# 1. 前言与目录

## *1. Linux驱动开发入门与实战 前言*

摘要：《Linux驱动开发入门与实战》本书由浅入深，全面、系统地介绍了Linux驱动开发技术，提供了大量实例供读者实战演练。另外，作者在实例讲解中详细分析了各种重要的理论知识，让读者能够举一反三。本节为前言部分。 标签：Linux驱动开发
Linux驱动开发入门与实战
前言
Linux驱动程序开发是当前一个非常热门的领域，大多数基于Linux操作系统的嵌入式系统都需要编写驱动程序。随着嵌入式系统的广泛应用，出现了越来越多的硬件产品，必须有人不断地编写驱动使设备在Linux操作系统上工作。但是，Linux驱动程序开发相对较难，高水平的开发人员也比较少，所以导致驱动程序跟不上硬件发展的问题。基于这个原因，笔者编写了本书，希望借助本书能使驱动程序的开发更容易被开发人员所理解，从而迅速高效地开发出相关的驱动程序来。
笔者结合自己多年的Linux驱动程序开发经验和心得体会，花费了一年多的时间写作本书。希望各位读者能在本书的引领下跨入Linux驱动开发大门，并成为一名驱动程序开发高手。本书结合大量基础知识，全面、系统、深入地介绍了Linux驱动程序开发技术，并以大量实例贯穿于全书的讲解之中，使读者对驱动开发有一个深入的了解。学习完本书后，读者应该可以具备独立进行驱动程序开发的能力。
本书特色
1．多媒体语音视频讲解，高效、直观
笔者专门为本书重点内容录制了多媒体教学视频，便于高效直观地学习。这些视频和本书源代码需要读者自行下载。
2．最新内核，了解最新开发技术
本书基于Linux 2.6.29内核，这是目前较新的一个内核。该内核包含了大多数常用的驱动程序，便于学习和移植。
3．内容全面、系统、深入
本书介绍了Linux驱动开发的基础知识、核心技术和一些驱动程序开发实例。内容的安排上力求全面、系统。在实例的选择上力求深入。
4．讲解由浅入深、循序渐进，适合各个层次的读者阅读
本书从Linux驱动程序开发的基础开始讲解，逐步深入到Linux驱动的高级开发技术及应

用，内容安排从易到难，讲解由浅入深、循序渐进，适合各个层次的读者阅读。
5．贯穿大量的开发实例和技巧，迅速提升开发水平
本书在讲解知识点时穿插了大量驱动程序的典型实例，并给出了大量的开发技巧，以便让读者更好地理解各种概念和开发技术，体验实际编程，迅速提高开发水平。
6．从工程应用出发，具有很强的实用性
本书详细介绍多个驱动开发实例。通过这些应用实例，可以提高读者的驱动开发水平，从而具备独立进行驱动程序开发的能力。
本书内容及知识体系
第1篇　Linux驱动开发基础（第1～6章）
本篇主要内容包括：Linux驱动开发概述、嵌入式处理器和开发板、构建嵌入式驱动程序开发环境、构建嵌入式Linux操作系统、构建第一个驱动程序、简单的字符设备驱动程序。通过对本篇内容的学习，读者可以掌握Linux驱动开发的基本概念和基本环境。
第2篇　Linux驱动开发核心技术（第7～10章）
本篇主要内容包括：设备驱动中的并发控制、设备驱动中的阻塞和同步机制、中断与时钟机制、内存访问等内容。通过本篇的学习，读者可以掌握Linux驱动开发的基础知识和核心技术。
第3篇　Linux驱动开发应用实战（第11～19章）
本篇主要内容包括：设备驱动模型、RTC实时时钟驱动程序、看门狗驱动程序、IIC设备驱动程序、LCD设备驱动程序、触摸屏设备驱动程序、输入子系统驱动程序、块设备驱动程序、USB设备驱动程序等。通过对本篇内容的学习，读者可以掌握编写各种设备驱动程序的方法。
本书读者对象
Linux内核爱好者；
想学习Linux驱动开发的入门人员；
Linux驱动程序专业开发人员；
嵌入式工程师；
大中专院校的学生；
社会培训班的学员；
需要了解驱动程序开发的技术人员。
本书作者及编委会成员
本书由郑强主笔编写。其他参与编写的人员有毕梦飞、蔡成立、陈涛、陈晓莉、陈燕、崔栋栋、冯国良、高岱明、黄成、黄会、纪奎秀、江莹、靳华、李凌、李胜君、李雅娟、刘大林、刘惠萍、刘水珍、马月桂、闵智和、秦兰、汪文君、文龙。在此一并表示感谢。
本书编委会成员有欧振旭、陈杰、陈冠军、顼宇峰、张帆、陈刚、程彩红、毛红娟、聂庆亮、王志娟、武文娟、颜盟盟、姚志娟、尹继平、张昆、张薛。
本书技术支持
您在阅读本书的过程中若碰到什么问题，请通过以下方式联系我们，我们会及时地答复您。

E-mail：bookservice2008@163.com（编辑） QQ:917603226
论坛网址：http://www.wanjuanchina.net
编著者

## 2. Linux驱动开发入门与实战 目录

摘要：《Linux驱动开发入门与实战》本书由浅入深，全面、系统地介绍了Linux驱动开发技术，提供了大量实例供读者实战演练。另外，作者在实例讲解中详细分析了各种重要的理论知识，让读者能够举一反三。本节为目录部分。 标签：Linux驱动开发
Linux驱动开发入门与实战
目录

# 2. 第6章　简单的字符设备驱动程序

## *1. 6.1.1 字符设备和块设备*

摘要：《Linux驱动开发入门与实战》第6章简单的字符设备驱动程序，本章首先从整体上介绍字符设备的框架结构，然后介绍字符设备结构体struct cdev，接着介绍字符设备的组成，最后详细讲解一个VirtualDisk字符设备驱动程序。本节为大家介绍字符设备和块设备。 标签：字符设备　块设备　Linux驱动开发　Linux驱动开发入门与实战

第6章　简单的字符设备驱动程序

在Linux设备驱动程序的家族中，字符设备驱动程序是较为简单的驱动程序，同时也是应用非常广泛的驱动程序。所以学习字符设备驱动程序，对构建Linux设备驱动程序的知识结构非常重要。本章将带领读者编写一个完整的字符设备驱动程序。

6.1　字符设备驱动程序框架

本节对字符设备驱动程序框架进行了简要的分析。字符设备驱动程序中有许多非常重要的概念，下面将从最简单的概念讲起：字符设备和块设备。

6.1.1　字符设备和块设备

Linux系统将设备分为3种类型：字符设备、块设备和网络接口设备。其中字符设备和块设备难以区分，下面将对其进行重要讲解。

1．字符设备

字符设备是指那些只能一个字节一个字节读写数据的设备，不能随机读取设备内存中的某一数据。其读取数据需要按照先后顺序，从这点来看，字符设备是面向数据流的设备。常见的字符有鼠标、键盘、串口、控制台和LED等设备。

2．块设备

块设备是指那些可以从设备的任意位置读取一定长度数据的设备。其读取数据不必按照先后顺序，可以定位到设备的某一具体位置，读取数据。常见的块设备有硬盘、磁盘、U盘、SD卡等。

3．字符设备和块设备的区分

每一个字符设备或者块设备都在/dev目录下对应一个设备文件。读者可以通过查看/dev目录下的文件的属性，来区分设备是字符设备还是块设备。使用cd命令进入/dev目录，并执行ls -l命令就可以看到设备的属性。[root@tom /]# cd /dev

/*进入/dev目录*/　[root@tom dev]# ls -l

/*列出/dev中文件的信息*/、 /*第1字段 2 3 4 5 6 7 8 */ crw-rw----+ 1 root root 14, 12 12-21 22:56 adsp crw------- 1 root root 10, 175 12-21 22:56 agpgart crw-rw----+ 1 root root 14, 4 12-21 22:56 audio brw-r----- 1 root disk 253, 0 12-21 22:56 dm-0 brw-r----- 1 root disk 253, 1 12-21 22:56 dm-1 crw-rw---- 1 root root 14, 9 12-21 22:56 dmmidi

ls -l命令的第一字段中的第一个字符c表示设备是字符设备，b表示设备是块设备。第234字段对驱动程序开发来说没有关系。第5，6字段分别表示设备的主设备号和次设备号，将在6.1.2节讲解。第7字段表示文件的最后修改时间。第8字段表示设备的名字。由第1和8字段可知，adsp是字符设备，dm-0是块设备。其中adsp设备的主设备号是14，次设备号是12。

## *2. 6.1.2 主设备号和次设备号*

摘要：《Linux驱动开发入门与实战》第6章简单的字符设备驱动程序，本章首先从整体上介绍字符设备的框架结构，然后介绍字符设备结构体struct cdev，接着介绍字符设备的组成，最后详细讲解一个VirtualDisk字符设备驱动程序。本节为大家介绍主设备号和次设备号。 标签：主设备号 次设备号 Linux驱动开发 Linux驱动开发入门与实战

6.1.2 主设备号和次设备号

一个字符设备或者块设备都有一个主设备号和次设备号。主设备号和次设备号统称为设备号。主设备号用来表示一个特定的驱动程序。次设备号用来表示使用该驱动程序的各设备。例如一个嵌入式系统，有两个LED指示灯，LED灯需要独立的打开或者关闭。那么，可以写一个LED灯的字符设备驱动程序，可以将其主设备号注册成5号设备，次设备号分别为1和2。这里，次设备号就分别表示两个LED灯。

1．主设备号和次设备号的表示

在Linux内核中，dev_t类型用来表示设备号。在Linux 2.6.29.4中，dev_t定义为一个

无符号长整型变量，如下：typedef u_long dev_t;
u_long在32位机中是4个字节，在64位机中是8字节。以32位机为例，其中高12表示主设备号，低20为表示次设备号，如图6.1所示。
图6.1 dev_t结构
2．主设备号和次设备号的获取
为了写出可移植的驱动程序，不能假定主设备号和次设备号的位数。不同的机型中，主设备号和次设备号的位数可能是不同的。应该使用MAJOR宏得到主设备号，使用MINOR宏来得到次设备号。下面是两个宏的定义：

```
#define MINORBITS   20                                  /*次设备号位数*/
#define MINORMASK   ((1U << MINORBITS) - 1)             /*次设备号掩码*/
#define MAJOR(dev)  ((unsigned int) ((dev) >> MINORBITS))
                        /*dev右移20位得到主设备号*/  #define MINOR(dev)
((unsigned int) ((dev) & MINORMASK))
     /*与次设备掩码与，得到次设备号*/
```

MAJOR宏将dev_t向右移动20位，得到主设备号；MINOR宏将dev_t的高12位清零，得到次设备号。相反，可以将主设备号和次设备号转换为设备号类型（dev_t），使用宏MKDEV可以完成这个功能。#define MKDEV(ma,mi) (((ma) << MINORBITS) | (mi))
MKDEV宏将主设备号（ma）左移20位，然后与次设备号（mi）相与，得到设备号。
3．静态分配设备号
静态分配设备号，就是驱动程序开发者，静态地指定一个设备号。对于一部分常用的设备，内核开发者已经为其分配了设备号。这些设备号可以在内核源码documentation/devices.txt文件中找到。如果只有开发者自己使用这些设备驱动程序，那么其可以选择一个尚未使用的设备号。在不添加新硬件的时候，这种方式不会产生设备号冲突。但是当添加新硬件时，则很可能造成设备号冲突，影响设备的使用。
4．动态分配设备号
由于静态分配设备号存在冲突的问题，所以内核社区建议开发者使用动态分配设备号的方法。动态分配设备号的函数是alloc_chrdev_region()，该函数将在"申请和释放设备号"一节讲述。
5．查看设备号
当静态分配设备号时，需要查看系统中已经存在的设备号，从而决定使用哪个新设备号。可以读取/proc/devices文件获得设备的设备号。/proc/devices文件包含字符设备和块设备的设备号，如下所示。[root@tom /]# cat /proc/devices /*cat命令查看/proc/devices文件的内容*/  Character devices:  /*字符设备*/
  1 mem    4 /dev/vc/0    7 vcs    13 input    14 sound    21 sg  Block devices:  /*块设备*/    1 ramdisk    2 fd    8 sd    253 device-mapper    254 mdp

## *3. 6.1.3 申请和释放设备号*

摘要：《Linux驱动开发入门与实战》第6章简单的字符设备驱动程序，本章首先从整体上介绍字符设备的框架结构，然后介绍字符设备结构体struct cdev，接着介绍字符设备的组成，最后详细讲解一个VirtualDisk字符设备驱动程序。本节为大家介绍申请和释放设备号。 标签：释放设备号 Linux驱动开发 Linux驱动开发入门与实战

6.1.3 申请和释放设备号

内核维护着一个特殊的数据结构，用来存放设备号与设备的关系。在安装设备时，应该给设备申请一个设备号，使系统可以明确设备对应的设备号。设备驱动程序中的很多功能，是通过设备号来操作设备的。下面，首先对申请设备号进行简述。

1. 申请设备号

在构建字符设备之前，首先要向系统申请一个或者多个设备号。完成该工作的函数是register_chrdev_region()，该函数在<fs/char_dev.c>中定义：

```
int register_chrdev_region(dev_t from, unsigned count, const char *name);
```

其中，from是要分配的设备号范围的起始值。一般只提供from的主设备号，from的次设备号通常被设置成0。count是需要申请的连续设备号的个数。最后name是和该范围编号关联的设备名称，该名称不能超过64字节。

和大多数内核函数一样，register_chrdev_region()函数成功时返回0。错误时，返回一个负的错误码，并且不能为字符设备分配设备号。下面是一个例子代码，其申请了CS5535_GPIO_COUNT个设备号。retval = register_chrdev_region(dev_id, CS5535_GPIO_COUNT,NAME);

