基于misc框架的驱动分析

XiaoBaWu 于 2016-09-02 14:57:12 发布 775 收藏 1

分类专栏： 【Linux内核与驱动】 文章标签： 框架 内核 硬件

版权声明：本文为博主原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接和本声明。

本文链接：https://blog.csdn.net/qq_28992301/article/details/52413648

版权

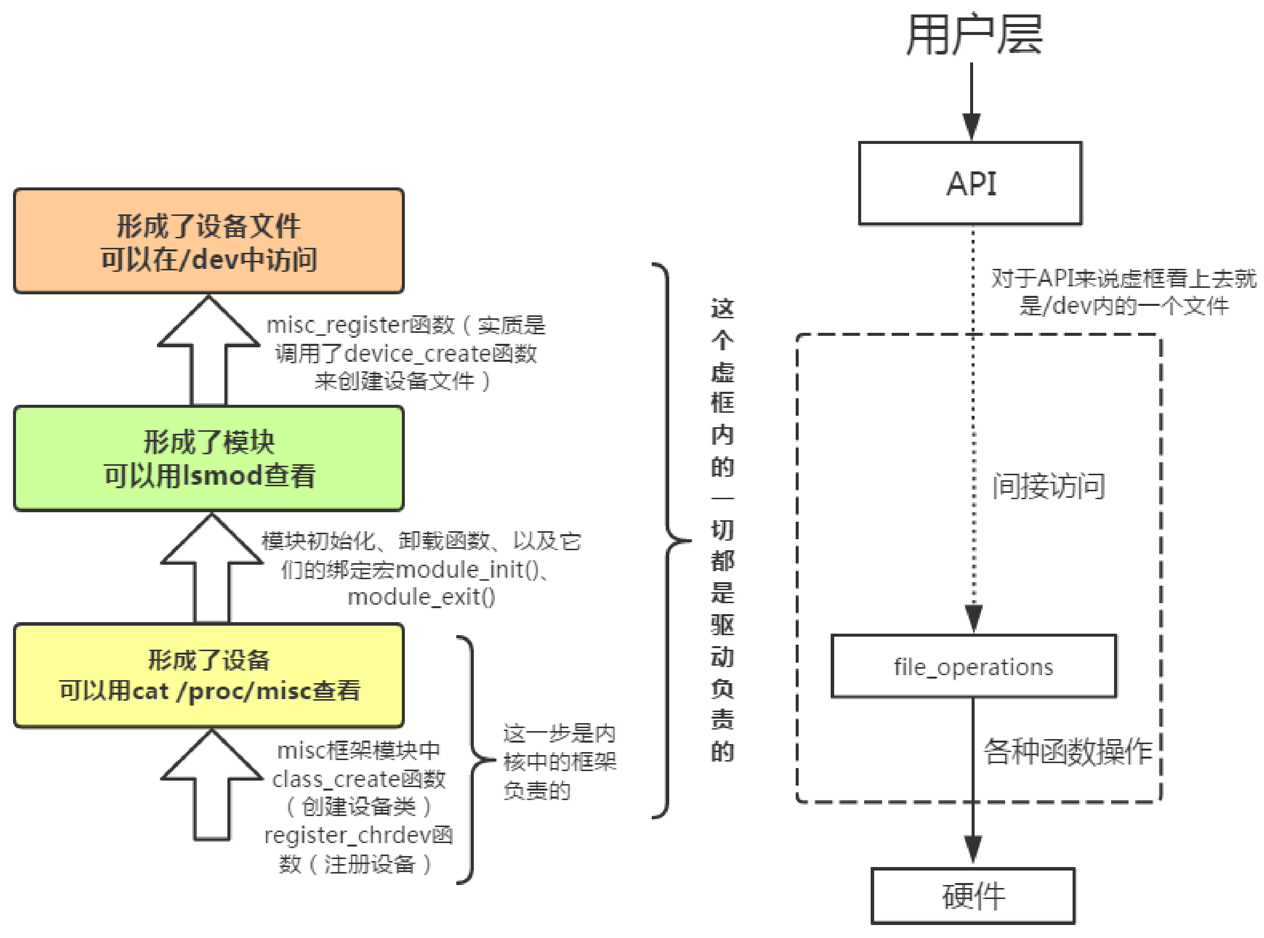
【Linux内核与驱动】 专栏收录该内容

28 篇文章 18 订阅
订阅专栏

基于misc框架的驱动分析

所谓的misc设备，就是很难于分类的杂散类设备，比如蜂鸣器、adc等。
一般来说，misc设备都是字符设备，所以led也能划分入misc设备，led驱动也能用misc设备驱动实现，这也印证了驱动的实现是自由的。
其本质是：通过读写/dev内的设备文件，来间接访问file_operations结构体内的函数，从而实现操作硬件

通过对比普通字符设备驱动的结构图，我们不难发现，misc框架是对普通字符设备驱动的一次封装罢了，它们从本质上并没有什么不同



值得注意的一点：因为历史原因，在这个版本的misc框架中，所有的misc设备公用一个主设备号（都是10），它们之间以次设备号互相区分。所以在框架中使用register_chrdev注册了一个主设备号为10的设备，而在驱动中device_create创建设备文件主设备号都为10，次设备号不同

1.misc框架接口的使用条件

- 如果要使用内核的框架来写驱动的话，必须要在menuconfig中添加框架模块，这样才能够调用框架接口函数

2.misc框架接口的实现原理

- misc.c这个文件提供了有关misc框架的接口，里面是misc框架模块
- misc_init是框架模块的加载函数，它主要负责了创建misc类，和misc设备的注册

```
static int __init misc_init(void)
{

    /*无关代码就不贴了*/
    ...

    /*创建misc类*/
    misc_class = class_create(THIS_MODULE, "misc");

    /*无关代码就不贴了*/
    ...

    /*misc设备的注册*/
    if (register_chrdev(MISC_MAJOR,"misc",&misc_fops))
        goto fail_printk;

    misc_class->devnode = misc_devnode;
    return 0;

    /*无关代码就不贴了*/
    ...
}
```

- 关于misc设备的注册，有一个值得注意的地方。因为历史原因，在这个版本的misc框架中，**所有的misc设备公用一个主设备号（都是10），它们之间以次设备号互相区分**。所以上面那段代码中，使用字符设备旧注册接口（详见两种注册方式），注册了一个名为misc的设备，其实是把我们所有的misc设备都注册了

