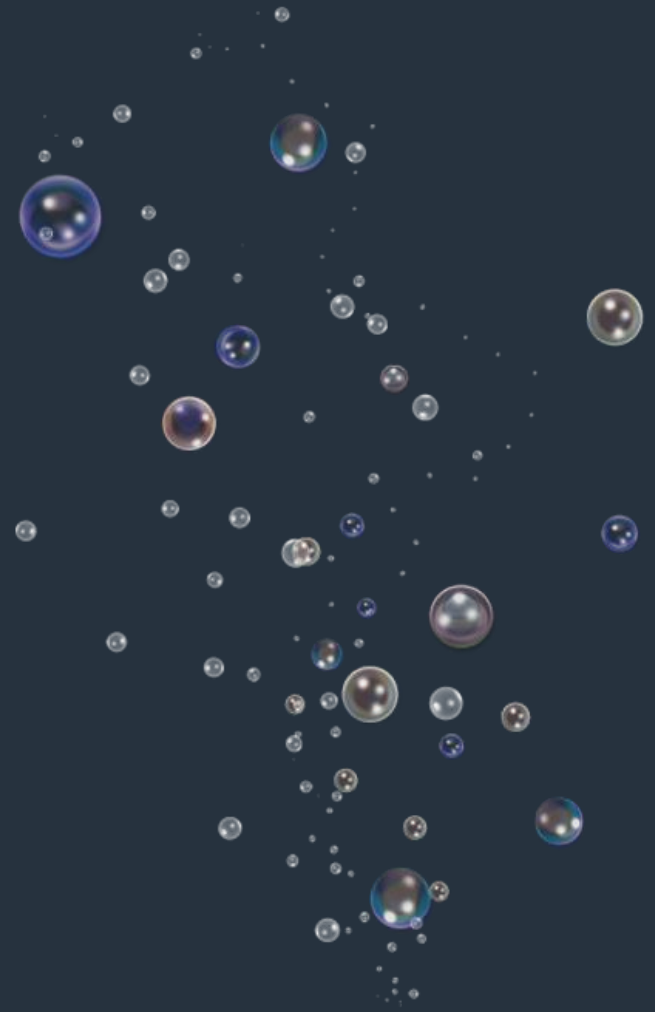


内核模块简介

大连理工大学 赖晓晨

主流操作系统类型

- 微内核体系结构

  内核只负责进程管理、内存管理、中断管理，文件系统、网络协议等其他部分运行于用户空间。可扩展性好，不同层次之间消息传递开销比较大。

- 单一体系结构内核

  内核是一个大程序，包括了操作系统的所有部分。所有模块都集成在一起，系统的速度、性能比较好，可扩展性和维护性相对差

- 采用模块机制

  内核可以做的很小，在内核中设计一些模块的接口，可以动态载入或移出模块，内核管理所有模块的运行，而系统功能的可扩展性留给模块去完成。

什么是模块

- 模块全称：动态可加载内核模块（loadable kernel module， LKM）
- 模块（module）是在内核空间运行的程序，实际是一种**目标对象文件**，没有链接，不能独立运行，但是其代码可以在运行时链接到系统中作为内核的一部分运行或从内核中取下，从而可以动态扩充内核的功能。
- 模块一般由一组函数或数据结构组成，模块运行于**核心态**，不会被交换出内存。
- Linux的设备驱动程序大都采用模块方式实现

使用模块机制的优点

- 使得内核结构更加紧凑和灵活。
- 系统如果需要新功能，只要编译相应的模块然后插入即可。
- 模块一旦链接到内核，就与内核中原有的代码完全等价。

模块实用程序介绍

- insmod：向正在运行的内核加载模块。
- lsmod：显示当前加载的内核模块信息。
- rmmod： 从当前运行的内核中卸载内核模块.
- depmod：处理可加载内核模块的依赖关系。
- modprobe： 利用depmod创建的依赖文件来自动加载相关的模块。
- modinfo： 获取模块信息。

Linux内核模块程序结构

- 模块加载函数 **init_module()**：当通过insmod或者modprobe命令加载模块时，会被内核自动执行，主要用于完成本模块的初始化工作。
- 模块卸载函数 **cleanup_module()**：当通过rmmod命令卸载某模块时，会被内核自动执行，完成与模块加载相反的工作。
- 模块许可证声明 **MODULE_LICENSE()**：描述内核模块的许可权限，如不加则提示内核被污染的警告，常用许可证有“GPL”、“GPL v2”等。
- 模块参数（可选）
- 模块导出符号（可选）
- 模块作者声明等（可选）

最简模块实例

```
#include <linux/init.h>
#include <linux/module.h>

int init_module(void)
{
    printk(KERN_ALERT "hello world!\n");
    return 0;
}

void cleanup_module(void)
{
    printk(KERN_ALERT "goodbye world!\n");
}

MODULE_LICENSE("GPL v2");
```

相关头文件

module1/hello.c

最简模块实例

```
#include <linux/init.h>
#include <linux/module.h>

int init_module(void)
{
    printk(KERN_ALERT "hello world!\n");
    return 0;
}

void cleanup_module(void)
{
    printk(KERN_ALERT "goodbye world!\n");
}

MODULE_LICENSE("GPL v2");
```

模块加载函数

module1/hello.c

最简模块实例

```
#include <linux/init.h>
#include <linux/module.h>

int init_module(void)
{
    printk(KERN_ALERT "hello world!\n");
    return 0;
}

void cleanup_module(void)
{
    printk(KERN_ALERT "goodbye world!\n");
}

MODULE_LICENSE("GPL v2");
```

内核空间输出函数

module1/hello.c

最简模块实例

```
#include <linux/init.h>
#include <linux/module.h>

int init_module(void)
{
    printk(KERN_ALERT "hello world!\n");
    return 0;
}

void cleanup_module(void)
{
    printk(KERN_ALERT "goodbye world!\n");
}

MODULE_LICENSE("GPL v2");
```

模块卸载函数

module1/hello.c

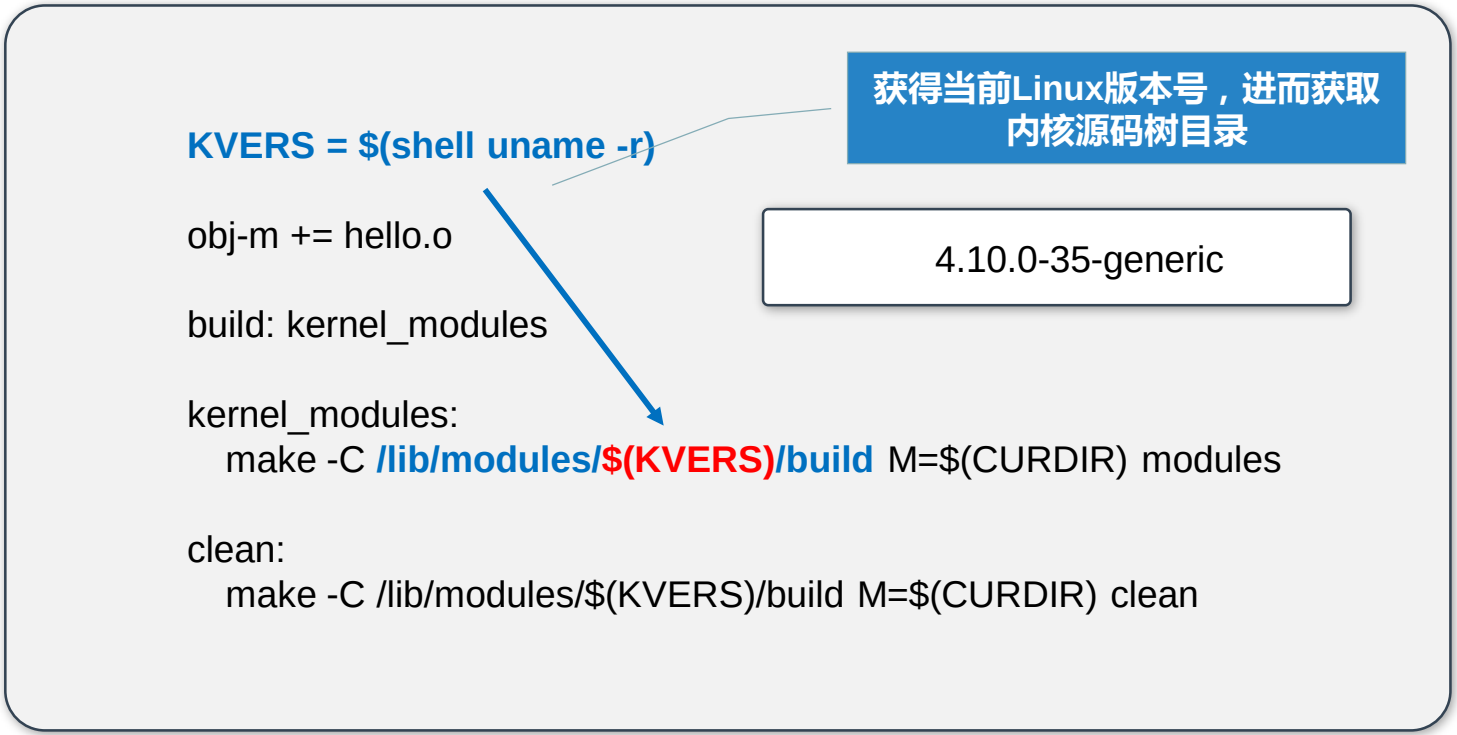
最简模块实例

```
#include <linux/init.h>
#include <linux/module.h>

int init_module(void)
{
    printk(KERN_ALERT "hello world!\n");
    return 0;
}

void cleanup_module(void)
{
    printk(KERN_ALERT "goodbye world!\n");
}

MODULE_LICENSE("GPL v2");
```

