misc_register

原创

方长存 于 2015-12-29 17:19:04 发布
分类专栏： C 文章标签： misc misc_register

在Linux系统中，存在一类字符设备，他们共享一个主设备号（10），但此设备号不同，我们称这类设备为混杂设备（miscdeivce），查看/proc/device中可以看到一个名为misc的主设备号为10.所有的混杂设备形成一个链表，对设备访问时内核根据次设备号找到对应的miscdevice设备。相对于普通字符设备驱动，它不需要自己去生成设备文件。

杂项设备（misc device）

杂项设备也是在嵌入式系统中用得比较多的一种设备驱动。在 Linux 内核的include\linux\miscdevice.h文件，要把自己定义的misc device从设备定义在这里。其实是因为这些字符设备不符合预先确定的字符设备范畴，所有这些设备采用主设备号10，一起归于misc device，其实misc_register就是用主设备号10调用register_chrdev()的。也就是说，misc设备其实也就是特殊的字符设备，可自动生成设备节点。

misc_device是特殊的字符设备。注册驱动程序时采用misc_register函数注册，此函数中会自动创建设备节点，即设备文件。无需mknod指令创建设备文件。因为misc_register()会调用class_device_create()或者device_create()。

字符设备(char device)

使用register_chrdev(LED_MAJOR,DEVICE_NAME,&dev_fops)注册字符设备驱动程序时，如果有多个设 备使用该函数注册驱动程序，LED_MAJOR不能相同，否则几个设备都无法注册(我已验证)。如果模块使用该方式注册并且 LED_MAJOR为0(自动分配主设备号)，使用insmod命令加载模块时会在终端显示分配的主设备号和次设备号，在/dev目录下建立该节点，比如 设备leds，如果加载该模块时分配的主设备号和次设备号为253和0，则建立节点:mknod leds c 253 0。使用register_chrdev (LED_MAJOR,DEVICE_NAME,&dev_fops)注册字符设备驱动程序时都要手动建立节点 ，否则在应用程序无法打开该设备。

阅读led驱动程序的代码的时候，没有发现ldd3中提到的各种字符设备注册函数，而是发现了一个misc_register函数，这说明led设备是作为杂项设备出现在内核中的，在内核中，misc杂项设备驱动接口是对一些字符设备的简单封装，他们共享一个主设备号，有不同的次设备号，共享一个open调用，其他的操作函数在打开后运用linux驱动程序的方法重载进行装载。

1. 主要数据结构

在Linux驱动中把无法归类的五花八门的设备定义为混杂设备(用miscdevice结构体表述)。miscdevice共享一个主设备号MISC_MAJOR(即10)，但次设备号不同。 所有的miscdevice设备形成了一个链表，对设备访问时内核根据次设备号查找对应的miscdevice设备，然后调用其file_operations结构中注册的文件操作接口进行操作。 在内核中用struct miscdevice表示miscdevice设备，然后调用其file_operations结构中注册的文件操作接口进行操作。miscdevice的API实现在drivers/char/misc.c中。

内核维护一个misc_list链表，misc设备在misc_register注册的时候链接到这个链表，在misc_deregister中解除链接。主要的设备结构就是miscdevice。定义如下：

```
 struct miscdevice  {
    int minor;                                  //次设备号
    const char *name;                           //设备的名称
    const struct file_operations *fops;         //文件操作
    struct list_head list;                      //misc_list的链表头
    struct device *parent;                      //父设备(Linux设备模型中的东东了，哈哈)
    struct device *this_device;                 //当前设备，是device_create的返回值，下边会看到
    const char *nodename;
    mode_t mode;
};
```

这个结构体是misc设备基本的结构体，在注册misc设备的时候必须要声明并初始化一个这样的结构体，但其中一般只需填充name minor fops字段就可以了。下面就是led驱动程序中初始化miscdevice的代码：

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

一般的时候在fops不用实现open方法，因为最初的方法misc_ops包含了open方法。其中minor如果填充MISC_DYNAMIC_MINOR，则是动态次设备号，次设备号由misc_register动态分配的。

scull 设备驱动只实现最重要的设备方法. 它的 file_operations 结构是如下初始化的:

```
struct file_operations scull_fops = {
     .owner =  THIS_MODULE,
     .llseek =  scull_llseek,
     .read =  scull_read,
     .write =  scull_write,
     .ioctl =  scull_ioctl,
     .open =  scull_open,
     .release =  scull_release,
   };
```

minor是这个混杂设备的次设备号，若由系统自动配置，则可以设置为MISC_DYNANIC_MINOR，name是设备名.使用时只需填写minor次设备号，*name设备名，*fops文件操作函数集即可。

