国嵌

www.enjoylinux.cn

LINUX
总线设备驱动模型

Contents

Contents

Sysfs文件系统

"sysfs is a ram-based filesystem initially based on ramfs. It provides a means to export kernel data structures, their attributes, and the linkages between them to userspace.”

--- documentation/filesystems/sysfs.txt

Sysfs文件系统

Linux2.6内核引入了 sysfs 文件系统。sysfs 被看成是与 proc同类别的文件系统。sysfs 把连接在系统上的设备和总线组织成分级的文件，使其从用户空间可以访问到。

Sysfs文件系统

Sysfs 被加载在 /sys/ 目录下，它的子目录包括:

- **Block**: 在系统中发现的每个块设备在该目录下对应一个子目录。每个子目录中又包含一些属性文件，它们描述了这个块设备的各方面属性，如：设备大小。

  （loop块设备是使用文件来模拟的）

Sysfs文件系统

- **Bus**：在内核中注册的每条总线在该目录下对应一个子目录，如：

  ide pci scsi usb pcmcia

其中每个总线目录内又包含两个子目录：

devices 和 **drivers**， **devices**目录包含了在整个系统中发现的属于该总线类型的设备，

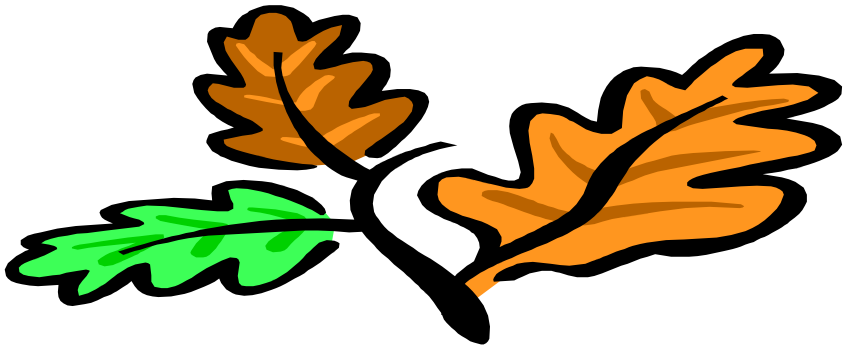
drivers目录包含了注册到该总线的所有驱动。

Sysfs文件系统

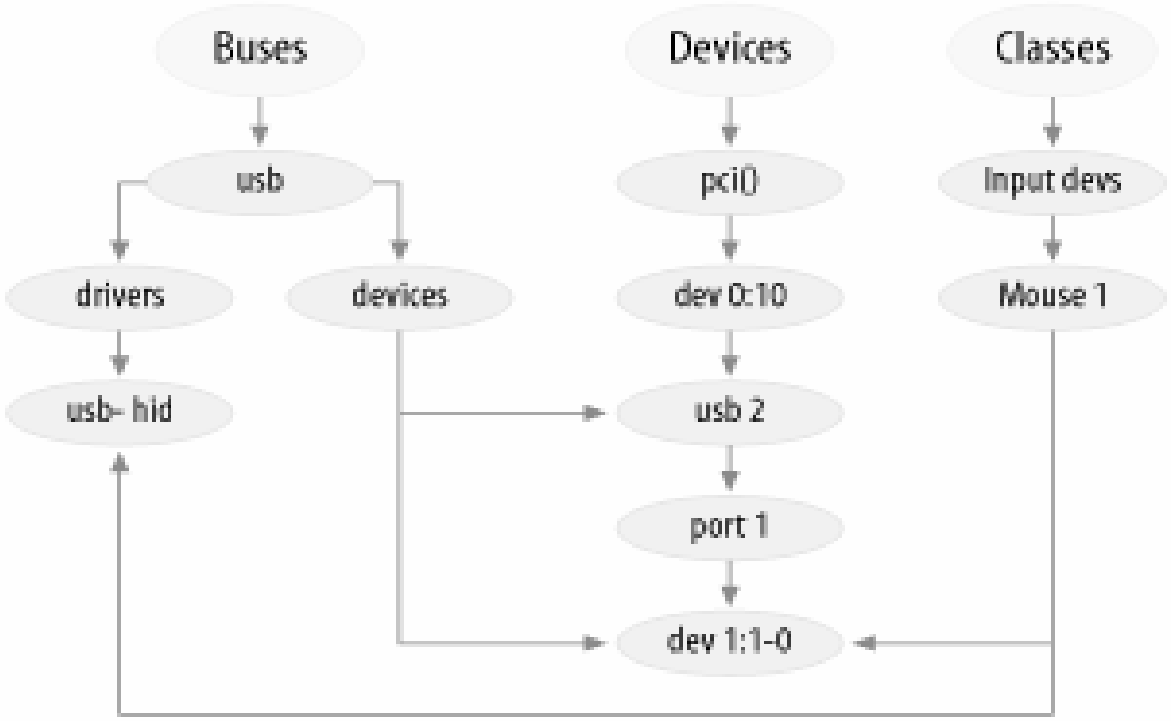
- **Class**：将设备按照功能进行的分类，如/sys/class/net目录下包含了所有网络接口。
- **Devices**：包含系统所有的设备。
- **Kernel**：内核中的配置参数
- **Module**：系统中所有模块的信息

