\_list buttons\_timer;

int major;

static int \_\_init Irq\_key\_init(void)

{

/\* 初始化定时器 \*/

init\_timer(&buttons\_timer);

/\* 当定时时间到达时 buttons\_timer\_function 就会被调用 \*/

buttons\_timer.function = buttons\_timer\_function;

/\* 向内核注册一个定时器 \*/

add\_timer(&buttons\_timer);

/\* 注册一个字符设备 \*/

major = register\_chrdev(0, "key\_drv", &irq\_key\_fops);

/\* 成功创建类后,可在/sys/class/目录下找到 key\_drv 类 \*/

irq\_key\_class = class\_create(THIS\_MODULE, "key\_drv");

/\* 在 key\_drv 类下创建/dev/IRQ\_KEY 设备,供应用程序打开设备\*/

device\_create(irq\_key\_class, NULL,

MKDEV(major, 0), NULL, DEVICE\_NAME);

return 0;

}

在中断处理函数里,使用 mod\_timer 修改定时器时间,并启动定时器。

static irqreturn\_t key\_interrupt(int irq, void \*dev\_id)

{

irq\_pd = (struct button\_irq\_desc \*)dev\_id;

/\* 修改定时器定时时间,定时 10ms,即 10 秒后启动定时器

\* HZ 表示 100 个 jiffies,jiffies 的单位为 10ms,即 HZ = 100\*10ms = 1s

\* 这里 HZ/100 即定时 10ms

\*/

mod\_timer(&buttons\_timer, jiffies + (HZ /100));

return IRQ\_RETVAL(IRQ\_HANDLED);

}

当定时时间到达时，buttons\_timer\_function 就会被调用,这个函数的工作,其实就是以前的中断处理函数的工作,只不过现在由它来承担了。

/\* 定时器处理函数 \*/

static void buttons\_timer\_function(unsigned long data)

{

struct button\_irq\_desc \*button\_irqs = irq\_pd;

unsigned int pinval;

pinval = gpio\_get\_value(button\_irqs->pin);

if (pinval)

{

/\* 松开 \*/

key\_val = 0x80 | button\_irqs->key\_val;

}

else

{

/\* 按下 \*/

key\_val = button\_irqs->key\_val;

}

ev\_press = 1;

/\* 唤醒休眠的进程 \*/

wake\_up\_interruptible(&button\_waitq);

/\* 用 kill\_fasync 函数告诉应用程序,有数据可读了

\* button\_fasync 结构体里包含了发给谁(PID 指定)

\* SIGIO : 表示要发送的信号类型

\* POLL\_IN: 表示发送的原因(有数据可读了)

\*/

kill\_fasync(&button\_fasync, SIGIO, POLL\_IN);

}

< driver / timer\_key.c >

/\*

\* Name:timer\_key.c

\* Copyright (C) 2014 Webee.JY (2483053468@qq.com)

\*

\* This program is free software; you can redistribute it and/or modify

\* it under the terms of the GNU General Public License version 2 as

\* published by the Free Software Foundation.

\*/

#include <linux/device.h>

#include <linux/interrupt.h>

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/fs.h>

#include <linux/init.h>

#include <linux/irq.h>

#include <asm/uaccess.h>

#include <asm/irq.h>

#include <asm/io.h>

#include <linux/gpio.h>

#include <linux/sched.h>

#include <linux/poll.h>

#include <linux/fcntl.h>

#define DEVICE\_NAME "IRQ\_KEY"

struct button\_irq\_desc {

int irq; /\* 中断号 \*/

int pin; /\* GPIO引脚 \*/

int key\_val; /\* 按键初始值 \*/

char \*name; /\* 名字 \*/

};