在Linux中有非常多的字符设备，在人为的为字符设备分配设备号时，很可能发生冲突。Linux内核开发者一直在努力将设备号变为动态的。可以使用alloc_chrdev_region()函数达到这个目的。

```
int alloc_chrdev_region(dev_t *dev, unsigned baseminor,
unsigned count, const char *name)
```

在上面的函数中，dev作为输出参数，在函数成功返回后将保存已经分配的设备号。函数有可能申请一段连续的设备号，这是dev返回第一个设备号。baseminor表示要申请的第一个次设备号，其通常设为0。count和name与register_chrdev_region()函数的对应参数一样。count表示要申请的连续设备号个数，name表示设备的名字。下面是一个例

子代码，其申请了CS5535_GPIO_COUNT个设备号。retval = alloc_chrdev_region(&dev_id, 0, CS5535_GPIO_COUNT, NAME);
2. 释放设备号
使用上面两种方式申请的设备号，都应该在不使用设备时，释放设备号。设备号的释放统一使用下面的函数：void unregister_chrdev_region(dev_t from, unsigned count);
在上面这个函数中，from表示要释放的设备号，count表示从from开始要释放的设备号个数。通常，在模块的卸载函数中调用unregister_chrdev_region()函数。

## *4. 6.2.1 cdev结构体*

摘要：《Linux驱动开发入门与实战》第6章简单的字符设备驱动程序，本章首先从整体上介绍字符设备的框架结构，然后介绍字符设备结构体struct cdev，接着介绍字符设备的组成，最后详细讲解一个VirtualDisk字符设备驱动程序。本节为大家介绍cdev结构体。 标签：cdev结构体 Linux驱动开发 Linux驱动开发入门与实战
6.2 初识cdev结构
当申请字符设备的设备号后，这时，需要将字符设备注册到系统中，才能使用字符设备。为了理解这个实现过程，首先解释一下cdev结构体。
6.2.1 cdev结构体
在Linux内核中使用cdev结构体描述字符设备。该结构体是所有字符设备的抽象，其包含了大量字符设备所共有的特性。cdev结构体定义如下：struct cdev { struct kobject kobj; /*内嵌的kobject结构，用于内核设备驱动模型的管理*/ struct module *owner;
/*指向包含该结构的模块的指针，用于引用计数*/ const struct file_operations *ops; /*指向字符设备操作函数集的指针*/ struct list_head list; /*该结构将使用该驱动的字符设备连接成一个链表*/ dev_t dev; /*该字符设备的起始设备号，一个设备可能有多个设备号*/ unsigned int count; /*使用该字符设备驱动的设

备数量*/ };
cdev结构中的kobj结构用于内核管理字符设备，驱动开发人员一般不使用该成员。
ops是指向file_operations结构的指针，该结构定义了操作字符设备的函数。由于此结构体较为复杂，所以将在6.2.2 file_operations结构体一节讲解。
dev就是用来存储字符设备所申请的设备号。count表示目前有多少个字符设备在使用该驱动程序。当使用rmmod卸载模块时，如果count成员不为0，那么系统不允许卸载模块。
list结构是一个双向链表，用于将其他结构体连接成一个双向链表。该结构在Linux内核中广泛使用，需要读者掌握。
struct list_head { struct list_head *next, *prev; }; 图6.2 cdev与inode的关系
如图6.2所示，cdev结构体的list成员连接到了inode结构体i_devices成员。其中i_devices也是一个list_head结构。这样，使cdev结构与inode结点组成了一个双向链表。inode结构体表示/dev目录下的设备文件，该结构体较为复杂，所以将在下面讲述。
每一个字符设备在/dev目录下都有一个设备文件，打开设备文件就相当于打开相应的字符设备。例如应用程序打开设备文件A，那么系统会产生一个inode结点。这样可以通过inode结点的i_cdev字段找到cdev字符结构体。通过cdev的ops指针，就能找到设备A的操作函数。对操作函数的讲解，将放在后面的内容中。

## 5. 6.2.2 file_operations结构体

摘要：《Linux驱动开发入门与实战》第6章简单的字符设备驱动程序，本章首先从整体上介绍字符设备的框架结构，然后介绍字符设备结构体struct cdev，接着介绍字符设备的组成，最后详细讲解一个VirtualDisk字符设备驱动程序。本节为大家介绍file_operations结构体。 标签：file_operations结构体 Linux驱动开发 Linux驱动开发入门与实战

6.2.2 file_operations结构体

file_operations是一个对设备进行操作的抽象结构体。Linux内核的设计非常巧妙。内核允许为设备建立一个设备文件，对设备文件的所有操作，就相当于对设备的操作。这样的好处是，用户程序可以使用访问普通文件的方法访问设备文件，进而访问设备。这样的方法，极大地减轻了程序员的编程负担，程序员不必去熟悉新的驱动接口，就能够访问设备。

对普通文件的访问常常使用open()、read()、write()、close()、ioctl()等方法。同样对设备文件的访问，也可以使用这些方法。这些调用最终会引起对file_operations结构体中对应函数的调用。对于程序员来说，只要为不同的设备编写不同的操作函数就可以了。

为了增加file_operations的功能，所以将很多函数集中在了该结构中。该结构的定义目前已经比较庞大了，其定义如下：struct file_operations { struct module *owner; loff_t (*llseek) (struct file *, loff_t, int); ssize_t (*read) (struct file *, char __user *, size_t, loff_t *); ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *); ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t); ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t); int (*readdir) (struct file *, void *, filldir_t); unsigned int (*poll) (struct file *, struct poll_table_struct *); int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long); long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long); long (*compat_ioctl) (struct file *, unsigned int, unsigned long); int (*mmap) (struct file *, struct vm_area_struct *); int (*open) (struct inode *, struct file *); int (*flush) (struct file *, fl_owner_t id); int (*release) (struct inode *, struct file *); int (*fsync) (struct file *, struct dentry *, int datasync); int (*aio_fsync) (struct kiocb *, int datasync); int (*fasync) (int, struct file *, int); int (*lock) (struct file *, int, struct file_lock *); ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int); unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long); int (*check_flags)(int); int (*flock) (struct file *, int, struct file_lock *); ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int); ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int); int

(*setlease)(struct file *, long, struct file_lock **); };
下面对file_operations结构体的重要成员进行讲解。
owner成员根本不是一个函数；它是一个指向拥有这个结构模块的指针。这个成员用来维持模块的引用计数，当模块还在使用时，不能用rmmod卸载模块。几乎所有时刻，它被简单初始化为 THIS_MODULE，一个在<linux/module.h>中定义的宏。
llseek()函数用来改变文件中的当前读/写位置，并将新位置返回。loff_t参数是一个"long long"类型，"long long"类型即使在32位机上也是64位宽。这是为了与64位机兼容而定的，因为64位机的文件大小完全可以突破4G。
read()函数用来从设备中获取数据，成功时函数返回读取的字节数，失败时返回一个负的错误码。
write()函数用来写数据到设备中。成功时该函数返回写入的字节数，失败时返回一个负的错误码。
ioctl()函数提供了一种执行设备特定命令的方法。例如使设备复位，这既不是读操作也不是写操作，不适合用read()和write()方法来实现。如果在应用程序中给ioctl传入没有定义的命令，那么将返回-ENOTTY的错误，表示该设备不支持这个命令。
open()函数用来打开一个设备，在该函数中可以对设备进行初始化。如果这个函数被复制NULL，那么设备打开永远成功，并不会对设备产生影响。
release()函数用来释放open()函数中申请的资源，将在文件引用计数为0时，被系统调用。其对应应用程序的close()方法，但并不是每一次调用close()方法，都会触发release()函数，在对设备文件的所有打开都释放后，才会被调用。

## *6. 6.2.3 cdev和file_operations结构体的*

摘要：《Linux驱动开发入门与实战》第6章简单的字符设备驱动程序，本章首先从整体上介绍字符设备的框架结构，然后介绍字符设备结构体struct cdev，接着介绍字符设备的组成，最后详细讲解一个VirtualDisk字符设备驱动程序。本节为大家介绍cdev和file_operations结构体的关系。 标签：cdev结构体 Linux驱动开发 Linux驱

动开发入门与实战
6.2.3　cdev和file_operations结构体的关系
一般来说，驱动开发人员会将特定设备的特定数据放到cdev结构体后，组成一个新的结构体。如图6.3所示，"自定义字符设备"中就包含特定设备的数据。该"自定义设备"中有一个cdev结构体。cdev结构体中有一个指向file_operations的指针。这里，file_operations中的函数就可以用来操作硬件，或者"自定义字符设备"中的其他数据，从而起到控制设备的作用。
（点击查看大图）图6.3　cdev与file_operations结构体的关系

## 7. 6.2.4 inode结构体

摘要：《Linux驱动开发入门与实战》第6章简单的字符设备驱动程序，本章首先从整体上介绍字符设备的框架结构，然后介绍字符设备结构体struct cdev，接着介绍字符设备的组成，最后详细讲解一个VirtualDisk字符设备驱动程序。本节为大家介绍inode结构体。 标签：inode结构体　Linux驱动开发　Linux驱动开发入门与实战
6.2.4　inode结构体
内核使用inode结构在内部表示文件。inode一般作为file_operations结构中函数的参数传递过来。例如，open()函数将传递一个inode指针进来，表示目前打开的文件结点。需要注意的是，inode的成员已经被系统赋予了合适的值，驱动程序只需要使用该结点中的信息，而不用更改。Oepn()函数为：int (*open) (struct inode *, struct file *);
inode结构中包含大量的有关文件的信息。这里，只对编写驱动程序有用的字段进行介绍，对于该结构更多的信息，可以参看内核源码。
dev_t i_rdev，表示设备文件对应的设备号。
struct list_head i_devices，如图6.2所示，该成员使设备文件连接到对应的cdev结构，从而对应到自己的驱动程序。
struct cdev *i_cdev，如图6.2所示，该成员也指向cdev设备。
除了从dev_t得到主设备号和次设备号外，这里还可以使用imajor()和iminor()函数从

i_rdev中得到主设备号和次设备号。
imajor()函数在内部调用MAJOR宏，如下代码所示。static inline unsigned
imajor(const struct inode *inode) { return MAJOR(inode->i_rdev);
/*从inode->i_rdev中提取主设备号*/ }
同样，iminor()函数在内部调用MINOR宏，如下代码所示。static inline unsigned
iminor(const struct inode *inode) { return MINOR(inode->i_rdev); ;
/*从inode->i_rdev中提取次设备号*/ }

## *8. 6.3.1 字符设备加载和卸载函数*

摘要：《Linux驱动开发入门与实战》第6章简单的字符设备驱动程序，本章首先从整体上介绍字符设备的框架结构，然后介绍字符设备结构体struct cdev，接着介绍字符设备的组成，最后详细讲解一个VirtualDisk字符设备驱动程序。本节为大家介绍字符设备加载和卸载函数。 标签：字符设备 Linux驱动开发 Linux驱动开发入门与实战
6.3 字符设备驱动的组成
了解字符设备驱动程序的组成，对编写驱动程序非常有用。因为字符设备在结构上都有很多相似的地方，所以只要会编写一个字符设备驱动程序，那么相似的字符设备驱动程序的编写，就不难了。在Linux系统中，字符设备驱动程序由以下几个部分组成。
6.3.1 字符设备加载和卸载函数
在字符设备的加载函数中，应该实现字符设备号的申请和cdev的注册。相反，在字符设备的卸载函数中应该实现字符设备号的释放和cdev的注销。
cdev是内核开发者对字符设备的一个抽象。除了cdev中的信息外，特定的字符设备还需要特定的信息，常常将特定的信息放在cdev之后，形成一个设备结构体，如代码中的xxx_dev。
常见的设备结构体、加载函数和卸载函数如下面的代码：struct xxx_dev
/*自定义设备结构体*/ { struct cdev cdev;
/*cdev结构体*/ ...
/*特定设备的特定数据*/ }; static int __init xxx_init(void)

```
/*设备驱动模块加载函数*/ {  ...  /* 申请设备号，当xxx_major不为0时，表示静态指定；当为0时，表示动态申请*/  if (xxx_major)  result = register_chrdev_region(xxx_devno, 1, "DEV_NAME");  /*静态申请设备号*/  else  /*动态申请设备号*/  {  result = alloc_chrdev_region(&xxx_devno, 0, 1, " DEV_NAME ");  xxx_major = MAJOR(xxx_devno);  /*获得申请的主设备号*/  } /*初始化cdev结构，并传递file_operations结构指针*/  cdev_init(&xxx_dev.cdev, &xxx_fops);  dev->cdev.owner = THIS_MODULE;  /*指定所属模块*/  err = cdev_add(&xxx_dev .cdev, xxx_devno, 1);  /*注册设备*/ } static void __exit xxx_exit(void)  /*模块卸载函数*/ {  cdev_del(&xxx_dev.cdev);  /*注销cdev*/  unregister_chrdev_region(xxx_devno, 1);  /*释放设备号*/ }
```

## *9. 6.3.2 file_operations结构体和其成员*

摘要：《Linux驱动开发入门与实战》第6章简单的字符设备驱动程序，本章首先从整体上介绍字符设备的框架结构，然后介绍字符设备结构体struct cdev，接着介绍字符设备的组成，最后详细讲解一个VirtualDisk字符设备驱动程序。本节为大家介绍file_operations结构体和其成员函数。 标签：file_operations结构体 Linux驱动开发 Linux驱动开发入门与实战