Sysfs文件系统

- **Firmware**: 系统中的固件
- **Fs**: 描述系统中的文件系统
- **Power**:系统中电源选项



Sysfs文件系统



Kobject

Kobject 实现了基本的面向对象管理机制，是构成Linux2.6设备模型的核心结构。它与sysfs文件系统紧密相连，在内核中注册的每个kobject对象对应sysfs文件系统中的一个目录。

Kobject

类似于C++中的基类，Kobject常被嵌入于其他类型（即：容器）中。如bus, devices, drivers 都是典型的容器。这些容器通过kobject连接起来，形成了一个树状结构。

Kobject

www.enjoylinux.cn

```
struct kobject {
    const char              *name;
    struct list_head        entry;
    struct kobject                  *parent; //指向父对象
    struct kset             *kset;
    struct kobj_type        *ktype;
    struct sysfs_dirent     *sd;
    struct kref             kref; //对象引用计数
    unsigned int state_initialized:1;
    unsigned int state_in_sysfs:1;
    unsigned int state_add_uevent_sent:1;
    unsigned int state_remove_uevent_sent:1;
};
```


Kobject操作

- void kobject_init(struct kobject * kobj)

  初始化kobject结构

- int kobject_add(struct kobject * kobj)

  将kobject对象注册到Linux系统

- int kobject_init_and_add(struct kobject *kobj, struct kobj_type *ktype,struct kobject *parent, const char *fmt, ...)

  初始化kobject，并将其注册到linux系统

Kobject操作

- void kobject_del(struct kobject * kobj)

  从Linux系统中删除kobject对象

- struct kobject *kobject_get(struct kobject *kobj)

  将kobject对象的引用计数加1，同时返回该对象指针。

- void kobject_put(struct kobject * kobj)

  将kobject对象的引用计数减1，如果引用计数降为0，则调用release方法释放该kobject对象。

Struct kobj_type

Kobject的ktype成员是一个指向kobj_type结构的指针，该结构中记录了kobject对象的一些属性。

```
struct kobj_type {
    void (*release)(struct kobject *kobj);
    struct sysfs_ops *sysfs_ops;
    struct attribute **default_attrs;
};
```

release：用于释放kobject占用的资源，当kobject的引用计数为0时被调用。

Struct attribute


```
struct attribute {
    char * name; /*属性文件名*/
    struct module * owner;
    mode_t mode; /*属性的保护位*/
};
```

struct attribute （属性）：对应于kobject的目录下的一个文件，Name成员就是文件名。

Struct sysfs_ops


```
struct sysfs_ops
{
ssize_t (*show)(struct kobject *, struct attribute *,char *);
ssize_t (*store)(struct kobject *,struct attribute *,const char *,
    size_t);
};
```

- **Show:** 当用户读属性文件时，该函数被调用，该函数将属性值存入buffer中返回给用户态;
- **Store:** 当用户写属性文件时，该函数被调用，用于存储用户传入的属性值。

