# 1. 一个简单的字符设备驱动

Linux是Unix操作系统的一种变种，在Linux下编写驱动程序的原理和思想完全类似于其他的Unix系统，但它dos或window环境下的驱动程序有很大的区别。

在Linux操作系统下有两类主要的设备文件类型，一种是字符设备，另一种是块设备。字符设备和块设备的主要区别是：在对字符设备发出读/写请求时，实际的硬件I/O一般就紧接着发生了，块设备则不然，它利用一块系统内存作缓冲区，当用户进程对设备请求能满足用户的要求，就返回请求的数据，如果不能，就调用请求函数来进行实际的I/O操作。块设备是主要针对磁盘等慢速设备设计的，以免耗费过多的CPU时间来等待。

本节简单介绍如何写一个简单的字符设备驱动，实现一个与硬件设备无关的字符设备驱动，仅仅操作从内核中分配的一些内存。

## 1.1主设备号和次设备号

用户进程是通过设备文件来与实际的硬件打交道。每个设备文件都都有其文件属性(c/b)，表示是字符设备还是块设备。另外每个文件都有两个设备号，第一个是主设备号，标识驱动程序，第二个是从设备号，标识使用同一个设备驱动程序的不同的硬件设备，比如有两个软盘，就可以用从设备号来区分他们，也就是多个驱动程序共享主设备号的情况。而次设备号有内核使用，用于确定/dev下的设备文件对应的具体设备。设备文件的主设备号必须与设备驱动程序在登记时申请的主设备号一致，否则用户进程将无法访问到驱动程序。

对于字符设备的访问是通过文件系统中的设备名称进行的。他们通常位于/dev目录下。

[root@Loongson:/dev]#ls /dev -l

total 0

crw-rw---- 1 root root 251, 0 Jan 1 00:03 cdevdemo

crw-rw---- 1 root root 5, 1 Jan 1 00:00 console

crw-rw---- 1 root root 10, 61 Jan 1 00:00 cpu\_dma\_latency

crw-rw---- 1 root root 14, 3 Jan 1 00:00 dsp

crw-rw---- 1 root root 29, 0 Jan 1 00:00 fb0

crw-rw---- 1 root root 1, 7 Jan 1 00:00 full

crw-rw---- 1 root root 89, 0 Jan 1 00:00 i2c-0

crw-rw---- 1 root root 89, 1 Jan 1 00:00 i2c-1

crw-rw---- 1 root root 89, 2 Jan 1 00:00 i2c-2

drwxr-xr-x 2 root root 60 Jan 1 00:00 input

crw-rw---- 1 root root 1, 11 Jan 1 00:00 kmsg

crw-rw---- 1 root root 10, 63 Jan 1 00:00 ls1f-pwm

其中首字母b代表块设备，c代表字符设备。对于普通文件来说，ls -l会列出文件的长度，而对于设备文件来说，上面的251等代表的是对应设备的主设备号，而后面的0,1,2,61等则是对应设备的次设备号。举一个例子，虚拟控制台和串口终端有驱动程序管理，而不同的终端分别有不同的次设备号。

### 1)设备编号的表达

在内核中，dev\_t用来保存设备编号，包括主设备号和次设备号。在内核中，dev\_t是一个32位的数，其中12位用来表示主设备号，其余20位用来标识次设备号。

通过dev\_t获取主设备号和次设备号使用下面的宏：

MAJOR(dev\_t dev);

MINOR(dev\_t dev);

相反，通过主设备号和次设备号转换为dev\_t类型使用：

MKDEV(int major, int minor);

### 2)分配和释放设备编号

在构建一个字符设备之前，驱动程序首先要获得一个或者多个设备编号，这类似一个营业执照，有了营业执照才在内核中正常工作营业。完成此工作的函数是：

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

first是要分配的设备编号范围的起始值。count是连续设备的编号的个数。name是和该设备编号范围关联的设备名称，他将出现在/proc/devices和sysfs中。此函数成功返回0，失败返回负的错误码。/proc/devices存放着系统中所有的设备编号。如果加载成功，就会在/proc/devices中看到该设备。

