字符设备驱动结构

#include <linux/init.h>

#include <linux/kernel.h>

#include <linux/module.h>

#include <linux/types.h>

#include <linux/cdev.h>

#include <linux/slab.h>

#include <linux/fs.h>

#include <linux/device.h>

#define DEVICE\_NAME "cdevdemo" //设备节点（设备文件）名字的定义

#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

{

/\* linux/cdev.h \*/

struct cdev cdev;

};

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

//ioctl函数

static long cdevdemo\_ioctl(struct file \*file, unsigned int cmd, unsigned long arg)

{

switch(cmd)

{

case 1:break;

}

return 0;

}

/\*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,

.unlocked\_ioctl = cdevdemo\_ioctl

};

/\*初始化并注册cdev\*/

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

{

int err;

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

/\*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);

dev->cdev.owner = THIS\_MODULE;

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

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

if(err)

{

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

}

}

static int \_\_init cdevdemo\_init(void)

{

int ret;

/\* <linux/types.h>

宏定义 dev\_t 1+20

<linux/cdev.h> -> kdev\_t.h

#define MKDEV(ma,mi) (((ma) << MINORBITS) | (mi)) \*/

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

/\* <linux/fs.h>

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

静态申请设备号

在 /linux/Documentation/devices.txt 确认空设备号

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

from：希望申请使用的设备号

count：希望申请使用设备号数目

name：设备名(在/proc/devices) \*/

if(cdevdemo\_major)

{

ret = register\_chrdev\_region(devno, 1, DEVICE\_NAME);

}

else

{

/\* int alloc\_chrdev\_region(dev\_t \*dev, unsigned int firstminor,

unsigned int count, const char \*name); \*/

ret = alloc\_chrdev\_region(&devno, 0, 1, DEVICE\_NAME);

/\* <linux/cdev.h> -> kdev\_t.h

#define MAJOR(dev) ((unsigned int) ((dev) >> MINORBITS))\*/

cdevdemo\_major = MAJOR(devno); /\*获得申请的主设备号\*/

}

if(ret < 0)

{

printk(KERN\_ALERT " -- chrdev\_region failed\n\n");

return ret;

}

/\* 为设备描述结构分配内存

<linux/slab.h>\*/

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));

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, DEVICE\_NAME);

//device\_create(cdevdemo\_class, NULL, MKDEV(cdevdemo\_major, 0), NULL, DEVICE\_NAME);

return 0;

fail\_malloc:

unregister\_chrdev\_region(devno, 1);

return ret;

}

static void \_\_exit cdevdemo\_exit(void)

{

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

/\*<linux/slab.h>\*/

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\_param(cdevdemo\_major, int, S\_IRUGO);//?????

module\_init(cdevdemo\_init);

module\_exit(cdevdemo\_exit);

MODULE\_LICENSE("GPL");

**简单混杂设备驱动结构**

#include <linux/init.h>

#include <linux/kernel.h>

#include <linux/module.h>

#include <linux/miscdevice.h>

#include <linux/fs.h>

#define DEVICE\_NAME "my\_buzzer" //设备节点（设备文件）名字的定义

//ioctl函数

static long pwm\_ioctl(struct file \*file, unsigned int cmd, unsigned long arg){

switch(cmd)

{

case 1:break;

}

return 0;

}

//设备的文件操作结构体

static struct file\_operations pwm\_fops = {

.owner = THIS\_MODULE,

.unlocked\_ioctl = pwm\_ioctl,

}

//注册混杂设备的3个必须参数

static struct miscdevice misc = {

.minor = MISC\_DYNAMIC\_MINOR,

.name = DEVICE\_NAME,

.fops = &pwm\_fops,

}

static int \_\_init pwm\_init(void){

misc\_register(&misc); //注册 misc 设备

//misc\_register 函数会自动创建设备节点(设备文件),无需mknod手动创建设备文件。

}

static void \_\_exit pwm\_exit(void){

misc\_deregister(&misc); ////注销 misc 设备

}

module\_init(pwm\_init);

module\_exit(pwm\_exit);

MODULE\_LICENSE("GPL");