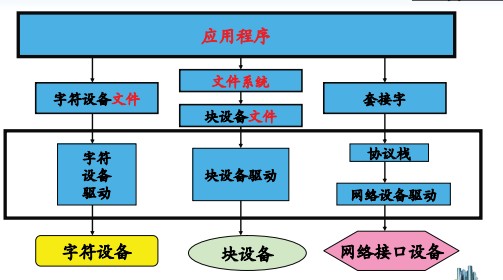
李沛冀，1212300209，Unix系统分析作业3

参考网站 http://blog.jobbole.com/86531/

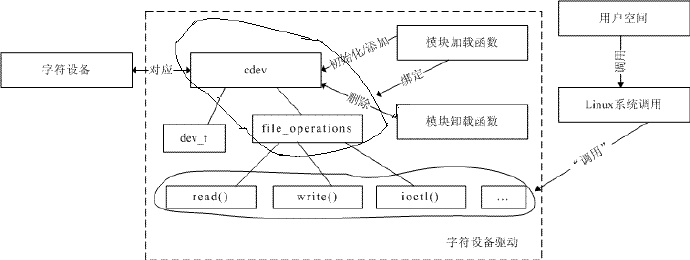
**一、linux系统将设备**分为3类：字符设备、块设备、网络设备。使用驱动程序：



* 字符设备：是指只能一个字节一个字节读写的设备，不能随机读取设备内存中的某一数据，读取数据需要按照先后数据。字符设备是面向流的设备，常见的字符设备有鼠标、键盘、串口、控制台和LED设备等。
* 块设备：是指可以从设备的任意位置读取一定长度数据的设备。块设备包括硬盘、磁盘、U盘和SD卡等。

每一个字符设备或块设备都在/dev目录下对应一个设备文件。linux用户程序通过设备文件（或称设备节点）来使用驱动程序操作字符设备和块设备。

**二、字符设备、字符设备驱动与用户空间访问该设备的程序三者之间的关系。**

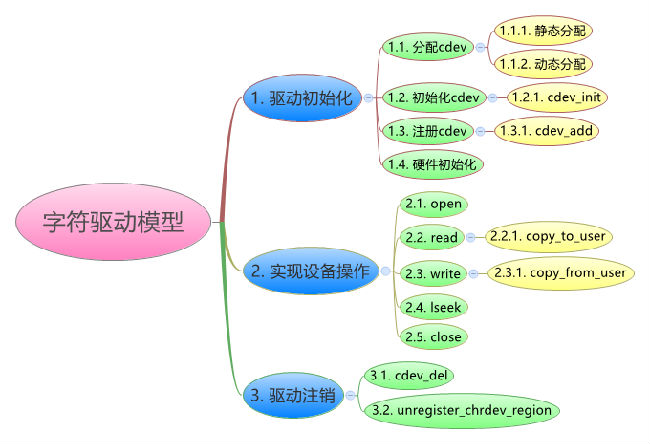


如图，在Linux内核中使用cdev结构体来描述字符设备，通过其成员dev\_t来定义设备号（分为主、次设备号）以确定字符设备的唯一性。通过其成员file\_operations来定义字符设备驱动提供给VFS的接口函数，如常见的open()、read()、write()等。

在Linux字符设备驱动中，模块加载函数通过register\_chrdev\_region( ) 或alloc\_chrdev\_region( )来静态或者动态获取设备号，通过cdev\_init( )建立cdev与file\_operations之间的连接，通过cdev\_add( )向系统添加一个cdev以完成注册。模块卸载函数通过cdev\_del( )来注销cdev，通过unregister\_chrdev\_region( )来释放设备号。

用户空间访问该设备的程序通过Linux系统调用，如open( )、read( )、write( )，来“调用”file\_operations来定义字符设备驱动提供给VFS的接口函数。

**三、字符设备驱动模型**



**1. 驱动初始化**

**1.1. 分配cdev**

在2.6的内核中使用cdev结构体来描述字符设备，在驱动中分配cdev,主要是分配一个cdev结构体与申请设备号，以按键驱动为例：

|  |
| --- |
| 1 /\*……\*/   2 /\* 分配cdev\*/   3 struct cdev btn\_cdev;   4 /\*……\*/   5 /\* 1.1 申请设备号\*/   6 if(major){   7 //静态   8 dev\_id = MKDEV(major, 0);   9 register\_chrdev\_region(dev\_id, 1, "button");  10 } else {  11 //动态  12 alloc\_chardev\_region(&dev\_id, 0, 1, "button");  13 major = MAJOR(dev\_id);  14 }  15 /\*……\*/ |