6.3.2 file_operations结构体和其成员函数

file_operations结构体中的成员函数都对应着驱动程序的接口，用户程序可以通过内核来调用这些接口，从而控制设备。大多数字符设备驱动都会实现read()、write()和ioctl()函数，这3个函数的常见写法如下面的代码所示。

```
/*文件操作结构体*/ static const struct file_operations xxx_fops = { .owner = THIS_MODULE, /*模块引用，任何时候都赋值THIS_MODULE */ .read = xxx_read,
```

```
/*指定设备的读函数 */    .write = xxx_write,          /*指定设备的写函数 */
   .ioctl = xxx_ioctl,          /*指定设备的控制函数 */  };  /*读函数*/
static ssize_t xxx_read(struct file *filp, char __user *buf,
size_t size,loff_t *ppos)  {    ...  if(size>8)
copy_to_user(buf,...,...);  /*当数据较大时，使用copy_to_user()，效率较高*/
esle  put_user(...,buf);            /*当数据较小时，使用put_user()，效率较
高*/  ...  }  /*写函数*/  static ssize_t xxx_write(struct file *filp, const
char __user *buf,size_t size, loff_t *ppos)  {    ...  if(size>8)
copy_from_user(..., buf,...);   /*当数据较大时，使用copy_to_user()，效率较高
*/  else      get_user(..., buf);         /*当数据较小时，使用put_user()，效
率较高*/  ...  }  /* ioctl设备控制函数 */  static long xxx_ioctl(struct file
*file, unsigned int cmd,
unsigned long arg)  {    ...    switch (cmd)    {      case xxx_cmd1:  ...
                          /*命令1执行的操作*/       break;      case
xxx_cmd1:  ...                                  /*命令2执行的操作*/
break;      default:       return  - EINVAL;         /*内核和驱动程序都不支
持该命令时，
返回无效的命令*/    }    return 0;  }
```

文件操作结构体xxx_fops中保存了操作函数的指针。对于没有实现的函数，被赋值为NULL。xxx_fops结构体在字符设备加载函数中，作为cdev_init()的参数，与cdev建立了关联。

设备驱动的read()和write()函数有同样的参数。filp是文件结构体的指针，指向打开的文件。buf是来自用户空间的数据地址，该地址不能在驱动程序中直接读取。size是要读的字节。ppos是读写的位置，其相对于文件的开头。

xxx_ioctl控制函数的cmd参数是事先定义的I/O控制命令，arg对应该命令的参数。

### *10. 6.3.3 驱动程序与应用程序的数据交换*

摘要：《Linux驱动开发入门与实战》第6章简单的字符设备驱动程序，本章首先从整体上介绍字符设备的框架结构，然后介绍字符设备结构体struct cdev，接着介绍字符设备的组成，最后详细讲解一个VirtualDisk字符设备驱动程序。本节为大家介绍驱动程序与应用程序的数据交换。 标签：驱动程序 Linux驱动开发 Linux驱动开发入门与实战

6.3.3 驱动程序与应用程序的数据交换

驱动程序和应用程序的数据交换是非常重要的。file_operations中的read()和write()函数，就是用来在驱动程序和应用程序间交换数据的。通过数据交换，驱动程序和应用程序可以彼此了解对方的情况。但是驱动程序和应用程序属于不同的地址空间。驱动程序不能直接访问应用程序的地址空间；同样应用程序也不能直接访问驱动程序的地址空间，否则会破坏彼此空间中的数据，从而造成系统崩溃，或者数据损坏。

安全的方法是使用内核提供的专用函数，完成数据在应用程序空间和驱动程序空间的交换。这些函数对用户程序传过来的指针进行了严格的检查和必要的转换，从而保证用户程序与驱动程序交换数据的安全性。这些函数有：unsigned long copy_to_user(void __user *to,

const void *from, unsigned long n); unsigned long copy_from_user(void *to, const

void __user *from, unsigned long n); put_user(local,user);

get_user(local,user);

## *11. 6.3.4 字符设备驱动程序组成小结*

摘要：《Linux驱动开发入门与实战》第6章简单的字符设备驱动程序，本章首先从整体上介绍字符设备的框架结构，然后介绍字符设备结构体struct cdev，接着介绍字符设备的组成，最后详细讲解一个VirtualDisk字符设备驱动程序。本节为字符设备驱动程序组成小结。 标签：字符设备 Linux驱动开发 Linux驱动开发入门与实战

6.3.4 字符设备驱动程序组成小结

字符设备是3大类设备（字符设备、块设备、网络设备）中较简单的一类设备，其驱动

程序中完成的主要工作是初始化、添加和删除cdev结构体，申请和释放设备号，以及填充file_operation结构体中操作函数，并实现file_operations结构体中的read()、write()、ioctl()等重要函数。如图6.4所示为cdev结构体、file_operations和用户空间调用驱动的关系。　（点击查看大图）图6.4　字符设备与用户空间关系

## *12. 6.4.1 VirtualDisk的头文件、宏和设备*

摘要：《Linux驱动开发入门与实战》第6章简单的字符设备驱动程序，本章首先从整体上介绍字符设备的框架结构，然后介绍字符设备结构体struct cdev，接着介绍字符设备的组成，最后详细讲解一个VirtualDisk字符设备驱动程序。本节为大家介绍VirtualDisk的头文件、宏和设备结构体。 标签：Linux驱动开发　Linux驱动开发入门与实战

6.4　VirtualDisk字符设备驱动

从本节开始，后续的几节都将以一个VirtualDisk设备为蓝本进行讲解。VirtualDisk是一个虚拟磁盘设备，在这个虚拟磁盘设备中分配了8K的连续内存空间，并定义了两个端口数据（port1和port2）。驱动程序可以对设备进行读写、控制和定位操作，用户空间的程序可以通过Linux系统调用访问VirtualDisk设备中的数据。

6.4.1　VirtualDisk的头文件、宏和设备结构体

VirtualDisk驱动程序应该包含必要的头文件和宏信息，并定义一个与实际设备相对应的设备结构体，相关的定义如下面的代码所示。01　#include <linux/module.h> 02　#include <linux/types.h> 03　#include <linux/fs.h> 04　#include <linux/errno.h> 05　#include <linux/mm.h> 06　#include <linux/sched.h> 07　#include <linux/init.h> 08　#include <linux/cdev.h> 09　#include <asm/io.h> 10　#include <asm/system.h> 11　#include <asm/uaccess.h> 12　#define VIRTUALDISK_SIZE　0x2000　/*全局内存最大8K字节*/　13　#define MEM_CLEAR 0x1　/*全局内存清零*/　14　#define PORT1_SET 0x2　/*将port1端口清零*/　15　#define PORT2_SET 0x3　/*将port2端口清零*/　16　#define VIRTUALDISK_MAJOR 200　/*预设的

VirtualDisk的主设备号为200*/ 17 static int VirtualDisk_major = VIRTUALDISK_MAJOR; 18 /*VirtualDisk设备结构体*/ 19 struct VirtualDisk 20 { 21 struct cdev cdev; /*cdev结构体*/ 22 unsigned char mem[VIRTUALDISK_SIZE]; /*全局内存8K*/ 23 int port1; /*两个不同类型的端口*/ 24 long port2; 25 long count; /*记录设备
目前被多少设备打开*/ 26 };
从第01～11行列出了必要的头文件，这些头文件中包含驱动程序可能使用的函数。
从第19～26行代码，定义了VirtualDisk设备结构体。其中包含了cdev字符设备结构体，和一块连续的8K的设备内存。另外定义了两个端口port1和port2，用来模拟实际设备的端口。count表示设备被打开的次数。在驱动程序中，可以不将这些成员放在一个结构中，但放在一起的好处是借助了面向对象的封装思想，将设备相关的成员封装成了一个整体。
第22行定义了一个8K的内存块，驱动程序中一般不静态的分配内存，因为静态分配的内存的生命周期非常长，随着驱动程序生和死。而驱动程序一般运行在系统的整个开机状态中，所以驱动程序分配的内存，一直不会得到释放。所以，编写驱动程序，应避免申请大块内存和静态分配内存。这里，只是为了演示方便，所以分配了静态内存。

## *13. 6.4.2 加载和卸载驱动程序*

摘要：《Linux驱动开发入门与实战》第6章简单的字符设备驱动程序，本章首先从整体上介绍字符设备的框架结构，然后介绍字符设备结构体struct cdev，接着介绍字符设备的组成，最后详细讲解一个VirtualDisk字符设备驱动程序。本节为大家介绍加载和卸载驱动程序。 标签：驱动程序 Linux驱动开发 Linux驱动开发入门与实战
6.4.2 加载和卸载驱动程序
第6.3节已经对字符设备驱动程序的加载和卸载模板进行了介绍。VirtualDisk的加载和卸载函数也和6.3节介绍的类似，其实现代码如下：
01 /*设备驱动模块加载函数*/ 02 int VirtualDisk_init(void) 03 { 04

```
int result;  05    dev_t devno = MKDEV(VirtualDisk_major, 0);    /*构建设备号*/  06    /* 申请设备号*/  07    if (VirtualDisk_major)                /*如果不为0，则静态申请*/  08      result = register_chrdev_region(devno, 1, "VirtualDisk");  09    else                                  /* 动态申请设备号 */  10    {  11      result = alloc_chrdev_region(&devno, 0, 1, "VirtualDisk");  12      VirtualDisk_major = MAJOR(devno);   /*从申请设备号中得到主设备号 */  13    }  14    if (result < 0)  15      return result;  16    /* 动态申请设备结构体的内存*/  17    Virtualdisk_devp = kmalloc(sizeof(struct VirtualDisk), GFP_KERNEL);  18    if (!Virtualdisk_devp)                /*申请失败*/  19    {  20      result = - ENOMEM;  21      goto fail_kmalloc;  22    }  23    memset(Virtualdisk_devp, 0, sizeof(struct VirtualDisk));/*将内存清零*/  24    /*初始化并且添加cdev结构体*/  25    VirtualDisk_setup_cdev(Virtualdisk_devp, 0);  26    return 0;  27  fail_kmalloc:  28      unregister_chrdev_region(devno, 1);  29    return result;  30  }  31  /*模块卸载函数*/  32  void VirtualDisk_exit(void)  33  {  34    cdev_del(&Virtualdisk_devp->cdev);            /*注销cdev*/  35    kfree(Virtualdisk_devp);                      /*释放设备结构体内存*/  36    unregister_chrdev_region(MKDEV(VirtualDisk_major, 0), 1);                                  /*释放设备号*/  37  }
```

第07～13行，使用两种方式申请设备号。VirtualDisk_major变量被静态定义为200。当加载模块时不使VirtualDisk_major等于0，那么就执行register_chrdev_ region()函数静态分配一个设备号；如果VirtualDisk_major等于0，那么就使用alloc_chrdev_ region()函数动态分配一个设备号，并由参数devno返回。第12行，使用MAJOR宏返回得到的主设备号。

第17～22行，分配一个VirtualDisk设备结构体。

第23行，将分配的VirtualDisk设备结构体清零。

第25行，调用自定义的VirtualDisk_setup_cdev()函数初始化cdev结构体，并加入内核中。该函数将在下面讲到。

第32～37行是卸载函数，该函数中注销了cdev结构体，释放了VirtualDisk设备所占的内存，并且释放了设备占用的设备号。

# 14. 6.4.3 cdev的初始化和注册

摘要：《Linux驱动开发入门与实战》第6章简单的字符设备驱动程序，本章首先从整体上介绍字符设备的框架结构，然后介绍字符设备结构体struct cdev，接着介绍字符设备的组成，最后详细讲解一个VirtualDisk字符设备驱动程序。本节为大家介绍cdev的初始化和注册。 标签：Linux驱动开发 Linux驱动开发入门与实战

6.4.3 cdev的初始化和注册

6.4.2节代码中第25行调用的VirtualDisk_setup_cdev()函数完成了cdev的初始化和注册，其代码如下：01 /*初始化并注册cdev*/ 02 static void VirtualDisk_setup_cdev(struct VirtualDisk *dev, int minor) 03 { 04 int err; 05 devno = MKDEV(VirtualDisk_major, minor); /*构造设备号*/ 06 cdev_init(&dev->cdev, &VirtualDisk_fops); /*初始化cdev设备*/ 07 dev->cdev.owner = THIS_MODULE; /*使驱动程序属于该模块*/ 08 dev->cdev.ops = &VirtualDisk_fops; /*cdev连接file_operations指针*/ 09 err = cdev_add(&dev->cdev, devno, 1); /*将cdev注册到系统中*/ 10 if (err) 11 printk(KERN_NOTICE "Error in cdev_add()\n"); 12 }

下面对该函数进行简要的解释：

第05行，使用MKDEV宏构造一个主设备号为VirtualDisk_major，次设备号为minor的设备号。

第06行，调用cdev_init()函数，将设备结构体cdev与file_operators指针相关联。这个文件操作指针定义如下代码所示。/*文件操作结构体*/ static const struct file_operations VirtualDisk_fops = { .owner = THIS_MODULE, .llseek = VirtualDisk_llseek, /*定位偏移量函数*/ .read = VirtualDisk_read, /*读设备函数*/ .write = VirtualDisk_write, /*写设备函数*/ .ioctl = VirtualDisk_ioctl, /*控制函数*/ .open = VirtualDisk_open, /*打开设备函数*/ .release = VirtualDisk_release, /*释放设备函数*/ };

第08行，指定VirtualDisk_fops为字符设备的文件操作函数指针。

第09行，调用cdev_add()函数将字符设备加入到内核中。

第10、11行，如果注册字符设备失败，则返回。

## *15. 6.4.4 打开和释放函数*

摘要：《Linux驱动开发入门与实战》第6章简单的字符设备驱动程序，本章首先从整体上介绍字符设备的框架结构，然后介绍字符设备结构体struct cdev，接着介绍字符设备的组成，最后详细讲解一个VirtualDisk字符设备驱动程序。本节为大家介绍打开和释放函数。 标签：Linux驱动开发 Linux驱动开发入门与实战