[root@Loongson:/proc]#cat /proc/devices

Character devices:

1 mem

4 /dev/vc/0

4 tty

4 ttyS

5 /dev/tty

5 /dev/console

5 /dev/ptmx

7 vcs

10 misc

13 input

14 sound

29 fb

81 video4linux

89 i2c

90 mtd

128 ptm

136 pts

153 spi

180 usb

189 usb\_device

251 cdevdemo

252 hidraw

253 bsg

254 rtc

Block devices:

259 blkext

8 sd

31 mtdblock

65 sd

66 sd

67 sd

68 sd

69 sd

70 sd

71 sd

128 sd

129 sd

130 sd

131 sd

132 sd

133 sd

134 sd

135 sd

179 mmc

register\_chrdev\_region函数是在已知主设备号的情况下使用，在未知主设备号的情况下，使用下面的函数：

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

dev用于输出申请到的设备编号，firstminor要使用的第一个此设备编号。最好不要随机选择一个当前未使用的设备号，而应该用动态分配机制去获取主设备号。

在不使用时需要释放这些设备编号，已提供其他设备程序使用：

**void** unregister\_chrdev\_region(dev\_t dev, unsigned **int** count);

此函数多在模块的清除函数中调用。

分配到设备编号之后，只是拿到了营业执照，虽说现在已经准备的差不多了,但是只是从内核中申请到了设备号,应用程序还是不能对此设备作任何事情,需要一个简单的函数来把设备编号和此设备能实现的功能连接起来,这样模块才能提供具体的功能.这个操作很简单，稍后就会提到，在此之前先介绍几个重要的数据结构。

## 1.2 重要的数据结构

注册设备编号仅仅是完成一个字符设备驱动的第一步。下面介绍大部分驱动都会包含的三个重要的内核的数据结构。

Linux系统中，设备驱动程序是操作系统内核的重要组成部分，它与硬件设备之间建立了标准的抽象接口。通过这个接口，用户可以像处理普通文件一样，对硬件设备进行打开(open)、关闭(close)、读写(read/write)、控制（ioctl）等操作。

设备驱动程序接口是由结构file\_operations结构体向系统说明的，它定义在include/linux/fs.h中。

### 1)文件操作file\_operations

file\_operations是第一个重要的结构，定义在 <linux/fs.h>, 是一个函数指针的集合，设备所能提供的功能大部分都由此结构提供。这些操作也是设备相关的系统调用的具体实现。此结构的具体实现如下所示：

**struct** file\_operations {

//它是一个指向拥有这个结构的模块的指针. 这个成员用来在它的操作还在被使用时阻止模块被卸载. 几乎所有时间中, 它被简单初始化为 THIS\_MODULE

**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);

ssize\_t (\*read\_iter) (**struct** kiocb \*, **struct** iov\_iter \*);

ssize\_t (\*write\_iter) (**struct** kiocb \*, **struct** iov\_iter \*);

**int** (\*iterate) (**struct** file \*, **struct** dir\_context \*);

unsigned **int** (\*poll) (**struct** file \*, **struct** poll\_table\_struct \*);

**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 \*, loff\_t, loff\_t, **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 \*\*);

**long** (\*fallocate)(**struct** file \*file, **int** mode, loff\_t offset,

loff\_t len);

**int** (\*show\_fdinfo)(**struct** seq\_file \*m, **struct** file \*f);

};

需要说明的是这里面的函数在驱动中不用全部实现，不支持的操作留置为NULL。

几个主要的入口函数打开(open)、关闭(close)、读写(read/write)、控制（ioctl），编写驱动时需要填充。

⑴ open 入口点：

open函数负责打开设备、准备I/O。任何时候对设备文件进行打开操作，都会调用设备的open入口点。所以，open函数必须对将要进行的I/O操作做好必要的准备工作，如清除缓冲区等。如果设备是独占的。则open函数必须将设备标记成忙状态。