实例分析

Kobject.c

实验

编写内核模块，在sys目录中创建一个目录，并在该目录下创建一个可读写的文件

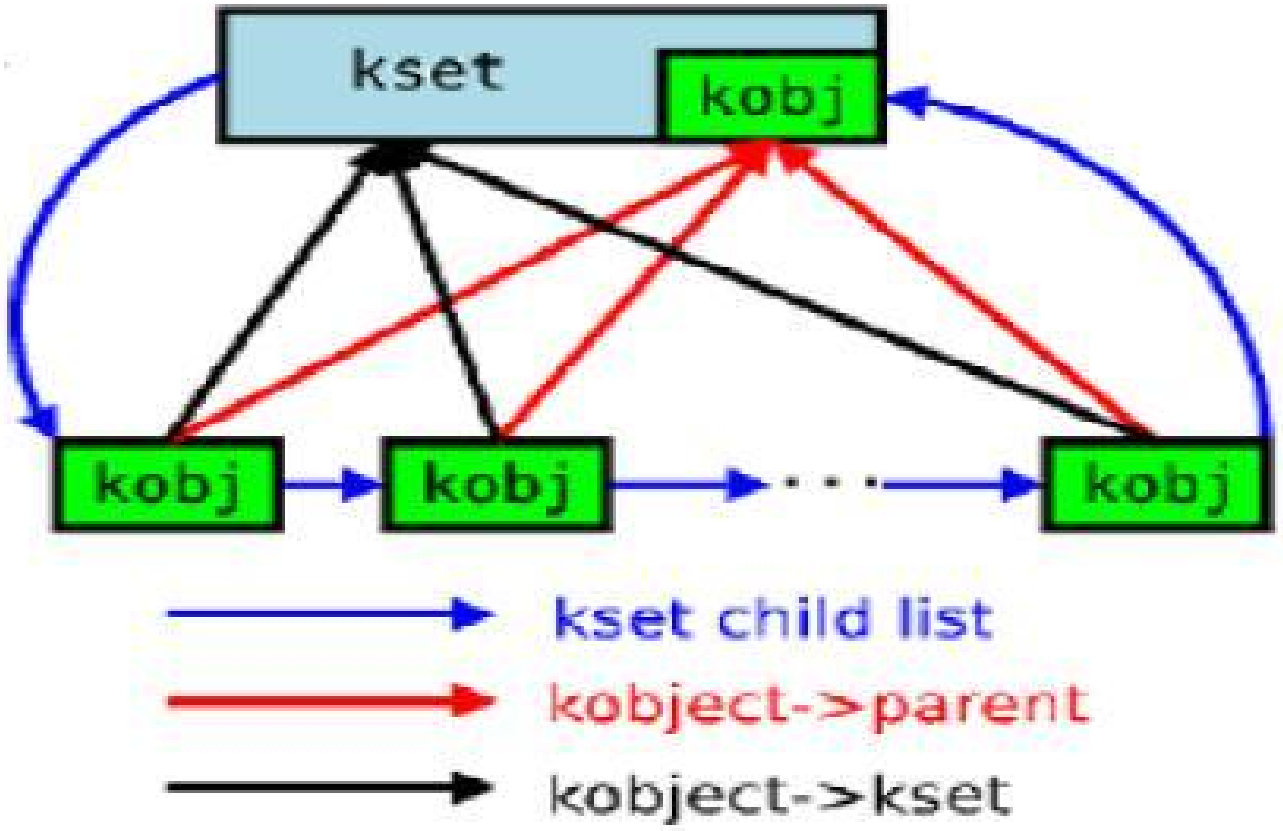
（在mini2440平台实现）

Kset

kset是具有相同类型的kobject的集合，在sysfs中体现成一个目录，在内核中用kset数据结构表示，定义为：

```
struct kset {
    struct list_head list; //连接该kset中所有kobject的链表头
    spinlock_t list_lock;
    struct kobject kobj; //内嵌的kobject
    struct kset_uevent_ops *uevent_ops; //处理热插拔事件的操
    作集合
}
```


Kset



Kset操作

- int kset_register(struct kset *kset)

  在内核中注册一个kset

- void kset_unregister(struct kset *kset)

  从内核中注销一个kset

热插拔事件

在Linux系统中，当系统配置发生变化时,如:添加kset到系统；移动kobject,一个通知会从内核空间发送到用户空间,这就是热插拔事件。热插拔事件会导致用户空间中相应的处理程序(如udev,mdev)被调用,这些处理程序会通过加载驱动程序,创建设备节点等来响应热插拔事件。

