[L22]Linux 中断和阻塞/通知入门

但凡有过单片机开发经验都知道中断这个东西,在单片机上管理中断是非常复杂的,比如配置寄存器,还要看中断临界区之类,不过在Linux就简单很多了,我们只需要申请中断,然后注册中断处理函数,就这么简单,不需要管那么多谁以来谁,优先怎么处理.因为这些都在Linux系统中有一个预先设定,通常效果都会比较好,而且处理器上的中断也相对单片机少很多,很多中断上访还要共享一条线.

在Linux中除了硬中断还有软中断,在Linux中处理中断的最大难点,因为会分为上半部下半部这样来分,我的经验是,上半部只干时间敏感的事情,因为那个时候CPU干不了其他事情,而且中断来得很快,所以尽量不要打乱系统的节奏,然后把中断还没处理完的放在下半部,这部分可以比较放心地处理,因为他不会暂停调度器.

每个中断都有自己的中断号,并没不同的硬件中断号也不一样,但是我们不需要记,因为SOC厂商的包内通常已经有指引,我们知道有这么一回事就行.

下面讲一下常用的Linux中断API,请结合官方文档(链接:https://www.kernel.org/doc/html/latest/core-api/genericirq.html)一起食用.

```
int request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags, const char *name, void *dev);
const void * free_irq(unsigned int irq, void *dev_id)
void enable_irq(unsigned int irq);
void disable_irq(unsigned int irq);

// irq 中断号,通过其他API获取.
// handler 发生中断后会调用此函数.
// flags 常用状态有IRQF_TRIGGER_RISING,IRQF_TRIGGER_FALLING,IRQF_TRIGGER_HIGH,IRQF_TRIGGER_LOW,IRQF_ONESHOT,IRQF_SHARED
// name 注册的中断名 - 用户空间在/proc中可以查到.
// dev 如果共用中断,即flags包含IRQF_SHARED,这里需要传入dev来区分.
// 返回值:request_irq < 0 (失败) request_irq = 0 (成功)
```

另外传入的中断函数是有返回值irq_handler_t的,所以我们传进去的函数必须给予返回,通常如下返回代表执行OK,其他情况后续再说.

```
return IRQ_RETVAL(IRQ_HANDLED);
```

另外禁用IRQ的方法会等待所有CPU中断处理完毕,特别是他是一个共享中断的话,会占用很多时间,所以一半情况会使用他的不等待版本.

注:单核CPU无此问题!

```
void disable_irq_nosync(unsigned int irq);
```

在内核文档中提到一个不太常用且很危险的函数,但是如果确实有非常关键任务要保护也可以用,具体查看以下文档,他们都是local_开头的,意味着全局操作,切记不可滥用.

链接:https://www.kernel.org/doc/html/latest/translations/zh_CN/kernel-hacking/hacking.html

记得一开始说过的中断分上下部吗?其实这个说法是过时的,因为BH(Bottom Half)在2.5版本以后就没了,但是一直都这么叫,社区里文档也没改,就都这么叫了,现在一般通过硬中断呼起软中断,或者使用tasklet,当然如果中断内任务非常短,完全在硬中断处理完也行,我认为硬中断的处理临界值在1us内,当然根据各自经验选择就可以了,没有一个标准的要求.

系统中软中断有10个,这里不用过多解释.

```
enum
{
  HI_SOFTIRQ=0,
  TIMER_SOFTIRQ,
  NET_TX_SOFTIRQ,
  NET_RX_SOFTIRQ,
  BLOCK_SOFTIRQ,
  IRQ_POLL_SOFTIRQ,
  TASKLET_SOFTIRQ,
  SCHED_SOFTIRQ,
  HRTIMER_SOFTIRQ,
  RCU_SOFTIRQ,    /* Preferable RCU should always be the last softirq */
```

```
  NR_SOFTIRQS
};
```

那么如何发起软中断呢?

```
void open_softirq(int nr, void (*action)(struct softirq_action *));
void raise_softirq(unsigned int nr);

// nr从上面的enum中选择,action是对应的处理函数.
```

软中断有一个问题,就是必须静态地注册,所以内核手册中也并不推荐.

```
// 文件路径:kernel/softirq.c
void __init softirq_init(void)
{
  int cpu;

  for_each_possible_cpu(cpu) {
    per_cpu(tasklet_vec, cpu).tail =
      &per_cpu(tasklet_vec, cpu).head;
    per_cpu(tasklet_hi_vec, cpu).tail =
      &per_cpu(tasklet_hi_vec, cpu).head;
  }

  open_softirq(TASKLET_SOFTIRQ, tasklet_action);
  open_softirq(HI_SOFTIRQ, tasklet_hi_action);
}
```

那我们可以选择的唯一最优就是tasklet了,不过这里应该注意到,tasklet其实也是软中断所发起的,而tasklet的处理函数就在softirq.c文件中.

```
void tasklet_init(struct tasklet_struct *t,
      void (*func)(unsigned long), unsigned long data)
{
  t->next = NULL;
  t->state = 0;
  atomic_set(&t->count, 0);
  t->func = func;
  t->use_callback = false;
  t->data = data;
}
EXPORT_SYMBOL(tasklet_init);
```

那么怎么使用tasklet?既可以调用上面的函数,也可以使用宏定义.

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

之后只需要在需要发起中断的地方执行以下函数就可以了.

```
void tasklet_schedule(name)
```

而func的定义也有一定的要求,并且可以传递data进去.

```
void func(unsigned long data){

}
```

实际调用(伪代码):

```
void func(unsigned long data){

}

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

除了tasklet这种比较推荐的方法之外.还有工作队列方式,不过这种方式任务延迟会更大一些,他可以被工作队列打断,如果你的工作允许工作队列打断,那么完全可以用工作队列方式.不过只要不是CPU忙着送死,workqueue实际响应效率和tasklet是没区别的.

他使用起来和tasklet也没多大区别.

