字符设备文件主要完成的几个功能：
打开文件：memdev为打开设备的函数，系统调用open()，将设备结构体指针赋值给文件私有数据指针，最终落实到这个memdev_open()
读文件：文件读函数，系统调用read()最终落实到这个memdev_read()，首先将获得字符结构体指针，判断MEMDEV的值是否小于等于缓冲buf在内存中最后的位置，如果偏移位置大于MEMDEV的值则返回，否则将文件的size-buf当前的偏移量赋值给count，如果调取copy_to_use成功执行以偏移量p为起始位置读取count个字符
写文件：文件写函数，系统调用write()最终落实到这个memdev_write()，首先将获得字符结构体指针，判断MEMDEV的值是否小于等于缓冲buf在内存中最后的位置，如果偏移位置大于MEMDEV的值则返回，否则将文件的size-buf当前的偏移量赋值给count，如果调取copy_from_user成功执行以偏移量p为起始位置写入count个字符
设备控制：设备控制函数，系统调用ioctl()最终落实到这个memdev_ioctl()，控制设备是否需要完成内存清0
设备驱动加载：设备驱动模块加载函数，完成申请设备号，申请成功加载模块后，在/dev/目录下会新生成memdev这个文件节点，并申请结构体所需内存；若申请设备失败，直接返回
设备驱动卸载：驱动模块卸载函数，完成注销设备，释放结构体内存，释放设备号

```
/*头文件*/
#include<linux/module.h>
#include<linux/init.h>
#include <linux/moduleparam.h>
#include<linux/kernel.h>    /* printk() */
#include <linux/slab.h>     /* kmalloc() */
#include<linux/types.h>  /* size_t */
#include<linux/fs.h>    /* everything... */
#include<linux/errno.h> /* error codes */
#include<linux/mm.h>
#include<linux/sched.h>
#include<linux/cdev.h>
#include<asm/io.h>
#include<asm/system.h>
#include<asm/uaccess.h>  /* copy_*_user */

#define MEMDEV_SIZE  0x1000  /* size=4KB */
#define MEM_CLEAR  0x1  /* a command of clear memory for ioctl() */
#define MEMDEV_MAJOR 0

int memdev_major = MEMDEV_MAJOR;
module_param(memdev_major, int, S_IRUGO);
struct memdev_dev{
   struct cdev cdev;       /*定义字符设备cdev 结构体*/
   unsigned char mem[MEMDEV_SIZE];    /*定义字符设备内存的大小size=4KB*/
};

struct memdev_dev  *memdev_devp;  /*指向字符设备结构体指针*/

/*memdev为打开设备的函数，系统调用open()，将设备结构体指针赋值给文件私有数据指针，最终落实到这个memdev_open()*/
static int memdev_open(struct inode *inodep,struct file *filp)
{
```

```
    /*将设备结构体指针赋值给文件私有数据指针*/
    filp->private_data = memdev_devp;
    return 0;
}


/*文件释放函数,系统调用close()最终落实到这个memdev_release()*/
static int memdev_release(struct inode *inodep,struct file *filp)
{
    return 0;
}

/*文件读函数，系统调用read()最终落实到这个memdev_read()，首先将获得字符结构体指针，判断MEMDEV的值是否小于等于缓冲buf在内存中最后的位置，如果偏移位置大于MEMDEV的值则返回，否则将文件的size-buf当前的偏移量赋值给count，如果调取copy_to_use成功执行以偏移量p为起始位置读取count个字符*/
static ssize_t memdev_read(struct file *filp,char __user *buf,size_t size,loff_t *ppos)
{
    unsigned long p = *ppos;
    unsigned int count = size;
    int ret = 0;
    /*获得设备结构体指针*/
    struct memdev_dev *dev = filp->private_data;
    if(p >= MEMDEV_SIZE)
        return 0;
    if(count > MEMDEV_SIZE - p)
        count = MEMDEV_SIZE - p;
    if(copy_to_user(buf,(void *)dev->mem+p,count)){
        ret = -EFAULT;
    }
    else{
        *ppos += count;
        ret = count;
        printk(KERN_WARNING "Read %u byte(s) from %lu \n",count,p);
    }
    return ret;
}

/*文件写函数，系统调用write()最终落实到这个memdev_write()，首先将获得字符结构体指针，判断MEMDEV的值是否小于等于缓冲buf在内存中最后的位置，如果偏移位置大于MEMDEV的值则返回，否则将文件的size-buf当前的偏移量赋值给count，如果调取copy_from_user成功执行以偏移量p为起始位置写入count个字符*/
static ssize_t memdev_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;

    /*获得设备结构体指针*/
    struct memdev_dev *dev = filp->private_data;
    if(p >= MEMDEV_SIZE)
        return 0;
    if(count > MEMDEV_SIZE - p)
```

```
        count = MEMDEV_SIZE - p;
    if(copy_from_user(dev->mem + p,buf,count))
        ret = -EFAULT;
    else{
        *ppos += count;
        ret = count;
        printk(KERN_WARNING "Written %u byte(s) from %lu \n",count,p);
    }
    return ret;

}
/*设备控制函数，系统调用ioctl()最终落实到这个memdev_ioctl()，控制设备是否需要完成内存清0*/
