整理--Misc类设备驱动

原创

liam.li 于 2017-02-28 21:05:33 发布 1382 收藏 1
分类专栏： # 设备驱动基础 文章标签： MISC类设备驱动

本文链接：https://blog.csdn.net/TongxinV/article/details/58657530

知识整理–MISC类设备驱动

> misc类设备的本质是字符设备，在驱动框架中使用register_chrdev注册了一个主设备号为10的设备

misc类设备介绍

1. 何为misc

(1)中文名：杂散设备
(2)/sys/class/misc
(3)典型的字符设备。misc是对原始的字符设备注册接口的一个类层次的封装，很多典型字符设备都可以归类到misc类中，使用misc驱动框架来管理
(4)像LED一样也有一套驱动框架，内核实现一部分（misc.c，led是led-class.c），驱动实现一部分（x210-buzzer.c）

1. 涉及文件

/drivers/char/misc.c
/driver/char/buzzer/x210-buzzer.c

misc驱动框架源码分析

> 源码框架的主要工作：注册misc类，使用老接口注册字符设备（主设备号10），开放device注册的接口misc_register给驱动工程师

驱动框架模块的注册

> 在框架中使用 `register_chrdev` 注册了一个主设备号为10的设备，而在设备驱动中 `device_create` 创建设备文件主设备号都为10，次设备号不同，从而实现分类（分为misc类）。我们也可以模仿这种方式创建自己的分类，但是其实不简单，还有好多东西需要积累！

以下文件位于/drivers/char/misc.c：

下列代码位于/drivers/char/misc.c

```
static struct class *misc_class;

static const struct file_operations misc_fops = {
    .owner      = THIS_MODULE,
    .open       = misc_open,
};

static int __init misc_init(void)
{
    int err;

#ifdef CONFIG_PROC_FS
    proc_create("misc", 0, NULL, &misc_proc_fops);
#endif
    misc_class = class_create(THIS_MODULE, "misc");//创建设备类，一定在某个地方有 device_create
    err = PTR_ERR(misc_class);
    if (IS_ERR(misc_class))
        goto fail_remove;

    err = -EIO;
    if (register_chrdev(MISC_MAJOR,"misc",&misc_fops))
                goto fail_printk;

    misc_class->devnode = misc_devnode;
    return 0;

fail_printk:
    printk("unable to get major %d for misc devices\n", MISC_MAJOR);
    class_destroy(misc_class);
fail_remove:
    remove_proc_entry("misc", NULL);
    return err;
}
subsys_initcall(misc_init);
```

//这是字符设备驱动开发基础的提到过的涉及到的旧的注册接口去注册我们设备内部通过 __register_chrdev_region 将 misc 对应的 char_device_struct 结构体类型的变量的地址挂到内核维护的 char_device_struct 结构体类型的指针数组
从这里我们也可以看出 misc 类设备本质是字符设备

注册了一个设备号为 MISC_MAJOR 的设备 misc，为什么这里要注册一个设备？ 分析开放出来的注册接口 misc_register，其中有一句 dev = MKDEV(MISC_MAJOR, misc->minor);使用了主设备号+次设备号生成一个设备号。也就是说在驱动中 device_create 创建设备文件主设备号都为 MISC_MAJOR，次设备号不同。从而实现分类(分出 misc 类的字符设备)!!!
但是这个 misc 设备和后面的注册的杂散设备本质上是一样的，没有说 misc 是老大，也就是说 misc 这个设备的 fops 并不是缺省

misc_open：分析misc_open，发现其最终指向**某一个设备对应的miscdevice结构体变量**中的 `file_operations结构体变量` 中的open。（没具体分析应用层打开misc类设备是通过这里的open间接调用具体的open，还是直接调用某一具体misc设备的open）