```
DECLARE_WORK(n,f);
schedule_work(struct work_struct *work);
```

实际调用(伪代码):

```
void f(unsigned long arg)
{

}

DECLARE_WORK(n,f);
schedule_work(&n);
```

最后要说的是,系统内需要用中断的东西很多,比如说软件定时器等也要用中断,因此要特别注意实际使用情况,我们结合实际情况来学习一下,比如从zynq-7000.dtsi中能看到,中断控制器定义如下.

```
intc: interrupt-controller@f8f01000 {
      compatible = "arm,cortex-a9-gic";
      #interrupt-cells = <3>;
      interrupt-controller;
      reg = <0xF8F01000 0x1000>,
            <0xF8F00100 0x100>;
    };
```

可以看到他强调了这是一个interrupt-controller,所以他是一个中断控制器,其中设置cells=3说明要传入3个参数,而具体参数参考gic.txt文档可以看到.

```
* ARM Generic Interrupt Controller

ARM SMP cores are often associated with a GIC, providing per processor
interrupts (PPI), shared processor interrupts (SPI) and software
generated interrupts (SGI).

Primary GIC is attached directly to the CPU and typically has PPIs and SGIs.
Secondary GICs are cascaded into the upward interrupt controller and do not
have PPIs or SGIs.

Main node required properties:

- compatible : should be one of:
  "arm,gic-400"
  "arm,cortex-a15-gic"
  "arm,cortex-a9-gic"
  "arm,cortex-a7-gic"
  "arm,arm11mp-gic"
  "brcm,brahma-b15-gic"
  "arm,arm1176jzf-devchip-gic"
  "qcom,msm-8660-qgic"
  "qcom,msm-qgic2"
- interrupt-controller : Identifies the node as an interrupt controller
- #interrupt-cells : Specifies the number of cells needed to encode an
  interrupt source.  The type shall be a <u32> and the value shall be 3.
```

```
  The 1st cell is the interrupt type; 0 for SPI interrupts, 1 for PPI
  interrupts.

  The 2nd cell contains the interrupt number for the interrupt type.
  SPI interrupts are in the range [0-987].  PPI interrupts are in the
  range [0-15].

  The 3rd cell is the flags, encoded as follows:
  bits[3:0] trigger type and level flags.
    1 = low-to-high edge triggered
    2 = high-to-low edge triggered (invalid for SPIs)
    4 = active high level-sensitive
    8 = active low level-sensitive (invalid for SPIs).
  bits[15:8] PPI interrupt cpu mask.  Each bit corresponds to each of
  the 8 possible cpus attached to the GIC.  A bit set to '1' indicated
  the interrupt is wired to that CPU.  Only valid for PPI interrupts.
  Also note that the configurability of PPI interrupts is IMPLEMENTATION
  DEFINED and as such not guaranteed to be present (most SoC available
  in 2014 seem to ignore the setting of this flag and use the hardware
  default value).

- reg : Specifies base physical address(s) and size of the GIC registers. The
  first region is the GIC distributor register base and size. The 2nd region is
  the GIC cpu interface register base and size.

Optional
- interrupts  : Interrupt source of the parent interrupt controller on
  secondary GICs, or VGIC maintenance interrupt on primary GIC (see
  below).

- cpu-offset  : per-cpu offset within the distributor and cpu interface
  regions, used when the GIC doesn't have banked registers. The offset is
  cpu-offset * cpu-nr.

Example:

  intc: interrupt-controller@fff11000 {
    compatible = "arm,cortex-a9-gic";
    #interrupt-cells = <3>;
    #address-cells = <1>;
    interrupt-controller;
    reg = <0xfff11000 0x1000>,
          <0xfff10100 0x100>;
  };


* GIC virtualization extensions (VGIC)

For ARM cores that support the virtualization extensions, additional
properties must be described (they only exist if the GIC is the
primary interrupt controller).

Required properties:

- reg : Additional regions specifying the base physical address and
  size of the VGIC registers. The first additional region is the GIC
  virtual interface control register base and size. The 2nd additional
  region is the GIC virtual cpu interface register base and size.

- interrupts : VGIC maintenance interrupt.

Example:

  interrupt-controller@2c001000 {
    compatible = "arm,cortex-a15-gic";
    #interrupt-cells = <3>;
    interrupt-controller;
    reg = <0x2c001000 0x1000>,
          <0x2c002000 0x1000>,
          <0x2c004000 0x2000>,
          <0x2c006000 0x2000>;
    interrupts = <1 9 0xf04>;
  };


* GICv2m extension for MSI/MSI-x support (Optional)

Certain revisions of GIC-400 supports MSI/MSI-x via V2M register frame(s).
This is enabled by specifying v2m sub-node(s).
```

```
Required properties:

- compatible      : The value here should contain "arm,gic-v2m-frame".

- msi-controller    : Identifies the node as an MSI controller.

- reg       : GICv2m MSI interface register base and size

Optional properties:

- arm,msi-base-spi  : When the MSI_TYPER register contains an incorrect
            value, this property should contain the SPI base of
          the MSI frame, overriding the HW value.

- arm,msi-num-spis  : When the MSI_TYPER register contains an incorrect
            value, this property should contain the number of
          SPIs assigned to the frame, overriding the HW value.