⑵ close入口点

close函数负责关闭设备的操作，当最后一次使用设备完成后，调用close函数，关闭设备文件。独占设备必须标记为可再次使用。

close()函数作用是关闭打开的文件。

⑶ read入口点

read函数负责从设备上读数据和命令，有缓冲区的I/O设备操作一般是从缓冲区里读数据。

⑷ write入口点

write函数负责往设备上写数据。对于有缓冲区的I/O设备操作，一般是把数据写入缓冲区里。对字符设备文件进行写操作将调用write函数。

⑸ ioctl入口点

ioctl函数执行读、写之外的操作，主要实现对设备的控制。

### 2)文件结构struct file

struct file, 定义于 <linux/fs.h>, 是设备驱动中第二个最重要的数据结构。文件结构代表一个打开的文件. (它不特定给设备驱动; 系统中每个打开的文件有一个关联的 struct file 在内核空间). 它由内核在 open 时创建, 并传递给在文件上操作的任何函数, 直到最后的关闭. 在文件的所有实例都关闭后, 内核释放这个数据结构。file结构的详细可参考fs.h，这里列出来几个重要的成员。

struct file\_operations \*f\_op：就是上面刚刚介绍的文件操作的集合结构。

mode\_t f\_mode：文件模式确定文件是可读的或者是可写的(或者都是), 通过位 FMODE\_READ 和 FMODE\_WRITE. 你可能想在你的 open 或者 ioctl 函数中检查这个成员的读写许可, 但是你不需要检查读写许可, 因为内核在调用你的方法之前检查. 当文件还没有为那种存取而打开时读或写的企图被拒绝, 驱动甚至不知道这个情况

loff\_t f\_pos：当前读写位置. loff\_t 在所有平台都是 64 位。驱动可以读这个值, 如果它需要知道文件中的当前位置, 但是正常地不应该改变它。

unsigned int f\_flags：这些是文件标志, 例如 O\_RDONLY, O\_NONBLOCK, 和 O\_SYNC. 驱动应当检查 O\_NONBLOCK 标志来看是否是请求非阻塞操作。

void \*private\_data：open 系统调用设置这个指针为 NULL, 在为驱动调用 open 方法之前. 你可自由使用这个成员或者忽略它; 你可以使用这个成员来指向分配的数据, 但是接着你必须记住在内核销毁文件结构之前, 在 release 方法中释放那个内存. private\_data 是一个有用的资源, 在系统调用间保留状态信息, 我们大部分例子模块都使用它

### 3)inode 结构

inode 结构由内核在内部用来表示文件。因此, 它和代表打开文件描述符的文件结构是不同的。 可能有代表单个文件的多个打开描述符的许多文件结构, 但是它们都指向一个单个 inode 结构。

inode 结构包含大量关于文件的信息比如文件的创建者、文件的创建日期、文件的大小等等。中文译名为"索引节点"。

inode包含文件的元信息，具体来说有以下内容：

\* 文件的字节数  
　　\* 文件拥有者的User ID  
　　\* 文件的Group ID  
　　\* 文件的读、写、执行权限  
　　\* 文件的时间戳，共有三个：ctime指inode上一次变动的时间，mtime指文件内容上一次变动的时间，atime指文件上一次打开的时间。  
　　\* 链接数，即有多少文件名指向这个inode  
　　\* 文件数据block的位置

可以用stat命令，查看某个文件的inode信息：

[root@Loongson:/]#cat hello.c

hello,I'm Loongson,This is file io test!