开放出来的注册接口

> misc_register

以下文件位于/drivers/char/misc.c：

下列代码在/drivers/char/misc.c

```
static LIST_HEAD(misc_list);          //定义和初始化一个维护链表(可以理解为用于管理入口 entry)
static DEFINE_MUTEX(misc_mtx);        //定义和初始化一个互斥锁
```

下列代码在 include/linux/miscdevice.h

```
struct miscdevice  {
    int minor;   /*次设备号*/
    const char *name;
    const struct file_operations *fops;
    struct list_head list;
            /*将内核链表内嵌到结构体*//
    struct device *parent;
    struct device *this_device;
    const char *nodename;
    mode_t mode;
};
```

将来写misc设备类就是定义一个这样的一个结构体变量然后填充，然后调用 misc_register 进行注册
作用就是类似于 led 框架的 led_classdev 结构体，只不过 led_classdev 没有 fops.因为 led 走的是 atrribute 那条路

```
int misc_register(struct miscdevice * misc)
{
    struct miscdevice *c;
    dev_t dev;
    int err = 0;

    INIT_LIST_HEAD(&misc->list);
                    //节点对应链表入口(entry)初始化
    mutex_lock(&misc_mtx);

    list_for_each_entry(c, &misc_list, list) {
        if (c->minor == misc->minor) {
            mutex_unlock(&misc_mtx);
            return -EBUSY;
        }
    }
```

在本文件中你是找不到 misc_list 这个变量的，怎么回事呢？在开头的 LIST_HEAD(misc_list); 把它展开后

```
static struct list_head misc_list = {
    &(misc_list), &(misc_list)
}
struct list_head {
    struct list_head *next, *prev;
}
```

也就是说 misc_list 是内核维护的一个 misc 设备类的链表，将来添加一个设备就往这个链表添加相应的 miscdevice 的结构体变量的地址(指针)

list_for_each_entry:是一个宏定义展开是一个 for 循环，实现的是去内核维护的 misc_list 链表中取出一个一个 miscdevice 的结构体变量的指针，比较 minor 与传进来的 minor，如果相等就说明这个 minor 被使用了；遍历了一遍之后都没有相等则说明.....

```
    if (misc->minor == MISC_DYNAMIC_MINOR) {
        int i = find_first_zero_bit(misc_minors, DYNAMIC_MINORS);
        if (i >= DYNAMIC_MINORS) {
            mutex_unlock(&misc_mtx);
            return -EBUSY;
        }
        misc->minor = DYNAMIC_MINORS - i - 1;
        set_bit(i, misc_minors);
    }
```

当传进来的次设备号为 MISC_DYNAMIC_MINOR 宏定义的那个数字，说明是要让内核帮我们分配次设备号，怎么帮？bitmap

```
    dev = MKDEV(MISC_MAJOR, misc->minor);  /* 使用固定的主设备号 + 次设备号构造设备号 */

    misc->this_device = device_create(misc_class, misc->parent, dev,
                      misc, "%s", misc->name);
    if (IS_ERR(misc->this_device)) {
        int i = DYNAMIC_MINORS - misc->minor - 1;
        if (i < DYNAMIC_MINORS && i >= 0)
            clear_bit(i, misc_minors);
        err = PTR_ERR(misc->this_device);
        goto out;
    }

    /*
     * Add it to the front, so that later devices can "override"
     * earlier defaults
     */
    list_add(&misc->list, &misc_list);
 out:
    mutex_unlock(&misc_mtx);
    return err;
}
```

struct miscdevice：misc类设备的抽象

misc_list：内核维护的管理misc设备类的链表，将来添加一个设备就往这个链表添加相应的**miscdevice的结构体变量的地址(指针)**

misc设备驱动源码分析

以misc类设备蜂鸣器为例

> 定义一个`struct miscdevice`结构体变量并进行填充，然后调用驱动框架提供的`misc_register`进行注册。

下列代码位于/driver/char/buzzer/x210-buzzer.c：

miscdevice：

```
struct miscdevice  {
    int minor;  /*次设备号*/
    const char *name;
    const struct file_operations *fops;
    struct list_head list;/*将内核链表内嵌到结构体*/

    struct device *parent;
    struct device *this_device;
    const char *nodename;
    mode_t mode;
};
```