6.4.4 打开和释放函数

当用户程序调用open()函数打开设备文件时，内核会最终调用VirtualDisk_open()函数。该函数的代码如下：01 /*文件打开函数*/ 02 int VirtualDisk_open(struct inode *inode, struct file *filp) 03 { 04 /*将设备结构体指针赋值给文件私有数据指针*/ 05 filp->private_data = Virtualdisk_devp; 06 struct VirtualDisk *devp = filp->private_data; /*获得设备结构体指针*/ 07 devp->count++; /*增加设备打开次数*/ 08 return 0; 09 }

下面对该函数进行简要的解释：

第05、06行，将Virtualdisk_devp赋给私有数据指针，在后面将用到这个指针。

第07行，将设备打开计数增加1。

当用户程序调用close()函数关闭设备文件时，内核会最终调用VirtualDisk_release()函数。这个函数主要是将计数器减1。该函数的代码如下：01 /*文件释放函数*/ 02 int VirtualDisk_release(struct inode *inode, struct file *filp) 03 { 04 struct VirtualDisk *devp = filp->private_data; /*获得设备结构体指针*/ 05 devp->count--; /*减少设备打开次数*/ 06 return 0; 07 }

## *16. 6.4.5 读写函数*

摘要：《Linux驱动开发入门与实战》第6章简单的字符设备驱动程序，本章首先从整体上介绍字符设备的框架结构，然后介绍字符设备结构体struct cdev，接着介绍字符设备的组成，最后详细讲解一个VirtualDisk字符设备驱动程序。本节为大家介绍读写函数。 标签：Linux驱动开发 Linux驱动开发入门与实战

6.4.5 读写函数

当用户程序调用read()函数读设备文件中的数据时，内核会最终调用VirtualDisk_read()函数。该函数的代码如下：01 /*读函数*/ 02 static ssize_t VirtualDisk_read(struct file *filp, char __user *buf, size_t size, 03 loff_t *ppos) 04 { 05 unsigned long p = *ppos; /*记录文件指针偏移位置*/ 06 unsigned int count = size; /*记录需要读取的字节数*/ 07 int ret = 0;/*返回值*/ 08 struct VirtualDisk *devp = filp->private_data; /*获得设备结构体指针*/ 09 /*分析和获取有效的读长度*/ 10 if (p >= VIRTUALDISK_SIZE) /*要读取的偏移大于设备的内存空间*/ 11 return count ? - ENXIO: 0; /*读取地址错误*/ 12 if (count > VIRTUALDISK_SIZE - p) /*要读取的字节大于设备的内存空间*/ 13 count = VIRTUALDISK_SIZE - p; /*将要读取的字节数设为剩余的字节数*/ 14 /*内核空间->用户空间交换数据*/ 15 if (copy_to_user(buf, (void*)(devp->mem + p), count)) 16 { 17 ret = - EFAULT; 18 } 19 else 20 { 21 *ppos += count; 22 ret = count; 23 printk(KERN_INFO "read %d bytes(s) from %d\n", count, p); 24 } 25 return ret; 26 }

下面对该函数进行简要的分析：

第05～07行，定义了一些局部变量。

第08行，从文件指针中获得设备结构体指针。

第10行，如果要读取的位置大于设备的大小，则出错。

第12行，如果要读的数据的位置大于设备的大小，则只读到设备的末尾。

第15～24行，从用户空间复制数据到设备中。如果复制数据成功，就将文件的偏移位置加上读出的数据个数。

当用户程序调用write()函数向设备文件写入数据时，内核会最终调用VirtualDisk_write()函数。该函数的代码如下：01 /*写函数*/ 02 static ssize_t

```
VirtualDisk_write(struct file *filp, const char __user   *buf,  03    size_t size, loff_t *ppos)  04  {  05    unsigned long p =  *ppos;
/*记录文件指针偏移位置*/  06    int ret = 0;
/*返回值*/  07    unsigned int count = size;            /*记录需要写入的字节数*/  08    struct VirtualDisk *devp = filp->private_data;  /*获得设备结构体指针*/  09    /*分析和获取有效的写长度*/  10    if (p >= VIRTUALDISK_SIZE)
        /*要写入的偏移大于设备的内存空间*/  11      return count ?  - ENXIO: 0;          /*写入地址错误*/  12    if (count > VIRTUALDISK_SIZE - p)
   /*要写入的字节大于设备的内存空间*/  13      count = VIRTUALDISK_SIZE - p;
      /*将要写入的字节数设为剩余的字节数*/  14    /*用户空间->内核空间*/  15
   if (copy_from_user(devp->mem + p, buf, count))  16      ret =  - EFAULT;
17    else  18    {  19      *ppos += count;                    /*增加偏移位置*/  20      ret = count;                       /*返回实际的写入字节数*/  21      printk(KERN_INFO "written %d bytes(s) from %d\n", count, p);
22    }  23    return ret;  24  }
```

下面对该函数进行简要的介绍：

第05～07行，定义了一些局部变量。

第08行，从文件指针中获得设备结构体指针。

第10行，如果要读取的位置大于设备的大小，则出错。

第12行，如果要读的数据的位置大于设备的大小，则只读到设备的末尾。

第15～24行，从设备中复制数据到用户空间中。如果复制数据成功，就将文件的偏移位置加上写入的数据个数。

## *17. 6.4.6 seek()函数*

摘要：《Linux驱动开发入门与实战》第6章简单的字符设备驱动程序，本章首先从整体上介绍字符设备的框架结构，然后介绍字符设备结构体struct cdev，接着介绍字符

设备的组成，最后详细讲解一个VirtualDisk字符设备驱动程序。本节为大家介绍seek()函数。 标签：seek()函数 Linux驱动开发 Linux驱动开发入门与实战

6.4.6 seek()函数

当用户程序调用fseek()函数在设备文件中移动文件指针时，内核会最终调用VirtualDisk_llseek()函数。该函数的代码如下：

```
01  /* seek文件定位函数 */
02  static loff_t VirtualDisk_llseek(struct file *filp, loff_t offset, int orig)
03  {
04      loff_t ret = 0;                                    /*返回的位置偏移*/
05      switch (orig)
06      {
07          case SEEK_SET:                                 /*相对文件开始位置偏移*/
08              if (offset < 0)                            /*offset不合法*/
09              {
10                  ret = - EINVAL;                        /*无效的指针*/
11                  break;
12              }
13              if ((unsigned int)offset > VIRTUALDISK_SIZE)   /*偏移大于设备内存*/
14              {
15                  ret = - EINVAL;                        /*无效的指针*/
16                  break;
17              }
18              filp->f_pos = (unsigned int)offset;        /*更新文件指针位置*/
19              ret = filp->f_pos;                         /*返回的位置偏移*/
20              break;
21          case SEEK_CUR:                                 /*相对文件当前位置偏移*/
22              if ((filp->f_pos + offset) > VIRTUALDISK_SIZE)   /*偏移大于设备内存*/
23              {
24                  ret = - EINVAL;                        /*无效的指针*/
25                  break;
26              }
27              if ((filp->f_pos + offset) < 0)            /*指针不合法*/
28              {
29                  ret = - EINVAL;                        /*无效的指针*/
30                  break;
31              }
32              filp->f_pos += offset;                     /*更新文件指针位置*/
33              ret = filp->f_pos;                         /*返回的位置偏移*/
34              break;
35          default:
36              ret = - EINVAL;                            /*无效的指针*/
37              break;
38      }
39      return ret;
40  }
```

下面对该函数进行简要的介绍：

第04行，定义了一个返回值，用来表示文件指针现在的偏移量。

第05行，用来选择文件指针移动的方向。

第07～20行，表示文件指针移动的类型是SEEK_SET，表示相对于文件的开始移动指针offset个位置。

第08～12行，如果偏移小于0，则返回错误。

第13～17行，如果偏移值大于文件的长度，则返回错误。

第18行，设置文件的偏移值到 filp->f_pos，这个指针表示文件的当前位置。

第21～34行，表示文件指针移动的类型是SEEK_CUR，表示相对于文件的当前位置移动指针offset个位置。

第22～26行，如果偏移值大于文件的长度，则返回错误。

第27～31行，表示指针小于0的情况，这种情况指针是不合法的。
第32行，将文件的偏移值filp->f_pos加上offset个偏移。
第35、36行，表示命令不是SEEK_SET或者SEEK_CUR，这种情况下表示传入了非法的命令，直接返回。

## 18. 6.4.7 ioctl()函数

摘要：《Linux驱动开发入门与实战》第6章简单的字符设备驱动程序，本章首先从整体上介绍字符设备的框架结构，然后介绍字符设备结构体struct cdev，接着介绍字符设备的组成，最后详细讲解一个VirtualDisk字符设备驱动程序。本节为大家介绍ioctl()函数。 标签：ioctl()函数 Linux驱动开发 Linux驱动开发入门与实战
6.4.7 ioctl()函数
当用户程序调用ioctl()函数改变设备的功能时，内核会最终调用VirtualDisk_ioctl()函数。该函数的代码如下：01 /* ioctl设备控制函数 */ 02 static int VirtualDisk_ioctl(struct inode *inodep, struct file *filp, unsigned 03 int cmd, unsigned long arg) 04 { 05 struct VirtualDisk *devp = filp->private_data; /*获得设备结构体指针*/ 06 switch (cmd) 07 { 08 case MEM_CLEAR: /*设备内存清零*/ 09 memset(devp->mem, 0, VIRTUALDISK_SIZE); 10 printk(KERN_INFO "VirtualDisk is set to zero\n"); 11 break; 12 case PORT1_SET: /*将端口1置0*/ 13 devp->port1=0; 14 break; 15 case PORT2_SET: /*将端口2置0*/ 16 devp->port2=0; 17 break; 18 default: 19 return - EINVAL; 20 } 21 return 0; 22 }
下面对该函数进行简要的介绍：
第05行，得到文件的私有数据，私有数据中存放的是VirtualDisk设备的指针。

第06～20行，根据ioctl()函数传进来的参数判断将要执行的操作。这里的字符设备支持3个操作，第1个操作是将字符设备的内存全部清0，第2个操作是将端口1设置为0，第3个操作是将端口2设置成0。

## *19. 6.5 小结*

摘要：《Linux驱动开发入门与实战》第6章简单的字符设备驱动程序，本章首先从整体上介绍字符设备的框架结构，然后介绍字符设备结构体struct cdev，接着介绍字符设备的组成，最后详细讲解一个VirtualDisk字符设备驱动程序。本节为这一章的小结部分。 标签：Linux驱动开发 Linux驱动开发入门与实战

6.5 小结

本章主要讲解了字符设备驱动程序。字符设备是Linux中的三大设备之一，很多设备都可以看成是字符设备，所以学习字符设备驱动程序的编程是很有用的。本章首先从整体上介绍了字符设备的框架结构，然后介绍了字符设备结构体struct cdev，接着介绍了字符设备的组成，最后详细讲解了一个VirtualDisk字符设备驱动程序。

# 3. 第17章　输入子系统设计

## *1. 17.1 input子系统入门*

摘要：《Linux驱动开发入门与实战》第17章输入子系统设计，本章将介绍Linux输入子系统的驱动开发。Linux的输入子系统不仅支持鼠标、键盘等常规输入设备，而且还支持蜂鸣器、触摸屏等设备。本章将对Linux输入子系统进行详细的分析。本节为大家介绍input子系统。 标签：input子系统　Linux驱动开发　Linux驱动开发入门与实战
第17章　输入子系统设计
本章将介绍Linux输入子系统的驱动开发。Linux的输入子系统不仅支持鼠标、键盘等常规输入设备，而且还支持蜂鸣器、触摸屏等设备。本章将对Linux输入子系统进行详细的分析。
17.1　input子系统入门
输入子系统又叫input子系统。其构建非常灵活，只需要调用一些简单的函数，就可以将一个输入设备的功能呈现给应用程序。本节将从一个实例开始，介绍编写输入子系统驱动程序的方法。

## *2. 17.1.1 简单的实例*

摘要：《Linux驱动开发入门与实战》第17章输入子系统设计，本章将介绍Linux输入

子系统的驱动开发。Linux的输入子系统不仅支持鼠标、键盘等常规输入设备，而且还支持蜂鸣器、触摸屏等设备。本章将对Linux输入子系统进行详细的分析。本节为大家介绍简单的实例。 标签：input子系统 Linux驱动开发 Linux驱动开发入门与实战