操作集合


```
Struct kset_uevent_ops {
    int (*filter)(struct kset *kset, struct kobject *kobj);
    const char *(*name)(struct kset *kset, struct kobject *kobj);
    int (*uevent)(struct kset *kset, struct kobject *kobj,
            struct kobj_uevent_env *env);
}
```


kset_uevent_ops

这三个函数什么时候调用？

当该kset所管理的kobject和kset状态发生变化时（如被加入，移动），这三个函数将被调用。

（例： kobject_uevent调用）

kset_uevent_ops

这三个函数的功能是什么？

- filter：决定是否将事件传递到用户空间。如果 filter 返回 0,将不传递事件。（例： uevent_filter）
- name：用于将字符串传递给用户空间的热插拔处理程序。
- uevent：将用户空间需要的参数添加到环境变量中。（例：dev_uevent）

实例分析

Kset.c

Contents

设备模型

随着技术的不断进步，系统的拓扑结构也越来越复杂，对智能电源管理、热插拔的支持要求也越来越高，2.4内核已经难以满足这些需求。为适应这种形势的需要，Linux 2.6内核提供了全新的内核设备模型。

设备模型元素

www.enjoylinux.cn

总线

驱动

设备

总线

总线是处理器和设备之间的通道，在设备模型中，所有的设备都通过总线相连，甚至是内部的虚拟“platform”总线。 在 Linux 设备模型中，总线由 bus_type 结构表示，定义在 <linux/device.h>

总线描述

www.enjoylinux.cn

```
struct bus_type {
        const char              *name; /*总线名称*/
        struct bus_attribute    *bus_attrs; /*总线属性*/
        struct device_attribute *dev_attrs; /*设备属性*/
        struct driver_attribute *drv_attrs; /*驱动属性*/
        int (*match)(struct device *dev, struct device_driver *drv);
        int (*uevent)(struct device *dev, struct kobj_uevent_env *env);
        int (*probe)(struct device *dev);
        int (*remove)(struct device *dev);
        void (*shutdown)(struct device *dev);
        int (*suspend)(struct device *dev, pm_message_t state);
        int (*suspend_late)(struct device *dev, pm_message_t state);
        int (*resume_early)(struct device *dev);
        int (*resume)(struct device *dev);
        struct dev_pm_ops *pm;
        struct bus_type_private *p;
}
```


总线注册/删除

总线的注册使用：

bus_register(struct bus_type * bus)

若成功，新的总线将被添加进系统，并可在 sysfs 的 /sys/bus 下看到。

总线的删除使用：

void bus_unregister(struct bus_type *bus)

总线方法


```
int (*match)(struct device * dev, struct device_driver * drv)
```

当一个新设备或者驱动被添加到这个总线时，该方法被调用。用于判断指定的驱动程序是否能处理指定的设备。若可以，则返回非零值。

总线方法


```
int (*uevent)(struct device *dev, char **envp, int num_envp,
char *buffer, int buffer_size)
```

在为用户空间产生热插拔事件之前，这个方法允许总线添加环境变量。

总线属性

总线属性由结构bus_attribute 描述，定义如下：

```
struct bus_attribute {
        struct attribute    attr;
        ssize_t (*show)(struct bus_type *, char * buf);
        ssize_t (*store)(struct bus_type *, const char *
    buf, size_t count);
}
```


总线属性

- int bus_create_file(struct bus_type *bus, struct bus_attribute *attr)

  创建属性

- void bus_remove_file(struct bus_type *bus, struct bus_attribute *attr)

  删除属性

实例分析

Bus_basic.c

实验

编写内核模块，在系统中创建一条总线。

（在mini2440平台实现）

设备描述

Linux 系统中的每个设备由一个 struct device 描述:

```
struct device {
  …… …… …… …… …… ……
    struct kobject kobj;
    char  bus_id[BUS_ID_SIZE];   /*在总线上唯一标识该设备的字符串 */
    struct bus_type          *bus;    /* 设备所在总线 */
    struct device_driver *driver;   /*管理该设备的驱动*/
    void   *driver_data;             /*该设备驱动使用的私有数据成员 *
    struct klist_node        knode_class;
    struct class             *class;
    struct attribute_group           **groups;
    void (*release)(struct device *dev);
}
```