x210_pwm_ioctl：

```
// TCFG0在Uboot中设置，这里不再重复设置
// Timer0输入频率 Finput=pclk/(prescaler1+1)/MUX1
//                  =66M/16/16
// TCFG0 = tcnt = (pclk/16/16)/freq;
// PWM0输出频率 Foutput =Finput/TCFG0= freq
static void PWM_Set_Freq( unsigned long freq )
{
    unsigned long tcon;
    unsigned long tcnt;
    unsigned long tcfg1;

    struct clk *clk_p;
    unsigned long pclk;       /*以Hz为单位的时钟频率*/

    //unsigned tmp;

    //设置GPD0_2为PWM输出
    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);  /*要理解这边的代码需要配合Datasheet*/
    tcfg1 |= (0x4<<8);
    __raw_writel(tcfg1, S3C2410_TCFG1);

    clk_p = clk_get(NULL, "pclk");
    pclk  = clk_get_rate(clk_p);/*clk_get_rate是系统提供的函数，用来读取我们系统设置的各
            种时钟的值 要理解就需要去分析时钟模块 */
    tcnt  = (pclk/16/16)/freq;

    __raw_writel(tcnt, S3C2410_TCNTB(2));
    __raw_writel(tcnt/2, S3C2410_TCMPB(2));//占空比为50%

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

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

void PWM_Stop( void )
{
    //将GPD0_2设置为input
    s3c_gpio_cfgpin(S5PV210_GPD0(2), S3C_GPIO_SFN(0));
}
```

x210_pwm_open/x210_pwm_close：

```c
static int x210_pwm_open(struct inode *inode, struct file *file)
{
    if (!down_trylock(&lock))/* 说明应用层打开一个设备，就是去试图去上锁，上锁成功则说明
                                成功打开这个设备*/
                            /* 同时说明这样去实现的时候，这个设备就不能同时被两个进程或以上调用*/
        return 0;
    else
        return -EBUSY;

}
static int x210_pwm_close(struct inode *inode, struct file *file)
{
    up(&lock);
    return 0;
}
```

`static struct semaphore lock` / `down_trylock(&lock)` / `up(&lock)`：信号量使用旧式接口，关于信号量的使用可以看内核源码如何使用

板载蜂鸣器驱动测试

> 确认/dev目录下有buzzer设备，没有需要make menuconfig 进行配置(确保子Makefile和子Kconfig相关变量相一致)

简单测试app_buzzer.c：

app_buzzer.c

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>

#define DEVNAME  "/dev/buzzer"
#define PWM_IOCTL_SET_FREQ        1
#define PWM_IOCTL_STOP            0
int main(void)
{
    int fd = -1;
    fd = open(DEVNAME, O_RDWR);
    if(fd < 0){
        perror("open");
        return -1;
    }
    ioctl(fd, PWM_IOCTL_SET_FREQ, 10000);//这个函数来自于驱动
    sleep(3);
    ioctl(fd, PWM_IOCTL_STOP);
    sleep(3);
    close(fd);

    return 0;
}
```

注意0：关于文中涉及到的内核链表可以参考这篇文章《内核链表实现分析与使用(双向环形链表)》

注意1：register_chrdev和device_creator的区别。register_chrdev是向内核维护的一个指针数组添加一个字符设备抽象对应的地址，来注册驱动；device_create是利用给的设备号和类名去创建设备文件（创建后的设备文件位于/dev）。

注意2：分析LED驱动框架源码的时候，并没有register_chrdev，只有class_create和device_creator，而且走的是attribute路线。所以用attribute方式实现的驱动只能通过/sys/class下的文件进行访问，详细可以查看这篇文章谈论attribute驱动实现方式(及device_create与设备节点的关系)