misc设备驱动模型

转载

hanson69 于 2017-02-09 13:37:49 发布 612 收藏

首先看看misc设备驱动的结构体，定义在include/Linux/miscdevice.h中：

```
struct miscdevice  {
        int minor;                  //次设备号，若为 MISC_DYNAMIC_MINOR 自动分配
        const char *name;           //设备名
        const struct file_operations *fops; //设备文件操作结构体
        struct list_head list;          //misc_list链表头
        struct device *parent;
        struct device *this_device;
        const char *nodename;
        mode_t mode;
    };
```

为什么只有次设备号呢？misc类设备的主设备号都是10，通过次设备号来区分各个设备。内核将所有注册为misc的设备都归为一大类。

结构体中的list_head结构体类型的list成员的作用是什么呢？内核自己会维护一个misc_list链表，所有注册为misc的设备都必须挂在这个链表上，这个list就是该链表的链表头。

结构体中的两个device结构体类型指针作用是什么呢？作用就是创建设备文件，稍候就可以看到了！

我们如何定义自己的misc类型的设备呢？可如下定义：

```
static struct miscdevice misc = {
        .minor = MISC_DYNAMIC_MINOR,
        .name = DEVICE_NAME,
        .fops = &dev_fops,
    };
```

定义了自己的misc设备，使用如下两个函数向内核注册/注销设备：

```
    int misc_register(struct miscdevice * misc);    //在加载模块时会自动创建设备文件，是主
    int misc_deregister(struct miscdevice *misc);   //在卸载模块时会自动删除设备文件
```

至此，整个设备驱动的注册流程就完了，接下来深入了解一下misc设备模型的工作原理。

首先看看misc初始化函数：

```
static int __init misc_init(void)
{
    int err;

#ifdef CONFIG_PROC_FS
    /*如果使用proc文件系统，则创建misc项*/
    proc_create("misc", 0, NULL, &misc_proc_fops);
#endif
    /*在/sys/class/目录下创建一个名为misc的类*/
    misc_class = class_create(THIS_MODULE, "misc");
    err = PTR_ERR(misc_class);
    if (IS_ERR(misc_class))
        goto fail_remove;

    err = -EIO;
    /*咦，怎么misc设备驱动调用字符驱动的注册函数呢？设备的主设备号为MISC_MAJOR，为10*/
    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;
}
/*向内核注册misc子系统*/
subsys_initcall(misc_init);
```

接下来看看misc设备驱动的注册与注销函数：

注册函数：

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

        /*内核初始化一个链表头*/
        INIT_LIST_HEAD(&misc->list);

        mutex_lock(&misc_mtx);
        /*遍历已经注册的misc，如果和当前准备注册的相同(依据次设备号来判断),就返回设备忙*/
        list_for_each_entry(c, &misc_list, list) {
            if (c->minor == misc->minor) {
                mutex_unlock(&misc_mtx);
                return -EBUSY;
            }
```

```
        }

        /*动态分配设备的次设备号*/
        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);
        }

        /*使用固定的主设备号，动态分配的次设备号构造设备号*/
        dev = MKDEV(MISC_MAJOR, misc->minor);

        /*创建设备文件，这里就是使用miscdevice结构体中两个device类型指针的地方，(装置)(装置)
          当然，这是和linux设备驱动模型相关的*/
        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"(推翻)(推翻)(推翻)(
         * earlier defaults
         */
         /*到这一步也就注册成功了，将新注册的misc设备加入到内核维护的misc_list链表中*/
        list_add(&misc->list, &misc_list);
     out:
        mutex_unlock(&misc_mtx);
        return err;
    }
```

find_first_zero_bit 就是在位图中找第一个是0的位（256位的位图，如果有要分配的设备号，从低位到高位，依次设为1），表明这一位是可以分配设备号的。

注销函数：

```
int misc_deregister(struct miscdevice *misc)
{
    int i = DYNAMIC_MINORS - misc->minor - 1;

    if (WARN_ON(list_empty(&misc->list)))
        return -EINVAL;

    mutex_lock(&misc_mtx);
    /*删除链表节点*/
    list_del(&misc->list);
    /*销毁设备文件*/
    device_destroy(misc_class, MKDEV(MISC_MAJOR, misc->minor));
    if (i < DYNAMIC_MINORS && i >= 0)
        clear_bit(i, misc_minors);
    mutex_unlock(&misc_mtx);
    return 0;
}
```

open函数分析：

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

misc_fops中只有一个open函数，那其他的如read等操作是如何的工作的呢？