从上面的代码可以看出，申请设备号有动静之分，其实设备号还有主次之分。

在Linux中以主设备号用来标识与设备文件相连的驱动程序。次编号被驱动程序用来辨别操作的是哪个设备。cdev 结构体的 dev\_t 成员定义了设备号，为 32 位，其中高 12 位为主设备号，低20 位为次设备号。

设备号的获得与生成：

获得：主设备号：MAJOR(dev\_t dev);

次设备号：MINOR(dev\_t dev);

生成：MKDEV(int major,int minor);

设备号申请的动静之分：

静态：

|  |  |
| --- | --- |
| 1  2 | 1 int register\_chrdev\_region(dev\_t from, unsigned count, const char \*name)；  2 /\*功能：申请使用从from开始的count 个设备号(主设备号不变，次设备号增加）\*/ |

静态申请相对较简单，但是一旦驱动被广泛使用,这个随机选定的主设备号可能会导致设备号冲突，而使驱动程序无法注册。

动态：

|  |  |
| --- | --- |
| 1  2 | 1 int alloc\_chrdev\_region(dev\_t \*dev, unsigned baseminor, unsigned count,const char \*name)；  2 /\*功能：请求内核动态分配count个设备号，且次设备号从baseminor开始。\*/ |

动态申请简单，易于驱动推广，但是无法在安装驱动前创建设备文件（因为安装前还没有分配到主设备号）。

**1.2. 初始化cdev**

|  |  |
| --- | --- |
| 1  2 | 1 void cdev\_init(struct cdev \*, struct file\_operations \*);  2 cdev\_init()函数用于初始化 cdev 的成员，并建立 cdev 和 file\_operations 之间的连接。 |

**1.3. 注册cdev**

|  |  |
| --- | --- |
| 1  2 | 1 int cdev\_add(struct cdev \*, dev\_t, unsigned);  2 　　　　 cdev\_add()函数向系统添加一个 cdev，完成字符设备的注册。 |

**1.4. 硬件初始化**

硬件初始化主要是硬件资源的申请与配置，以TQ210的按键驱动为例：

|  |  |
| --- | --- |
| 1  2  3  4  5 | 1 /\* 1.4 硬件初始化\*/  2 //申请GPIO资源  3 gpio\_request(S5PV210\_GPH0(0), "GPH0\_0");  4 //配置输入  5 gpio\_direction\_input(S5PV210\_GPH0(0)); |

**2.实现设备操作**

用户空间的程序以访问文件的形式访问字符设备，通常进行open、read、write、close等系统调用。而这些系统调用的最终落实则是file\_operations结构体中成员函数，它们是字符设备驱动与内核的接口。以TQ210的按键驱动为例：

|  |
| --- |
| 1 /\*设备操作集合\*/  2 static struct file\_operations btn\_fops = {  3 .owner = THIS\_MODULE,  4 .open = button\_open,  5 .release = button\_close,  6 .read = button\_read  7 }; |

上面代码中的button\_open、button\_close、button\_read是要在驱动中自己实现的。file\_operations结构体成员函数有很多个，下面就选几个常见的来展示：

**2.1. open（）函数**

原型：

|  |
| --- |
| 1 int(\*open)(struct inode \*, struct file\*);  2 /\*打开\*/ |

**2.2. read( )函数**

原型：

|  |
| --- |
| ssize\_t(\*read)(struct file \*, char \_\_user\*, size\_t, loff\_t\*);  /\*用来从设备中读取数据，成功时函数返回读取的字节数，出错时返回一个负值\*/ |

**2.3. write( )函数**

原型：

|  |
| --- |
| 1 ssize\_t(\*write)(struct file \*, const char\_\_user \*, size\_t, loff\_t\*);  2 /\*向设备发送数据，成功时该函数返回写入的字节数。如果此函数未被实现，  3 当用户进行write()系统调用时，将得到-EINVAL返回值\*/ |