[root@Loongson:/]#stat hello.c

File: hello.c

Size: 41 Blocks: 1 IO Block: 4096 regular file

Device: 1f01h/7937d Inode: 779 Links: 1

Access: (0644/-rw-r--r--) Uid: ( 0/ root) Gid: ( 0/ root)

Access: 1970-01-01 00:02:45.000000000

Modify: 1970-01-01 00:02:45.000000000

Change: 1970-01-01 00:02:45.000000000

查看每个硬盘分区的inode总数和已经使用的数量，可以使用df命令。

[root@Loongson:/]#df

Filesystem 1K-blocks Used Available Use% Mounted on

/dev/root 51200 16604 34596 32% /

devtmpfs 64 0 64 0% /dev

tmpfs 64 0 64 0% /dev

tmpfs 11656 0 11656 0% /tmp

[root@Loongson:/]#df /dev/

Filesystem 1K-blocks Used Available Use% Mounted on

devtmpfs 64 0 64 0% /dev

由于每个文件都必须有一个inode，因此有可能发生inode已经用光，但是硬盘还未存满的情况。这时，就无法在硬盘上创建新文件。

每个inode都有一个号码，操作系统用inode号码来识别不同的文件。

这里值得重复一遍，Unix/Linux系统内部不使用文件名，而使用inode号码来识别文件。对于系统来说，文件名只是inode号码便于识别的别称或者绰号。表面上，用户通过文件名，打开文件。实际上，系统内部这个过程分成三步：首先，系统找到这个文件名对应的inode号码；其次，通过inode号码，获取inode信息；最后，根据inode信息，找到文件数据所在的block，读出数据。

使用ls -i命令，可以看到文件名对应的inode号码：

[root@Loongson:/]#ls -i hello.c

779 hello.c

**实际问题：**

在一台配置较低的Linux服务器（内存、硬盘比较小）的/data分区内创建文件时，系统提示磁盘空间不足，用df -h命令查看了一下磁盘使用情况，发现/data分区只使用了66%，还有12G的剩余空间，按理说不会出现这种问题。 后来用df -i查看了一下/data分区的索引节点(inode)，发现已经用满(IUsed=100%)，导致系统无法创建新目录和文件。

**查找原因：**

/data/cache目录中存在数量非常多的小字节缓存文件，占用的Block不多，但是占用了大量的inode。

解决方案：

1、删除/data/cache目录中的部分文件，释放出/data分区的一部分inode。  
　　2、用软连接将空闲分区/opt中的newcache目录连接到/data/cache，使用/opt分区的inode来缓解/data分区inode不足的问题：

ln -s /opt/newcache /data/cache

## 1.3字符设备的注册

内核在内部使用类型 struct cdev 的结构来代表字符设备. 在内核调用设备操作前, 编写分配并注册一个或几个这些结构。

有 2 种方法来分配和初始化一个这些结构. 如果想在运行时获得一个独立的 cdev 结构, 可使用这样的代码:

**struct** cdev \*my\_cdev = cdev\_alloc();

my\_cdev->ops = &my\_fops;

更多的情况是把cdv结构嵌入到你自己封装的设备结构中，这时需要使用下面的方法来分配和初始化：

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

后面的例子程序就是这么做的。一旦 cdev 结构建立, 最后的步骤是把它告诉内核：

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

这里, dev 是 cdev 结构, num 是这个设备响应的第一个设备号, count 是应当关联到设备的设备号的数目. 常常 count 是 1。

从系统去除一个字符设备, 调用:

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

内核在内部使用类型 struct cdev 的结构来代表字符设备. 在内核调用你的设备操作前, 你编写分配并注册一个或几个这些结构。

## 1.4 具体实例

上面大致介绍了实现一个字符设备所要做的工作，下面就来一个真实的例子来总结上面介绍的内容。设备驱动程序是I/O进程与设备控制器之间的通信程序。

本驱动程序的功能：

⑴ 接收由设备独立性软件发来的命令和参数，并将命令中的抽象要求转换为具体的要求。

⑵ 检查用户I/O请求的合法性，了解I/O设备的状态，传递有关参数，设置设备的工作方式。

⑶ 发出I/O命令。

⑷ 及时响应由控制器或通道发来的中断请求，并根据其中断类型调用相应的中断处理程序进行处理。

⑸ 对于设置有通道的计算机系统，驱动程序还应能够根据用户的I/O请求，自动地构建通道程序。

设备驱动程序的处理过程：

⑴ 将抽象要求转换为具体要求

⑵ 检查I/O设备请求的合法性

⑶ 读出和检查设备的状态

⑷ 传送必要的参数

⑸ 工作方式的设置

⑹ 启动I/O设备

源码中的关键地方已经作了注释。

/\*devdemo.c\*/

#include <linux/module.h>

#include <linux/types.h>

#include <linux/fs.h>

#include <linux/errno.h>

#include <linux/mm.h>

#include <linux/sched.h>

#include <linux/init.h>

#include <linux/cdev.h>

#include <asm/io.h>

#include <asm/uaccess.h>

#include <linux/timer.h>

#include <asm/atomic.h>

#include <linux/slab.h>

#include <linux/device.h>

#define CDEVDEMO\_MAJOR 255 /\*预设cdevdemo的主设备号\*/

**static** **int** cdevdemo\_major = CDEVDEMO\_MAJOR;

/\*设备结构体,此结构体可以封装设备相关的一些信息等

信号量等也可以封装在此结构中，后续的设备模块一般都

应该封装一个这样的结构体，但此结构体中必须包含某些

成员，对于字符设备来说，我们必须包含struct cdev cdev\*/

**struct** cdevdemo\_dev

{

**struct** cdev cdev;

};

**struct** cdevdemo\_dev \*cdevdemo\_devp; /\*设备结构体指针\*/

/\*文件打开函数，上层对此设备调用open时会执行\*/

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

{

printk(KERN\_NOTICE "======== cdevdemo\_open ");

**return** 0;

}

/\*文件释放，上层对此设备调用close时会执行\*/

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

{

printk(KERN\_NOTICE "======== cdevdemo\_release ");

**return** 0;

}

/\*文件的读操作，上层对此设备调用read时会执行\*/

**static** ssize\_t cdevdemo\_read(**struct** file \*filp, **char** \_\_user \*buf, **size\_t** count, loff\_t \*ppos)

{

printk(KERN\_NOTICE "======== cdevdemo\_read ");

}

/\* 文件操作结构体，文中已经讲过这个结构\*/

**static** **const** **struct** file\_operations cdevdemo\_fops =

{

.owner = THIS\_MODULE,

.open = cdevdemo\_open,

.release = cdevdemo\_release,

.read = cdevdemo\_read,

};

/\*初始化并注册cdev\*/

**static** **void** cdevdemo\_setup\_cdev(**struct** cdevdemo\_dev \*dev, **int** index)

{

printk(KERN\_NOTICE "======== cdevdemo\_setup\_cdev 1");

**int** err, devno = MKDEV(cdevdemo\_major, index);

printk(KERN\_NOTICE "======== cdevdemo\_setup\_cdev 2");

/\*初始化一个字符设备，设备所支持的操作在cdevdemo\_fops中\*/

cdev\_init(&dev->cdev, &cdevdemo\_fops);

printk(KERN\_NOTICE "======== cdevdemo\_setup\_cdev 3");

dev->cdev.owner = THIS\_MODULE;

dev->cdev.ops = &cdevdemo\_fops;

printk(KERN\_NOTICE "======== cdevdemo\_setup\_cdev 4");

err = cdev\_add(&dev->cdev, devno, 1);

printk(KERN\_NOTICE "======== cdevdemo\_setup\_cdev 5");

**if**(err)

{

printk(KERN\_NOTICE "Error %d add cdevdemo %d", err, index);

}

}

**int** cdevdemo\_init(**void**)

{

printk(KERN\_NOTICE "======== cdevdemo\_init ");

**int** ret;

dev\_t devno = MKDEV(cdevdemo\_major, 0);