设备注册

- int device_register(struct device *dev)

  注册设备

- void device_unregister(struct device *dev)

  注销设备

一条总线也是个设备，也必须按设备注册

设备属性

设备属性由struct device_attribute 描述:

```
struct device_attribute
{
    struct attribute attr;
    ssize_t (*show)(struct device *dev, struct device_attribute
    *attr,char *buf);
    ssize_t (*store)(struct device *dev, struct device_attribute *attr,
    const char *buf, size_t count);
}
```

设备属性

- int device_create_file(struct device *device, struct device_attribute * entry)

  创建属性

- void device_remove_file(struct device * dev, struct device_attribute * attr)

  删除属性

实例分析

Bus.c

Device.c

实验

编写内核模块，在系统中创建一条总线，并在该总线上添加一个设备。

（在mini2440平台实现）

驱动描述

驱动程序由struct device_driver 描述：

```
struct device_driver {
    const char     *name;  /*驱动程序的名字( 体现在 sysfs 中 )*/
    struct bus_type *bus;  /*驱动程序所在的总线*/
    struct module                   *owner;
    const char                 *mod_name;
    int (*probe) (struct device *dev);
    int (*remove) (struct device *dev);
    void (*shutdown) (struct device *dev);
    int (*suspend) (struct device *dev, pm_message_t state);
    int (*resume) (struct device *dev);
    struct attribute_group **groups;
    struct dev_pm_ops *pm;
    struct driver_private *p;
}
```


驱动注册/注销

- int driver_register(struct device_driver *drv)

  注册驱动

- void driver_unregister(struct device_driver *drv)

  注销驱动

驱动属性

驱动的属性使用struct driver_attribute 来描述:

```
struct driver_attribute {
    struct attribute attr;
    ssize_t (*show)(struct device_driver *drv,
    char *buf);
    ssize_t (*store)(struct device_driver *drv,
    const char *buf, size_t count);
}
```


驱动属性

- int driver_create_file(struct device_driver * drv, struct driver_attribute * attr)

  创建属性

- void driver_remove_file(struct device_driver * drv, struct driver_attribute * attr)

  删除属性

实例分析

driver.c

实验

编写内核模块，在系统中创建一条总线，并在该总线上注册一个驱动程序

（在mini2440平台实现）

Contents

Platform总线

Platform总线是linux2.6内核加入的一种虚拟总线。platform机制的本身使用并不复杂，由两部分组成:

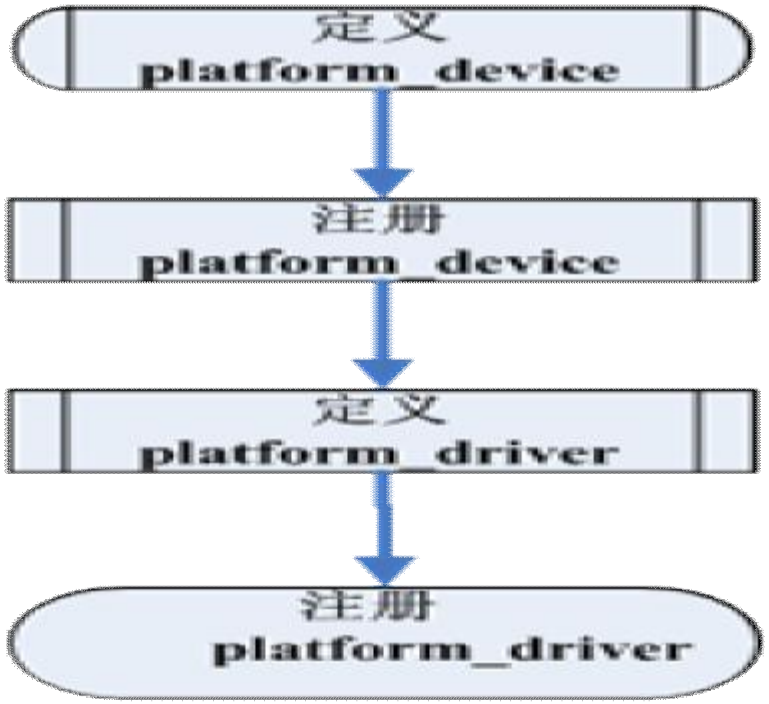
platform_device和platform_driver

Platform总线

Platform 驱动与传统的设备驱动模型相比，优势在于platform机制将设备本身的资源注册进内核，由内核统一管理，在驱动程序使用这些资源时使用统一的接口，这样提高了程序的可移植性。

