ioctl的使用方法

通过发命令,来实现设备的控制

要实现用ioctl来控制设备,在驱动中必须添加如下功能

```
app:ioctl(fd,cmd,0)
drv:static long led_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
    printk("cmd =%d\n",cmd);
    switch(cmd){
        case LED_ON:
            /*点灯*/
            writel(readl(fs210_led_dev->led_base+FS210_GPC0DAT)|(0x1<<3)|
(0x1<<4),fs210_led_dev->led_base+FS210_GPC0DAT);
            break;
        case LED_OFF:
            /*灭灯*/
            writel(readl(fs210_led_dev->led_base+FS210_GPC0DAT)&~((0x1<<3)|
(0x1<<4)),fs210_led_dev->led_base+FS210_GPC0DAT);
            break;
        default:
            break;
    }
    return 0;
}
```

命令构建(linut/ioctl.h)

/用如下宏来组建命令,保证命令唯一性/ //一个命令组建的时候包含以下几个部分,有些命令可以带数据

```
#define _IOC(dir,type,nr,size) \
    (((dir)  << _IOC_DIRSHIFT) | \
     ((type) << _IOC_TYPESHIFT) | \
     ((nr)   << _IOC_NRSHIFT) | \
     ((size) << _IOC_SIZESHIFT))
```

不带数据的命令组建宏

```
define IO(type,nr)      _IOC(IOC_NONE,(type),(nr),0)
```

带数据的读命令组建宏

```
define IOR(type,nr,size)    _IOC(IOC_READ,(type),(nr),(_IOC_TYPECHECK(size)))
```

带数据的写命令组建宏

```
define IOW(type,nr,size)    _IOC(IOC_WRITE,(type),(nr),(_IOC_TYPECHECK(size))
```

带命令的可读可写组建宏

```
define IOWR(type,nr,size)   _IOC(IOC_READ|IOC_WRITE,(type),(nr),(IOC_TYPECHECK(size)))
```

2.6.36 之前的内核

```
 long (ioctl) (struct inode node ,struct file* filp, unsigned int cmd,unsigned long arg)
```

2.6.36之后的内核

```
long (*unlocked_ioctl) (struct file *filp, unsigned int cmd, unsigned long arg)
```

命令从其实质而言就是一个整数, 但为了让这个整数具备更好的可读性，我们通常会把这个整数分为几个段：类型（8位），序号，参数传送方向，参数长度。

Type(类型/幻数): 表明这是属于哪个设备的命令

Number(序号)，用来区分同一设备的不同命令

Direction：参数传送的方向，可能的值是_IOC_NONE(没有数据传输), _IOC_READ, _IOC_WRITE（向设备写入参数）

Size： 参数长度

Linux系统提供了下面的宏来帮助定义命令:

_IO(type,nr)：不带参数的命令

_IOR(type,nr,datatype)：从设备中读参数的命令

_IOW(type,nr,datatype)：向设备写入参数的命令

例：

#define MEM_MAGIC ‘m’ //定义幻数

#define MEM_SET _IOW(MEM_MAGIC, 0, int)

unlocked_ioctl函数的实现通常是根据命令执行的一个switch语句。但是，当命令号不能匹配任何一个设备所支持的命令时，返回-EINVAL.

编程模型：

```
Switch cmd

Case 命令A：
```

```
//执行A对应的操作

Case 命令B：

//执行B对应的操作

Default:

// return -EINVAL
```

地址映射

动态映射：

所谓动态映射，是指在驱动程序中采用ioremap函数将物理地址映射为虚拟地址。

原型：void * ioremap(physaddr, size)

参数：

Physaddr：待映射的物理地址

Size: 映射的区域长度

返回值：映射后的虚拟地址

静态映射：

所谓静态映射，是指Linux系统根据用户事先指定的映射关系，在内核启动时，自动地将物理地址映射为虚拟地址。

在静态映射中，用户是通过map_desc结构来指明物理地址与虚拟地址的映射关系。

```
struct map_desc {

unsigned long virtual; /* 映射后的虚拟地址*/

unsigned long pfn; /* 物理地址所在的页帧号*/

unsigned long length; /* 映射长度*/

unsigned int type; /* 映射的设备类型*/

};
```

pfn: 利用__phys_to_pfn(物理地址)可以计算出物理地址所在的物理页帧号,在完成地址映射后，就可以读写寄存器了，Linux内核提供了一系列函数，来读写寄存器。

```
unsigned ioread8(void *addr)

unsigned ioread16(void *addr)
```

```
unsigned ioread32(void *addr)

unsigned readb(address)

unsigned readw(address)

unsigned readl(address)

void iowrite8(u8 value, void *addr)

void iowrite16(u16 value, void *addr)

void iowrite32(u32 value, void *addr)

void writeb(unsigned value, address)    //写入1字节数据

void writew(unsigned value, address)    //写入2字节数据

void writel(unsigned value, address)    //写入4字节数据
```