Linux内核使用misc_register函数注册一个混杂设备，使用misc_deregister移除一个混杂设备。注册成功后，linux内核为自动为该设备创建设备节点，在/dev/下会产生相应的节点。

2. misc_init 函数

misc也是作为一个模块被加载到内核的，只不过是静态模块。这个函数是misc静态模块加载时的初始化函数。

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

#ifdef CONFIG_PROC_FS
    /*创建一个proc入口项*/
    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;
    /*注册设备，其中设备的主设备号为MISC_MAJOR，为10。设备名为misc，misc_fops是操作函数的集合；包含了open方法*/
    if (register_chrdev(MISC_MAJOR,"misc",&misc_fops))
        goto fail_printk;
    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;
}
/*misc作为一个子系统被注册到linux内核中*/
subsys_initcall(misc_init);
```

可以看出，这个初始化函数，最主要的功能就是注册字符设备 ，所用的注册接口是2.4内核的register_chrdev。它注册了主设备号为MISC_MAJOR，次设备号为0-255的256个设备。并且创建了一个misc类。

3. misc_register（）函数

misc_register()函数在misc.c中，最主要的功能是基于misc_class构造一个设备，将miscdevice结构挂载到misc_list列表上，并初始化与linux设备模型相关的结构，它的参数是miscdevice结构体。

```
int misc_register(struct miscdevice * misc)
{
    struct miscdevice *c;
    dev_t dev;
    int err = 0;
    /*初始化misc_list链表*/
    INIT_LIST_HEAD(&misc->list);
    mutex_lock(&misc_mtx);
    /*遍历misc_list链表，看这个次设备号以前有没有被用过，如果次设备号已被占有则退出*/
    list_for_each_entry(c, &misc_list, list) {
        if (c->minor == misc->minor) {
            mutex_unlock(&misc_mtx);
            return -EBUSY;
        }
    }
    /*看是否是需要动态分配次设备号*/
    if (misc->minor == MISC_DYNAMIC_MINOR) {
        /*
         *#define DYNAMIC_MINORS 64 /* like dynamic majors */
         *static unsigned char misc_minors[DYNAMIC_MINORS / 8];
         *这里存在一个次设备号的位图，一共64位。下边是遍历每一位，
         *如果这位为0，表示没有被占有，可以使用，为1表示被占用。
         */
        int i = DYNAMIC_MINORS;
        while (--i >= 0)
            if ( (misc_minors[i>>3] & (1 << (i&7))) == 0)
                break;
        if (i<0) {
            mutex_unlock(&misc_mtx);
            return -EBUSY;
        }
        /*得到这个次设备号*/
        misc->minor = i;
    }
    /*设置位图中相应位为1*/
    if (misc->minor < DYNAMIC_MINORS)
        misc_minors[misc->minor >> 3] |= 1 << (misc->minor & 7);
    /*计算出设备号*/
    dev = MKDEV(MISC_MAJOR, misc->minor);
    /*在/dev下创建设备节点，这就是有些驱动程序没有显式调用device_create，却出现了设备节点的原因*/
    misc->this_device = device_create(misc_class, misc->parent, dev, NULL,
                      "%s", misc->name);
    if (IS_ERR(misc->this_device)) {
        err = PTR_ERR(misc->this_device);
        goto out;
    }

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

可以看出，这个函数首先遍历misc_list链表，查找所用的次设备号是否已经被注册，防止冲突。如果是动态次设备号则分配一个，然后调用MKDEV生成设备号,从这里可以看出所有的misc设备共享一个主设备号MISC_MAJOR，然后调用device_create，生成设备文件。最后加入到misc_list链表中。

关于device_create，class_create 作用： class_create函数在misc.c中的模块初始化中被调用，现在一起说一下。这两个函数看起来很陌生，没有在ldd3中发现过，看源代码的时候发现class_create会调用底层组件__class_regsiter()是说明它是注册一个类。而device_create是创建一个设备，他是创建设备的便捷实现调用了device_register函数。他们都提供给linux设备模型使用，从linux内核2.6的某个版本之后，devfs不复存在，udev成为devfs的替代。相比devfs，udev有很多优势。

```
struct class *myclass = class_create(THIS_MODULE, “my_device_driver”);
class_device_create(myclass, NULL, MKDEV(major_num, 0), NULL, “my_device”);
```