static struct button\_irq\_desc button\_irqs [] = {

{IRQ\_EINT(16), S5PV210\_GPH2(0), 0x01, "S1"}, /\* S1 \*/

{IRQ\_EINT(17), S5PV210\_GPH2(1), 0x02, "S2"}, /\* S2 \*/

{IRQ\_EINT(18), S5PV210\_GPH2(2), 0x03, "S3"}, /\* S3 \*/

{IRQ\_EINT(19), S5PV210\_GPH2(3), 0x04, "S4"}, /\* S4 \*/

{IRQ\_EINT(24), S5PV210\_GPH3(0), 0x05, "S5"}, /\* S5 \*/

{IRQ\_EINT(25), S5PV210\_GPH3(1), 0x06, "S6"}, /\* S6 \*/

{IRQ\_EINT(26), S5PV210\_GPH3(2), 0x07, "S7"}, /\* S7 \*/

{IRQ\_EINT(27), S5PV210\_GPH3(3), 0x08, "S8"}, /\* S8 \*/

};

static struct button\_irq\_desc \*irq\_pd;

static DECLARE\_WAIT\_QUEUE\_HEAD(button\_waitq);

static struct class \*irq\_key\_class;

/\* 键值: 按下时, 0x01, 0x02, 0x03, 0x04,0x05,0x06,0x07,0x08 \*/

/\* 键值: 松开时, 0x81, 0x82, 0x83, 0x84,0x85,0x86,0x87,0x88 \*/

static unsigned char key\_val;

/\* 中断事件标志, 中断服务程序将它置1，irq\_key\_read将它清0 \*/

static volatile int ev\_press = 0;

static struct fasync\_struct \*button\_fasync;

static struct timer\_list buttons\_timer; /\* 定义一个定时器结构体 \*/

/\* 中断处理函数 \*/

static irqreturn\_t key\_interrupt(int irq, void \*dev\_id)

{

irq\_pd = (struct button\_irq\_desc \*)dev\_id;

/\* 修改定时器定时时间，定时10ms，即10秒后启动定时器

\* HZ 表示100个jiffies，jiffies的单位为10ms，即HZ = 100\*10ms = 1s

\* 这里HZ/100即定时10ms

\*/

mod\_timer(&buttons\_timer, jiffies + (HZ /100));

return IRQ\_RETVAL(IRQ\_HANDLED);

}

static int irq\_key\_open(struct inode \*inode, struct file \*file)

{

int i;

int err = 0;

/\* 使用request\_irq函数注册中断 \*/

for (i = 0; i < sizeof(button\_irqs)/sizeof(button\_irqs[0]); i++)

{

err = request\_irq(button\_irqs[i].irq, key\_interrupt, IRQF\_TRIGGER\_FALLING|IRQF\_TRIGGER\_RISING,

button\_irqs[i].name, (void \*)&button\_irqs[i]);

}

/\* 注册中断失败处理 \*/

if (err)

{

i--;

for (; i >= 0; i--)

{

disable\_irq(button\_irqs[i].irq);

free\_irq(button\_irqs[i].irq, (void \*)&button\_irqs[i]);

}

return -EBUSY;

}

return 0;

}

static ssize\_t irq\_key\_read(struct file \*file, char \_\_user \*buf, size\_t count, loff\_t \*ppos)

{

if (count != 1)

return -EINVAL;

/\* 如果没有按键动作, 休眠，即不会马上执行copy\_to\_user

\* ev\_press = 0时，进程会休眠，当有按键动作时，

\* 会进入按键中断处理函数，里面将ev\_press = 1，

\* 然后唤醒进程，然后马上执行copy\_to\_user，继续往下跑。

\*/

wait\_event\_interruptible(button\_waitq, ev\_press);

/\* 如果有按键动作, 返回键值给应用程序 \*/

copy\_to\_user(buf, &key\_val, 1);

ev\_press = 0;

return 1;

}

static int irq\_key\_close(struct inode \*inode, struct file \*file)

{

int i;

/\* 注销中断 \*/

for (i = 0; i < sizeof(button\_irqs)/sizeof(button\_irqs[0]); i++)

{

free\_irq(button\_irqs[i].irq, (void \*)&button\_irqs[i]);

}

return 0;

}

static unsigned irq\_key\_poll(struct file \*file, poll\_table \*wait)

{

unsigned int mask = 0;