```
static int misc_open(struct inode * inode, struct file * file)
{
 int minor = iminor(inode);
 struct miscdevice *c;
 int err = -ENODEV;
 const struct file_operations *old_fops, *new_fops = NULL;

 mutex_lock(&misc_mtx);

        /*在misc_list链表中查找这个文件是否已经注册*/
 list_for_each_entry(c, &misc_list, list) {
  if (c->minor == minor) {
   new_fops = fops_get(c->fops);
   break;
  }
 }

        /*如果没找到则request_module，并且重新查找*/
 if (!new_fops) {
  mutex_unlock(&misc_mtx);
  request_module("char-major-%d-%d", MISC_MAJOR, minor);
  mutex_lock(&misc_mtx);

  list_for_each_entry(c, &misc_list, list) {
   if (c->minor == minor) {
    new_fops = fops_get(c->fops);
    break;
   }
  }
  if (!new_fops)
   goto fail;
 }

 err = 0;
 old_fops = file->f_op;
 file->f_op = new_fops;
 if (file->f_op->open) {
  file->private_data = c;
  err=file->f_op->open(inode,file);//这个函数是核心
  if (err) {          //定义new_fops和old_fops就是在这里发生错误时可以还原
   fops_put(file->f_op);
   file->f_op = fops_get(old_fops);
  }
 }
 fops_put(old_fops);
fail:
 mutex_unlock(&misc_mtx);
 return err;
}
```

file->f_op->open(inode,file) 是整个函数的重点，通过这句调用真正的open。open打开一个miscdevice的文件（就是最开始介绍的那个结构体），然后应用层就可以通过其中的fileoperation就可以进行read等操作。

misc设备驱动实例

这里贴一个简单的misc设备驱动程序，方便大家对照上面的理论部分进行分析，此驱动程序是X210开发板的蜂鸣器驱动程序，可以看看：

```
#define DEVICE_NAME     "buzzer"

//这里的宏定义可以放在头文件中，应用层程序只要包含这个头文件就可以调用这个宏定义了
#define PWM_IOCTL_SET_FREQ  1
#define PWM_IOCTL_STOP   0

static struct semaphore lock;

// 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

 //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);
 tcfg1 |= (0x4<<8);
 __raw_writel(tcfg1, S3C2410_TCFG1);

 clk_p = clk_get(NULL, "pclk");          //获得pclk时钟的结构体
 pclk  = clk_get_rate(clk_p);            //获得pclk时钟的频率

 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, s
   raw_writel(tcon, S3C2410_TCON);
```

```
 __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));
}

//这里的open主要功能是判断是否上锁
static int x210_pwm_open(struct inode *inode, struct file *file)
{
 if (!down_trylock(&lock))
  return 0;
 else
  return -EBUSY;

}

//关闭之前一定要记得解锁，否则这个资源将会一直被占用，其他人都用不了
static int x210_pwm_close(struct inode *inode, struct file *file)
{
 up(&lock);
 return 0;
}

// PWM:GPF14->PWM0，应用层将通过ioctl函数来调用这里的x210_pwm_ioctl函数来控制蜂鸣器
static int x210_pwm_ioctl(struct inode *inode, struct file *file, unsigned int cmd, ur
{
 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;
}


static struct file_operations dev_fops = {
    .owner   =   THIS_MODULE,
    .open    =   x210_pwm_open,
```

```
    .release =   x210_pwm_close,
    .ioctl   =   x210_pwm_ioctl,
};

static struct miscdevice misc = {
 .minor = MISC_DYNAMIC_MINOR,
 .name = DEVICE_NAME,
 .fops = &dev_fops,
};

static int __init dev_init(void)
{
 int ret;

 init_MUTEX(&lock);
 ret = misc_register(&misc);

 /* GPD0_2 (PWMTOUT2) */
 ret = gpio_request(S5PV210_GPD0(2), "GPD0");
 if(ret)
  printk("buzzer-x210: request gpio GPD0(2) fail");

 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);

 printk ("x210 "DEVICE_NAME" initialized\n");
     return ret;
}

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

module_init(dev_init);
module_exit(dev_exit);
MODULE_LICENSE("GPL");
```

下面介绍应用层如何来调用驱动层来控制蜂鸣器：

```c
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.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);
 sleep(3);
 ioctl(fd, PWM_IOCTL_STOP);
 sleep(3);
 ioctl(fd, PWM_IOCTL_SET_FREQ, 3000);
 sleep(3);
 ioctl(fd, PWM_IOCTL_STOP);
 sleep(3);

 close(fd);

 return 0;
}
```