Example:

  interrupt-controller@e1101000 {
    compatible = "arm,gic-400";
    #interrupt-cells = <3>;
    #address-cells = <2>;
    #size-cells = <2>;
    interrupt-controller;
    interrupts = <1 8 0xf04>;
    ranges = <0 0 0 0xe1100000 0 0x100000>;
    reg = <0x0 0xe1110000 0 0x01000>,
          <0x0 0xe112f000 0 0x02000>,
          <0x0 0xe1140000 0 0x10000>,
          <0x0 0xe1160000 0 0x10000>;
    v2m0: v2m@0x8000 {
      compatible = "arm,gic-v2m-frame";
      msi-controller;
      reg = <0x0 0x80000 0 0x1000>;
    };

    ....

    v2mN: v2m@0x9000 {
      compatible = "arm,gic-v2m-frame";
      msi-controller;
      reg = <0x0 0x90000 0 0x1000>;
    };
  };
```

第一个参数指示SPI(Shared)和PPI(Private),第二个参数指示中断号,第三个是中断触发类型(标志),具体可以看txt文档中说的.

继续看到gpio小节,能看到这样的描述.

```
gpio0: gpio@e000a000 {
      compatible = "xlnx,zynq-gpio-1.0";
      #gpio-cells = <2>;
      clocks = <&clkc 42>;
      gpio-controller;
      interrupt-controller;
      #interrupt-cells = <2>;
      interrupt-parent = <&intc>;
      interrupts = <0 20 4>;
      reg = <0xe000a000 0x1000>;
    };
```

他竟然也是个中断控制器,竟然也要传入两个参数,我们看到往上传递的是0 20 4,说明是SPI,中断号20,触发类型是4,即高电平触发,看到手册中断号是52,因为SPI起始中断号是32,所以这里填20,即是GPIO中断52.

has a 2-bit field, which specifies sensitivity type and handling model.

The SPI interrupts are listed in Table 7-4.

Table 7-4: **PS and PL Shared Peripheral Interrupts (SPI)**

Source	Interrupt Name	IRQ ID#	Status Bits (mpcore Registers)	Required Type	PS-PL Signal Name	I/O
APU	CPU 1, 0 (L1, TLB, BTAC)	33:32	spi_status_0[1:0]	Rising edge	~	~
	L2 Cache	34	spi_status_0[2]	High level	~	~
	OCM	35	spi_status_0[3]	High level	~	~
Reserved	~	36	spi_status_0[3]	~	~	~
PMU	PMU [1,0]	38, 37	spi_status_0[6:5]	High level	~	~
XADC	XADC	39	spi_status_0[7]	High level	~	~
DevC	DevC	40	spi_status_0[8]	High level	~	~
SWDT	SWDT	41	spi_status_0[9]	Rising edge	~	~
Timer	TTC 0	44:42	spi_status_0[12:10]	High level	~	~
DMAC	DMAC Abort	45	spi_status_0[13]	High level	IRQP2F[28]	Output
	DMAC [3:0]	49:46	spi_status_0[17:14]	High level	IRQP2F[23:20]	Output
Memory	SMC	50	spi_status_0[18]	High level	IRQP2F[19]	Output
	Quad SPI	51	spi_status_0[19]	High level	IRQP2F[18]	Output
Reserved	~	~	~	Always driven Low	IRQP2F[17]	Output
IOP	GPIO	52	spi_status_0[20]	High level	IRQP2F[16]	Output
	USB 0	53	spi_status_0[21]	High level	IRQP2F[15]	Output
	Ethernet 0	54	spi_status_0[22]	High level	IRQP2F[14]	Output
	Ethernet 0 Wake-up	55	spi_status_0[23]	Rising edge	IRQP2F[13]	Output
	SDIO 0	56	spi_status_0[24]	High level	IRQP2F[12]	Output
	I2C 0	57	spi_status_0[25]	High level	IRQP2F[11]	Output

GPIO可以发起中断,因此我们要写一个关于自己GPIO中断的配置,当然根据自己实际进行调整,我这里选用EMIO第五个引脚,当然要要记得修改bd并重新生成PL端相关的内容,为什么不用LEVEL_LOW,是因为一旦低电平持续,会锁死整个系统,这是单片机基础知识了.

```
#include <dt-bindings/interrupt-controller/irq.h>

key {
    compatible = "taterli,key";
    key-gpio = <&gpio0 58 GPIO_ACTIVE_LOW>;
    interrupt-parent = <&gpio0>;
    interrupts = <58 IRQ_TYPE_EDGE_FALLING>;
};
```

我们要实现一个驱动功能,按下按键触发中断,并在中断时候发起tasklet,并且创建一个字符设备供用户态读取,我会从LED例子中改,大家也能看得到结构基本是一样的,另外只读设备不用锁也是没关系的.

```
#include <linux/types.h>
#include <linux/kernel.h>
#include <linux/delay.h>
#include <linux/ide.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/errno.h>
#include <linux/gpio.h>
#include <asm/mach/map.h>
#include <asm/uaccess.h>
#include <asm/io.h>
#include <linux/cdev.h>
#include <linux/of.h> /* dts操作相关 */
#include <linux/of_address.h> /* dts地址相关 */
#include <linux/of_gpio.h> /* gpio子系统相关 */
#include <linux/of_irq.h> /* 新增:中断相关 */
#include <linux/mutex.h> /* 信号量头文件 */

#define KERNEL_KEY_DEVIE_CNT 1
#define KERNEL_KEY_NAME "kernel_emio_key"

struct kernel_key_dev
```

```
{
    dev_t devid;
    struct cdev cdev;
    struct class *class;
    struct device *device;
    int major;
    int minor;
    struct device_node *nd; /* 设备节点 */
    int gpio; /* gpio编号 */
    int irq; /* IRQ编号 */
    int status; /* 按键状态 */
    struct mutex used; /* 信号量 */
    struct timer_list timer; /* 消抖定时器 */
};

static struct kernel_key_dev dev;