17.1.1 简单的实例

本节将讲述一个简单的输入设备驱动实例。这个输入设备只有一个按键，按键被连接到一条中断线上，当按键被按下时，将产生一个中断，内核将检测到这个中断，并对其进行处理。该实例的代码如下：01 #include <asm/irq.h> 02 #include <asm/io.h> 03 static struct input_dev *button_dev; /*输入设备结构体*/ 04 static irqreturn_t button_interrupt(int irq, void *dummy)
/*中断处理函数*/ 05 { 06
input_report_key(button_dev, BTN_0, inb(BUTTON_PORT) & 1);
/*向输入子系统报告产生按键事件*/ 07 input_sync(button_dev);
/*通知接收者，一个报告发送完毕*/
08 return IRQ_HANDLED; 09 } 10 static int __init button_init(void)
/*加载函数*/ 11 { 12 int error; 13 if (request_irq(BUTTON_IRQ, button_interrupt, 0, "button", NULL))
/*申请中断处理函数*/ 14 { 15 /*申请失败，则打印出错信息*/
16 printk(KERN_ERR "button.c: Can't allocate irq %d\n", button_irq); 17 return -EBUSY;
18 } 19 button_dev = input_allocate_device(); /*分配一个设备结构体*/ 20 if (!button_dev) /*判断分配是否成功*/
21 { 22 printk(KERN_ERR "button.c: Not enough memory\n");
23 error = -ENOMEM; 24 goto err_free_irq; 25 } 26
button_dev->evbit[0] = BIT_MASK(EV_KEY); /*设置按键信息*/ 27
button_dev->keybit[BIT_WORD(BTN_0)] = BIT_MASK(BTN_0); 28 error = input_register_device(button_dev); /*注册一个输入设备*/ 29 if (error)
30 { 31 printk(KERN_ERR "button.c: Failed to register device\n"); 32 goto err_free_dev; 33 } 34 return 0;
35 err_free_dev: /*以下是错误处理*/ 36
input_free_device(button_dev); 37 err_free_irq: 38
free_irq(BUTTON_IRQ, button_interrupt); 39 return error; 40 } 41
static void __exit button_exit(void) /*卸载函数*/ 42 { 43
input_unregister_device(button_dev); /*注销按键设备*/ 44
free_irq(BUTTON_IRQ, button_interrupt); /*释放按键占用的中断线*/ 45 } 46
module_init(button_init); 47 module_exit(button_exit);

这个实例程序代码比较简单，在初始化函数button_init()中注册了一个中断处理函数，然后调用input_allocate_device()函数分配了一个input_dev结构体，并调用input_register_device()函数对其进行了注册。在中断处理函数button_interrupt()中，实例将接收到的按键信息上报给input子系统。从而通过

input子系统，向用户态程序提供按键输入信息。
本实例采用了中断方式，除了中断相关的代码外，实例中包含了一些input子系统提供的函数，现对其中一些重要的函数进行分析。
第19行的input_allocate_device()函数在内存中为输入设备结构体分配一个空间，并对其主要的成员进行了初始化。驱动开发人员为了更深入的了解input子系统，应该对其代码有一点的认识，该函数的代码如下：

```
struct input_dev *input_allocate_device(void)  {      struct input_dev *dev;
     dev = kzalloc(sizeof(struct input_dev), GFP_KERNEL);
/*分配一个input_dev结构体，并初始化为0*/      if (dev) {          dev-
>dev.type = &input_dev_type;         /*初始化设备的类型*/          dev-
>dev.class = &input_class;          /*设置为输入设备类*/
device_initialize(&dev->dev);           /*初始化device结构*/
mutex_init(&dev->mutex);                /*初始化互斥锁*/
spin_lock_init(&dev->event_lock);       /*初始化事件自旋锁*/
INIT_LIST_HEAD(&dev->h_list);           /*初始化链表*/
INIT_LIST_HEAD(&dev->node);             /*初始化链表*/
__module_get(THIS_MODULE);              /*模块引用技术加1*/      }
return dev;  }
```

该函数返回一个指向input_dev类型的指针，该结构体是一个输入设备结构体，包含了输入设备的一些相关信息，如设备支持的按键码、设备的名称、设备支持的事件等。在本章用到这个结构体时，将对其进行详细介绍。此处将注意力集中在实例中的函数上。

## *3. 17.1.2 注册函数input_register_devic*

摘要：《Linux驱动开发入门与实战》第17章输入子系统设计，本章将介绍Linux输入子系统的驱动开发。Linux的输入子系统不仅支持鼠标、键盘等常规输入设备，而且还支持蜂鸣器、触摸屏等设备。本章将对Linux输入子系统进行详细的分析。本节为大家

介绍注册函数input_register_device()。 标签：input子系统 Linux驱动开发
Linux驱动开发入门与实战
17.1.2 注册函数input_register_device()（1）
button_init()函数中的28行调用了input_register_device()函数注册输入设备结构体。input_register_device()函数是输入子系统核心（input core）提供的函数。该函数将input_dev结构体注册到输入子系统核心中，input_dev结构体必须由前面讲的input_allocate_device()函数来分配。input_register_device()函数如果注册失败，必须调用input_free_device()函数释放分配的空间。如果该函数注册成功，在卸载函数中应该调用input_unregister_device()函数来注销输入设备结构体。
1. input_register_device()函数
input_register_device()函数的代码如下：01 int input_register_device(struct input_dev *dev) 02 { 03 static atomic_t input_no = ATOMIC_INIT(0); 04 struct input_handler *handler; 05 const char *path; 06 int error; 07 __set_bit(EV_SYN, dev->evbit); 08 init_timer(&dev->timer); 09 if (!dev->rep[REP_DELAY] && !dev->rep[REP_PERIOD]) { 10 dev->timer.data = (long) dev; 11 dev->timer.function = input_repeat_key; 12 dev->rep[REP_DELAY] = 250; 13 dev->rep[REP_PERIOD] = 33; 14 } 15 if (!dev->getkeycode) 16 dev->getkeycode = input_default_getkeycode; 17 if (!dev->setkeycode) 18 dev->setkeycode = input_default_setkeycode; 19 dev_set_name(&dev->dev, "input%ld", 20 (unsigned long) atomic_inc_return(&input_no) - 1); 21 error = device_add(&dev->dev); 22 if (error) 23 return error; 24 path = kobject_get_path(&dev->dev.kobj, GFP_KERNEL); 25 printk(KERN_INFO "input: %s as %s\n", 26 dev->name ? dev->name : "Unspecified device", path ?
path : "N/A"); 27 kfree(path); 28 error = mutex_lock_interruptible(&input_mutex); 29 if (error) { 30 device_del(&dev->dev); 31 return error; 32 } 33 list_add_tail(&dev->node, &input_dev_list); 34 list_for_each_entry(handler, &input_handler_list, node) 35 input_attach_handler(dev, handler); 36 input_wakeup_procfs_readers(); 37 mutex_unlock(&input_mutex); 38 return 0; 39 }
下面对该函数的主要代码进行分析。
第03～06行，定义了一些函数中将要用到的局部变量。
第07行，调用__set_bit()函数设置input_dev所支持的事件类型。事件类型由input_dev的evbit成员来表示，在这里将其EV_SYN置位，表示设备支持所有的事件。注意，一个设备可以支持一种或者多种事件类型。常用的事件类型如下：#define EV_SYN 0x00 /*表示设备支持所有的事件*/ #define EV_KEY 0x01

/*键盘或者按键，表示一个键码*/ #define EV_REL 0x02 /*鼠标设备，表示一个相对的光标位置结果*/ #define EV_ABS 0x03 /*手写板产生的值，其是一个绝对整数值*/ #define EV_MSC 0x04 /*其他类型*/ #define EV_LED 0x11 /*LED灯设备*/ #define EV_SND 0x12 /*蜂鸣器，输入声音*/ #define EV_REP 0x14 /*允许重复按键类型*/ #define EV_PWR 0x16 /*电源管理事件*/

第08行，初始化一个timer定时器，这个定时器是为处理重复击键而定义的。

第09～14行，如果dev->rep[REP_DELAY]和dev->rep[REP_PERIOD]没有设值，则将其赋默认值，这主要是为自动处理重复按键定义的。

第15～18行，检查getkeycode()函数和setkeycode()函数是否被定义，如果没定义，则使用默认的处理函数，这两个函数为input_default_getkeycode()和input_default_setkeycode()。input_default_getkeycode()函数用来得到指定位置的键值。input_default_setkeycode()函数用来设置键值。

第19行，设置input_dev中的device的名字，名字以input0、input1、input2、input3、input4等的形式出现在sysfs文件系统中。

第21行，使用device_add()函数将input_dev包含的device结构注册到Linux设备模型中，并可以在sysfs文件系统中表现出来。

第24～27行，打印设备的路径，输出调试信息。

第33行，调用list_add_tail()函数将input_dev加入input_dev_list链表中，input_dev_list链表中包含了系统中所有的input_dev设备。

第34～35行，调用了input_attach_handler()函数，该函数将在下面单独解释。

## *4. 17.1.2 注册函数input_register_devic*

摘要：《Linux驱动开发入门与实战》第17章输入子系统设计，本章将介绍Linux输入子系统的驱动开发。Linux的输入子系统不仅支持鼠标、键盘等常规输入设备，而且还支持蜂鸣器、触摸屏等设备。本章将对Linux输入子系统进行详细的分析。本节为大家

介绍注册函数input_register_device()。 标签：input子系统 Linux驱动开发 Linux驱动开发入门与实战
17.1.2 注册函数input_register_device()（2）
2. input_attach_handler()函数
input_attach_handler()函数用来匹配input_dev和handler，只有匹配成功，才能进行下一步的关联操作。input_attach_handler()函数的代码如下：01 static int input_attach_handler(struct input_dev *dev, struct input_handler *handler) 02 { 03 const struct input_device_id *id; /*输入设备的指针*/ 04 int error; 05 if (handler->blacklist && input_match_device(handler->blacklist, dev)) 06 return -ENODEV; /*设备和处理函数之间的匹配*/ 07 id = input_match_device(handler->id_table, dev); 08 if (!id) 09 return -ENODEV; 10 error = handler->connect(handler, dev, id);/*连接设备和处理函数*/ 11 if (error && error != -ENODEV) 12 printk(KERN_ERR 13 "input: failed to attach handler %s to device %s, " 14 "error: %d\n", 15 handler->name, kobject_name(&dev->dev.kobj), error); 16 return error; 17 }
下面对该函数进行简要的分析。
第03行，定义了一个input_device_id的指针。该结构体表示设备的标识，标识中存储了设备的信息，其定义如下：struct input_device_id { kernel_ulong_t flags; /*标志信息*/ __u16 bustype; /*总线类型*/ __u16 vendor; /*制造商ID*/ __u16 product; /*产品ID*/ __u16 version; /*版本号*/ ... kernel_ulong_t driver_info; /*驱动额外的信息*/ };
第05行，首先判断handle的blacklist是否被赋值，如果被赋值，则匹配blacklist中的数据跟dev->id的数据是否匹配。blacklist是一个input_device_id*的类型，其指向input_device_ids的一个表，这个表中存放了驱动程序应该忽略的设备。即使在id_table中找到支持的项，也应该忽略这种设备。
第07～09行，调用input_match_device()函数匹配handle->>id_table和dev->id中的数据。如果不成功则返回。handle->id_table也是一个input_device_id类型的指针，其表示驱动支持的设备列表。
第10行，如果匹配成功，则调用handler->connect()函数将handler与input_dev连接起来。
3. input_match_device ()函数
input_match_device ()函数用来与input_dev和handler进行匹配。handler的id_table表中定义了其支持的input_dev设备。该函数的代码如下：

01 static const struct input_device_id *input_match_device(const struct

```
 input_device_id *id,  02                                struct input_dev
*dev)  03  {  04      int i;  05      for (; id->flags || id->driver_info;
id++) {  06          if (id->flags & INPUT_DEVICE_ID_MATCH_BUS)  07
    if (id->bustype != dev->id.bustype)  08                  continue;  09
       if (id->flags & INPUT_DEVICE_ID_MATCH_VENDOR)  10              if
(id->vendor != dev->id.vendor)  11                   continue;  12
if (id->flags & INPUT_DEVICE_ID_MATCH_PRODUCT)  13              if (id-
>product != dev->id.product)  14                  continue;  15          if
(id->flags & INPUT_DEVICE_ID_MATCH_VERSION)  16              if (id->version
!= dev->id.version)  17                  continue;  18
MATCH_BIT(evbit,  EV_MAX);  19          MATCH_BIT(keybit, KEY_MAX);  20
    MATCH_BIT(relbit, REL_MAX);  21          MATCH_BIT(absbit, ABS_MAX);  22
        MATCH_BIT(mscbit, MSC_MAX);  23          MATCH_BIT(ledbit,
LED_MAX);  24          MATCH_BIT(sndbit, SND_MAX);  25
MATCH_BIT(ffbit,  FF_MAX);  26          MATCH_BIT(swbit,  SW_MAX);  27
   return id;  28      }  29      return NULL;  30  }
```

下面对该函数进行简要的解释。

第04行声明一个局部变量i，用于循环。

第05行，是一个for循环，用来匹配id和dev->id中的信息，只要有一项相同则返回。

第06～08行，用来匹配总线类型。id->flags中定义了要匹配的项，其中INPUT_DEVICE_ID_MATCH_BUS如果没有设置，则比较input device和input handler的总线类型。

第09～11行，匹配设备厂商的信息。

第12～14行，分别匹配设备号的信息。

第18～26行，使用MATCH_BIT匹配项。如果id->flags定义的类型匹配成功，或者id->flags没有定义，才会进入到MATCH_BIT的匹配项。MATCH_BIT宏的定义如下：

```
#define MATCH_BIT(bit, max) \          for (i = 0; i < BITS_TO_LONGS(max);
i++) \              if ((id->bit[i] & dev->bit[i]) != id->bit[i]) \
        break; \          if (i != BITS_TO_LONGS(max)) \
continue;
```

从MATCH_BIT宏的定义可以看出。只有当iput device和input handler的ID成员在evbit、keybit、… swbit项相同才会匹配成功。而且匹配的顺序是从evbit、keybit到swbit。只要有一项不同，就会循环到ID中的下一项进行比较。

简而言之，注册input device的过程就是为input device设置默认值，并将其挂以input_dev_list。与挂载在input_handler_list中的handler相匹配。如果匹配成功，就会调用handler的connect函数。

## 5. 17.1.3 向子系统报告事件（1）

摘要：《Linux驱动开发入门与实战》第17章输入子系统设计，本章将介绍Linux输入子系统的驱动开发。Linux的输入子系统不仅支持鼠标、键盘等常规输入设备，而且还支持蜂鸣器、触摸屏等设备。本章将对Linux输入子系统进行详细的分析。本节为大家介绍向子系统报告事件。 标签：子系统 Linux驱动开发 Linux驱动开发入门与实战