许可证声明

module1/hello.c

Makefile



module1/Makefile

Makefile

```
KVERS = $(shell uname -r)

obj-m += hello.o

build: kernel_modules

kernel_modules:
	make -C /lib/modules/$(KVERS)/build M=$(CURDIR) modules

clean:
	make -C /lib/modules/$(KVERS)/build M=$(CURDIR) clean
```

以模块形式编译该程序

module1/Makefile

Makefile

```
KVERS = $(shell uname -r)

obj-m += hello.o

build: kernel_modules

kernel_modules:
    make -C /lib/modules/$(KVERS)/build M=$(CURDIR) modules

clean:
    make -C /lib/modules/$(KVERS)/build M=$(CURDIR) clean
```

module1/Makefile

Makefile

```
KVERS = $(shell uname -r)

obj-m += hello.o

build: kernel_modules

kernel_modules:
    make -C /lib/modules/$(KVERS)/build M=$(CURDIR) modules

clean:
    make -C /lib/modules/$(KVERS)/build M=$(CURDIR) clean
```

清空目标文件规则

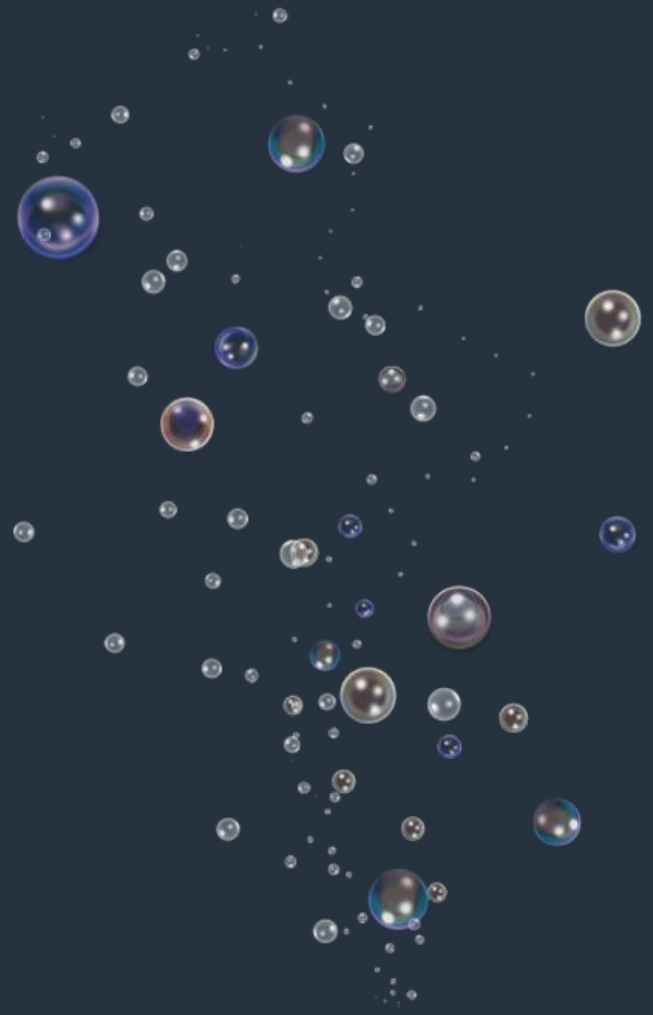
module1/Makefile

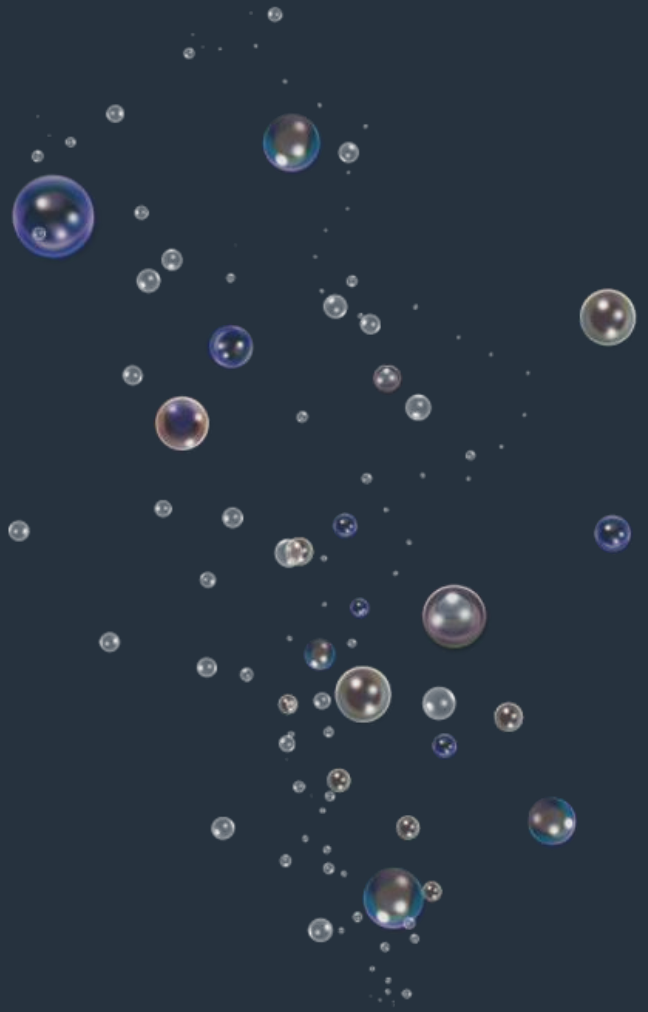
操作步骤

```
$ sudo –i
# make
# insmod hello.ko
# lsmod
# rmmod hello
# lsmod
# less /var/log/kern.log | tail -5
```

嵌入式软件设计

大连理工大学 赖晓晨





内核模块设计

大连理工大学 赖晓晨

多文件模块实例

```
#include <linux/init.h>
#include <linux/module.h>
int init_module(void)
{
    printk(KERN_ALERT "Hi,hello world\n");
    return 0;
}
MODULE_LICENSE("GP
```

module2/t1.c

清空目标文件规则

```
#include <linux/init.h>
#include <linux/module.h>
void cleanup_module(void)
{
    printk(KERN_ALERT "Hi goodbye world!\n");
}
MODULE_LICENSE("GPL v2");
```

module2/t2.c

多文件模块的Makefile

```
KVERS = $(shell uname -r)

obj-m += test.o
test-objs := t1.o t2.o

build: kernel_modules

kernel_modules:
    make -C /lib/modules/$(KVERS)/build M=$(CURDIR) modules

clean:
    make -C /lib/modules/$(KVERS)/build M=$(CURDIR) clean
```

列出所有目标文件

module2/Makefile

操作步骤

```
$ sudo –i
# make
# ls
# insmod test.ko
# rmmod test
# less /var/log/kern.log | tail -5
```

一个更具普遍性的模块实例

```
#include <linux/init.h>
#include <linux/module.h>

static int number = 100;
module_param(number, int, S_IRUGO);
static int __init normal_init(void)
{
    printk(KERN_INFO "the number is: %d\n", number);
    return 0;
}
module_init(normal_init);
static void __exit normal_exit(void)
{
    printk(KERN_INFO "module normal finished!\n");
}
module_exit(normal_exit);
void just_a_try(void)
{
   printk(KERN_INFO "just a try!\n");
}
EXPORT_SYMBOL_GPL(just_a_try);
MODULE_LICENSE("GPL v2");
```

定义模块参数

module3/normal.c

一个更具普遍性的模块实例

```
#include <linux/init.h>
#include <linux/module.h>

static int number = 100;
module_param(number, int, S_IRUGO);
static int __init normal_init(void)
{
    printk(KERN_INFO "the number is: %d\n", number);
    return 0;
}
module_init(normal_init);
static void __exit normal_exit(void)
{
    printk(KERN_INFO "module normal finished!\n");
}
module_exit(normal_exit);
void just_a_try(void)
{
    printk(KERN_INFO "just a try!\n");
}
EXPORT_SYMBOL_GPL(just_a_try);
MODULE_LICENSE("GPL v2");
```

模块加载函数

性能优化

module3/normal.c

一个更具普遍性的模块实例

```
#include <linux/init.h>
#include <linux/module.h>

static int number = 100;
module_param(number, int, S_IRUGO);
static int __init normal_init(void)
{
    printk(KERN_INFO "the number is: %d\n", number);
    return 0;
}
module_init(normal_init);
static void __exit normal_exit(void)
{
    printk(KERN_INFO "module normal finished!\n");
}
module_exit(normal_exit);
void just_a_try(void)
{
    printk(KERN_INFO "just a try!\n");
}
EXPORT_SYMBOL_GPL(just_a_try);
MODULE_LICENSE("GPL v2");
```