工作流程

通过platform机制开发底层设备驱动的流程如图：



平台设备描述

平台设备使用Struct Platform_device来描述：

```
struct platform_device {
    const char *name;  /*设备名*/
    int id;    /*设备编号，配合设备名使用*/
    struct device dev;
    u32 num_resources;
    struct resource  *resource;  /*设备资源*/
}
```

平台设备描述

Struct Platform_device的分配使用：

```
struct platform_device
    *platform_device_alloc(const char *name, int id)
```

参数：

name: 设备名

id: 设备id，一般为-1

平台设备注册

注册平台设备，使用函数：

int platform_device_add(struct platform_device *pdev)

设备资源

平台设备资源使用struct resource来描述:

```
struct resource {
    resource_size_t start;   //资源的起始物理地址
    resource_size_t end;   //资源的结束物理地址
    const char *name;      //资源的名称
    unsigned long flags;  //资源的类型，比如MEM，IO，IRQ类型
    struct resource *parent, *sibling, *child;   //资源链表指针
}
```


设备资源-例


```
static struct resource s3c_wdt_resource1 = {
        .start = 0x44100000,
        .end   = 0x44200000,
        .flags = IORESOURCE_MEM,
    }
static struct resource s3c_wdt_resource2 = {
        .start = 20,
        .end   = 20,
        .flags = IORESOURCE_IRQ,
    }
```


获取资源

struct resource *platform_get_resource(struct platform_device *dev, unsigned int type, unsigned int num)

参数:

- dev: 资源所属的设备
- type: 获取的资源类型
- num: 获取的资源数

例:

platform_get_resource(pdev, IORESOURCE_IRQ, 0)

获取中断号

平台驱动描述

平台驱动使用struct platform_driver 描述:

```
struct platform_driver {
int (*probe)(struct platform_device *);
int (*remove)(struct platform_device *);
void (*shutdown)(struct platform_device *);
int (*suspend)(struct platform_device *, pm_message_t state);
int (*suspend_late)(struct platform_device *, pm_message_t state);
int (*resume_early)(struct platform_device *);
int (*resume)(struct platform_device *);
struct device_driver driver;
}
```


平台驱动注册

平台驱动注册使用函数:

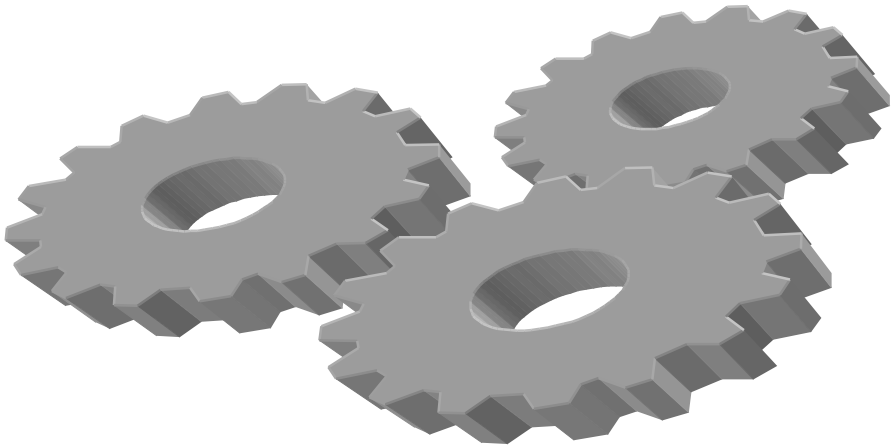
int platform_driver_register(struct platform_driver *)

Platform驱动

www.enjoylinux.cn

Platform_driver_register

原理分析



实例分析

plat_dev.c

Plat_drv.c

s3c2410_wdt.c

Contents

中断概念

为什么需要中断?