/* 设备打开时候会被调用 */
static int key_open(struct inode *inode,struct file *filp){
    /* 获取不到会去休眠不会死等,如果用mutex_lock()会死等,用户空间看起来没区别都在死等. */

    if(mutex_lock_interruptible(&dev.used)){
        return -ERESTARTSYS;
    }

    filp->private_data = &dev;

    return 0;
}

/* 设备读取时候会被调用 */
static ssize_t key_read(struct file *filp,char __user *buf,size_t cnt,loff_t *offset){
    int ret;

    if (cnt != 1){
        return -EFAULT;
    }

    /* 由于无中转,也可以直接传递. */
    ret = copy_to_user(buf,&dev.status,cnt);
    if(ret){
        /* 复制失败了 */
        return -EFAULT;
    }
    dev.status = 0;

    return 0;
}

/* 设备写入时候会被调用 */
static ssize_t key_write(struct file *filp,const char __user *buf,size_t cnt,loff_t *offset){
    return 0;
}

/* 设备释放时候会被调用 */
static int key_release(struct inode *inode,struct file *filp){
    mutex_unlock(&dev.used);

    return 0;
}

static struct file_operations fops =
{
    .owner = THIS_MODULE,
    .open = key_open,
    .read = key_read,
    .write = key_write,
    .release = key_release,
};

static void key_timer_function(unsigned long arg)
{
  /* 消抖后再次检查. */
  if (0 == gpio_get_value(dev.gpio))
    dev.status = 1; /* 已经按下按键 */
  else
    dev.status = 0;
}
```

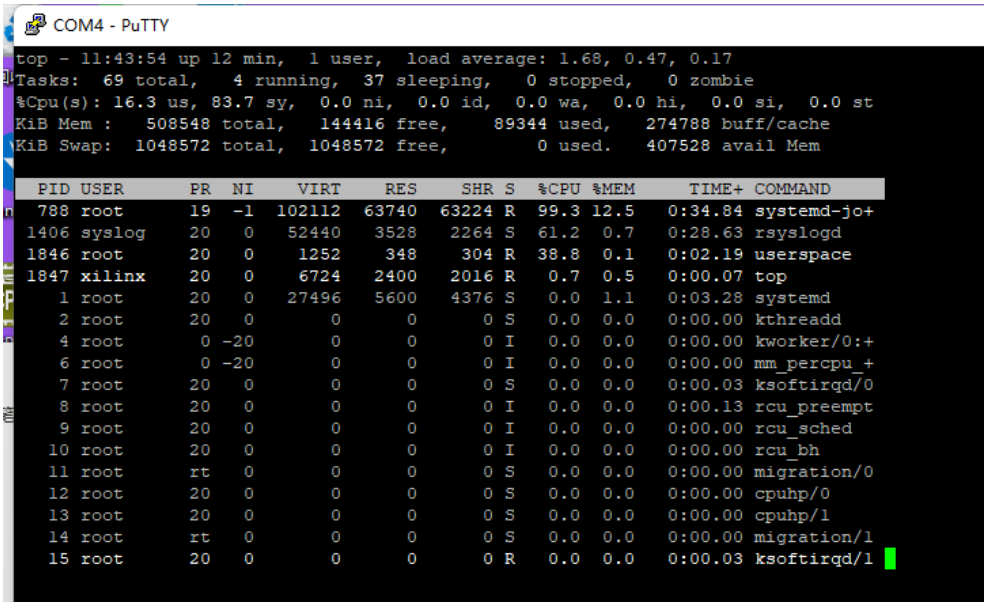
```
static irqreturn_t key_interrupt(int irq, void *dev_id)
{
  /* 按键防抖处理，开启定时器延时15ms */
  mod_timer(&dev.timer, jiffies + msecs_to_jiffies(15));
  return IRQ_HANDLED;
}

/* key_init是系统内置函数,我只能换一个了. */
static int __init emio_key_init(void){
    const char *str;
    int ret;
    unsigned long irq_flags;

    /* 新增的从dts获取数据的过程 */
    dev.nd = of_find_node_by_path("/key");
    if(dev.nd == NULL){
        return -EINVAL;
    }

    ret = of_property_read_string(dev.nd,"status",&str);
    if(ret < 0){
        return -EINVAL;
    }

    if(strcmp(str,"okay")){
        return -EINVAL;
    }

    ret = of_property_read_string(dev.nd,"compatible",&str);
    if(ret < 0){
        return -EINVAL;
    }

    if(strcmp(str,"taterli,key")){
        return -EINVAL;
    }

    /* IO当然也可以是一个数组 */
    dev.gpio = of_get_named_gpio(dev.nd,"key-gpio",0);
    if(!gpio_is_valid(dev.gpio)){
        /* IO是独占资源,因此可能申请失败! */
        return -EINVAL;
    }

    /* 申请IO并给一个名字 */
    ret = gpio_request(dev.gpio,"taterli-kernel-key");
    if(ret < 0){
        /* 除了返回EINVAL,也可以返回上一层传递的错误. */
        return ret;
    }

    /* 将GPIO设置为输入模式 */
  gpio_direction_input(dev.gpio);

  dev.irq = irq_of_parse_and_map(dev.nd, 0);
  if (!dev.irq){
    return -EINVAL;
    }

    /* 获取设备树中指定的中断触发类型 */
  irq_flags = irq_get_trigger_type(dev.irq);
  if (IRQF_TRIGGER_NONE == irq_flags)
    irq_flags = IRQF_TRIGGER_FALLING | IRQF_TRIGGER_RISING;

    printk("irq_num = %d,irq_flags = %ld\n",dev.irq, irq_flags);

  /* 申请中断 */
  ret = request_irq(dev.irq, key_interrupt, irq_flags, "PS EMIO IRQ", NULL);

  if (ret) {
        /* 实在没什么好办法做goto */
    gpio_free(dev.gpio);
    return ret;
  }