模块卸载函数

性能优化

module3/normal.c

一个更具普遍性的模块实例

```
#include <linux/init.h>
#include <linux/module.h>

static int number = 100;
module_param(number, int, S_IRUGO);
static int __init normal_init(void)
{
    printk(KERN_INFO "the number is: %d\n", number);
    return 0;
}
module_init(normal_init);
static void __exit normal_exit(void)
{
    printk(KERN_INFO "module normal finished!\n");
}
module_exit(normal_exit);
void just_a_try(void)
{
    printk(KERN_INFO "just a try!\n");
}
EXPORT_SYMBOL_GPL(just_a_try);
MODULE_LICENSE("GPL v2");
```

导出该函数到内核符号表

module3/normal.c

一个更具普遍性的模块实例

```
#include <linux/init.h>
#include <linux/module.h>

static int number = 100;
module_param(number, int, S_IRUGO);
static int __init normal_init(void)
{
    printk(KERN_INFO "the number is: %d\n", number);
    return 0;
}
module_init(normal_init);
static void __exit normal_exit(void)
{
    printk(KERN_INFO "module normal finished!\n");
}
module_exit(normal_exit);
void just_a_try(void)
{
    printk(KERN_INFO "just a try!\n");
}
EXPORT_SYMBOL_GPL(just_a_try);
MODULE_LICENSE("GPL v2");
```

许可证说明

module3/normal.c

引用其他模块导出的符号实例

```
#include <linux/init.h>
#include <linux/module.h>

extern void just_a_try(void)

static int __init test_init(void)
{
    printk(KERN_ALERT “Can you see the symbol exported before?\n");
    just_a_try();
    return 0;
}
module_init(test_init);

static void __exit test_exit(void)
{
    printk(KERN_ALERT "module test finished\n");
}
module_exit(test_exit);

MODULE_LICENSE("GPL v2");
```

声明引用

函数调用

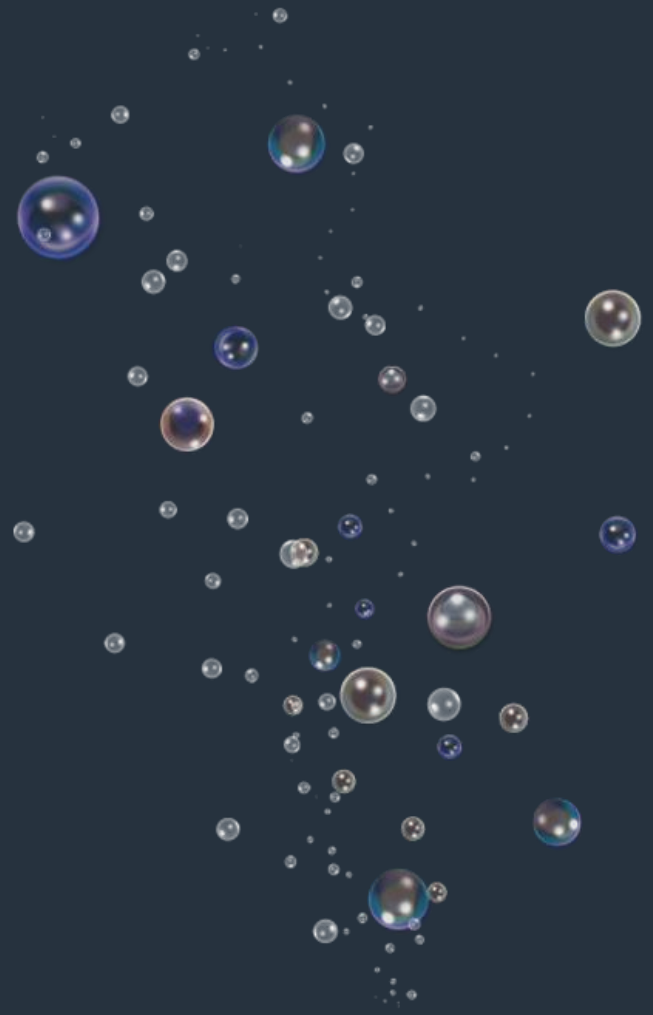
module4/test.c

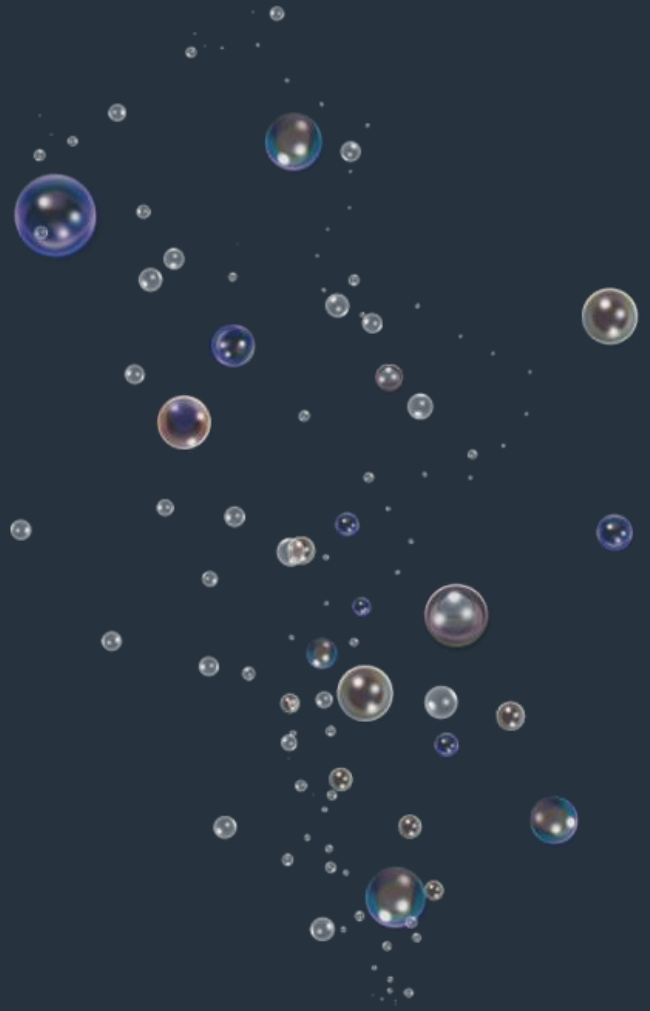
操作步骤

```
$ sudo –i
# make
# insmod normal.ko number=123
# insmod test.ko
# lsmod
# less /var/log/kern.log
# rmmod test
# rmmod normal
```

嵌入式软件设计

大连理工大学 赖晓晨





Linux设备驱动程序简介

大连理工大学 赖晓晨

Linux设备驱动程序的作用

- 操作系统一般提供设备驱动程序来专门完成对特定硬件的控制，设备驱动程序实际是处理或操作硬件控制器的软件，是内核中具有高特权级的、驻留内存的、可共享的底层硬件处理例程。
- 驱动程序使硬件细节对应用程序员“透明”。

Linux设备驱动程序实现机制

- Linux系统中每一类设备都有一个驱动程序，设备驱动程序存在于**内核中**，不同的应用可以共享这些代码。
- Linux系统中，每一个设备体现为/dev目录下的一个**文件**。
- 一个驱动程序就是一个函数和数据结构的集合，它封装了控制的细节，并通过一组特殊接口定义了一个经典**操作集**，内核通过访问该设备的文件节点来调用相关处理函数。

Linux设备的分类

- Linux系统的设备分为字符设备，块设备和网络设备三种。
- **字符设备**是指存取时没有缓存的设备，如系统的串口设备/dev/cua0、/dev/cua1。
- **块设备**的读写则都有缓存来支持，只能以块为单位进行读写，并且块设备能够随机存取，即不管块处于设备的什么地方都可以对它进行读写，字符设备则没有这个要求。块设备主要包括硬盘、软盘、CD-ROM等。
- **网络设备**在Linux里做专门的处理、主要是基于socket机制。

设备号

- 传统方式的设备管理中，除了设备类型（字符设备和块设备）以外，内核还需要一对参数，称为主、次设备号。
- 主设备号决定使用何种设备驱动程序。 每种不同的设备都被分配了不同的主设备号； 所有具有相同主设备号的设备文件都是被同一个驱动程序控制（2.4内核下是8位，2.6及之后内核为12位）。
- 次（从）设备号用来区别驱动程序控制的多个设备（2.4内核下为8位，2.6及之后内核为20位）。

