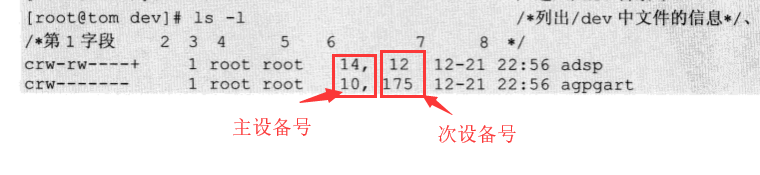
1. **设备号**

**字符设备与块设备**

字符设备：只能按字节顺序读取的设备，如鼠标，键盘

块设备：能够随机读取一定长度数据的设备，如磁盘，u盘

每个字符设备或块设备在/dev下都有一个对应的设备文件



**主设备号和次设备号**

一个字符设备或块设备都有一个设备号（由主设备号和次设备号组成）

在Linux中用dev\_t类型（在32位系统中是32bit）表示设备号，其中高12位是主设备号，低20位是次设备号

typedef u\_long deb\_t

一般一个主设备号对应一个驱动程序，次设备号对应驱动程序所实现的设备

从设备号中获取主、次设备号（dev为设备号）：

获取主设备号：MAJOR(dev)

获取次设备号：MINOR(dev)

主、次设备号合成设备号（ma为主设备号，mi为次设备号）：MKDEV(ma, mi)

**分配设备号**

编写驱动程序第一步首先要获取设备号

1. 静态分配设备号

静态分配设备号指程序员自己指定一个设备号，指定的设备号不应该已存在

cat /proc/devices查看系统中以存在的设备号

1. 动态分配设备号

动态分配函数如下，该函数定义在<fs/char\_dev.c>中

int register\_chrdev\_region(dev\_t from, unsigned count, const char \* name)

From：要分配的起始设备号，一般只提供主设备号，次设备号设为0

Count：要分配设备号的个数

Name：分配的这些设备号的名称

返回值：成功0

人指定主设备号可能会发生冲突，所以引入另一个函数

int alloc\_chrdev\_region(dev\_t \*dev, unsigned baseminor, unsigned count, const char \* name)

Dev：作为输出参数，存放动态分配连续设备号的第一个设备号

Baseminor：次设备号的开始号数，一般为0

Count：分配设备号的个数

Name：分配的这些设备号的名称

返回值：成功0

释放设备号，在模块卸载时应释放设备号

void unregister\_chrdev\_region(dev\_t from, unsigned count)

1. **cdev结构体**

使用cdev结构体表示字符设备，该结构如下

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;

    /\*该结构是inode链表的链表头，将使用该驱动的字符设备文件连接成一个链表\*/

    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\*/

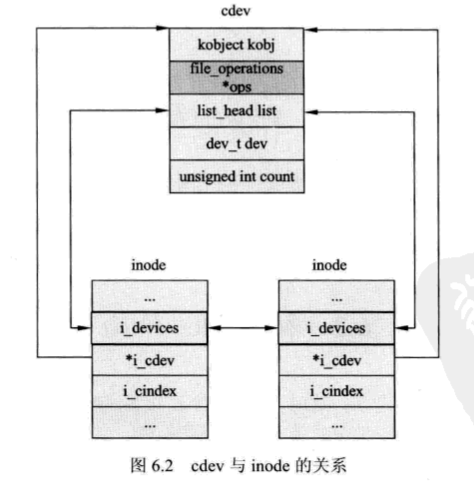
    unsigned int count;

    /\*使用该字符设备驱动的设备数量\*/

};