**2.4. close( )函数**

原型：

|  |
| --- |
| 1 int(\*release)(struct inode \*, struct file\*);  2 /\*关闭\*/ |

**2.5. 补充说明**

1. 在Linux字符设备驱动程序设计中，有3种非常重要的数据结构：struct file、struct inode、struct file\_operations。

struct file 代表一个打开的文件。系统中每个打开的文件在内核空间都有一个关联的struct file。它由内核在打开文件时创建, 在文件关闭后释放。其成员loff\_t f\_pos 表示文件读写位置。

struct inode 用来记录文件的物理上的信息。因此,它和代表打开文件的file结构是不同的。一个文件可以对应多个file结构,但只有一个inode结构。其成员dev\_t i\_rdev表示设备号。

struct file\_operations 一个函数指针的集合，定义能在设备上进行的操作。结构中的成员指向驱动中的函数,这些函数实现一个特别的操作, 对于不支持的操作保留为NULL。

2. 在read( )和write( )中的buff 参数是用户空间指针。因此,它不能被内核代码直接引用，因为用户空间指针在内核空间时可能根本是无效的——没有那个地址的映射。因此，内核提供了专门的函数用于访问用户空间的指针：

|  |
| --- |
| 1 unsigned long copy\_from\_user(void \*to, const void \_\_user \*from, unsigned long count);  2 unsigned long copy\_to\_user(void \_\_user \*to, const void \*from, unsigned long count); |

**3. 驱动注销**

**3.1. 删除cdev**

在字符设备驱动模块卸载函数中通过cdev\_del()函数向系统删除一个cdev，完成字符设备的注销。

|  |  |
| --- | --- |
| 1  2  3  4 | /\*原型：\*/  void cdev\_del(struct cdev \*);  /\*例：\*/  cdev\_del(&btn\_cdev); |

**3.2. 释放设备号**

在调用cdev\_del()函数从系统注销字符设备之后，unregister\_chrdev\_region()应该被调用以释放原先申请的设备号。

|  |  |
| --- | --- |
| 1  2  3  4 | /\*原型：\*/  void unregister\_chrdev\_region(dev\_t from, unsigned count);  /\*例：\*/  unregister\_chrdev\_region(MKDEV(major, 0), 1); |

**四、字符设备驱动程序基础:**

**4.1 cdev结构体**

在Linux2.6 内核中，使用cdev结构体来描述一个字符设备，cdev结构体的定义如下：

|  |
| --- |
| 1 struct cdev {   2   3 struct kobject kobj;   4   5 struct module \*owner; /\*通常为THIS\_MODULE\*/   6   7 struct file\_operations \*ops; /\*在cdev\_init()这个函数里面与cdev结构联系起来\*/   8   9 struct list\_head list;  10  11 dev\_t dev; /\*设备号\*/  12  13 unsigned int count;  14  15 }； |

cdev 结构体的dev\_t 成员定义了设备号，为32位，其中12位是主设备号，20位是次设备号，我们只需使用二个简单的宏就可以从dev\_t 中获取主设备号和次设备号：

MAJOR(dev\_t dev)

MINOR(dev\_t dev)

相反地，可以通过主次设备号来生成dev\_t：

MKDEV(int major,int minor)

**4.2 Linux 2.6内核提供一组函数用于操作cdev 结构体**

|  |
| --- |
| 1：void cdev\_init(struct cdev\*,struct file\_operations \*);  2：struct cdev \*cdev\_alloc(void);  3：int cdev\_add(struct cdev \*,dev\_t,unsigned);  4：void cdev\_del(struct cdev \*); |

其中（1）用于初始化cdev结构体，并建立cdev与file\_operations 之间的连接。（2）用于动态分配一个cdev结构，（3）向内核注册一个cdev结构，（4）向内核注销一个cdev结构

**4.3 Linux 2.6内核分配和释放设备号**

在调用cdev\_add()函数向系统注册字符设备之前，首先应向系统申请设备号，有二种方法申请设备号，一种是静态申请设备号：

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

另一种是动态申请设备号：

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

其中，静态申请是已知起始设备号的情况，如先使用cat /proc/devices 命令查得哪个设备号未事先使用（不推荐使用静态申请）；动态申请是由系统自动分配，只需设置major = 0即可。