    /* 申请一个设备号 */
    ret = alloc_chrdev_region(&dev.devid,0,KERNEL_KEY_DEVIE_CNT,KERNEL_KEY_NAME);
    if(ret){
        goto alloc_fail;
```

```
    }
    dev.major = MAJOR(dev.devid);
    dev.minor = MINOR(dev.devid);

    dev.cdev.owner = THIS_MODULE;
    cdev_init(&dev.cdev,&fops);

    ret = cdev_add(&dev.cdev,dev.devid,KERNEL_KEY_DEVIE_CNT);
    if(ret){
        goto add_fail;
    }

    dev.class = class_create(THIS_MODULE,KERNEL_KEY_NAME);
    if(IS_ERR(dev.class)){
        ret = PTR_ERR(dev.class);
        goto class_fail;
    }

    dev.device = device_create(dev.class,NULL,dev.devid,NULL,KERNEL_KEY_NAME);
    if(IS_ERR(dev.device)){
        ret = PTR_ERR(dev.class);
        goto dev_fail;
    }

    mutex_init(&dev.used); /* 初始设置 */

    /* 初始化定时器(要在中断之前做好!) */
  init_timer(&dev.timer);
  dev.timer.function = key_timer_function;

    return 0;

dev_fail:
    class_destroy(dev.class);

class_fail:
    cdev_del(&dev.cdev);

add_fail:
    unregister_chrdev_region(dev.devid,KERNEL_KEY_DEVIE_CNT);

alloc_fail:
    /* 释放IO之前释放IRQ */
    free_irq(dev.irq,NULL);

    /* 这里就清爽很多了,释放IO就行. */
    gpio_free(dev.gpio);
    return ret;
}

static void __exit emio_key_exit(void){
    /* 添加:删除定时器(虽然不再触发,但是也占着资源!) */
    del_timer_sync(&dev.timer);

    device_destroy(dev.class,dev.devid);

    class_destroy(dev.class);

    cdev_del(&dev.cdev);

    unregister_chrdev_region(dev.devid,KERNEL_KEY_DEVIE_CNT);

    /* 添加:释放IO之前释放中断 */
    free_irq(dev.irq,NULL);

    gpio_free(dev.gpio);
}

module_init(emio_key_init);
module_exit(emio_key_exit);

MODULE_AUTHOR("Taterli <admin@taterli.com>");
MODULE_DESCRIPTION("Key GPIO");
MODULE_LICENSE("GPL");
```

用户程序:

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

int main(int argc,char **argv){
    int fd;
    int val = 0;

    fd = open("/dev/kernel_emio_key",O_RDONLY);
    if(fd < 0){
        return -1;
    }

  for (;;) {
    read(fd, &val, 1);
    if(val == 1){
      printf("Key Press!");
    }
  }

  close(fd);
}
```

这个时候整个CPU都忙于POLL这个设备了,非常占用CPU,实际上读了很多次都是没按下,这和查询没什么意义区别.



那怎么改进呢,就要用到等待队列,就比如你要买面包,然后每一分钟打电话去问有没有面包,非常麻烦,但是如果面包店有新鲜出炉的面包,就通知我们,就不用我们那么累去问面包做好没有了,等待队列简单又好用,主要有以下几个方法,他在内核驱动指南的基础里,说明他其实是很常用的,注意条件必须原子操作.

链接:https://www.kernel.org/doc/html/latest/driver-api/basics.html#c.wait_event

```
wait_event(wq_head, condition);
wait_event_timeout(wq_head, condition, timeout);
wait_event_interruptible(wq_head, condition);
wait_event_interruptible_timeout(wq_head, condition, timeout);

// wq_head 等待队列
// condition 测试条件,弱条件为真,则唤醒队列.
// timeout 要是条件timeout事件之后,依然没有真,也要跳出.
// interruptible 可被打断(释放CPU,交出CPU.)
```

驱动上要改动的就是添加原子变量,因为等待队列的条件必须是原子性的,然后在必要时候唤起队列,驱动改动如下(省略部分).

```
// ...

struct kernel_key_dev
{
    dev_t devid;
    struct cdev cdev;
    struct class *class;
    struct device *device;
    int major;
    int minor;
    struct device_node *nd; /* 设备节点 */
    int gpio; /* gpio编号 */
    int irq; /* IRQ编号 */
    atomic_t status; /* 使用等待队列只能用原子操作! */
    struct timer_list timer; /* 消抖定时器 */
    wait_queue_head_t wait; /* 等待队列 */
};

static struct kernel_key_dev dev;

/* 设备读取时候会被调用 */
static ssize_t key_read(struct file *filp,char __user *buf,size_t cnt,loff_t *offset){
    int ret;

    if (cnt != 1){
        return -EFAULT;
    }

  /* 加入等待队列，当有按键按下或松开动作发生时，才会被唤醒 */
  ret = wait_event_interruptible(dev.wait, atomic_read(&dev.status) == 1);
  if (ret)
    return ret;

    printk("dev.status = %d\n",atomic_read(&dev.status));

    /* 由于无中转,也可以直接传递. */
    ret = copy_to_user(buf,&dev.status,cnt);
    if(ret){
        /* 复制失败了 */
        return -EFAULT;
    }
    atomic_set(&dev.status, 0);

    return 0;
}

static void key_timer_function(unsigned long arg)
{
  /* 消抖后再次检查. */
  if (!gpio_get_value(dev.gpio)){
    atomic_set(&dev.status, 1); /* 已经按下按键 */
    wake_up_interruptible(&dev.wait); /* 遇到这个时候会条件成立因此唤醒队列,如果不确定是否成立也应该唤醒,唤醒后会进行检查. */
  }
  else
    atomic_set(&dev.status, 0);
}

static int __init emio_key_init(void){
    // ...

  atomic_set(&dev.status, 0); /* 初始设置 */

    init_waitqueue_head(&dev.wait);

    /* 初始化定时器(要在中断之前做好!) */
  init_timer(&dev.timer);
  dev.timer.function = key_timer_function;

    return 0;

    // ...
}
```