Ops：该结构体下面介绍

List：该设备模块对应的设备文件链表



Inode结构体表示/dev目录下的设备文件

Conunt：目前有多数字符设备在使用该驱动程序，如果count不为0，则内核不会卸载该模块

**file\_operations 结构体**

file\_operations是对设备进行操作的抽象结构体，其提供的方法使我们能像操作文件一样操作设备

下面对file\_ operations 结构体的重要成员进行讲解。

lseek()：函数用来改变文件中的当前读/写位置,并将新位置返回。

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

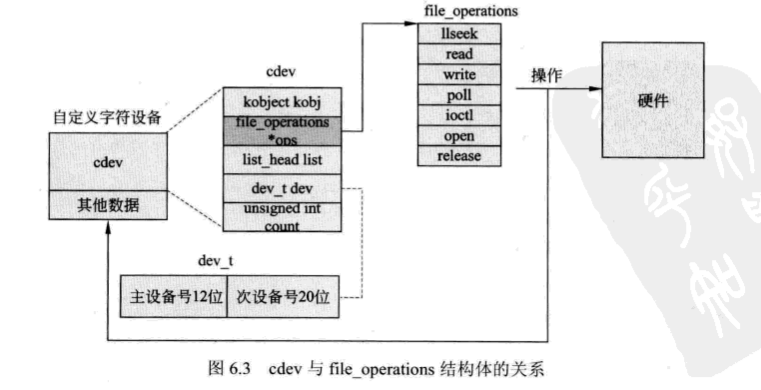
个负的错误码。

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。

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。

**file\_operations与cdev结构体的关系**



**inode结构体**

内核使用inode结构体表示一个文件。inode 一般作为file\_operations 结构中函数的参数。

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。

dev\_t i\_rdev：表示设备文件对应的设备号。

struct list\_head i\_devices：链表头，用于将inode结构体连接成列表

struct cdev\* i\_ cdev：该成员指向cdev设备

**file 结构体**

内核使用file结构体表示一个以打开的文件，file一般作为file\_operations 结构中函数的参数。

**cdev的注册于注销**

初始化：

Void cdev\_init(struct cdev \*dev, struct file\_operations \*fops);

添加cdev到内核：

Int cdev\_add(struct cdev \*dev, dev\_t num, unsigned int count);

删除：

Void cdev\_del(struct cdev \*dev);

**Scull\_dev结构**

一般驱动程序会自定义自己的cdev结构，在接下来的示例中，我们会使用scull\_dev结构。

这里对scull\_dev做一个介绍

每个scull\_dev结构最多有1000个scull\_qset指针数组，每个scull\_qset指向4000 字节的区域。我们称每个内存区域为一个量子（quantum）

struct scull\_dev

{

    struct scull\_qset \*data; /\* Pointer to first quantum set \*/

    int quantum;             /\* 每个量子的大小 \*/

    int qset;                /\* 每个scull\_qset中含有的量子的数量 \*/

    unsigned long size;      /\* 设备保存的数据大小 \*/

    unsigned int access\_key; /\* used by sculluid and scullpriv \*/

    struct semaphore sem;    /\* mutual exclusion semaphore \*/

    struct cdev cdev;        /\* Char device structure \*/

};

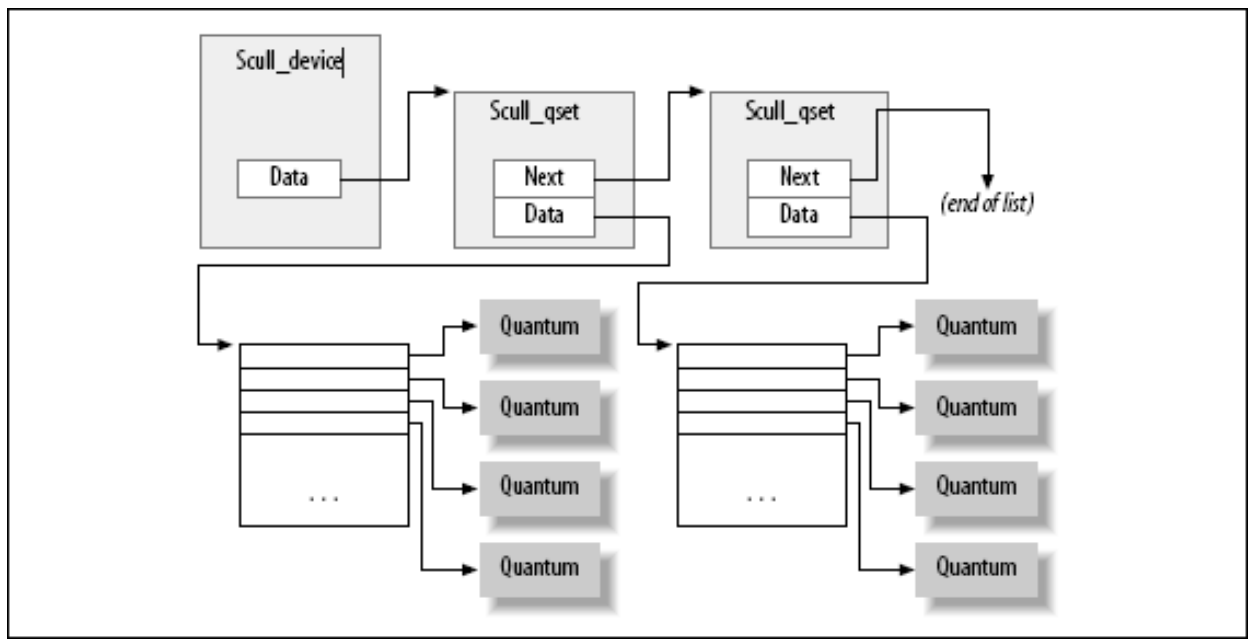
struct scull\_qset

{

    void \*\*data;

    struct scull\_qset \*next;