相反地，在调用cdev\_del()函数从系统中注销字符设备之后，应该向系统申请释放原先申请的设备号，使用：

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

**4.4 cdev结构的file\_operations结构体**

这个结构体是字符设备当中最重要的结构体之一，file\_operations 结构体中的成员函数指针是字符设备驱动程序设计的主体内容，这些函数实际上在应用程序进行Linux 的 open()、read()、write()、close()、seek()、ioctl()等系统调用时最终被调用。

|  |
| --- |
| 1 struct file\_operations {   2   3 /\*拥有该结构的模块计数，一般为THIS\_MODULE\*/   4 struct module \*owner;   5   6 /\*用于修改文件当前的读写位置\*/   7 loff\_t (\*llseek) (struct file \*, loff\_t, int);   8   9 /\*从设备中同步读取数据\*/  10 ssize\_t (\*read) (struct file \*, char \_\_user \*, size\_t, loff\_t \*);  11  12 /\*向设备中写数据\*/  13 ssize\_t (\*write) (struct file \*, const char \_\_user \*, size\_t, loff\_t \*);  14  15  16 ssize\_t (\*aio\_read) (struct kiocb \*, const struct iovec \*, unsigned long, loff\_t);  17 ssize\_t (\*aio\_write) (struct kiocb \*, const struct iovec \*, unsigned long, loff\_t);  18 int (\*readdir) (struct file \*, void \*, filldir\_t);  19  20 /\*轮询函数，判断目前是否可以进行非阻塞的读取或写入\*/  21 unsigned int (\*poll) (struct file \*, struct poll\_table\_struct \*);  22  23 /\*执行设备的I/O命令\*/  24 int (\*ioctl) (struct inode \*, struct file \*, unsigned int, unsigned long);  25  26  27 long (\*unlocked\_ioctl) (struct file \*, unsigned int, unsigned long);  28 long (\*compat\_ioctl) (struct file \*, unsigned int, unsigned long);  29  30 /\*用于请求将设备内存映射到进程地址空间\*/  31 int (\*mmap) (struct file \*, struct vm\_area\_struct \*);  32  33 /\*打开设备文件\*/  34 int (\*open) (struct inode \*, struct file \*);  35 int (\*flush) (struct file \*, fl\_owner\_t id);  36  37 /\*关闭设备文件\*/  38 int (\*release) (struct inode \*, struct file \*);  39  40  41 int (\*fsync) (struct file \*, struct dentry \*, int datasync);  42 int (\*aio\_fsync) (struct kiocb \*, int datasync);  43 int (\*fasync) (int, struct file \*, int);  44 int (\*lock) (struct file \*, int, struct file\_lock \*);  45 ssize\_t (\*sendpage) (struct file \*, struct page \*, int, size\_t, loff\_t \*, int);  46 unsigned long (\*get\_unmapped\_area)(struct file \*, unsigned long, unsigned long, unsigned long, unsigned long);  47 int (\*check\_flags)(int);  48 int (\*flock) (struct file \*, int, struct file\_lock \*);  49 ssize\_t (\*splice\_write)(struct pipe\_inode\_info \*, struct file \*, loff\_t \*, size\_t, unsigned int);  50 ssize\_t (\*splice\_read)(struct file \*, loff\_t \*, struct pipe\_inode\_info \*, size\_t, unsigned int);  51 int (\*setlease)(struct file \*, long, struct file\_lock \*\*);  52 }; |

**4.5 file结构**

file 结构代表一个打开的文件，它的特点是一个文件可以对应多个file结构。它由内核再open时创建，并传递给在该文件上操作的所有函数，直到最后close函数，在文件的所有实例都被关闭之后，内核才释放这个数据结构。

在内核源代码中，指向 struct file 的指针通常比称为filp，file结构有以下几个重要的成员：

|  |
| --- |
| 1 struct file{   2   3 mode\_t fmode; /\*文件模式，如FMODE\_READ，FMODE\_WRITE\*/   4   5 ......   6   7 loff\_t f\_pos; /\*loff\_t 是一个64位的数，需要时，须强制转换为32位\*/   8   9 unsigned int f\_flags; /\*文件标志，如：O\_NONBLOCK\*/  10  11 struct file\_operations \*f\_op;  12  13 void \*private\_data; /\*非常重要，用于存放转换后的设备描述结构指针\*/  14  15 .......  16  17 }; |