3.驱动代码分析

下面以蜂鸣器的驱动作为实例，其实整体结构和普通字符设备驱动差不多，只有一些小区别，主要是用了misc_register这个框架提供的接口函数

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/delay.h>
#include <linux/poll.h>
#include <asm/irq.h>
#include <asm/io.h>
#include <linux/interrupt.h>
#include <asm/uaccess.h>
#include <plat/regs-timer.h>
#include <mach/regs-irq.h>
#include <asm/mach/time.h>
#include <linux/clk.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/miscdevice.h>
#include <linux/gpio.h>
#include <plat/gpio-cfg.h>

#define DEVICE_NAME     "buzzer"

#define PWM_IOCTL_SET_FREQ      1
#define PWM_IOCTL_STOP          0
```

```
/*定义了一个信号量。其实这个驱动是把信号量当成互斥锁来用了。这是落伍的方法不如直接用互斥锁*/
static struct semaphore lock;

/*这里开始是ioctl会调用的的设置函数*/
static void PWM_Set_Freq( unsigned long freq )
{
    unsigned long tcon;
    unsigned long tcnt;
    unsigned long tcfg1;

    struct clk *clk_p;
    unsigned long pclk;

    /*这里开始是利用原厂的静态映射结合寄存器读写函数来操作寄存器*/
    s3c_gpio_cfgpin(S5PV210_GPD0(2), S3C_GPIO_SFN(2));

    tcon = __raw_readl(S3C2410_TCON);
    tcfg1 = __raw_readl(S3C2410_TCFG1);

    //mux = 1/16
    tcfg1 &= ~(0xf<<8);
    tcfg1 |= (0x4<<8);
    __raw_writel(tcfg1, S3C2410_TCFG1);

    clk_p = clk_get(NULL, "pclk");
    pclk  = clk_get_rate(clk_p);

    tcnt  = (pclk/16/16)/freq;