因为默认的程序都是阻塞的,如果用户非要强调使用非阻塞方法呢,那也很简单,无非多实现一个poll方法嘛,当然,非阻塞的read,是需要立即返回的,就如我们一开始的驱动一样.

```
// ...
#include <linux/poll.h> /* poll相关 */

/* 设备读取时候会被调用 */
static ssize_t key_read(struct file *filp,char __user *buf,size_t cnt,loff_t *offset){
    int ret;

    if (cnt != 1){
        return -EFAULT;
    }

    if (filp->f_flags & O_NONBLOCK) { /* 非阻塞方式访问 */
    if(atomic_read(&dev.status) == 0) /* 按键还没就绪 */
      return -EAGAIN;
  }else{
        /* 加入等待队列，当有按键按下或松开动作发生时，才会被唤醒 */
        ret = wait_event_interruptible(dev.wait, atomic_read(&dev.status) == 1);
        if (ret)
            return ret;
    }

    printk("dev.status = %d\n",atomic_read(&dev.status));

    /* 由于无中转,也可以直接传递. */
    ret = copy_to_user(buf,&dev.status,cnt);
    if(ret){
        /* 复制失败了 */
        return -EFAULT;
    }
    atomic_set(&dev.status, 0);

    return 0;
}

/* 用户强调非阻塞时候 */
static unsigned int key_poll(struct file *filp, struct poll_table_struct *wait)
{
  unsigned int mask = 0;

  poll_wait(filp, &dev.wait, wait);

  if(atomic_read(&dev.status) == 1) /* 按键按下 */
    mask = POLLIN | POLLRDNORM;     /* 返回PLLIN,说明有数据可以取走了! */

  return mask;
}

static struct file_operations fops =
{
    .owner = THIS_MODULE,
    .open = key_open,
    .read = key_read,
    .write = key_write,
    .release = key_release,
    .poll = key_poll,
};
```

这里引入知识点,当我们POLL到有有效数据时候返回了一个POLLIN,这个是什么,他和用户端的poll函数返回值同定义.

1. POLLIN → 有数据准备好
2. POLLPRI → 有数据准备好且这些数据需要立马取走
3. POLLOUT → 准备好坑了可以开始写数据
4. POLLERR → 文件描述符错误.
5. POLLHUP → 文件描述符挂起.
6. POLLNVAL → 无效请求.
7. POLLRDNORM → 这是常规请求,常和POLLIN一起用.

用户端除了poll还有epoll,epoll的返回和poll差不多,不过多了一个E开头,比如EPOLLIN,epoll效率更高,不需要遍历全部描述符,但是并发度不高的情况下,建议还是使用poll,对于epoll需要分多步来完成.

```
struct pollfd{
  int fd; /* 文件描述符 */
  short events; /* 请求的事件 */
  short revents; /* 返回的事件 */
}

struct epoll_event {
  __uint32_t events; /* EPOLL 事件,可以是EPOLLIN,EPOLLOUT,EPOLLPRI,EPOLLERR,EPOLLHUP,EPOLLET,EPOLLONESHOT. */
  epoll_data_t data; /* 用户数据 */
};

// nfds => 要监视的文件描述符数量
// timeout => 超时(毫秒)

int poll(struct pollfd *fds,nfds_t nfds,int timeout);
int epoll_create(int size); // 随便传入一个大于0的数值就可以.
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event); // 可以EPOLL_CTL_ADD,EPOLL_CTL_MOD,EPOLL_CTL_DEL一个感兴趣的event到fd中.
int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout); // 能等maxevents个events.
```

除了poll还可以用select函数,具体也可以网上搜来看看,下面是poll例子.

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

int main(int argc, char **argv)
{
  int fd;
  int ret;
  int val = 0;
  struct pollfd fds;

  fd = open("/dev/kernel_emio_key", O_RDONLY | O_NONBLOCK);
  if (fd < 0)
  {
    return -1;
  }

  fds.fd = fd;
  fds.events = POLLIN;

  for (;;)
  {
    ret = poll(&fds, 1, 5 * 1000);
    if (ret == 0)
    {
      printf("No key event with 5sec!\n");
    }
    else if (ret == -1)
    {
      printf("Some error occurred!\n");
    }
    else
    {

      read(fd, &val, 1);
      if (val == 1)
      {
        printf("key_val = %d\n", val);
      }
    }
  }

  close(fd);
}
```

虽然通过各种方法,都可以以很轻的CPU占用率来读取中断状态,但是本质依然是一个查询,能不能有一个方法,也通过"中断"方式同时用户呢.

可以,不过他叫"信号",我们执行程序时候发送Ctrl+C,那是一个SIGINT信号,Linux中的信号很多,在内核include/uapi/asm-generic/signal.h可以看到,除了列表上的,还可以有用户空间通知用户空间的USR1,比如dd命令就可以通过USR1知道当前进度,用户空间可以通过pkill等发送信号,而内核可以通过fasync发送信号,本质上是一种类似"中断"的通信方式,这里发现没有,发送给信号的命令居然包含kill,没错,内核中发送命令的函数也都和kill这个词离不开.

dd的相关代码:https://github.com/coreutils/coreutils/blob/master/src/dd.c

信号的基本用法就是使用signal(USR1,handler)这样注册,当遇到USR1信号,发起handler函数,其他时候摸鱼.

内核驱动代码(完整版,主要添加key_fasync和定时器中添加fasync发起!):