**4.6 inode 结构**

内核用inode 结构在内部表示文件，它是实实在在的表示物理硬件上的某一个文件，且一个文件仅有一个inode与之对应，同样它有二个比较重要的成员：

|  |
| --- |
| 1 struct inode{   2   3 dev\_t i\_rdev; /\*设备编号\*/   4   5 struct cdev \*i\_cdev; /\*cdev 是表示字符设备的内核的内部结构\*/   6   7 };   8   9 可以从inode中获取主次设备号，使用下面二个宏：  10  11 /\*驱动工程师一般不关心这二个宏\*/  12  13 unsigned int imajor(struct inode \*inode);  14  15 unsigned int iminor(struct inode \*inode); |

**4.7字符设备驱动模块加载与卸载函数**

在字符设备驱动模块加载函数中应该实现设备号的申请和cdev 结构的注册，而在卸载函数中应该实现设备号的释放与cdev结构的注销。

我们一般习惯将cdev内嵌到另外一个设备相关的结构体里面，该设备包含所涉及的cdev、私有数据及信号量等等信息。常见的设备结构体、模块加载函数、模块卸载函数形式如下：

|  |
| --- |
| 1/\*设备结构体\*/   2   3 struct xxx\_dev{   4   5 struct cdev cdev;   6   7 char \*data;   8   9 struct semaphore sem;  10  11 ......  12  13 };  14  15  16  17 /\*模块加载函数\*/  18  19 static int \_\_init xxx\_init(void)  20  21 {  22  23 .......  24  25 初始化cdev结构;  26  27 申请设备号；  28  29 注册设备号；  30  31  32  33 申请分配设备结构体的内存； /\*非必须\*/  34  35 }  36  37  38  39 /\*模块卸载函数\*/  40  41 static void \_\_exit xxx\_exit(void)  42  43 {  44  45 .......  46  47 释放原先申请的设备号；  48  49 释放原先申请的内存；  50  51 注销cdev设备；  52  53 }  54  55 |

4.8字符设备驱动的 file\_operations 结构体重成员函数

|  |
| --- |
| 1 /\*读设备\*/   2   3 ssize\_t xxx\_read(struct file \*filp, char \_\_user \*buf, size\_t count, loff\_t \*f\_pos)   4   5 {   6   7 ......   8   9 使用filp->private\_data获取设备结构体指针；  10  11 分析和获取有效的长度；  12  13 /\*内核空间到用户空间的数据传递\*/  14  15 copy\_to\_user(void \_\_user \*to, const void \*from, unsigned long count);  16  17 ......  18  19 }  20  21 /\*写设备\*/  22  23 ssize\_t xxx\_write(struct file \*filp, const char \_\_user \*buf, size\_t count, loff\_t \*f\_pos)  24  25 {  26  27 ......  28  29 使用filp->private\_data获取设备结构体指针；  30  31 分析和获取有效的长度；  32  33 /\*用户空间到内核空间的数据传递\*/  34  35 copy\_from\_user(void \*to, const void \_\_user \*from, unsigned long count);  36  37 ......  38  39 }  40  41 /\*ioctl函数\*/  42  43 static int xxx\_ioctl(struct inode \*inode,struct file \*filp,unsigned int cmd,unsigned long arg)  44  45 {  46  47 ......  48  49 switch(cmd){  50  51 case xxx\_CMD1:  52  53 ......  54  55 break;  56  57 case xxx\_CMD2:  58  59 .......  60  61 break;  62  63 default:  64  65 return -ENOTTY; /\*不能支持的命令\*/  66  67 }  68  69 return 0;  70  71 } |

**4.9、字符设备驱动文件操作结构体模板**

|  |
| --- |
| 1 struct file\_operations xxx\_fops = {   2   3 .owner = THIS\_MODULE,   4   5 .open = xxx\_open,   6   7 .read = xxx\_read,   8   9 .write = xxx\_write,  10  11 .close = xxx\_release,  12  13 .ioctl = xxx\_ioctl,  14  15 .lseek = xxx\_llseek,  16  17 };  18  19 上面的写法需要注意二点，一：结构体成员之间是以逗号分开的而不是分号，结构体字段结束时最后应加上分号。 |

**五、字符设备驱动小结：**

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。如图所示为cdev结构体、file\_operations和用户空间调用驱动的关系。