    __raw_writel(tcnt, S3C2410_TCNTB(2));
    __raw_writel(tcnt/2, S3C2410_TCMPB(2));//ռ?¿Ô?50%

    tcon &= ~(0xf<<12);
    tcon |= (0xb<<12);      //disable deadzone, auto-reload, inv-off, update TCNTB0&TCMPB0, start timer
    __raw_writel(tcon, S3C2410_TCON);

    tcon &= ~(2<<12);           //clear manual update bit
    __raw_writel(tcon, S3C2410_TCON);
}

void PWM_Stop( void )
{
    s3c_gpio_cfgpin(S5PV210_GPD0(2), S3C_GPIO_SFN(0));
}

/*这里开始是file_operations内的操作函数*/

/*这里的open和close函数分别执行了上锁和解锁。目的是防止文件被重复打开*/
static int x210_pwm_open(struct inode *inode, struct file *file)
{
        /*把互斥锁上锁。down的本质是把锁-1*/
    if (!down_trylock(&lock))
        return 0;
    else
        return -EBUSY;

}


static int x210_pwm_close(struct inode *inode, struct file *file)
```

```c
static int x210_pwm_close(struct inode *inode, struct file *file)
{
        /*把互斥锁解锁。up的本质是把锁+1*/
    up(&lock);
    return 0;
}

/*这里没有用read、write。而是用了ioctl。这是一种对文件发送命令的方式。广泛运用在驱动程序中*/
/*通过这种方式应用层可以对驱动发送命令。虽然用read、write也能实现。不过用ioctl更加合理(因为蜂鸣器不涉及数据的读写)*
static int x210_pwm_ioctl(struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg)
{
    switch (cmd)
    {
        case PWM_IOCTL_SET_FREQ:
            printk("PWM_IOCTL_SET_FREQ:\r\n");
            if (arg == 0)
                return -EINVAL;
            PWM_Set_Freq(arg);
            break;

        case PWM_IOCTL_STOP:
        default:
            printk("PWM_IOCTL_STOP:\r\n");
            PWM_Stop();
            break;
    }

    return 0;
}

/*定义一个file_operations设备体*/
static struct file_operations dev_fops = {
    .owner   =   THIS_MODULE,
    .open    =   x210_pwm_open,
    .release =   x210_pwm_close,
    .ioctl   =   x210_pwm_ioctl,
};


/*定义一个misc设备体*/
static struct miscdevice misc = {
    /*.minor是次设备号的意思MISC_DYNAMIC_MINOR这个宏是255。
     *这样设置可以让内核为我们自动分配次设备号
     */
    .minor = MISC_DYNAMIC_MINOR,
    .name = DEVICE_NAME,
    .fops = &dev_fops,
};

static int __init dev_init(void)
{
    int ret;
    /*初始化了一个互斥锁。其实是把信号量当成互斥锁来用了。这是落伍的方法不如直接用互斥锁*/
    init_MUTEX(&lock);

    /*注册misc。其实做了两件事。分配了次设备号并注册设备、创建了设备文件*/
    ret = misc_register(&misc);

    /*蜂鸣器的硬件初始化*/
    ret = gpio_request(S5PV210_GPD0(2), "GPD0");
```

```
    if(ret)
    printk("buzzer-x210: request gpio GPD0(2) fail");

    /*这里不是很规范，严格来说硬件的初始化应该放在open函数内比较好*/
    s3c_gpio_setpull(S5PV210_GPD0(2), S3C_GPIO_PULL_UP);
    s3c_gpio_cfgpin(S5PV210_GPD0(2), S3C_GPIO_SFN(1));
    gpio_set_value(S5PV210_GPD0(2), 0);

    return ret;
}

static void __exit dev_exit(void)
{
    gpio_free(S5PV210_GPD0(2));
    misc_deregister(&misc);
}

module_init(dev_init);
module_exit(dev_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("www.9tripod.com");
MODULE_DESCRIPTION("x210 PWM Driver");
```