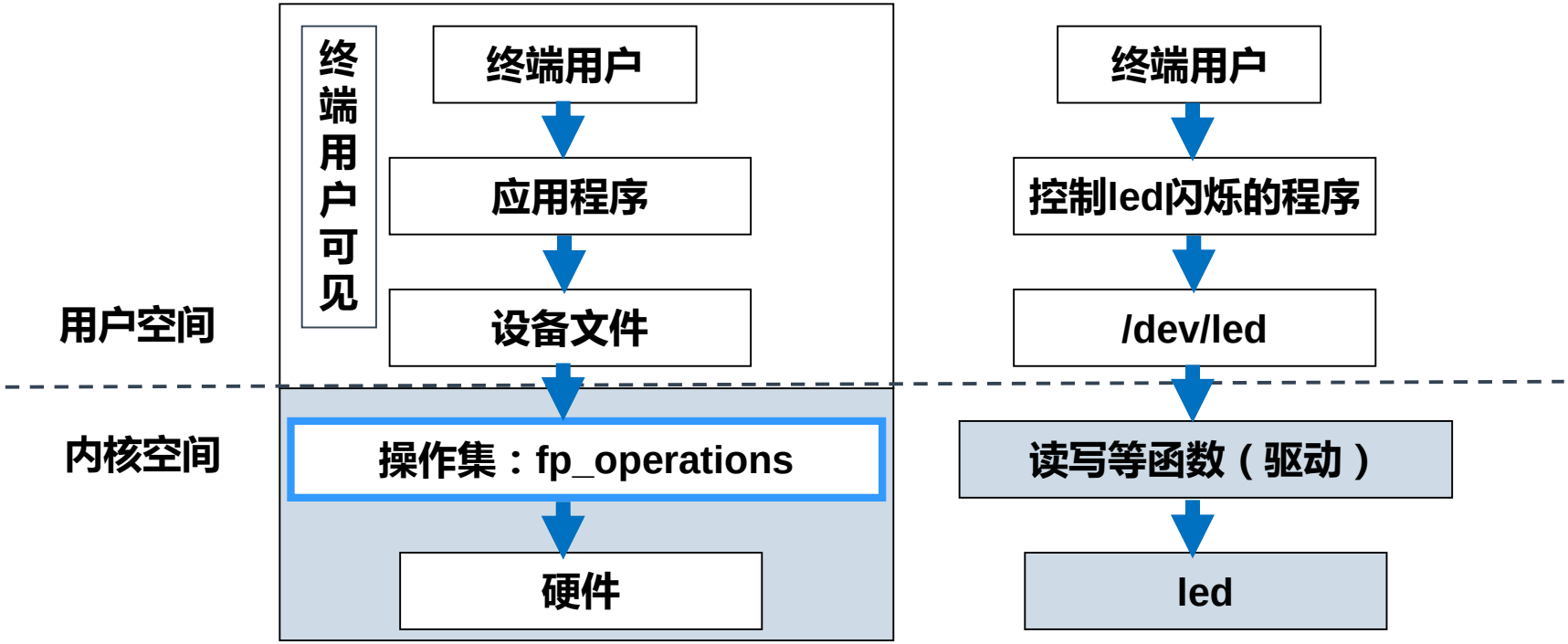
常用主设备号

- Linux有关方面已经就常用设备号达成了一致。
- 软驱：2
- 硬盘：3
- 并口：6
- 声卡：14
- joystick：15

设备文件

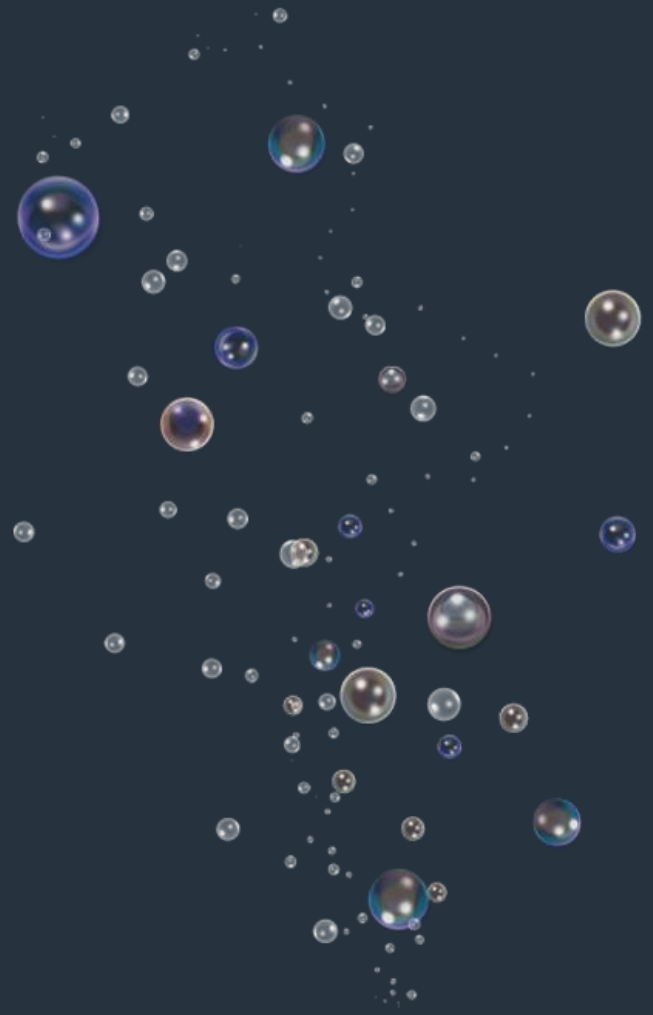
- Linux内核中用主、次设备号标识一个设备，但是，从用户角度而言，这一方法不大实用，因为用户不可能记住每一个设备号。
- 用户希望用统一的方式来访问各个设备，因此Linux中的设备管理应用了**设备文件**这个概念。
- 系统试图使它对各类设备的输出、输入看起来就象是对普通文件一样。因此，把设备映射为一种特殊的文件。

（字符设备）驱动程序在系统中的位置



嵌入式软件设计

大连理工大学 赖晓晨



驱动程序的数据结构（2.6+）

大连理工大学 赖晓晨

file结构体

file结构体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，在文件的所有实例都关闭之后，内核释放这个数据结构。

```
struct file
{
    …
    struct path  f_path;
    struct inode  *f_inode;
    const struct file_operations  *f_op;     //文件关联的操作
    …
    fmode_t  f_mode;                         //文件的访问模式
    …
    void*  private_data;                     //文件私有数据
    …
}
```

cdev结构体

在Linux内核中，使用cdev结构体来描述一个字符设备。

```
struct cdev
{
    struct kobject kobj;                    //内嵌的kobject对象
    struct module *owner;                   //所属模块
    struct file_operations *ops;            //文件操作结构体
    dev_t  dev;                             //设备号
    unsigned int count ;
}
```

MAJOR(dev_t dev)

MINOR(dev_t dev)

MKDEV(int major, int minor)

cdev结构体操作函数

- cdev_alloc()函数用于动态申请一块cdev类型内存。
- cdev_init()函数用于初始化cdev成员。
- cdev_add()函数用于向系统添加一个cdev，完成字符设备的注册
- cdev_del()函数用于从系统删除一个cdev，完成字符设备的注销

```
struct cdev
{
    struct kobject kobj;                //内嵌的kobject对象
    struct module *owner;               //所属模块
    struct file_operations *ops;        //文件操作结构体
    dev_t  dev;                         //设备号
    unsigned int count ;
}
```

分配和释放设备号

- 在使用cdev_add()向系统注册字符设备之前应先申请设备号，采用如下函数：

已知设备号：

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

由系统自动分配设备号

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

- 在使用cdev_del()函数注销字符设备之后，应释放设备号，采用如下函数：

```
void unregister_chrdev_region(dev_t from, unsigned count);
```

file_operations结构体

```
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 *);
    int (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
    …
};
```

修改一个文件的当前读写位置，并将新位置返回

file_operations结构体

```
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 *);
    int (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
    …
};
```

从设备中读数据，成功时返回读到的字节数

file_operations结构体

```
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 *);
    int (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
    …
};
```

向设备发送数据，成功时返回写入的字节数

file_operations结构体

```
struct file_operations {
    struct module *owner;
    loff_t(*llseek) (struct file *, loff_t, int);
    ssize_t(*read) (struct file *, char __user *, si
    ssize_t(*write) (struct file *, const char __user *, size_t, loff_t *);
    int (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
    …
};
```

提供设备相关控制命令

file_operations结构体

```
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 cha
    int (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
    …
};
```

将设备内存映射到进程的虚拟地址空间中

file_operations结构体

```
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 *);
    int (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
    …
};
```