```
#include <linux/types.h>
#include <linux/kernel.h>
#include <linux/delay.h>
#include <linux/ide.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/errno.h>
#include <linux/gpio.h>
#include <asm/mach/map.h>
#include <asm/uaccess.h>
#include <asm/io.h>
#include <linux/cdev.h>
#include <linux/of.h>         /* dts操作相关 */
#include <linux/of_address.h> /* dts地址相关 */
#include <linux/of_gpio.h>    /* gpio子系统相关 */
#include <linux/of_irq.h>     /* 新增:中断相关 */
#include <linux/mutex.h>      /* 信号量头文件 */
#include <linux/poll.h>       /* poll相关 */
#include <linux/fcntl.h>      /* 文件操作(包括异步通知) */

#define KERNEL_KEY_DEVIE_CNT 1
#define KERNEL_KEY_NAME "kernel_emio_key"

struct kernel_key_dev
{
    dev_t devid;
    struct cdev cdev;
    struct class *class;
    struct device *device;
    int major;
    int minor;
    struct device_node *nd;       /* 设备节点 */
    int gpio;                     /* gpio编号 */
    int irq;                      /* IRQ编号 */
    atomic_t status;              /* 使用等待队列只能用原子操作! */
    struct timer_list timer;      /* 消抖定时器 */
    wait_queue_head_t wait;       /* 等待队列 */
    struct fasync_struct *fasync; /* fasync_struct结构体(异步通知用途) */
};

static struct kernel_key_dev dev;

/* 设备打开时候会被调用 */
static int key_open(struct inode *inode, struct file *filp)
{
    filp->private_data = &dev;

    return 0;
}

/* 设备读取时候会被调用 */
static ssize_t key_read(struct file *filp, char __user *buf, size_t cnt, loff_t *offset)
{
    int ret;

    if (cnt != 1)
    {
        return -EFAULT;
    }

    if (filp->f_flags & O_NONBLOCK)
```

```
    {                                       /* 非阻塞方式访问 */
        if (atomic_read(&dev.status) == 0) /* 按键还没就绪 */
            return -EAGAIN;
    }
    else
    {
        /* 加入等待队列，当有按键按下或松开动作发生时，才会被唤醒 */
        ret = wait_event_interruptible(dev.wait, atomic_read(&dev.status) == 1);
        if (ret)
            return ret;
    }

    printk("dev.status = %d\n", atomic_read(&dev.status));

    /* 由于无中转,也可以直接传递. */
    ret = copy_to_user(buf, &dev.status, cnt);
    if (ret)
    {
        /* 复制失败了 */
        return -EFAULT;
    }
    atomic_set(&dev.status, 0);

    return 0;
}

/* 设备写入时候会被调用 */
static ssize_t key_write(struct file *filp, const char __user *buf, size_t cnt, loff_t *offset)
{
    return 0;
}

/* 用户强调非阻塞时候 */
static unsigned int key_poll(struct file *filp, struct poll_table_struct *wait)
{
    unsigned int mask = 0;

    poll_wait(filp, &dev.wait, wait);

    if (atomic_read(&dev.status) == 1) /* 按键按下 */
        mask = POLLIN | POLLRDNORM;    /* 返回PLLIN,说明有数据可以取走了! */

    return mask;
}

/* 直接套娃绑定用户传递过来的信息! */
static int key_fasync(int fd, struct file *filp, int on)
{
    return fasync_helper(fd, filp, on, &dev.fasync);
}

/* 设备释放时候会被调用 */
static int key_release(struct inode *inode, struct file *filp)
{
    /* 只有一个东西需要释放,他成功自然全部成功. */
    return key_fasync(-1, filp, 0);
}

static struct file_operations fops =
    {
        .owner = THIS_MODULE,
        .open = key_open,
        .read = key_read,
        .write = key_write,
        .release = key_release,
        .poll = key_poll,
        .fasync = key_fasync,
};

static void key_timer_function(unsigned long arg)
{
    /* 消抖后再次检查. */
    if (!gpio_get_value(dev.gpio))
    {
        atomic_set(&dev.status, 1);       /* 已经按下按键 */
        wake_up_interruptible(&dev.wait); /* 遇到这个时候会条件成立因此唤醒队列,如果不确定是否成立也应该唤醒,唤醒后会进行检查. */
        if (dev.fasync)
        { /* 如果有定义异步通知也不要忘了通知应用程序 */
            /* 可以读时候设置成POLL_IN,可以写时候设置成POLL_OUT,第二个参数就是信号. */
```

```
            kill_fasync(&dev.fasync, SIGIO, POLL_IN);
        }
    }
    else
        atomic_set(&dev.status, 0);
}

static irqreturn_t key_interrupt(int irq, void *dev_id)
{
    /* 按键防抖处理，开启定时器延时15ms */
    mod_timer(&dev.timer, jiffies + msecs_to_jiffies(15));
    return IRQ_HANDLED;
}

/* key_init是系统内置函数,我只能换一个了. */
static int __init emio_key_init(void)
{
    const char *str;
    int ret;
    unsigned long irq_flags;

    /* 新增的从dts获取数据的过程 */
    dev.nd = of_find_node_by_path("/key");
    if (dev.nd == NULL)
    {
        return -EINVAL;
    }

    ret = of_property_read_string(dev.nd, "status", &str);
    if (ret < 0)
    {
        return -EINVAL;
    }

    if (strcmp(str, "okay"))
    {
        return -EINVAL;
    }

    ret = of_property_read_string(dev.nd, "compatible", &str);
    if (ret < 0)
    {
        return -EINVAL;
    }

    if (strcmp(str, "taterli,key"))
    {
        return -EINVAL;
    }

    /* IO当然也可以是一个数组 */
    dev.gpio = of_get_named_gpio(dev.nd, "key-gpio", 0);
    if (!gpio_is_valid(dev.gpio))
    {
        /* IO是独占资源,因此可能申请失败! */
        return -EINVAL;
    }