**struct** **class** \*cdevdemo\_class;

/\*申请设备号，如果申请失败采用动态申请方式\*/

**if**(cdevdemo\_major)

{

printk(KERN\_NOTICE "======== cdevdemo\_init 1");

ret = register\_chrdev\_region(devno, 1, "cdevdemo");

}**else**

{

printk(KERN\_NOTICE "======== cdevdemo\_init 2");

ret = alloc\_chrdev\_region(&devno,0,1,"cdevdemo");

cdevdemo\_major = MAJOR(devno);

}

**if**(ret < 0)

{

printk(KERN\_NOTICE "======== cdevdemo\_init 3");

**return** ret;

}

/\*动态申请设备结构体内存\*/

cdevdemo\_devp = kmalloc(**sizeof**(**struct** cdevdemo\_dev), GFP\_KERNEL);

**if**(!cdevdemo\_devp) /\*申请失败\*/

{

ret = -ENOMEM;

printk(KERN\_NOTICE "Error add cdevdemo");

**goto** fail\_malloc;

}

memset(cdevdemo\_devp,0,**sizeof**(**struct** cdevdemo\_dev));

printk(KERN\_NOTICE "======== cdevdemo\_init 3");

cdevdemo\_setup\_cdev(cdevdemo\_devp, 0);

/\*下面两行是创建了一个总线类型，会在/sys/class下生成cdevdemo目录

这里的还有一个主要作用是执行device\_create后会在/dev/下自动生成

cdevdemo设备节点。而如果不调用此函数，如果想通过设备节点访问设备

需要手动mknod来创建设备节点后再访问。\*/

cdevdemo\_class = class\_create(THIS\_MODULE, "cdevdemo");

device\_create(cdevdemo\_class, NULL, MKDEV(cdevdemo\_major, 0), NULL, "cdevdemo");

printk(KERN\_NOTICE "======== cdevdemo\_init 4");

**return** 0;

fail\_malloc:

unregister\_chrdev\_region(devno,1);

}

**void** cdevdemo\_exit(**void**) /\*模块卸载\*/

{

printk(KERN\_NOTICE "End cdevdemo");

cdev\_del(&cdevdemo\_devp->cdev); /\*注销cdev\*/

kfree(cdevdemo\_devp); /\*释放设备结构体内存\*/

unregister\_chrdev\_region(MKDEV(cdevdemo\_major,0),1); //释放设备号

}

MODULE\_LICENSE("Dual BSD/GPL");

module\_param(cdevdemo\_major, **int**, S\_IRUGO);

module\_init(cdevdemo\_init);

module\_exit(cdevdemo\_exit);

具体步骤总结：

### 1) file\_operations结构体设计

/\* 文件操作结构体，文中已经讲过这个结构\*/

**static** **const** **struct** file\_operations cdevdemo\_fops =

{

.owner = THIS\_MODULE,

.open = cdevdemo\_open,

.release = cdevdemo\_release,

.read = cdevdemo\_read,

};

### 2) 模块初始化、模块卸载函数实现

(1) 注册设备号

分配设备编号，注册设备与注销设备的函数均在fs.h中声明，如下：

extern int register\_chrdev\_region(dev\_t,unsigned int,const char\*);//表示静态的申请和注册设备号

extern int alloc\_chrdev\_region(dev\_t,unsigned int,const char\*);//表示动态的申请和注册设备号

extern int register\_chrdev(unsigned int,const char\*,struct file\_operations\*);//表示int为0时动态注册，非零静态注册。

在linux2.6版本里面，register\_chrdev\_region是register\_chrdev的升级版。

使用register\_chrdev\_region函数时，首先要定义一个dev\_t变量来作为一个设备号，dev\_t dev\_num；如果想静态申请，那么

dev\_num=MKDEV(major\_no,0);//major\_no表示设备号的变量，然后便可以使用register\_chrdev\_region(dev\_num,2,"my\_dev");//第二个参数表示注册的设备数量，第三个表示驱动名