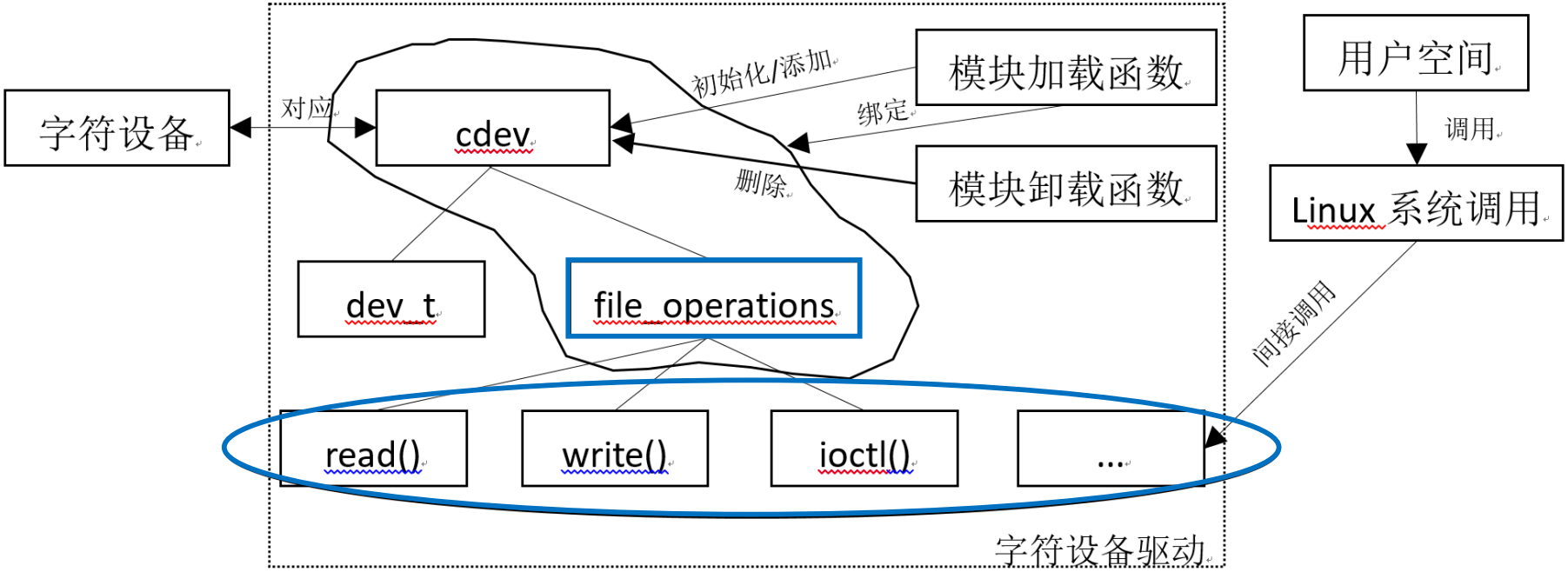
打开、关闭设备

内核空间与用户空间之间复制数据的方法

在用户空间不能直接访问内核空间的内存，因此借助下面两个函数，分别用来把数据从用户空间拷贝到内核空间，以及把数据从内核空间拷贝到用户空间。

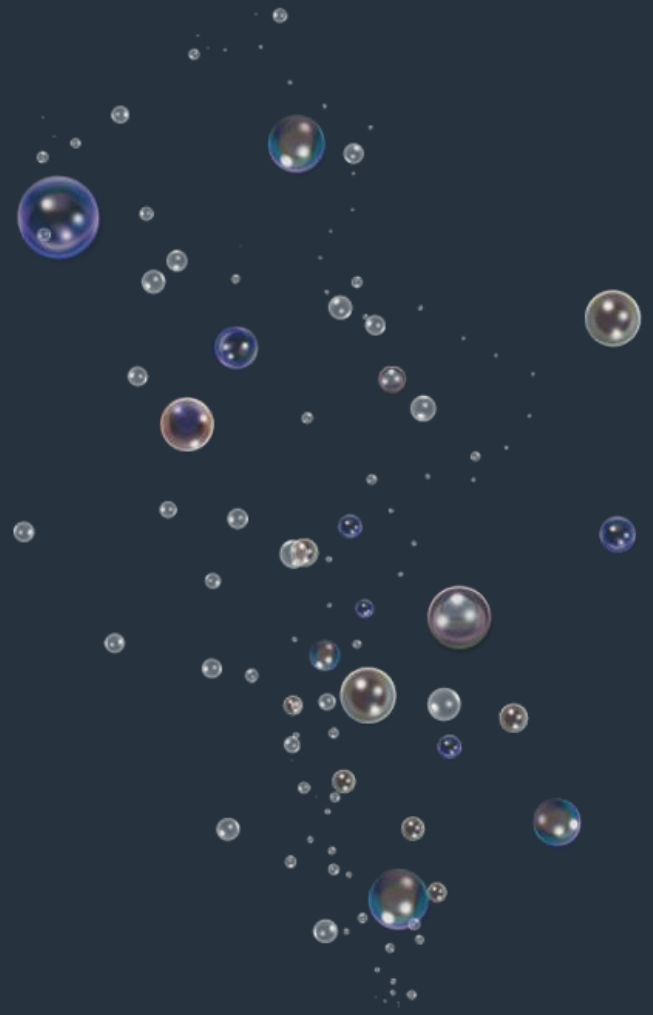
```
unsigned long copy_from_user(void *to, const void __user *from,
                             unsigned long count);
unsigned long copy_to_user (void __user *to, const void *from,
                            unsigned long count);
```

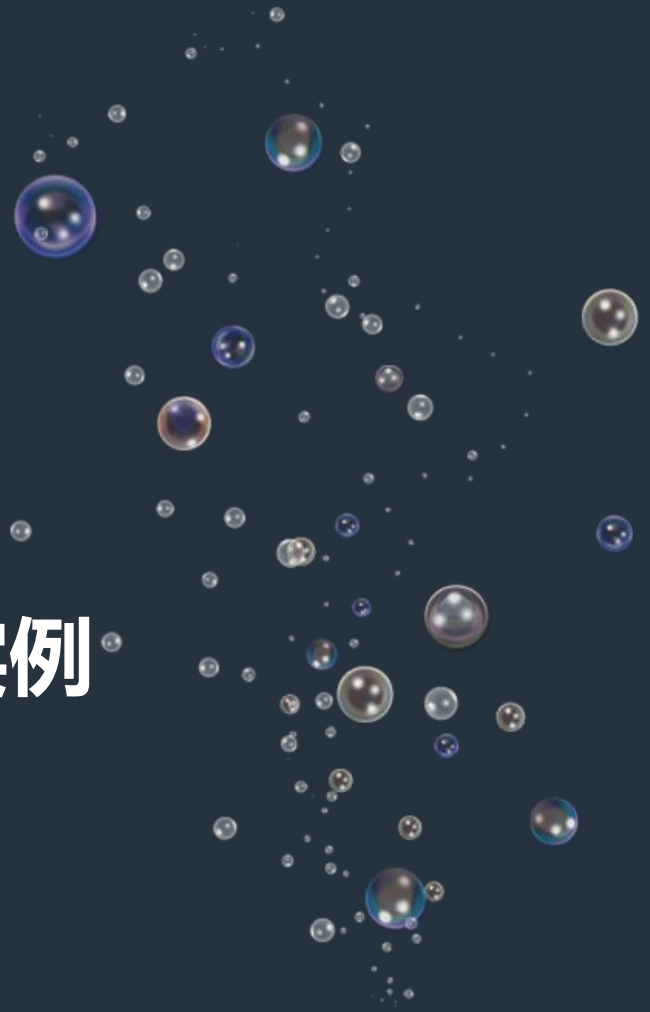
字符设备驱动结构


字符设备
对应
cdev
初始化/添加
绑定
删除
模块加载函数
模块卸载函数
dev_t
file_operations
read()
write()
ioctl()
...
字符设备驱动
用户空间
调用
Linux 系统调用
间接调用

嵌入式软件设计

大连理工大学 赖晓晨





虚拟字符设备驱动程序实例

大连理工大学 赖晓晨

Globalmem虚拟设备驱动实例

- 在内核空间申请一块4KB内存用于模拟一个设备，并在驱动中提供针对这块内存的读、写、控制和定位函数，以供用户空间的进程能通过Linux系统调用获取或者设置这块内存的内容。

- 虚拟设备驱动 vs 真实硬件设备驱动

Globalmem虚拟设备驱动实例

```
/*
 * a simple char device driver: globalmem without mutex
 *
 * Copyright (C) 2014 Barry Song  (baohua@kernel.org)
 *
 * Licensed under GPLv2 or later.
 */
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/cdev.h>
#include <linux/slab.h>
#include <linux/uaccess.h>

#define GLOBALMEM_SIZE        0x1000
#define MEM_CLEAR 0x1
#define GLOBALMEM_MAJOR 230
```

Globalmem虚拟设备驱动实例

```
/*
 * a simple char device driver: globalmem without mutex
 *
 * Copyright (C) 2014 Barry Song  (baohua@kernel.org)
 *
 * Licensed under GPLv2 or later.
 */
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/cdev.h>
#include <linux/slab.h>
#include <linux/uaccess.h>

#define GLOBALMEM_SIZE      0x1000
#define MEM_CLEAR 0x1
#define GLOBALMEM_MAJOR 230
```

虚拟设备内存：4K

Globalmem虚拟设备驱动实例

```
/*
 * a simple char device driver: globalmem without mutex
 *
 * Copyright (C) 2014 Barry Song  (baohua@kernel.org)
 *
 * Licensed under GPLv2 or later.
 */
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/cdev.h>
#include <linux/slab.h>
#include <linux/uaccess.h>

#define GLOBALMEM_SIZE	0x1000
#define MEM_CLEAR 0x1
#define GLOBALMEM_MAJOR 230
```