这样就创建了一个类和设备，模块被加载时，udev daemon就会自动在/dev下创建my_device设备文件节点。这样就省去了自己创建设备文件的麻烦。这样也有助于动态设备的管理。
这个是miscdevice的卸载函数：

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

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

        mutex_lock(&misc_mtx);
        /*在misc_list链表中删除miscdevice设备*/
        list_del(&misc->list);
        /*删除设备节点*/
        device_destroy(misc_class, MKDEV(MISC_MAJOR, misc->minor));
        if (i < DYNAMIC_MINORS && i>0) {
            /*释放位图相应位*/
            misc_minors[i>>3] &= ~(1 << (misc->minor & 7));
        }
        mutex_unlock(&misc_mtx);
        return 0;
    }
```

4.下边是register_chrdev函数的实现：

```
int register_chrdev(unsigned int major, const char *name,
            const struct file_operations *fops)
{
    struct char_device_struct *cd;
    struct cdev *cdev;
    char *s;
    int err = -ENOMEM;
    /*主设备号是10，次设备号为从0开始，分配256个设备*/
    cd = __register_chrdev_region(major, 0, 256, name);
    if (IS_ERR(cd))
        return PTR_ERR(cd);
    /*分配字符设备*/
    cdev = cdev_alloc();
    if (!cdev)
        goto out2;

    cdev->owner = fops->owner;
    cdev->ops = fops;
    /*Linux设备模型中的,设置kobject的名字*/
    kobject_set_name(&cdev->kobj, "%s", name);
    for (s = strchr(kobject_name(&cdev->kobj),'/'); s; s = strchr(s, '/'))
        *s = '!';
    /*把这个字符设备注册到系统中*/
    err = cdev_add(cdev, MKDEV(cd->major, 0), 256);
    if (err)
        goto out;

    cd->cdev = cdev;

    return major ? 0 : cd->major;
out:
    kobject_put(&cdev->kobj);
out2:
    kfree(__unregister_chrdev_region(cd->major, 0, 256));
    return err;
}
```

来看看这个设备的操作函数的集合：

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

可以看到这里只有一个打开函数，用户打开miscdevice设备是通过主设备号对应的打开函数，在这个函数中找到次设备号对应的相应的具体设备的open函数。它的实现如下：

```
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;

    lock_kernel();
    mutex_lock(&misc_mtx);
    /*找到次设备号对应的操作函数集合，让new_fops指向这个具体设备的操作函数集合*/
    list_for_each_entry(c, &misc_list, list) {
        if (c->minor == minor) {
            new_fops = fops_get(c->fops);
            break;
        }
    }

    if (!new_fops) {
        mutex_unlock(&misc_mtx);
        /*如果没有找到，则请求加载这个次设备号对应的模块*/
        request_module("char-major-%d-%d", MISC_MAJOR, minor);
        mutex_lock(&misc_mtx);
        /*重新遍历misc_list链表，如果没有找到就退出，否则让new_fops指向这个具体设备的操作函数集合*/
        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;
    /*保存旧打开函数的地址*/
    old_fops = file->f_op;
    /*让主设备号的操作函数集合指针指向具体设备的操作函数集合*/
    file->f_op = new_fops;
    if (file->f_op->open) {
        /*使用具体设备的打开函数打开设备*/
        err=file->f_op->open(inode,file);
        if (err) {
            fops_put(file->f_op);
            file->f_op = fops_get(old_fops);
        }
    }
    fops_put(old_fops);
fail:
    mutex_unlock(&misc_mtx);
    unlock_kernel();
    return err;
}
```

5. 总结

总结一下miscdevice驱动的注册和卸载流程：

misc_register:

匹配次设备号->找到一个没有占用的次设备号(如果需要动态分配的话)->计算设号->创建设备文-

miscdevice结构体添加到misc_list链表中。

misc_deregister:

从mist_list中删除miscdevice->删除设备文件->位图位清零。

杂项设备作为字符设备的封装，为字符设备提供的简单的编程接口，如果编写新的字符驱动，可以考虑使用杂项设备接口，方便简单，只需要初始化一个miscdevice的结构，调用misc_register就可以了。系统最多有255个杂项设备，因为杂项设备模块自己占用了一个次设备号。可以发现，mini2440很多字符设备都是以杂项设备注册到内核的，如mini2440_buttons,mini2440_adc,mini2440_pwm等。