17.1.3 向子系统报告事件（1）

在17.1.1节button_interrupt()函数的06行，调用了input_report_key()函数向输入子系统报告发生的事件，这里就是一个按键事件。在button_interrupt()中断函数中，不需要考虑重复按键的重复点击情况，input_report_key()函数会自动检查这个问题，并报告一次事件给输入子系统。该函数的代码如下：

```
01 static inline void input_report_key(struct input_dev *dev,
unsigned int code, int value) 02 { 03 input_event(dev, EV_KEY,
code, !!value); 04 }
```

该函数的第1个参数是产生事件的输入设备，第2个参数是产生的事件，第3个参数是事件的值。需要注意的是，第2个参数可以取类似BTN_0、BTN_1、BTN_LEFT、BTN_RIGHT等值，这些键值被定义在include/linux/input.h文件中。当第2个参数为按键时，第3个参数表示按键的状态，value值为0表示按键释放，非0表示按键按下。

1. input_report_key()函数

在input_report_key()函数中正在起作用的函数是input_event()函数，该函数用来向输入子系统报告输入设备产生的事件，这个函数非常重要，它的代码如下：

```
01 void input_event(struct input_dev *dev, 02 unsigned int type,
unsigned int code, int value) 03 { 04 unsigned long flags; 05
if (is_event_supported(type, dev->evbit, EV_MAX)) { 06
spin_lock_irqsave(&dev->event_lock, flags); 07
add_input_randomness(type, code, value); 08
input_handle_event(dev, type, code, value); 09
spin_unlock_irqrestore(&dev->event_lock, flags); 10 } 11 }
```

该函数第1个参数是input_device设备，第2个参数是事件的类型，可以取EV_KEY、EV_REL、EV_ABS等值，在上面的按键时间报告函数input_report_key()中传递的就是

EV_KEY值，表示发生一个按键事件。第3、4个函数与input_report_key()函数的参数相同，下面对这个函数进行简要的分析。
第04行，调用is_event_supported()函数检查输入设备是否支持该事件。该函数的代码如下：

```
01  static inline int is_event_supported(unsigned int code,  02
        unsigned long *bm, unsigned int max)  03  {  04      return code <=
max && test_bit(code, bm);  05  }
```

该函数检查input_dev.evbit中的相应位是否设置，如果设置返回1，否则返回0。每一种类型的事件都在input_dev.evbit中用一个位来表示，构成一个位图，如果某位为1，表示该输入设备支持这类事件，如果为0，表示输入设备不支持这类事件。如图17.1所示，表示各位支持的事件，其中省略了一些事件类型，目前Linux支持十多种事件类型，所以用一个long型变量就可以全部表示了。　（点击查看大图）图17.1
input_dev.evbit支持的事件表示方法需要注意的是，这里可以回顾一下17.1.1节button_init()函数的第26行，如下所示。

```
26      button_dev->evbit[0] = BIT_MASK(EV_KEY);        /*设置按键信息*/
```

该行就是设置输入设备button_dev所支持的事件类型，BIT_MASK是用来构造input_dev.evbit这个位图的宏，宏代码如下：#define BIT_MASK(nr)        (1UL << ((nr) % BITS_PER_LONG))
回到17.1.1节input_event()函数的第06行，调用spin_lock_irqsave()函数对将事件锁锁定。
第07行，add_input_randomness()函数对事件发送没有一点用处，只是用来对随机数熵池增加一些贡献，因为按键输入是一种随机事件，所以对熵池是有贡献的。
第08行，调用input_handle_event()函数来继续输入子系统的相关模块发送数据。该函数较为复杂，下面单独进行分析。
2. input_handle_event()函数
input_handle_event()函数向输入子系统传送事件信息。第1个参数是输入设备input_dev，第2个参数是事件的类型，第3个参数是键码，第4个参数是键值。该函数的代码如下：

```
01  static void input_handle_event(struct input_dev *dev,  02
     unsigned int type, unsigned int code, int value)  03  {  04      int
disposition = INPUT_IGNORE_EVENT;  05      switch (type) {  06      case
EV_SYN:  07          switch (code) {  08          case SYN_CONFIG:  09
       disposition = INPUT_PASS_TO_ALL;  10              break;  11
case SYN_REPORT:  12              if (!dev->sync) {  13
dev->sync = 1;  14                  disposition = INPUT_PASS_TO_HANDLERS;
15              }  16              break;  17          }  18          break;
 19      case EV_KEY:  20          if (is_event_supported(code, dev->keybit,
KEY_MAX) &&  21              !!test_bit(code, dev->key) != value) {  22
        if (value != 2) {  23                  __change_bit(code, dev->key);
```

```
 24                 if (value)  25
input_start_autorepeat(dev, code);  26             }  27
disposition = INPUT_PASS_TO_HANDLERS;  28         }  29          break;  30
     case EV_SW:  31          if (is_event_supported(code, dev->swbit,
SW_MAX) &&  32              !!test_bit(code, dev->sw) != value) {  33
      __change_bit(code, dev->sw);  34             disposition =
INPUT_PASS_TO_HANDLERS;  35         }  36          break;  37      case
EV_ABS:  38          if (is_event_supported(code, dev->absbit, ABS_MAX)) {
39              value = input_defuzz_abs_event(value,  40
  dev->abs[code], dev->absfuzz[code]);  41              if (dev->abs[code]
!= value) {  42                  dev->abs[code] = value;  43
 disposition = INPUT_PASS_TO_HANDLERS;  44             }  45         }  46
         break;  47      case EV_REL:  48          if
(is_event_supported(code, dev->relbit, REL_MAX) && value)  49
disposition = INPUT_PASS_TO_HANDLERS;  50          break;  51      case
EV_MSC:  52          if (is_event_supported(code, dev->mscbit, MSC_MAX))  53
             disposition = INPUT_PASS_TO_ALL;  54          break;  55
case EV_LED:  56          if (is_event_supported(code, dev->ledbit, LED_MAX)
&&  57              !!test_bit(code, dev->led) != value) {  58
__change_bit(code, dev->led);  59             disposition =
INPUT_PASS_TO_ALL;  60          }  61          break;  62      case EV_SND:
63          if (is_event_supported(code, dev->sndbit, SND_MAX)) {  64
      if (!!test_bit(code, dev->snd) != !!value)  65
__change_bit(code, dev->snd);  66             disposition =
INPUT_PASS_TO_ALL;  67          }  68          break;  69      case EV_REP:
70          if (code <= REP_MAX && value >= 0 && dev->rep[code] != value) {
71              dev->rep[code] = value;  72              disposition =
INPUT_PASS_TO_ALL;  73          }  74          break;  75      case EV_FF:
76          if (value >= 0)  77              disposition =
INPUT_PASS_TO_ALL;  78          break;  79      case EV_PWR:  80
disposition = INPUT_PASS_TO_ALL;  81          break;  82      }  83      if
(disposition != INPUT_IGNORE_EVENT && type != EV_SYN)  84          dev->sync
= 0;  85      if ((disposition & INPUT_PASS_TO_DEVICE) && dev->event)  86
      dev->event(dev, type, code, value);  87      if (disposition &
INPUT_PASS_TO_HANDLERS)  88          input_pass_event(dev, type, code,
value);  89  }
```

## 6. 17.1.3 向子系统报告事件（2）

摘要：《Linux驱动开发入门与实战》第17章输入子系统设计，本章将介绍Linux输入子系统的驱动开发。Linux的输入子系统不仅支持鼠标、键盘等常规输入设备，而且还支持蜂鸣器、触摸屏等设备。本章将对Linux输入子系统进行详细的分析。本节为大家介绍向子系统报告事件。 标签：子系统 Linux驱动开发 Linux驱动开发入门与实战
17.1.3 向子系统报告事件（2）
浏览一下该函数的大部分代码，主要由一个switch结构组成。该结构用来对不同的事件类型，分别处理。其中case语句包含了EV_SYN、EV_KEY、EV_SW、EV_SW、EV_SND等事件类型。在这么多事件中，本例只要关注EV_KEY事件，因为本节的实例发送的是键盘事件。其实，只要对一个事件的处理过程了解后，对其他事件的处理过程也就清楚了。下面对input_handle_event()函数进行简要的介绍：
第04行，定义了一个disposition变量，该变量表示使用什么样的方式处理事件。此处初始化为INPUT_IGNORE_EVENT，表示如果后面没有对该变量重新赋值，则忽略这个事件。
第05～82行是一个重要的switch结构，该结构中对各种事件进行了一些必要的检查，并设置了相应的disposition变量的值。其中只需要关心第19～29行的代码 即可。
第19～29行，对EV_KEY事件进行处理。第20行，调用is_event_supported()函数判断是否支持该按键。第21行，调用test_bit()函数来测试按键状态是否改变。第23行，调用__change_bit()函数改变键的状态。第25行，处理重复按键的情况。第27行，将disposition变量设置为INPUT_PASS_TO_HANDLERS，表示事件需要handler来处理。
disposition的取值有如下几种：#define INPUT_IGNORE_EVENT 0 #define INPUT_PASS_TO_HANDLERS 1 #define INPUT_PASS_TO_DEVICE 2 #define INPUT_PASS_TO_ALL(INPUT_PASS_TO_HANDLERS | INPUT_PASS_TO_DEVICE)
INPUT_IGNORE_EVENT表示忽略事件，不对其进行处理。INPUT_PASS_ TO_HANDLERS表示将事件交给handler处理。INPUT_PASS_TO_DEVICE表示将事件 交给input_dev处理。INPUT_PASS_TO_ALL表示将事件交给handler和input_dev共同处理。
第83、84行，处理EV_SYN事件，这里并不对其进行关心。
第85、86行，首先判断disposition等于INPUT_PASS_TO_DEVICE，然后判断dev->event是否对其指定了一个处理函数，如果这些条件都满足，则调用自定义的dev-

>event()函数处理事件。有些事件是发送给设备，而不是发送给handler处理的。
event()函数用来向输入子系统报告一个将要发送给设备的事件，例如让LED灯点亮事件、蜂鸣器鸣叫事件等。当事件报告给输入子系统后，就要求设备处理这个事件。
第87、88行，如果事件需要handler处理，则调用input_pass_event()函数，该函数将在下面详细解释。
3. input_pass_event()函数
input_pass_event()函数将事件传递到合适的函数，然后对其进行处理，该函数的代码如下：01 static void input_pass_event(struct input_dev *dev, 02 unsigned int type, unsigned int code, int value) 03 { 04 struct input_handle *handle; 05 rcu_read_lock(); 06 handle = rcu_dereference(dev->grab); 07 if (handle) 08 handle->handler->event(handle, type, code, value); 09 else 10 list_for_each_entry_rcu(handle, &dev->h_list, d_node) 11 if (handle->open) 12 handle->handler->event(handle, 13 type, code, value); 14 rcu_read_unlock(); 15 }
下面对该函数进行简要的分析。
第04行，分配一个input_handle结构的指针。
第06行，得到dev->grab的指针。grab是强制为input device的handler，这时要调用handler的event函数。
第10～13行，表示如果没有为input device强制指定handler，即为grab赋值，就会遍历input device->h_list上的handle成员。如果该handle被打开，表示该设备已经被一个用户进程使用。就会调用与输入设备对应的handler的event()函数。注意，只有在handle被打开的情况下才会接收到事件，这就是说，只有设备被用户程序使用时，才有必要向用户空间导出信息。事件的处理过程如图17.2所示。
（点击查看大图）图17.2 event执行过程

## 7. 17.2.1 输入子系统的组成

摘要：《Linux驱动开发入门与实战》第17章输入子系统设计，本章将介绍Linux输入子系统的驱动开发。Linux的输入子系统不仅支持鼠标、键盘等常规输入设备，而且还支持蜂鸣器、触摸屏等设备。本章将对Linux输入子系统进行详细的分析。本节为大家介绍输入子系统的组成。 标签：input子系统 Linux驱动开发 Linux驱动开发入门与实战

## 17.2 handler注册分析

input_handler是输入子系统的主要数据结构，一般将其称为handler处理器，表示对输入事件的具体处理。input_handler为输入设备的功能实现了一个接口，输入事件最终传递到handler处理器，handler处理器根据一定的规则，然后对事件进行处理，具体的规则将在下面详细介绍。在此之前，需要了解一下输入子系统的组成。

### 17.2.1 输入子系统的组成

前面主要讲解了input_dev相关的函数，本节将总结前面的知识，并引出新的知识。为了使读者对输入子系统有整体的了解，本节将对输入子系统的组成进行简要的介绍。后面的章节将围绕输入子系统的各个组成部分来学习。首先，看看图17.3所示，为输入子系统的组成。 （点击查看大图）图17.3 输入子系统的组成

输入子系统由驱动层、输入子系统核心层（Input Core）和事件处理层（Event Handler）3部分组成。一个输入事件，如鼠标移动，键盘按键按下等通过驱动层->系统核心层->事件处理层->用户空间的顺序到达用户空间并传给应用程序使用。其中Input Core即输入子系统核心层由driver/input/input.c及相关头文件实现。其对下提供了设备驱动的接口，对上提供了事件处理层的编程接口。输入子系统主要设计input_dev、input_handler、input_handle等数据结构，它们的用途和功能如表17.1所示。

表17.1 关键数据结构

| 数 据 结 构 | 位 置 | 说 明 |
| --- | --- | --- |
| struct input_dev | input.h | 物理输入设备的基本数据结构，包含设备相关的一些信息 |
| struct input_handler | input.h | 事件处理结构体，定义怎么处理事件的逻辑 |
| struct input_handle | input.h | 用来创建input_dev和input_handler之间关系的结构体 |