};



1. **字符设备驱动的组成**

**字符设备加载和卸载函数**

在设备的加载函数中应该完成设备号的申请和cdev的注册

在设备的卸载函数中应该完成设备号的释放和cdev的注销

如果需要特定的cdev结构，则应在cdev的末尾添加

常见的代码结构如下：

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\*/

    if (XXX\_major)

    {

        /\*静态申请设备号\*/

        result = register\_chrdev\_region(xxx\_devno, 1, "DEV NAME");

    }

    else

    {

        /\*动态申请设备号\*/

        result = alloc\_chrdev\_region(&xxx\_devno, 0, 1, "DEV NAME ");

}

    /\*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\*/

}

**file\_operations 结构体和其成员函数**

驱动程序实现file\_operations 结构体的成员函数，用户程序可以通过内核来调用这些函数，从而实现应用程序控制设备。大多数字符设备驱动都会实现read()、write()和 ioct()函数

**驱动程序与应用程序的数据交换**

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。

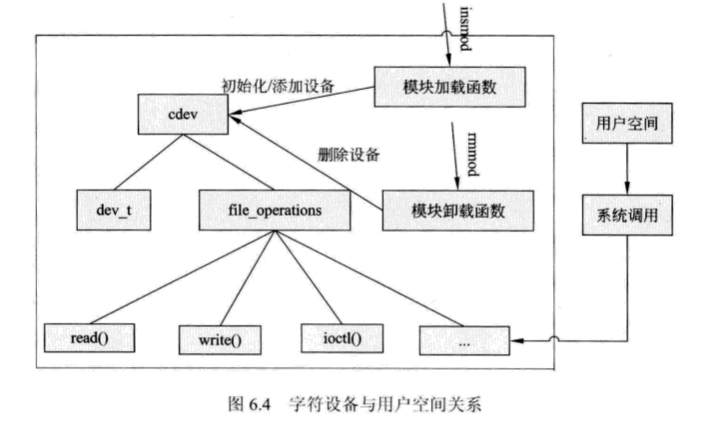
内核提供的专用函数，完成数据在应用程序空间和驱动程序空间的交换

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);



1. **file\_operations 结构体**

**open方法**

Int (\*open)(struct inode \*inode, struct file \*filp)

Inode：要打开的设备文件

其字段i\_cdev指向我们的cdev结构，但一般我们希望获取的是我们自定义的xxx\_cdev结构（如第3节所定义的xxx\_cdev），这里可以使用内核定义的宏container\_of

Struct xxx\_cdev \*dev = container\_of(inode->i\_cdev, struct xxx\_dev, cdev);

Filp：内核以初始化的 打开的设备文件 结构

我们一般会利用该结构的private\_data字段来保存我们设备文件的一些自定义数据

示例：

int scull\_open(struct inode \*inode, struct file \*filp)

{

    struct scull\_dev \*dev;

    dev = container\_of(inode->i\_cdev, struct scull\_dev, cdev);  // 获取自定义的cdev scull\_dev

    filp->private\_data = dev; // 在文件中保存 scull\_dev 的地址

    if ((filp->f\_flags & O\_ACCMODE) == O\_WRONLY)

    {

        scull\_trim(dev); /\* ignore errors \*/

    }

    return 0; /\* success \*/

}

**Release方法**

Release负责释放由open分配的资源

file结构有一个引用计数器，当执行fork或者dup是计数器会增加，当执行close系统调用时计数器会减少，当计数器为0时，内核才会调用release方法

我们上述open并未分配资源，所以release并不需要做什么

int scull\_release(struct inode \*inode, struct file \*filp)

{

 return 0;

}

**read方法**

ssize\_t read(struct file \*filp, char \_\_user \*buff, size\_t count, loff\_t \*offp);

filp 是文件指针

count 是请求的传输数据大小.

buff 参数 指向持有被写入数据的缓存, 或者放入新数据的空缓存.

buff 参数是用户空间指针. 因此, 它不能被内核代码直接解引用

内核与应用空间数据交换需要使用如下函数：

unsigned long copy\_to\_user(void \_\_user \*to,const void \*from,unsigned long count);

unsigned long copy\_from\_user(void \*to,const void \_\_user \*from,unsigned long count);

offp 是一个指 针指向一个"long offset type"对象, 它指出用户正在存取的文件位置.