static int memdev_ioctl(struct inode *inodep,struct file *filp,unsigned int cmd,unsigned long arg)
{
    struct memdev_dev *dev = filp->private_data;

    switch(cmd){
        case MEM_CLEAR:
            memset(dev->mem,0,MEMDEV_SIZE); /*清零*/
            printk(KERN_WARNING "memdev has set to zero.\n");
            break;

        default:
            return -EINVAL; /*暂不支持其他命令*/
    }
    return 0;
}
/*文件定位函数，系统调用seek()最终落实到这个memdev_ioctl()*/
static loff_t memdev_llseek(struct file *filp,loff_t offset,int whence)
{
    loff_t ret = 0;
    switch(whence){
        case 0:  /*相对文件开始位置偏移*/
            if(offset < 0){
                ret = -EINVAL;
                break;
            }
            if((unsigned int )offset > MEMDEV_SIZE){
                ret = -EINVAL;
                break;
            }

            filp->f_pos = (unsigned int)offset;
            ret = filp->f_pos;
            break;


        case 1:  /*相对文件当前位置偏移*/
            if((filp->f_pos + offset) < 0){
                ret = -EINVAL;
                break;
            }
            if((filp->f_pos + offset) > MEMDEV_SIZE){
                ret = -EINVAL;
```

```
            break;
        }

        filp->f_pos += (unsigned int)offset;
        ret = filp->f_pos;
</span><span style="font-size:14px;">        break;
    default:
        ret = -EINVAL;
        break;
  }
  return ret;

}

/*文件操作结构体--file_operations */
static const struct file_operations memdev_fops = {
  .owner = THIS_MODULE,
  .open = memdev_open,
  .release = memdev_release,
  .read = memdev_read,
  .write = memdev_write,
  .ioctl = memdev_ioctl,
  .llseek = memdev_llseek,
};

/*初始化cdev,添加注册cdev*/
static void memdev_setup_cdev(struct memdev_dev *dev,int index)
{
  int err,devno = MKDEV(memdev_major,index);

  cdev_init(&dev->cdev,&memdev_fops);
  dev->cdev.owner = THIS_MODULE;
  dev->cdev.ops = &memdev_fops;
  err = cdev_add(&dev->cdev,devno,1);

  if(err)
    printk(KERN_WARNING "Error %d adding memdev %d",err,index);
}

/*设备驱动模块加载函数，完成申请设备号，申请成功加载模块后，在/dev/目录下会新生成memdev
这个文件节点，并申请结构体所需内存；若申请设备失败，直接返回*/
static int __init memdev_init(void)
{
  int result;
  dev_t devno = MKDEV(memdev_major,0);  //分配一个dev_t 设备编号

  /*申请设备号*/
  if(memdev_major)  /*申请成功时，加载模块后，在/dev/目录下会新生成memdev这个文件节点*/
    result = register_chrdev_region(devno,1,"memdev");
  else{    /*动态申请设备号*/
      result = alloc_chrdev_region(&devno, 0, 1, "memdev");
      memdev_major = MAJOR(devno);
  }
  if(result<0)
```

```
        return result;

    /*动态申请设备结构体的内存*/
    memdev_devp = kmalloc(sizeof(struct memdev_dev),GFP_KERNEL);
    if(!memdev_devp){  /*动态申请设备结构体的内存失败*/
        result = -ENOMEM;
        goto fail_malloc;  /*失败处理*/
    }

    memset(memdev_devp,0,sizeof(struct memdev_dev)); //清零

    memdev_setup_cdev(memdev_devp,0);
    return 0;
fail_malloc:
                unregister_chrdev_region(devno, 1);
                return result;
}
/*驱动模块卸载函数，完成注销设备，释放结构体内存，释放设备号*/
static void __exit memdev_exit(void)
{
    cdev_del(&memdev_devp->cdev); //注销cdev结构
    kfree(memdev_devp);         //释放设备结构体内存
    unregister_chrdev_region(MKDEV(memdev_major,0), 1);  //释放设备号
}

module_init(memdev_init);
module_exit(memdev_exit);

MODULE_AUTHOR("lwj<http://blog.csdn.net/lwj103862095>");
MODULE_LICENSE("Dual BSD/GPL");
MODULE_DESCRIPTION("a simple char driver in memory");</span>
```