## 8. 17.2.2 input_handler结构体

摘要：《Linux驱动开发入门与实战》第17章输入子系统设计，本章将介绍Linux输入子系统的驱动开发。Linux的输入子系统不仅支持鼠标、键盘等常规输入设备，而且还支持蜂鸣器、触摸屏等设备。本章将对Linux输入子系统进行详细的分析。本节为大家介绍input_handler结构体。 标签：input子系统 Linux驱动开发 Linux驱动开发入门与实战

17.2.2 input_handler结构体

input_handler是输入设备的事件处理接口，为处理事件提供一个统一的函数模板，程序员应该根据具体的需要实现其中的一些函数，并将其注册到输入子系统中。该结构体的定义如下：01 struct input_handler { 02 void *private; 03 void (*event)(struct input_handle *handle, unsigned int type, unsigned int code, int value); 04 int (*connect)(struct input_handler *handler, struct input_dev *dev, const struct input_device_id *id); 05 void (*disconnect)(struct input_handle *handle); 06 void (*start)(struct input_handle *handle); 07 const struct file_operations *fops; 08 int minor; 09 const char *name; 10 const struct input_device_id *id_table; 11 const struct input_device_id *blacklist; 12 struct list_head h_list; 13 struct list_head node; 14 };

对该结构体简要分析如下。

第02行，定义了一个private指针，表示驱动特定的数据。这里的驱动指的就是handler处理器。

第03行，定义了一个event()处理函数，这个函数将被输入子系统调用去处理发送给设备的事件。例如将发送一个事件命令LED灯点亮，实际控制硬件的点亮操作就可以放在event()函数中实现。

第04行，定义了一个connect()函数，该函数用来连接handler和input_dev。在

input_attach_handler()函数的第10行，就是回调的这个自定义函数。
第05行，定义了一个disconnect()函数，这个函数用来断开handler和input_dev之间的联系。
第07行，表示handler实现的文件操作集，这里不是很重要。
第08行，表示设备的次设备号。
第09行，定义了一个name，表示handler的名字，显示在/proc/bus/input/handlers目录中。
第10行，定义了一个id_table表，表示驱动能够处理的表。
第11行，指向一个input_device_id表，这个表包含handler应该忽略的设备。
第12行，定义了一个链表h_list，表示与这个input_handler相联系的下一个handler。

第13行，定义了一个链表node，将其连接到全局的input_handler_list链表中，所有的input_handler都连接在其上。

## 9. 17.2.3 注册input_handler

摘要：《Linux驱动开发入门与实战》第17章输入子系统设计，本章将介绍Linux输入子系统的驱动开发。Linux的输入子系统不仅支持鼠标、键盘等常规输入设备，而且还支持蜂鸣器、触摸屏等设备。本章将对Linux输入子系统进行详细的分析。本节为大家介绍注册input_handler。 标签：Linux驱动开发 Linux驱动开发入门与实战
17.2.3 注册input_handler
input_register_handler()函数注册一个新的input handler处理器。这个handler将为输入设备使用，一个handler可以添加到多个支持它的设备中，也就是一个handler可以处理多个输入设备的事件。函数的参数传入简要注册的input_handler指针，该函数的代码如下：01 int input_register_handler(struct input_handler *handler) 02 { 03 struct input_dev *dev; 04 int retval; 05 retval = mutex_lock_interruptible(&input_mutex); 06 if (retval) 07

```
return retval;  08      INIT_LIST_HEAD(&handler->h_list);  09      if
(handler->fops != NULL) {  10          if (input_table[handler->minor >> 5])
{  11              retval = -EBUSY;  12              goto out;  13
}  14          input_table[handler->minor >> 5] = handler;  15      }  16
  list_add_tail(&handler->node, &input_handler_list);  17
list_for_each_entry(dev, &input_dev_list, node)  18
input_attach_handler(dev, handler);  19      input_wakeup_procfs_readers();
20   out:  21      mutex_unlock(&input_mutex);  22      return retval;  23
}
```

下面对这个函数进行简要的分析。

第03、04行，定义了一些局部变量。

第05～07行，对input_mutex进行了加锁。当加锁失败后，则返回。

第08行，初始化h_hlist链表，该链表连接与这个input_handler相联系的下一个handler。

第09～14行，其中的handler->minor表示对应input设备结点的次设备号。以handler->minor右移5位作为索引值插入到input_table[ ]中，

第16行，调用list_add_tail()函数，将handler加入全局的input_handler_list链表中，该链表包含了系统中所有的input_handler。

第17、18行，主要调用了input_attach_handler()函数。该函数在17.1.2节input_register_device()函数的第35行曾详细的介绍过。input_attach_handler()函数的作用是匹配input_dev_list链表中的input_dev与handler。如果成功会将input_dev与handler联系起来。

第19行，与procfs文件系统有关，这里不需要关心。

第20～22行，解互斥锁并退出。

## *10. 17.2.4 input_handle结构体*

摘要：《Linux驱动开发入门与实战》第17章输入子系统设计，本章将介绍Linux输入

子系统的驱动开发。Linux的输入子系统不仅支持鼠标、键盘等常规输入设备，而且还支持蜂鸣器、触摸屏等设备。本章将对Linux输入子系统进行详细的分析。本节为大家介绍input_handle结构体。 标签：input_handle结构体 Linux驱动开发 Linux驱动开发入门与实战

17.2.4 input_handle结构体

input_register_handle()函数用来注册一个新的handle到输入子系统中。

input_handle的主要功能是用来连接input_dev和input_handler。其结构如下：

```
01 struct input_handle { 02 void *private; 03 int open; 04 const char *name; 05 struct input_dev *dev; 06 struct input_handler *handler; 07 struct list_head d_node; 08 struct list_head h_node; 09 };
```

下面对该结构体的成员进行简要的介绍。

第02行，定义了private表示handler特定的数据。

第03行，定义了一个open变量，表示handle是否正在被使用，当使用时，会将事件分发给设备处理。

第04行，定义了一个name变量，表示handle的名字。

第05行，定义了dev变量指针，表示该handle依附的input_dev设备。

第06行，定义了一个handler变量指针，指向input_handler，该handler处理器就是与设备相关的处理器。

第07行，定义了一个d_node变量，使用这个变量将handle放到设备相关的链表中，也就是放到input_dev->h_list表示的链表中。

第08行，定义了一个h_node变量，使用这个变量将handle放到input_handler相关的链表中，也就是放到handler->h_list表示的链表中。

## *11. 17.2.5 注册input_handle*

摘要：《Linux驱动开发入门与实战》第17章输入子系统设计，本章将介绍Linux输入

子系统的驱动开发。Linux的输入子系统不仅支持鼠标、键盘等常规输入设备，而且还支持蜂鸣器、触摸屏等设备。本章将对Linux输入子系统进行详细的分析。本节为大家介绍注册input_handle。 标签：input_handle Linux驱动开发 Linux驱动开发入门与实战

17.2.5 注册input_handle

input_handle是用来连接input_dev和input_handler的一个中间结构体。事件通过input_handle从input_dev发送到input_handler，或者从input_handler发送到input_dev进行处理。在使用input_handle之前，需要对其进行注册，注册函数是input_register_handle()。

1. 注册函数input_register_handle()

input_register_handle()函数用来注册一个新的handle到输入子系统中。该函数接收一个input_handle类型的指针，该变量要在注册前对其成员初始化。

input_register_handle()函数的代码如下：

```
01 int input_register_handle(struct input_handle *handle) 02 { 03 struct input_handler *handlehandler = handle->handler; 04 struct input_dev *dev = handle->dev; 05 int error; 06 error = mutex_lock_interruptible(&dev->mutex); 07 if (error) 08 return error; 09 list_add_tail_rcu(&handle->d_node, &dev->h_list); 10 mutex_unlock(&dev->mutex); 11 synchronize_rcu(); 12 list_add_tail(&handle->h_node, &handler->h_list); 13 if (handler->start) 14 handler->start(handle); 15 return 0; 16 }
```

下面对该函数进行简要的解释。

第03行，从handle中取出一个指向input_handler的指针，为下面的操作使用。

第04行，从handle中取出一个指向input_dev的指针，为下面的操作使用。

第06行，给竞争区域加一个互斥锁。

第09行，调用list_add_tail_rcu()函数将handle加入输入设备的dev->h_list链表中。

第12行，调用list_add_tail()函数将handle加入input_handler的handler->h_list链表中。

第13、14行，如果定义了start()函数，则调用它。

2. input_dev、input_handler和input_handle之间的关系

从以上的代码分析可以看出，input_dev、input_handler和handle三者之间是相互联系的，如图17.4所示。 （点击查看大图）图17.4 input_dev、input_handler和input_handle的关系结点1、2、3表示input_dev设备，其通过input_dev->node变量连接到全局输入设备链表input_dev_list中。结点4、5、6表示input_handler处理器，其通过input_handler->node连接到全局handler处理器链表input_handler_list中。结点7是一个input_handle的结构体，其用来连接input_dev和input_handler。

input_handle的dev成员指向了对应的input_dev设备，input_handle的handler成员指向了对应的input_handler。另外，结点7的input_handle通过d_node连接到了结点2的

input_dev上的h_list链表上。另一方面，结点7的input_handle通过h_node连接到了结点5的input_handler的h_list链表上。通过这种关系，将input_dev和input_handler联系了起来。

## 12. 17.3.1 子系统初始化函数input_init()

摘要：《Linux驱动开发入门与实战》第17章输入子系统设计，本章将介绍Linux输入子系统的驱动开发。Linux的输入子系统不仅支持鼠标、键盘等常规输入设备，而且还支持蜂鸣器、触摸屏等设备。本章将对Linux输入子系统进行详细的分析。本节为大家介绍子系统初始化函数input_init()。 标签：子系统 Linux驱动开发 Linux驱动开发入门与实战

17.3 input子系统

为了对输入子系统有一个清晰的认识，本节将分析输入系统的初始化过程。在Linux中，输入子系统作为一个模块存在，向上，为用户层提供接口函数，向下，为驱动层程序提供统一的接口函数。这样，就能够使输入设备的事件通过输入子系统发送给用户层应用程序，用户层应用程序也可以通过输入子系统通知驱动程序完成某项功能。

17.3.1 子系统初始化函数input_init()

输入子系统作为一个模块存在，必然有一个初始化函数。在/drivers/input/input.c文件中定义了输入子系统的初始化函数input_init()，该函数的代码如下：

```
01 static int __init input_init(void) 02 { 03 int err; 04 err = class_register(&input_class); 05 if (err) { 06 printk(KERN_ERR "input: unable to register input_dev class\n"); 07 return err; 08 } 09 err = input_proc_init(); 10 if (err) 11 goto fail1; 12 err = register_chrdev(INPUT_MAJOR, "input", &input_fops); 13 if (err) { 14 printk(KERN_ERR "input: unable to register char major %d", INPUT_MAJOR); 15 goto fail2; 16 } 17 return 0; 18
```

fail2: input_proc_exit(); 19 fail1: class_unregister(&input_class); 20 return err; 21 }
下面对该函数进行简要的分析。
第04行，调用class_register()函数先注册了一个名为input的类。所有input device都属于这个类。在sysfs中表现就是，所有input device所代表的目录都位于/dev/class/input下面。input_class类的定义只是一个名字，代码如下：struct class input_class = { .name = "input", };
第09行，调用input_proc_init()在/proc下面建立相关的交互文件。
第12行，调用register_chrdev()注册了主设备号为INPUT_MAJOR(13)。次设备号为0～255的字符设备。它的操作指针为input_fops。在这里，可以看到所有主设备号13的字符设备的操作最终都会转入到input_fops中。例如/dev/input/event0~/dev/input/event4的主设备号为13，对其的操作会落在input_fops中。input_fops只定义了一个input_open_file()函数，input_fops的定义代码如下：static const struct file_operations input_fops = { .owner = THIS_MODULE, .open = input_open_file, };

## 13. 17.3.2 文件打开函数input_open_file(

摘要：《Linux驱动开发入门与实战》第17章输入子系统设计，本章将介绍Linux输入子系统的驱动开发。Linux的输入子系统不仅支持鼠标、键盘等常规输入设备，而且还支持蜂鸣器、触摸屏等设备。本章将对Linux输入子系统进行详细的分析。本节为大家介绍文件打开函数input_open_file()。 标签：Linux驱动开发 Linux驱动开发入门与实战
17.3.2 文件打开函数input_open_file()
文件操作指针中定义了input_open_file()函数，该函数将控制转到input_handler中定义的fops文件指针的open()函数。该函数在input_handler中实现，这样就使不同的handler处理器对应了不同的文件打开方法，为完成不同功能提供了方便。
input_open_file()函数的代码如下：01 static int input_open_file(struct inode

```
*inode, struct file *file)  02  {  03      struct input_handler *handler;
04      const struct file_operations *old_fops, *new_fops = NULL;  05
int err;  06      lock_kernel();  07      /* No load-on-demand here? */  08
    handler = input_table[iminor(inode) >> 5];  09      if (!handler ||
!(new_fops = fops_get(handler->fops))) {  10          err = -ENODEV;  11
     goto out;  12      }  13      if (!new_fops->open) {  14
fops_put(new_fops);  15          err = -ENODEV;  16          goto out;  17
   }  18      old_fops = file->f_op;  19      file->f_op = new_fops;  20
 err = new_fops->open(inode, file);  21      if (err) {  22
fops_put(file->f_op);  23          file->f_op = fops_get(old_fops);  24
}  25      fops_put(old_fops);  26  out:  27      unlock_kernel();  28
return err;  29  }
```

下面对该函数进行简要的分析。

第03～05行，声明一些局部变量，供下面的操作使用。

第08行，出现了熟悉的input_table[]数组。iminor(inode)为打开文件所对应的次设备号。input_table是一个struct input_handler全局数组，只有8个元素，其定义为：

```
static struct input_handler *input_table[8];
```