返回值

如果返回值等于count，则数据传输成功

如果返回值小于count，则数据部分传输成功

如果返回值等于0，则以达到文件末尾

如果小于0，则发生错误

示例：

ssize\_t scull\_read(struct file \*filp, char \_\_user \*buf, size\_t count,

                   loff\_t \*f\_pos)

{

    struct scull\_dev \*dev = filp->private\_data;     // 我们在open的时候将private\_data存放scull\_dev的指针

    struct scull\_qset \*dptr;                        /\* 第一个scull\_qset \*/

    int quantum = dev->quantum, qset = dev->qset;

    int itemsize = quantum \* qset;                  /\* 计算scull\_qset的大小 \*/

    int item, s\_pos, q\_pos, rest;

    ssize\_t retval = 0;

    if (down\_interruptible(&dev->sem))              // 上锁

        return -ERESTARTSYS;

    if (\*f\_pos >= dev->size)                        // 如果读取位置超过设备大小

        goto out;

    if (\*f\_pos + count > dev->size)                 // 如果读取的部分数据超过设备大小

        count = dev->size - \*f\_pos;

    item = (long)\*f\_pos / itemsize;                 // 读取的scull\_qset位置

    rest = (long)\*f\_pos % itemsize;

    s\_pos = rest / quantum;                         // 读取的quantum位置

q\_pos = rest % quantum;                         // 读取的quantum内的位置

    /\* follow the list up to the right position (defined elsewhere) \*/

    dptr = scull\_follow(dev, item);

    if (dptr == NULL || !dptr->data || !dptr->data[s\_pos])

        goto out;

    if (count > quantum - q\_pos)                    // 如果读取我的数据跨越两个量子（quantum），则修改读取大小

        count = quantum - q\_pos;

    if (copy\_to\_user(buf, dptr->data[s\_pos] + q\_pos, count))

    {

        retval = -EFAULT;

        goto out;

}

    \*f\_pos += count;

    retval = count;

out:

    up(&dev->sem);          // 解锁

    return retval;          // 返回实际读取的大小

}

**write方法**

ssize\_t write(struct file \*filp, const char \_\_user \*buff, size\_t count, loff\_t \*offp);

参数说明参考read方法

示例：

ssize\_t scull\_write(struct file \*filp, const char \_\_user \*buf, size\_t count, loff\_t \*f\_pos)

{

    struct scull\_dev \*dev = filp->private\_data;     // 在open中我们将private\_data保存为scull\_dev的指针

    struct scull\_qset \*dptr;

    int quantum = dev->quantum, qset = dev->qset;   // 取出量子大小与scull\_qset中的量子数据

    int itemsize = quantum \* qset;                  // 计算scull\_qset的大小

    int item, s\_pos, q\_pos, rest;

    ssize\_t retval = -ENOMEM;                       /\* 实际写入的大小 \*/

    if (down\_interruptible(&dev->sem))              // 上锁

        return -ERESTARTSYS;

    /\* 计算要写入数据的位置在哪一个scull\_qset的哪一个量子的哪一个位置 \*/

    item = (long)\*f\_pos / itemsize;

    rest = (long)\*f\_pos % itemsize;

    s\_pos = rest / quantum;

q\_pos = rest % quantum;

    /\* follow the list up to the right position \*/

    dptr = scull\_follow(dev, item);

    if (dptr == NULL)

        goto out;

    // 如果要写入数据的scull\_qset没有分配内存，则分配内存

    if (!dptr->data)

    {

        dptr->data = kmalloc(qset \* sizeof(char \*), GFP\_KERNEL);

        if (!dptr->data)

            goto out;

        memset(dptr->data, 0, qset \* sizeof(char \*));

}

    // 如果要写入的量子还没有分配内存，则分配内存

    if (!dptr->data[s\_pos])

    {

        dptr->data[s\_pos] = kmalloc(quantum, GFP\_KERNEL);

        if (!dptr->data[s\_pos])

            goto out;

}

    /\* 如果要写入的数据长度跨越两个量子，则修改写入的长度 \*/

    if (count > quantum - q\_pos)

        count = quantum - q\_pos;

    // 复制用户空间的数据到量子中

    if (copy\_from\_user(dptr->data[s\_pos] + q\_pos, buf, count))

    {

        retval = -EFAULT;

        goto out;

    }

    \*f\_pos += count;

    retval = count;

    /\* 更新设备数据大小 \*/

    if (dev->size < \*f\_pos)

        dev->size = \*f\_pos;

out:

    up(&dev->sem);

    return retval;      // 返回实际写入长度

}