控制命令

Globalmem虚拟设备驱动实例

```
/*
 * a simple char device driver: globalmem without mutex
 *
 * Copyright (C) 2014 Barry Song  (baohua@kernel.org)
 *
 * Licensed under GPLv2 or later.
 */
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/cdev.h>
#include <linux/slab.h>
#include <linux/uaccess.h>

#define GLOBALMEM_SIZE        0x1000
#define MEM_CLEAR 0x1
#define GLOBALMEM_MAJOR 230
```

主设备号

```
static int globalmem_major = GLOBALMEM_MAJOR;
module_param(globalmem_major, int, S_IRUGO);

struct globalmem_dev {
        struct cdev cdev;
        unsigned char mem[GLOBALMEM_SIZE];
};

struct globalmem_dev *globalmem_devp;

static int globalmem_open(struct inode *inode, struct file *filp)
{
        filp->private_data = globalmem_devp;
        return 0;
}

static int globalmem_release(struct inode *inode, struct file *filp)
{
        return 0;
}
```

可选的模块参数，用于改变主设备号

```
static int globalmem_major = GLOBALMEM_MAJOR;
module_param(globalmem_major, int, S_IRUGO);

struct globalmem_dev {
        struct cdev cdev;
        unsigned char mem[GLOBALMEM_SIZE];
};

struct globalmem_dev *globalmem_devp;

static int globalmem_open(struct inode *inode, struct file *filp)
{
        filp->private_data = globalmem_devp;
        return 0;
}

static int globalmem_release(struct inode *inode, struct file *filp)
{
        return 0;
}
```

定义结构体（虚拟设备），及对应指针

```
struct file
{
    …
    const struct file_operations  *f_op;
    fmode_t  f_mode;
    void*  private_data;    //文件私有数据
    …
}
```

```
LMEM_MAJOR;
 S_IRUGO);


MEM_SIZE];

struct globalmem_dev *globalmem_devp;

static int globalmem_open(struct inode *inode, struct file *filp)
{
        filp->private_data = globalmem_devp;
        return 0;
}

static int globalmem_release(struct inode *inode, struct file *filp)
{
        return 0;
}
```

打开设备函数，私有数据设置

```
static int globalmem_major = GLOBALMEM_MAJOR;
module_param(globalmem_major, int, S_IRUGO);

struct globalmem_dev {
        struct cdev cdev;
        unsigned char mem[GLOBALMEM_SIZE];
};

struct globalmem_dev *globalmem_devp;
static int globalmem_open(struct inode *inode, struct file *filp)
{
        filp->private_data = globalmem_devp;
        return 0;
}

static int globalmem_release(struct inode *inode, struct file *filp)
{
        return 0;
}
```

关闭设备

```
static long globalmem_ioctl(struct file *filp, unsigned int cmd,
                            unsigned long arg)
{
        struct globalmem_dev *dev = filp->private_data;

        switch (cmd) {
        case MEM_CLEAR:
                memset(dev->mem, 0, GLOBALMEM_SIZE);
                printk(KERN_INFO "globalmem is set to zero\n");
                break;

        default:
                return -EINVAL;
        }

        return 0;
}
```

ioctl()函数（清内存）

```
static long globalmem_ioctl(struct file *filp, unsigned int cmd,
                                                    unsigned long arg)
{
        struct globalmem_dev *dev = filp->private_data;

        switch (cmd) {
        case MEM_CLEAR:
                memset(dev->mem, 0, GLOBALMEM_SIZE);
                printk(KERN_INFO "globalmem is set to zero\n");
                break;

        default:
                return -EINVAL;
        }

        return 0;
}
```

获取私有数据指针

```
static long globalmem_ioctl(struct file *filp, unsigned int cmd,
                                                    unsigned long arg)
{
        struct globalmem_dev *dev = filp->private_data;

        switch (cmd) {
        case MEM_CLEAR:
                memset(dev->mem, 0, GLOBALMEM_SIZE);
                printk(KERN_INFO "globalmem is set to zero\n");
                break;

        default:
                return -EINVAL;
        }

        return 0;
}
```

如果是MEM_CLEAR命令，则清内存

```
static long globalmem_ioctl(struct file *filp, unsigned int cmd,
                                                    unsigned long arg)
{
        struct globalmem_dev *dev = filp->private_data;

        switch (cmd) {
        case MEM_CLEAR:
                memset(dev->mem, 0, GLOBALMEM_SIZE);
                printk(KERN_INFO "globalmem is set to zero\n");
                break;

        default:
                return -EINVAL;
        }

        return 0;
}
```

如果是其他命令，则返回错误标志

```
static ssize_t globalmem_read(struct file *filp, char __user * buf,
                              size_t size, loff_t * ppos)
{
        unsigned long p = *ppos;
        unsigned int count = size;
        int ret = 0;
        struct globalmem_dev *dev = filp->private_data;
        if (p >= GLOBALMEM_SIZE)
                return 0;
        if (count > GLOBALMEM_SIZE - p)
                count = GLOBALMEM_SIZE - p;
        if (copy_to_user(buf, dev->mem + p, count)) {
                ret = -EFAULT;
        } else {
                *ppos += count;
                ret = count;
                printk(KERN_INFO "read %u bytes(s) from %lu\n", count, p);
        }

        return ret;
}
```

读函数

```
static ssize_t globalmem_read(struct file *filp, char __user * buf,
                              size_t size, loff_t * ppos)
{
        unsigned long p = *ppos;          //读取位置相对于文件头的偏移量
        unsigned int count = size;
        int ret = 0;
        struct globalmem_dev *dev = filp->private_data;
        if (p >= GLOBALMEM_SIZE)
                return 0;
        if (count > GLOBALMEM_SIZE - p)
                count = GLOBALMEM_SIZE - p;
        if (copy_to_user(buf, dev->mem + p, count)) {
                ret = -EFAULT;
        } else {
                *ppos += count;
                ret = count;
                printk(KERN_INFO "read %u bytes(s) from %lu\n", count, p);
        }

        return ret;
}
```

若干变量定义

```
static ssize_t globalmem_read(struct file *filp, char __user * buf,
                              size_t size, loff_t * ppos)
{
        unsigned long p = *ppos;
        unsigned int count = size;
        int ret = 0;
        struct globalmem_dev *dev = filp->private_data;
        if (p >= GLOBALMEM_SIZE)
                return 0;
        if (count > GLOBALMEM_SIZE - p)
                count = GLOBALMEM_SIZE - p;
        if (copy_to_user(buf, dev->mem + p, count)) {
                ret = -EFAULT;
        } else {
                *ppos += count;
                ret = count;
                printk(KERN_INFO "read %u bytes(s) from %lu\n", count, p);
        }

        return ret;
}
```

当前读取位置超出文件尾则直接返回0

```
static ssize_t globalmem_read(struct file *filp, char __user * buf,
                              size_t size, loff_t * ppos)
{
        unsigned long p = *ppos;
        unsigned int count = size;
        int ret = 0;
        struct globalmem_dev *dev = filp->private_data;
        if (p >= GLOBALMEM_SIZE)
                return 0;
        if (count > GLOBALMEM_SIZE - p)
                count = GLOBALMEM_SIZE - p;
        if (copy_to_user(buf, dev->mem + p, count)) {
                ret = -EFAULT;
        } else {
                *ppos += count;
                ret = count;
                printk(KERN_INFO "read %u bytes(s) from %lu\n", count, p);
        }

        return ret;
}
```

如从当前位置开始读取的内容长度超出文件尾，则读取长度为当前位置之后的全部内容

```
static ssize_t globalmem_read(struct file *filp, char __user * buf,
                              size_t size, loff_t * ppos)
{
        unsigned long p = *ppos;
        unsigned int count = size;
        int ret = 0;
        struct globalmem_dev *dev = filp->private_data;
        if (p >= GLOBALMEM_SIZE)
                return 0;
        if (count > GLOBALMEM_SIZE - p)
                count = GLOBALMEM_SIZE - p;
        if (copy_to_user(buf, dev->mem + p, count)) {
                ret = -EFAULT;
        } else {
                *ppos += count;
                ret = count;
                printk(KERN_INFO "read %u bytes(s) from %lu\n", count, p);
        }

        return ret;
}
```

读取信息，复制到用户空间缓冲区

```
static ssize_t globalmem_read(struct file *filp, char __user * buf,
                              size_t size, loff_t * ppos)
{
        unsigned long p = *ppos;
        unsigned int count = size;
        int ret = 0;
        struct globalmem_dev *dev = filp->private_data;
        if (p >= GLOBALMEM_SIZE)
                return 0;
        if (count > GLOBALMEM_SIZE - p)
                count = GLOBALMEM_SIZE - p;
        if (copy_to_user(buf, dev->mem + p, count)) {
                ret = -EFAULT;
        } else {
                *ppos += count;
                ret = count;
                printk(KERN_INFO "read %u bytes(s) from %lu\n", count, p);
        }

        return ret;
}
```