在这里，首先将设备结点的次设备号右移5位做为索引值到input_table中取对应项，从这里也可以看到，一个handler代表32（1<<5）个设备结点，也就是一个handler最多可以处理32个设备结点。因为在input_table中取值是以次备号右移5位为索引的，即第5位相同的次备号对应的是同一个索引。回忆input_register_handler()函数的第14行input_table[handler->minor >> 5] = handler，其将handler赋给了input_table数组，所使用的规则也是右移5位。

第09～12行，在input_table中找到对应的handler之后，就会检验这个handler是否存在，如果没有，则返回一个设备不存在的错误。在存在的情况下，从handler->fops中获得新的文件操作指针file_operation，并增加引用计数。此后，对设备的操作都通过新的文件操作指针new_fops来完成。

第13～17行，判断new_fops->open()函数是否定义，如果没有定义，则表示设备不存在。

第20行，使用新的open()函数，重新打开设备。

## *14. 17.4.1 evdev的初始化*

摘要：《Linux驱动开发入门与实战》第17章输入子系统设计，本章将介绍Linux输入子系统的驱动开发。Linux的输入子系统不仅支持鼠标、键盘等常规输入设备，而且还支持蜂鸣器、触摸屏等设备。本章将对Linux输入子系统进行详细的分析。本节为大家介绍evdev的初始化。 标签：evdev Linux驱动开发 Linux驱动开发入门与实战

17.4 evdev输入事件驱动分析

evdev输入事件驱动，为输入子系统提供了一个默认的事件处理方法。其接收来自底层驱动的大多数事件，并使用相应的逻辑对其进行处理。evdev输入事件驱动从底层接收事件信息，将其反映到sys文件系统中，用户程序通过对sys文件系统的操作，就能够达到处理事件的能力。下面先对evdev的初始化进行简要的分析。

17.4.1 evdev的初始化

evdev以模块的方式被组织在内核中，与其他模块一样，也具有初始化函数和卸载函数。evdev的初始化主要完成一些注册工作，使内核认识evdev的存在。

1. evdev_init()初始化函数

evdev模块定义在/drivers/input/evdev.c文件中，该模块的初始化函数是evdev_init()。在初始化函数中注册了一个evdev_handler结构体，用来对一些通用的抽象事件进行统一处理，该函数的代码如下：01 static int __init evdev_init(void) 02 { 03 return input_register_handler(&evdev_handler); 04 }

第03行，调用input_register_handler()函数注册了evdev_handler事件处理器，input_register_handler()函数在前面已经详细解释过，这里将对其参数evdev_handler进行分析，其定义如下：01 static struct input_handler evdev_handler = { 02 .event = evdev_event, 03 .connect = evdev_connect, 04 .disconnect = evdev_disconnect, 05 .fops = &evdev_fops, 06 .minor = EVDEV_MINOR_BASE, 07 .name = "evdev", 08 .id_table = evdev_ids, 09 };

第06行，定义了minor为EVDEV_MINOR_BASE（64）。因为一个handler可以处理32个设备，所以evdev_handler所能处理的设备文件范围为（13,64）～（13,64+32），其中13是所有输入设备的主设备号。

第08行，定义了id_table结构。回忆前面几节的内容，由input_attach_handler()函数可知，input_dev与handler匹配成功的关键，在于handler中的blacklist和id_talbe.。Evdev_handler只定义了id_table，其定义如下：static const struct input_device_id evdev_ids[] = { { .driver_info = 1 }, /* Matches all devices */ { }, /* Terminating zero entry */ };

evdev_ids没有定义flags，也没有定义匹配属性值。这个evdev_ids的意思就是

：evdev_handler可以匹配所有input_dev设备，也就是所有的input_dev发出的事件，都可以由evdev_handler来处理。另外，从前面的分析可以知道，匹配成功之后会调用handler->connect()函数，对该函数的介绍如下：

2. evdev_connect()函数

evdev_handler的第3行定义了evdev_connect()函数。evdev_connect()函数主要用来连接input_dev和input_handler，这样事件的流通链才能建立。流通链建立后，事件才知道被谁处理，或者处理后将向谁返回结果。01 static int evdev_connect(struct input_handler *handler, struct input_dev *dev, 02 const struct input_device_id *id) 03 { 04 struct evdev *evdev; 05 int minor; 06 int error; 07 for (minor = 0; minor < EVDEV_MINORS; minor++) 08 if (!evdev_table[minor]) 09 break; 10 if (minor == EVDEV_MINORS) { 11 printk(KERN_ERR "evdev: no more free evdev devices\n"); 12 return -ENFILE; 13 } 14 evdev = kzalloc(sizeof(struct evdev), GFP_KERNEL); 15 if (!evdev) 16 return -ENOMEM; 17 INIT_LIST_HEAD(&evdev->client_list); 18 spin_lock_init(&evdev->client_lock); 19 mutex_init(&evdev->mutex); 20 init_waitqueue_head(&evdev->wait); 21 snprintf(evdev->name, sizeof(evdev->name), "event%d", minor); 22 evdev->exist = 1; 23 evdev->minorminor = minor; 24 evdev->handle.dev = input_get_device(dev); 25 evdev->handle.name = evdev->name; 26 evdev->handle.handler = handler; 27 evdev->handle.private = evdev; 28 dev_set_name(&evdev->dev, evdev->name); 29 evdev->dev.devt = MKDEV(INPUT_MAJOR, EVDEV_MINOR_BASE + minor); 30 evdev->dev.class = &input_class; 31 evdev->dev.parent = &dev->dev; 32 evdev->dev.release = evdev_free; 33 device_initialize(&evdev->dev); 34 error = input_register_handle(&evdev->handle); 35 if (error) 36 goto err_free_evdev; 37 error = evdev_install_chrdev(evdev); 38 if (error) 39 goto err_unregister_handle; 40 error = device_add(&evdev->dev); 41 if (error) 42 goto err_cleanup_evdev; 43 return 0; 44 err_cleanup_evdev: 45 evdev_cleanup(evdev); 46 err_unregister_handle: 47 input_unregister_handle(&evdev->handle); 48 err_free_evdev: 49 put_device(&evdev->dev); 50 return error; 51 }

下面对该函数进行简要的分析。

第04～06行，声明了一些必要的局部变量。

第07～13行，for循环中的EVDEV_MINORS定义为32，表示evdev_handler所表示的32个设备文件。evdev_talbe是一个struct evdev类型的数组，struct evdev是模块使用的封装结构，与具体的输入设备有关。第08行，这一段代码的在evdev_talbe找到为空的那

一项，当找到为空的一项，便结束for循环。这时，minor就是数组中第一项为空的序号。第10到13行，如果没有空闲的表项，则退出。
第14～16行，分配一个struct evdev的空间，如果分配失败，则退出。
第17～20行，对分配的evdev结构进行初始化，主要对链表、互斥锁和等待队列做必要的初始化。在evdev中，封装了一个handle结构，这个结构与handler是不同的。可以把handle看成是handler和input device的信息集合体，这个结构用来联系匹配成功的handler和input device。
第21行，对evdev命一个名字，这个设备的名字形如eventx，例如event1、event2和event3等。最大有32个设备，这个设备将在/dev/input/目录下显示。
第23～27行，对evdev进行必要的初始化。其中，主要对handle进行初始化，这些初始化的目的是使input_dev和input_handler联系起来。
第28～33行，在设备驱动模型中注册一个evdev->dev的设备，并初始化一个evdev->dev的设备。这里，使evdev->dev所属的类指向input_class。这样在/sysfs中创建的设备目录就会在/sys/class/input/下显示。
第34行，调用input_register_handle()函数注册一个input_handle结构体。
第37行，注册handle，如果成功，那么调用evdev_install_chrdev将evdev_table的minor项指向evdev.。
第40行，将evdev->device注册到sysfs文件系统中。
第41～50行，进行一些必要的错误处理。

## *15. 17.4.2 evdev设备的打开*

摘要：《Linux驱动开发入门与实战》第17章输入子系统设计，本章将介绍Linux输入子系统的驱动开发。Linux的输入子系统不仅支持鼠标、键盘等常规输入设备，而且还支持蜂鸣器、触摸屏等设备。本章将对Linux输入子系统进行详细的分析。本节为大家介绍evdev设备的打开。 标签：evdev Linux驱动开发 Linux驱动开发入门与实战
17.4.2 evdev设备的打开
用户程序通过输入子系统创建的设备结点函数open()、read()和write()等，打开和读

写输入设备。创建的设备结点显示在/dev/input/目录下，由eventx表示。

1. evdev_open()函数

对主设备号为INPUT_MAJOR的设备结点进行操作，会将操作集转换成handler的操作集。在evdev_handler中定义了一个fops集合，被赋值为evdev_fops的指针。evdev_fops就是设备结点的操作集，其定义代码如下：01 static const struct file_operations evdev_fops = { 02 .owner = THIS_MODULE, 03 .read = evdev_read, 04 .write = evdev_write, 05 .poll = evdev_poll, 06 .open = evdev_open, 07 .release = evdev_release, 08 .unlocked_ioctl = evdev_ioctl, 09 .fasync = evdev_fasync, 10 .flush = evdev_flush 11 };

evdev_fops结构体是一个file_operations的类型。当用户层调用类似代码open("/dev/input/event1", O_RDONLY)函数打开设备结点时，会调用evdev_fops中的evdev_read()函数，该函数的代码如下：01 static int evdev_open(struct inode *inode, struct file *file) 02 { 03 struct evdev *evdev; 04 struct evdev_client *client; 05 int i = iminor(inode) - EVDEV_MINOR_BASE; 06 int error; 07 if (i >= EVDEV_MINORS) 08 return -ENODEV; 09 error = mutex_lock_interruptible(&evdev_table_mutex); 10 if (error) 11 return error; 12 evdev = evdev_table[i]; 13 if (evdev) 14 get_device(&evdev->dev); 15 mutex_unlock(&evdev_table_mutex); 16 if (!evdev) 17 return -ENODEV; 18 client = kzalloc(sizeof(struct evdev_client), GFP_KERNEL); 19 if (!client) { 20 error = -ENOMEM; 21 goto err_put_evdev; 22 } 23 spin_lock_init(&client->buffer_lock); 24 client->evdevevdev = evdev; 25 evdev_attach_client(evdev, client); 26 error = evdev_open_device(evdev); 27 if (error) 28 goto err_free_client; 29 file->private_data = client; 30 return 0; 31 err_free_client: 32 evdev_detach_client(evdev, client); 33 kfree(client); 34 err_put_evdev: 35 put_device(&evdev->dev); 36 return error; 37 }

下面对该函数进行简要的分析。

第03、04行，定义了一些局部变量。

第05行，iminor(inode) - EVDEV_MINOR_BASE得到了在evdev_table[]中的序号，赋给变量i。

第09～17行，将数组evdev_table[]中对应的evdev取出，并调用get_device()增加引用计数。

第18～30行，分配并初始化一个client结构体，并将它和evdev关联起来。关联的内容是，将client->evdev指向它所表示的evdev，调用evdev_attach_client()将client挂

到evdev->client_list上。第29行，将client赋给file的private_data。在evdev中，这个操作集就是evdev_fops，对应的open()函数如下：static int evder_oper(struct inode *inode, struct file *file)
第26行，调用evdev_open_device()函数，打开输入设备。该函数的具体功能将在下面详细介绍。
第31～37行，进行一些错误处理。
2. evdev_open_device()函数
evdev_open_device()函数用来打开相应的输入设备，使设备准备好接收或者发送数据。evdev_open_device()函数先获得互斥锁，然后检查设备是否存在，并判断设备是否已经被打开。如果没有打开，则调用input_open_device()函数打开设备。
evdev_open_device()函数的代码如下：

```
01  static int evdev_open_device(struct evdev *evdev)  02  {  03      int retval;  04      retval = mutex_lock_interruptible(&evdev->mutex);  05  if (retval)  06          return retval;  07      if (!evdev->exist)  08      retval = -ENODEV;  09      else if (!evdev->open++) {  10  retval = input_open_device(&evdev->handle);  11          if (retval)  12          evdev->open--;  13      }  14      mutex_unlock(&evdev->mutex);  15      return retval;  16  }
```

下面对该函数进行简要的分析。
第07行，判断该设备是否存在，如果不存在则返回设备不存在。
第09～12行，如果evdev是第一次打开，就会调用input_open_device()打开evdev对应的handle；否则不做任何操作返回。
3. input_open_device()函数
在这个函数中，递增handle的打开计数。如果是第一次打开，则调用input_dev的open()函数。

```
01  int input_open_device(struct input_handle *handle)  02  {  03  struct input_dev *dev = handle->dev;  04      int retval;  05      handle->open++;  06      ...  07      if (!dev->users++ && dev->open)  08  retval = dev->open(dev);  09      ...  10      return retval;  11  }
```

## *16. 17.5 小结*

摘要：《Linux驱动开发入门与实战》第17章输入子系统设计，本章将介绍Linux输入子系统的驱动开发。Linux的输入子系统不仅支持鼠标、键盘等常规输入设备，而且还支持蜂鸣器、触摸屏等设备。本章将对Linux输入子系统进行详细的分析。本节为这一章的小结部分。 标签：Linux驱动开发 Linux驱动开发入门与实战

## 17.5 小结

在本章中，分析了整个输入子系统的架构。Linux设备驱动采用了分层的模式，从最下层的设备模型到设备、驱动、总线再到input子系统最后到input device。这样的分层结构使得最上层的驱动不必关心下层是怎么实现的，而下层驱动又为多种型号同样功能的驱动提供了一个统一的接口。