/\* 该函数，只是将进程挂在button\_waitq队列上，而不是立即休眠 \*/

poll\_wait(file, &button\_waitq, wait);

/\* 当没有按键按下时，即不会进入按键中断处理函数，此时ev\_press = 0

\* 当按键按下时，就会进入按键中断处理函数，此时ev\_press被设置为1

\*/

if (ev\_press)

mask |= POLLIN | POLLRDNORM; /\* POLLIN表示有数据可读 \*/

/\* 如果有按键按下时，mask |= POLLIN | POLLRDNORM,否则mask = 0 \*/

return mask;

}

/\* 当应用程序调用了fcntl(fd, F\_SETFL, Oflags | FASYNC);

\* 则最终会调用驱动的fasync函数，在这里则是irq\_key\_fasync

\* irq\_key\_fasync最终又会调用到驱动的fasync\_helper函数

\* fasync\_helper函数的作用是初始化/释放fasync\_struct结构体

\*/

static int irq\_key\_fasync(int fd, struct file \*filp, int on)

{

return fasync\_helper(fd, filp, on, &button\_fasync);

}

static struct file\_operations irq\_key\_fops = {

.owner = THIS\_MODULE,

.open = irq\_key\_open,

.read = irq\_key\_read,

.release = irq\_key\_close,

.poll = irq\_key\_poll,

.fasync = irq\_key\_fasync,

};

/\* 定时器中断处理函数 \*/

static void buttons\_timer\_function(unsigned long data)

{

struct button\_irq\_desc \*button\_irqs = irq\_pd;

unsigned int pinval;

pinval = gpio\_get\_value(button\_irqs->pin);

if (pinval)

{

/\* 松开 \*/

key\_val = 0x80 | button\_irqs->key\_val;

}

else

{

/\* 按下 \*/

key\_val = button\_irqs->key\_val;

}

ev\_press = 1;

/\* 唤醒休眠的进程 \*/

wake\_up\_interruptible(&button\_waitq);

/\* 用kill\_fasync函数告诉应用程序，有数据可读了

\* button\_fasync结构体里包含了发给谁(PID指定)

\* SIGIO : 表示要发送的信号类型

\* POLL\_IN: 表示发送的原因(有数据可读了)

\*/

kill\_fasync(&button\_fasync, SIGIO, POLL\_IN);

}

int major;

static int \_\_init Irq\_key\_init(void)

{

/\* 初始化定时器 \*/

init\_timer(&buttons\_timer);

/\* 当定时时间到达时buttons\_timer\_function就会被调用 \*/

buttons\_timer.function = buttons\_timer\_function;

/\* 向内核注册一个定时器 \*/

add\_timer(&buttons\_timer);

/\* 注册一个字符设备 \*/

major = register\_chrdev(0, "key\_drv", &irq\_key\_fops);

/\* 成功创建类后，可在/sys/class/目录下找到key\_drv类 \*/

irq\_key\_class = class\_create(THIS\_MODULE, "key\_drv");

/\* 在key\_drv类下创建/dev/IRQ\_KEY 设备，供应用程序打开设备\*/

device\_create(irq\_key\_class, NULL, MKDEV(major, 0), NULL, DEVICE\_NAME);

return 0;

}

static void Irq\_key\_exit(void)

{

unregister\_chrdev(major, "key\_drv");

device\_destroy(irq\_key\_class, MKDEV(major, 0));

class\_destroy(irq\_key\_class);

del\_timer(&buttons\_timer);

}

module\_init(Irq\_key\_init);

module\_exit(Irq\_key\_exit);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("webee");

MODULE\_DESCRIPTION("Character drivers for irq key");

< driver / Makefile >

ifneq ($(KERNELRELEASE),)

obj-m :=timer\_key.o

else

module-objs :=timer\_key.o

KERNELDIR :=/home/gec/linux\_kernel/linux2.6.35.7/

PWD :=$(shell pwd)

default:

$(MAKE) -C $(KERNELDIR) M=$(PWD) modules

endif

clean:

$(RM) \*.ko \*.mod.c \*.mod.o \*.o \*.order \*.symvers \*.cmd