如果要动态的注册设备号，使用下面alloc\_chrdev\_region(&dev\_num，0,2，"memdev");次设备号从0开始，注册两个设备，设备名为memdev。

(2) 添加设备

前面只是注册了设备号，后面要向内核添加设备了；

struct cdev devno;

cdev\_init(&devno,&file\_operations) // 初始化设备

devno.owner=THIS\_MODULE;

devno.ops=&mem\_fops

对于已经知道了主设备号，就用

cdev\_add(&devno,dev\_num,MEMDEV\_NR\_DEVS);//添加设备

如果是动态申请的设备号，就用

cdev\_add(&devno,MKDEV(mem\_major,0),MEMDEV\_NR\_DEVS);

由此可见，使用register\_chrdev\_region()比register\_chrdev()多了一步，就是向内核注册添加cdev设备的步骤。

(3) 添加设备结点

**自动添加、删除设备节点的方法**

以下创建了一个总线类型，在/sys/class下生成cdevdemo目录 后，在执行device\_create后会在/dev/下自动生成 cdevdemo设备节点。而如果不调用此函数，如果想通过设备节点访问设备 ， 需要手动mknod来创建设备节点后再访问。

cdevdemo\_class = class\_create(THIS\_MODULE, "cdevdemo");

device\_create(cdevdemo\_class, NULL, MKDEV(cdevdemo\_major, 0), NULL, "cdevdemo");

本实例使用的是这种方法。

**另外一种手动添加、删除设备节点的方法**

　　创建设备文件：

　　　　#mknod /dev/ devdemo c major minor

c是指字符设备，major是主设备号，就是在/proc/devices里看到的。

　　用shell命令：

　　　　$ cat /proc/devices

就可以获得主设备号。 minor是从设备号，设置成0就可以了。

mknod命令创建的设备节点，可以使用rm -f 设备文件名称（注意这个地方不是跟全路径，就是一个名字）可以删掉，重启电脑或者删除对应的文件都是无效的。

### 3) 读写函数的实现

/\*文件打开函数，上层对此设备调用open时会执行\*/

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

{

printk(KERN\_NOTICE "======== cdevdemo\_open ");

**return** 0;

}

/\*文件释放，上层对此设备调用close时会执行\*/

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

{

printk(KERN\_NOTICE "======== cdevdemo\_release ");

**return** 0;

}

/\*文件的读操作，上层对此设备调用read时会执行\*/

**static** ssize\_t cdevdemo\_read(**struct** file \*filp, **char** \_\_user \*buf, **size\_t** count, loff\_t \*ppos)

{

printk(KERN\_NOTICE "======== cdevdemo\_read ");

}

### 4) 驱动程序编译

编写Makefile，后在虚拟机中编译。

obj-m := devdemo1.o

#定义目录变量

KDIR := /Workstation/tools/kernel/linux-3.0.82-openloongson

PWD := $(shell pwd)

all:

# make文件

make -C $(KDIR) M=$(PWD) modules ARCH=mips CROSS\_COMPILE=mipsel-linux-

clean:

rm -rf \*.o \*.mod.c \*.ko

### 5) 驱动程序编译和加载

在用insmod命令将编译好的模块调入内存时，init\_module 函数被调用。在这里，编写makefile文件编译该设备驱动程序，编译结束后产生devdemo.ko文件。　

驱动程序已经编译好了，现在把它安装到系统中去：

[root@Loongson:/]#insmod devdemo.ko

======== cdevdemo\_init

======== cdevdemo\_init 1

======== cdevdemo\_init 3

======== cdevdemo\_setup\_cdev 1

======== cdevdemo\_setup\_cdev 2

======== cdevdemo\_setup\_cdev 3

======== cdevdemo\_setup\_cdev 4

======== cdevdemo\_setup\_cdev 5

======== cdevdemo\_init 4

如果安装成功，在/proc/devices文件中就可以看到设备devdemo，并可以看到它的主设备号。

[root@Loongson:/]#cat /proc/devices | grep demo

255 cdevdemo

要卸载的话，运行命令：

[root@Loongson:/]#rmmod devdemo

End cdevdemo

### 6) 驱动程序测试

现在可以通过设备文件来访问我们的驱动程序。可以写一个小小的测试程序：

#include <stdio.h>

#include <errno.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <string.h>

#include <unistd.h>

int main(void)