    /* 申请IO并给一个名字 */
    ret = gpio_request(dev.gpio, "taterli-kernel-key");
    if (ret < 0)
    {
        /* 除了返回EINVAL,也可以返回上一层传递的错误. */
        return ret;
    }

    /* 将GPIO设置为输入模式 */
    gpio_direction_input(dev.gpio);

    dev.irq = irq_of_parse_and_map(dev.nd, 0);
    if (!dev.irq)
    {
        return -EINVAL;
    }

    /* 获取设备树中指定的中断触发类型 */
    irq_flags = irq_get_trigger_type(dev.irq);
    if (IRQF_TRIGGER_NONE == irq_flags)
        irq_flags = IRQF_TRIGGER_FALLING | IRQF_TRIGGER_RISING;
```

```
    printk("irq_num = %d,irq_flags = %ld\n", dev.irq, irq_flags);

    /* 申请中断 */
    ret = request_irq(dev.irq, key_interrupt, irq_flags, "PS EMIO IRQ", NULL);

    if (ret)
    {
        /* 实在没什么好办法做goto */
        gpio_free(dev.gpio);
        return ret;
    }

    /* 申请一个设备号 */
    ret = alloc_chrdev_region(&dev.devid, 0, KERNEL_KEY_DEVIE_CNT, KERNEL_KEY_NAME);
    if (ret)
    {
        goto alloc_fail;
    }
    dev.major = MAJOR(dev.devid);
    dev.minor = MINOR(dev.devid);

    dev.cdev.owner = THIS_MODULE;
    cdev_init(&dev.cdev, &fops);

    ret = cdev_add(&dev.cdev, dev.devid, KERNEL_KEY_DEVIE_CNT);
    if (ret)
    {
        goto add_fail;
    }

    dev.class = class_create(THIS_MODULE, KERNEL_KEY_NAME);
    if (IS_ERR(dev.class))
    {
        ret = PTR_ERR(dev.class);
        goto class_fail;
    }

    dev.device = device_create(dev.class, NULL, dev.devid, NULL, KERNEL_KEY_NAME);
    if (IS_ERR(dev.device))
    {
        ret = PTR_ERR(dev.class);
        goto dev_fail;
    }

    atomic_set(&dev.status, 0); /* 初始设置 */

    init_waitqueue_head(&dev.wait);

    /* 初始化定时器(要在中断之前做好!) */
    init_timer(&dev.timer);
    dev.timer.function = key_timer_function;

    return 0;

dev_fail:
    class_destroy(dev.class);

class_fail:
    cdev_del(&dev.cdev);

add_fail:
    unregister_chrdev_region(dev.devid, KERNEL_KEY_DEVIE_CNT);

alloc_fail:
    /* 释放IO之前释放IRQ */
    free_irq(dev.irq, NULL);

    /* 这里就清爽很多了,释放IO就行. */
    gpio_free(dev.gpio);
    return ret;
}

static void __exit emio_key_exit(void)
{
    /* 添加:删除定时器(虽然不再触发,但是也占着资源!) */
    del_timer_sync(&dev.timer);

    device_destroy(dev.class, dev.devid);
```

```
    class_destroy(dev.class);

    cdev_del(&dev.cdev);

    unregister_chrdev_region(dev.devid, KERNEL_KEY_DEVIE_CNT);

    /* 添加:释放IO之前释放中断 */
    free_irq(dev.irq, NULL);

    gpio_free(dev.gpio);
}

module_init(emio_key_init);
module_exit(emio_key_exit);

MODULE_AUTHOR("Taterli <admin@taterli.com>");
MODULE_DESCRIPTION("Key GPIO");
MODULE_LICENSE("GPL");
```

用户驱动:

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

static int fd;

static void key_press(int signum)
{
  int val = 0;
  read(fd, &val, 1);
  if (val == 1)
  {
    printf("Key Press!\n");
  }
}

int main(int argc, char *argv[])
{
  int flags = 0;

  fd = open("/dev/kernel_emio_key", O_RDONLY);
  if (fd < 0)
  {
    return -1;
  }

  signal(SIGIO, key_press);
  fcntl(fd, F_SETOWN, getpid()); // 要告诉内核自己PID,不然通知谁呢.
  flags = fcntl(fd, F_GETFD);
  fcntl(fd, F_SETFL, flags | FASYNC); // 在基础状态(fd)上附加一个异步通知功能,此时的fd包含进程PID以及FASYNC请求,内核收到后就会执行.fasync的fops.

  for (;;)
  {
    sleep(1);
  }

  close(fd);
}
```

驱动越写越复杂,BUG和隐患就越来越多,这是教程所不能传授的,需要不断地训练来提高,除了上面说的read,write,open,release,poll,fasync,还有一个比较常用的是ioctl,刚才看到fasync基本就是套娃,用户空间写什么,这里就存什么,其实ioctl也差不多,大家不妨试试,这里给出一个例子.

```
// 共同定义
#define CMD_MYCMD_A    (_IO(0XEF, 0x1))
#define CMD_MYCMD_B    (_IO(0XEF, 0x2))
```

```
// 内核空间
static long kmod_ioctl(struct file *filp, unsigned int cmd,
                       unsigned long arg)
{
    switch (cmd)
    {

    case CMD_MYCMD_A:
        funcA();
        break;

    case CMD_MYCMD_B:
        funcB(arg);
        break;

    default:
        break;
    }
    return 0;
}

// 用户空间
ioctl(fd, CMD_MYCMD_B, arg);
```