移动当前读写位置，并提示读到的字节数

```
static ssize_t globalmem_write(struct file *filp, const char __user * buf,
                               size_t size, loff_t * ppos)
{
        unsigned long p = *ppos;
        unsigned int count = size;
        int ret = 0;
        struct globalmem_dev *dev = filp->private_data;
        if (p >= GLOBALMEM_SIZE)
                return 0;
        if (count > GLOBALMEM_SIZE - p)
                count = GLOBALMEM_SIZE - p;
        if (copy_from_user(dev->mem + p, buf, count))
                ret = -EFAULT;
        else {
                *ppos += count;
                ret = count;
                printk(KERN_INFO "written %u bytes(s) from %lu\n", count, p);
        }

        return ret;
}
```

写函数

```
static ssize_t globalmem_write(struct file *filp, const char __user * buf,
                               size_t size, loff_t * ppos)
{
        unsigned long p = *ppos;
        unsigned int count = size;
        int ret = 0;
        struct globalmem_dev *dev = filp->private_data;
        if (p >= GLOBALMEM_SIZE)
                return 0;
        if (count > GLOBALMEM_SIZE - p)
                count = GLOBALMEM_SIZE - p;
        if (copy_from_user(dev->mem + p, buf, count))
                ret = -EFAULT;
        else {
                *ppos += count;
                ret = count;
                printk(KERN_INFO "written %u bytes(s) from %lu\n", count, p);
        }

        return ret;
}
```

变量定义

```
static ssize_t globalmem_write(struct file *filp, const char __user * buf,
                               size_t size, loff_t * ppos)
{
        unsigned long p = *ppos;
        unsigned int count = size;
        int ret = 0;
        struct globalmem_dev *dev = filp->private_data;
        if (p >= GLOBALMEM_SIZE)
                return 0;
        if (count > GLOBALMEM_SIZE - p)
                count = GLOBALMEM_SIZE - p;
        if (copy_from_user(dev->mem + p, buf, count))
                ret = -EFAULT;
        else {
                *ppos += count;
                ret = count;
                printk(KERN_INFO "written %u bytes(s) from %lu\n", count, p);
        }

        return ret;
}
```

当前写入位置超出文件尾则直接返回0

```
static ssize_t globalmem_write(struct file *filp, const char __user * buf,
                               size_t size, loff_t * ppos)
{
        unsigned long p = *ppos;
        unsigned int count = size;
        int ret = 0;
        struct globalmem_dev *dev = filp->private_data;
        if (p >= GLOBALMEM_SIZE)
                return 0;
        if (count > GLOBALMEM_SIZE - p)
                count = GLOBALMEM_SIZE - p;
        if (copy_from_user(dev->mem + p, buf, count))
                ret = -EFAULT;
        else {
                *ppos += count;
                ret = count;
                printk(KERN_INFO "written %u bytes(s) from %lu\n", count, p);
        }

        return ret;
}
```

如从当前位置开始写入的数据长度超出文件尾，则写入数据长度为全部剩余空间

```
static ssize_t globalmem_write(struct file *filp, const char __user * buf,
                               size_t size, loff_t * ppos)
{
        unsigned long p = *ppos;
        unsigned int count = size;
        int ret = 0;
        struct globalmem_dev *dev = filp->private_data;
        if (p >= GLOBALMEM_SIZE)
                return 0;
        if (count > GLOBALMEM_SIZE - p)
                count = GLOBALMEM_SIZE - p;
        if (copy_from_user(dev->mem + p, buf, count))
                ret = -EFAULT;
        else {
                *ppos += count;
                ret = count;
                printk(KERN_INFO "written %u bytes(s) from %lu\n", count, p);
        }

        return ret;
}
```

写入信息，复制到内核空间缓冲区

```
static ssize_t globalmem_write(struct file *filp, const char __user * buf,
                               size_t size, loff_t * ppos)
{
        unsigned long p = *ppos;
        unsigned int count = size;
        int ret = 0;
        struct globalmem_dev *dev = filp->private_data;
        if (p >= GLOBALMEM_SIZE)
                return 0;
        if (count > GLOBALMEM_SIZE - p)
                count = GLOBALMEM_SIZE - p;
        if (copy_from_user(dev->mem + p, buf, count))
                ret = -EFAULT;
        else {
            *ppos += count;
            ret = count;
            printk(KERN_INFO "written %u bytes(s) from %lu\n", count, p);
        }

        return ret;
}
```

移动当前读写位置，并提示写入的字节数

```
static loff_t globalmem_llseek(struct file *filp, loff_t offset, int orig)
{
        loff_t ret = 0;
        switch (orig) {
        case 0:
                if (offset < 0) {
                        ret = -EINVAL;
                        break;
                }
                if ((unsigned int)offset > GLOBALMEM_SIZE) {
                        ret = -EINVAL;
                        break;
                }
                filp->f_pos = (unsigned int)offset;
                ret = filp->f_pos;
                break;
```

移动当前读写位置函数：seek()函数支持从文件头（orig为0），以及文件当前位置（orig为1）开始移动读写位置。

```
static loff_t globalmem_llseek(struct file *filp, loff_t offset, int orig)
{
        loff_t ret = 0;
        switch (orig) {
        case 0:
                if (offset < 0) {
                        ret = -EINVAL;
                        break;
                }
                if ((unsigned int)offset > GLOBALMEM_SIZE) {
                        ret = -EINVAL;
                        break;
                }
                filp->f_pos = (unsigned int)offset;
                ret = filp->f_pos;
                break;
```

从文件头开始，如偏移量为负

```
static loff_t globalmem_llseek(struct file *filp, loff_t offset, int orig)
{
        loff_t ret = 0;
        switch (orig) {
        case 0:
                if (offset < 0) {
                        ret = -EINVAL;
                        break;
                }
                if ((unsigned int)offset > GLOBALMEM_SIZE) {
                        ret = -EINVAL;
                        break;
                }
                filp->f_pos = (unsigned int)offset;
                ret = filp->f_pos;
                break;
```

从文件头开始，如偏移量超出文件长度

```
static loff_t globalmem_llseek(struct file *filp, loff_t offset, int orig)
{
        loff_t ret = 0;
        switch (orig) {
        case 0:
                if (offset < 0) {
                        ret = -EINVAL;
                        break;
                }
                if ((unsigned int)offset > GLOBALMEM_SIZE) {
                        ret = -EINVAL;
                        break;
                }
                filp->f_pos = (unsigned int)offset;
                ret = filp->f_pos;
                break;
```

移动当前读写位置

```
        case 1:
                if ((filp->f_pos + offset) > GLOBALMEM_SIZE) {
                        ret = -EINVAL;
                        break;
                }
                if ((filp->f_pos + offset) < 0) {
                        ret = -EINVAL;
                        break;
                }
                filp->f_pos += offset;
                ret = filp->f_pos;
                break;
        default:
                ret = -EINVAL;
                break;
        }
        return ret;
}
```

从当前位置开始，如加偏移量超出文件尾

```
        case 1:
                if ((filp->f_pos + offset) > GLOBALMEM_SIZE) {
                        ret = -EINVAL;
                        break;
                }
                if ((filp->f_pos + offset) < 0) {
                        ret = -EINVAL;
                        break;
                }
                filp->f_pos += offset;
                ret = filp->f_pos;
                break;
        default:
                ret = -EINVAL;
                break;
        }
        return ret;
}
```

从当前位置开始，如加偏移量小于0

```
        case 1:
                if ((filp->f_pos + offset) > GLOBALMEM_SIZE) {
                        ret = -EINVAL;
                        break;
                }
                if ((filp->f_pos + offset) < 0) {
                        ret = -EINVAL;
                        break;
                }
                filp->f_pos += offset;
                ret = filp->f_pos;
                break;
        default:
                ret = -EINVAL;
                break;
        }
        return ret;
}
```

移动当前读写位置

```
        case 1:
                if ((filp->f_pos + offset) > GLOBALMEM_SIZE) {
                        ret = -EINVAL;
                        break;
                }
                if ((filp->f_pos + offset) < 0) {
                        ret = -EINVAL;
                        break;
                }
                filp->f_pos += offset;
                ret = filp->f_pos;
                break;
        default:
                ret = -EINVAL;
                break;
        }
        return ret;
}
```

既不是从文件头开始，也不是从当前位置开始，返回错误标志

```
static const struct file_operations globalmem_fops = {
        .owner = THIS_MODULE,
        .llseek = globalmem_llseek,
        .read = globalmem_read,
        .write = globalmem_write,
        .unlocked_ioctl = globalmem_ioctl,
        .open = globalmem_open,
        .release = globalmem_release,
};

static void globalmem_setup_cdev(struct globalmem_dev *dev, int index)
{
        int err, devno = MKDEV(globalmem_major, index);

        cdev_init(&dev->cdev, &globalmem_fops);
        dev->cdev.owner = THIS_MODULE;
        err = cdev_add(&dev->cdev, devno, 1);
        if (err)
            printk(KERN_NOTICE "Error %d adding globalmem%d", err, index);
}
```