1.外设的处理速度一般慢于CPU

2.CPU不能一直等待外部事件

所以设备必须有一种方法来通知CPU它的工作进度，这种方法就是中断。

中断实现

在Linux驱动程序中,为设备实现一个中断包含两个步骤:

1.向内核注册中断

2.实现中断处理函数

中断注册

request_irq用于实现中断的注册功能:

```
int request_irq(unsigned int irq,
                void (*handler)(int, void*, struct
                    pt_regs *),
                unsigned long flags,
                const char *devname,
                void *dev_id)
```

返回0表示成功，或者返回一个错误码

中断注册(参数)

- unsigned int irq

  中断号。

- void (*handler)(int,void *,struct pt_regs *)

  中断处理函数。

- unsigned long flags

  与中断管理有关的各种选项。

- const char * devname

  设备名

- void *dev_id

  共享中断时使用。

中断注册(中断标志)

在flags参数中，可以选择一些与中断管理有关的选项,如:

- **IRQF_DISABLED（SA_INTERRUPT）**

  如果设置该位，表示是一个“快速”中断处理程序；如果没有设置这位，那么是一个“慢速”中断处理程序。

- **IRQF_SHARED（SA_SHIRQ）**

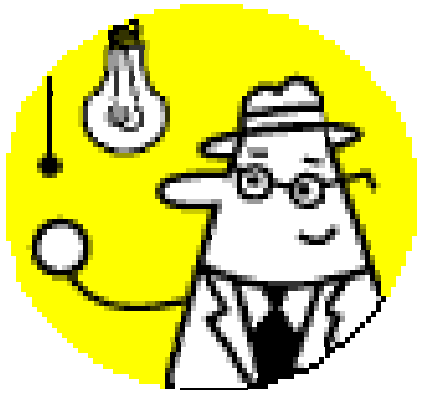
  该位表明中断可以在设备间共享。

快速/慢速中断

这两种类型的中断处理程序的主要区别在于：快速中断保证中断处理的原子性(不被打断)，而慢速中断则不保证。换句话说，也就是“开启中断”标志位(处理器IF)在运行快速中断处理程序时是关闭的，因此在服务该中断时,不会被其他类型的中断打断;而调用慢速中断处理时，其它类型的中断仍可以得到服务。

共享中断

共享中断就是将不同的设备挂到同一个中断信号线上。Linux对共享的支持主要是为PCI设备服务。



共享中断

共享中断也是通过request_irq函数来注册的，但有三个特别之处：

1. 申请共享中断时，必须在flags参数中指定IRQF_SHARED位

2. dev_id参数必须是唯一的。

Q: 为什么要唯一?

共享中断

3.共享中断的处理程序中，不能使用

disable_irq(unsigned int irq)

为什么？

如果使用了这个函数，共享中断信号线的其它设备将同样无法使用中断,也就无法正常工作了。

中断处理程序

什么是中断处理程序,有何特别之处?

中断处理程序就是普通的C代码。特别之处在于中断处理程序是在中断上下文中运行的，它的行为受到某些限制:

1. 不能向用户空间发送或接受数据
2. 不能使用可能引起阻塞的函数
3. 不能使用可能引起调度的函数

中断处理函数流程


```
void short_sh_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
   /* 判断是否是本设备产生了中断(为什么要做这样的检测?) */
    value = inb(short_base);
    if (!(value & 0x80)) return;

    /* 清除中断位(如果设备支持自动清除,则不需要这步) */
    outb(value & 0x7F, short_base);

    /* 中断处理,通常是数据接收 */
     。 。 。 。 。 。 。 。 。

      /* 唤醒等待数据的进程 */
      wake_up_interruptible(&short_queue);
```


释放中断

当设备不再需要使用中断时(通常在驱动卸载时)，应当把它们返还给系统，使用：

```
void free_irq(unsigned int irq, void *dev_id)
```

Contents

工作原理

S3c2440的GPIO_G0，GPIO_G3，GPIO_G5，GPIO_G6, GPIO_G7, GPIO_G11作为输入口，读取按键状态，这六个I/O口分别使用外部中断EINT8，EINT11，EINT13，EINT14，EINT15，EINT19。当按键松开时，I/O口处于高电平，得到逻辑1，当按键按下时，I/O被拉低，得到逻辑0。

实例分析

mini2440_buttons.c

实验

www.enjoylinux.cn

采用平台驱动模型实现按键驱动程序

（在mini2440平台实现）