{

int fd;

int i;

char buf[10] ;

fd = open("/dev/cdevdemo", O\_RDWR);

if (fd < 0)

{

perror("open error");

return -1;

}

read(fd，buf，10);

　　for (i = 0; i < 10;i++)

　　　　printf("%d\n"，buf[i]);

close(fd);

return 0;

}

编译运行，运行结果：

[root@Loongson:/]#insmod devdemo.ko

======== cdevdemo\_init

======== cdevdemo\_init 1

======== cdevdemo\_init 3

======== cdevdemo\_setup\_cdev 1

======== cdevdemo\_setup\_cdev 2

======== cdevdemo\_setup\_cdev 3

======== cdevdemo\_setup\_cdev 4

======== cdevdemo\_setup\_cdev 5

======== cdevdemo\_init 4[root@Loongson:/]#test\_devdemo

-/bin/sh: test\_devdemo: not found

[root@Loongson:/]#chmod u+x test\_devdemo

[root@Loongson:/]#./test\_devdemo

======== cdevdemo\_open

======== cdevdemo\_read the string is :

======== cdevdemo\_release hello

以上只是一个简单的演示。真正实用的驱动程序要复杂的多，要处理如中断，DMA，I/O port等问题。

## 1.5 一些有用的资料

[Linux 驱动 之 模块化编程](http://blog.chinaunix.net/uid-26833883-id-4366882.html)

<http://blog.chinaunix.net/uid-26833883-id-4366882.html>

[Linux 驱动 之 模块化编程](http://blog.chinaunix.net/uid-26833883-id-4366882.html)

<http://blog.chinaunix.net/uid-26833883-id-4366909.html>

[Linux 设备驱动之字符设备(一)](http://blog.chinaunix.net/uid-26833883-id-4369060.html)

<http://blog.chinaunix.net/uid-26833883-id-4369060.html>

[Linux 设备驱动之字符设备(二)](http://blog.chinaunix.net/uid-26833883-id-4369060.html)

<http://blog.chinaunix.net/uid-26833883-id-4369117.html>

[Linux 设备驱动之字符设备(三)](http://blog.chinaunix.net/uid-26833883-id-4369060.html)

<http://blog.chinaunix.net/uid-26833883-id-4371047.html>

## 1.6 例子修改成模块注销自动删除设备节点

实例文件：devdemo.c

添加设备节点时，添加的类设置其属性为 666。即将以下两句

cdevdemo\_class = class\_create(THIS\_MODULE, "cdevdemo");

device\_create(cdevdemo\_class, NULL, MKDEV(cdevdemo\_major, 0), NULL, "cdevdemo");

修改成：

cdevdemo\_class = class\_create(THIS\_MODULE, "cdevdemo");

if (IS\_ERR(cdevdemo\_class))

{

ret = PTR\_ERR(cdevdemo\_class);

printk(KERN\_ERR "class create error %d\n", ret);

goto fail\_malloc;

}

cdevdemo\_class->devnode = chardev\_devnode;

dev = device\_create(cdevdemo\_class, NULL, MKDEV(cdevdemo\_major, 0), NULL, "cdevdemo");

添加两个定义：

struct class \*cdevdemo\_class;

static char \*chardev\_devnode(struct device \*dev, umode\_t \*mode)

{

if (mode)

\*mode = 0666;

return NULL;

}

模块注销卸载中添加两句：

void cdevdemo\_exit(void) /\*模块卸载\*/

{

device\_destroy(cdevdemo\_class, MKDEV(cdevdemo\_major,0));

class\_destroy(cdevdemo\_class);

……

}