操作集定义及赋值

```
static const struct file_operations globalmem_fops = {
        .owner = THIS_MODULE,
        .llseek = globalmem_llseek,
        .read = globalmem_read,
        .write = globalmem_write,
        .unlocked_ioctl = globalmem_ioctl,
        .open = globalmem_open,
        .release = globalmem_release,
};

static void globalmem_setup_cdev(struct globalmem_dev *dev, int index)
{
        int err, devno = MKDEV(globalmem_major, index);

        cdev_init(&dev->cdev, &globalmem_fops);
        dev->cdev.owner = THIS_MODULE;
        err = cdev_add(&dev->cdev, devno, 1);
        if (err)
            printk(KERN_NOTICE "Error %d adding globalmem%d", err, index);
}
```

设备注册函数

```
static const struct file_operations globalmem_fops = {
        .owner = THIS_MODULE,
        .llseek = globalmem_llseek,
        .read = globalmem_read,
        .write = globalmem_write,
        .unlocked_ioctl = globalmem_ioctl,
        .open = globalmem_open,
        .release = globalmem_release,
};

static void globalmem_setup_cdev(struct globalmem_dev *dev, int index)
{
        int err, devno = MKDEV(globalmem_major, index);

        cdev_init(&dev->cdev, &globalmem_fops);
        dev->cdev.owner = THIS_MODULE;
        err = cdev_add(&dev->cdev, devno, 1);
        if (err)
            printk(KERN_NOTICE "Error %d adding globalmem%d", err, index);
}
```

生成设备号

```
static const struct file_operations globalmem_fops = {
```

```
struct cdev
{
    struct kobject kobj;                //内嵌的kobject对象
    struct module *owner;               //所属模块
    struct file_operations *ops;        //文件操作结构体
    dev_t  dev;                         //设备号
    unsigned int count ;
}
```

```
static void globalmem_setup_cdev(struct globalme
{
        int err, devno = MKDEV(globalmem_major, index);

        cdev_init(&dev->cdev, &globalmem_fops);
        dev->cdev.owner = THIS_MODULE;
        err = cdev_add(&dev->cdev, devno, 1);
        if (err)
            printk(KERN_NOTICE "Error %d adding globalmem%d", err, index);
}
```

初始化设备文件结构体

```
static const struct file_operations globalmem_fops = {
        .owner = THIS_MODULE,
        .llseek = globalmem_llseek,
        .read = globalmem_read,
        .write = globalmem_write,
        .unlocked_ioctl = globalmem_ioctl,
        .open = globalmem_open,
        .release = globalmem_release,
};

static void globalmem_setup_cdev(struct globalmem_dev *dev, int index)
{
        int err, devno = MKDEV(globalmem_
        cdev_init(&dev->cdev, &globalmem_f
        dev->cdev.owner = THIS_MODULE;
        err = cdev_add(&dev->cdev, devno, 1);
        if (err)
            printk(KERN_NOTICE "Error %d adding globalmem%d", err, index);
}
```

向系统注册设备

模块加载函数

```
static int __init globalmem_init(void)
{
        int ret;
        dev_t devno = MKDEV(globalmem_major, 0);
        if (globalmem_major)
                ret = register_chrdev_region(devno, 1, "globalmem");
        else {
                ret = alloc_chrdev_region(&devno, 0, 1, "globalmem");
                globalmem_major = MAJOR(devno);
        }
        if (ret < 0)
                return ret;
        globalmem_devp = kzalloc(sizeof(struct globalmem_dev), GFP_KERNEL);
        if (!globalmem_devp) {
                ret = -ENOMEM;
                goto fail_malloc;
        }
        globalmem_setup_cdev(globalmem_devp, 0);
        return 0;
 fail_malloc:
        unregister_chrdev_region(devno, 1);
        return ret;
}
module_init(globalmem_init);
```

```
static int __init globalmem_init(void)
{
        int ret;
        dev_t devno = MKDEV(globalmem_major, 0);
        if (globalmem_major)
                ret = register_chrdev_region(devno, 1, "globalmem");
        else {
                ret = alloc_chrdev_region(&devno, 0, 1, "globalmem");
                globalmem_major = MAJOR(devno);
        }
        if (ret < 0)
                return ret;
        globalmem_devp = kzalloc(sizeof(struct globalmem_dev), GFP_KERNEL);
        if (!globalmem_devp) {
                ret = -ENOMEM;
                goto fail_malloc;
        }
        globalmem_setup_cdev(globalmem_devp, 0);
        return 0;
 fail_malloc:
        unregister_chrdev_region(devno, 1);
        return ret;
}
module_init(globalmem_init);
```

生成主设备号

```
static int __init globalmem_init(void)
{
        int ret;
        dev_t devno = MKDEV(globalmem_major, 0);
        if (globalmem_major)
                ret = register_chrdev_region(devno, 1, "globalmem");
        else {
                ret = alloc_chrdev_region(&devno, 0, 1, "globalmem");
                globalmem_major = MAJOR(devno);
        }
        if (ret < 0)
                return ret;
        globalmem_devp = kzalloc(sizeof(struct globalmem_dev), GFP_KERNEL);
        if (!globalmem_devp) {
                ret = -ENOMEM;
                goto fail_malloc;
        }
        globalmem_setup_cdev(globalmem_devp, 0);
        return 0;
 fail_malloc:
        unregister_chrdev_region(devno, 1);
        return ret;
}
module_init(globalmem_init);
```

根据是否已有主设备号，选用不同函数向系统申请注册设备号

```
static int __init globalmem_init(void)
{
        int ret;
        dev_t devno = MKDEV(globalmem_major, 0);
        if (globalmem_major)
                ret = register_chrdev_region(devno, 1, "globalmem");
        else {
                ret = alloc_chrdev_region(&devno, 0, 1, "globalmem");
                globalmem_major = MAJOR(devno);
        }
        if (ret < 0)
                return ret;
        globalmem_devp = kzalloc(sizeof(struct globalmem_dev),GFP_KERNEL);
        if (!globalmem_devp) {
                ret = -ENOMEM;
                goto fail_malloc;
        }
        globalmem_setup_cdev(globalmem_devp, 0);
        return 0;
 fail_malloc:
        unregister_chrdev_region(devno, 1);
        return ret;
}
module_init(globalmem_init);
```

申请设备内存

```
static int     init globalmem_init(void)
static void globalmem_setup_cdev(struct globalmem_dev *dev, int index)
{
        int err, devno = MKDEV(globalmem_major, index);

        cdev_init(&dev->cdev, &globalmem_fops);
        dev->cdev.owner = THIS_MODULE;
        err = cdev_add(&dev->cdev, devno, 1);
        if (err)
            printk(KERN_NOTICE "Error %d adding globalmem%d", err, index);
}
                                                                    ERNEL);
                if (!globalmem_devp) {
                        ret = -ENOMEM;
                        goto fail_malloc;
                }
                globalmem_setup_cdev(globalmem_devp, 0);
                return 0;
         fail_malloc:
                unregister_chrdev_region(devno, 1);
                return ret;
        }
        module_init(globalmem_init);
```

注册设备

```
static void __exit globalmem_exit(void)
{
        cdev_del(&globalmem_devp->cdev);
        kfree(globalmem_devp);
        unregister_chrdev_region(MKDEV(globalmem_major, 0), 1);
}
module_exit(globalmem_exit);

MODULE_AUTHOR("Barry Song <baohua@kernel.org>");
MODULE_LICENSE("GPL v2");
```

模块卸载函数

```
static void __exit globalmem_exit(void)
{
        cdev_del(&globalmem_devp->cdev);
        kfree(globalmem_devp);
        unregister_chrdev_region(MKDEV(globalmem_major, 0), 1);
}
module_exit(globalmem_exit);

MODULE_AUTHOR("Barry Song <baohua@kernel.org>");
MODULE_LICENSE("GPL v2");
```

注销设备

```
static void __exit globalmem_exit(void)
{
        cdev_del(&globalmem_devp->cdev);
        kfree(globalmem_devp);
        unregister_chrdev_region(MKDEV(globalmem_major, 0), 1);
}
module_exit(globalmem_exit);

MODULE_AUTHOR("Barry Song <baohua@kernel.org>");
MODULE_LICENSE("GPL v2");
```

释放内存空间

```
static void __exit globalmem_exit(void)
{
        cdev_del(&globalmem_devp->cdev);
        kfree(globalmem_devp);
        unregister_chrdev_region(MKDEV(globalmem_major, 0), 1);
}
module_exit(globalmem_exit);

MODULE_AUTHOR("Barry Song <baohua@kernel.org>");
MODULE_LICENSE("GPL v2");
```

注销设备号

```
static void __exit globalmem_exit(void)
{
        cdev_del(&globalmem_devp->cdev);
        kfree(globalmem_devp);
        unregister_chrdev_region(MKDEV(globalmem_major, 0), 1);
}
module_exit(globalmem_exit);

MODULE_AUTHOR("Barry Song <baohua@kernel.org>");
MODULE_LICENSE("GPL v2");
```

作者说明、许可证声明

driver/char_dev.c

操作步骤

```
$ sudo –i
# insmod char_dev.ko
# lsmod
# cat /proc/devices
# mknod /dev/char_dev  c  230  0
# echo  “hello world”  > /dev/char_dev
# cat  /dev/char_dev
```

嵌入式软件设计

大连理工大学 赖晓晨