

# 第6章

## 字符设备驱动

### 本章导读

在整个 Linux 设备驱动的学习中，字符设备驱动较为基础。本章将讲解 Linux 字符设备驱动程序的结构，并解释其主要组成部分的编程方法。

6.1 节讲解了 Linux 字符设备驱动的关键数据结构 cdev 及 file\_operations 结构体的操作方法，并分析了 Linux 字符设备的整体结构，给出了简单的设计模板。

6.2 节描述了本章及后续各章节所基于的 globalmem 虚拟字符设备，第 6 ~ 9 章都将基于该虚拟设备实例进行字符设备驱动及并发控制等知识的讲解。

6.3 节依据 6.1 节的知识讲解 globalmem 的设备驱动编写方法，对读写函数、seek() 函数和 I/O 控制函数等进行了重点分析。该节的最后也讲解了 Linux 驱动编程“私有数据”的用法。

6.4 节给出了 6.3 节的 globalmem 设备驱动在用户空间的验证。

### 6.1 Linux 字符设备驱动结构

#### 6.1.1 cdev 结构体

在 Linux 内核中，使用 cdev 结构体描述一个字符设备，cdev 结构体的定义如代码清单 6.1。

代码清单 6.1 cdev 结构体

```
1 struct cdev {  
2     struct kobject kobj;           /* 内嵌的 kobject 对象 */  
3     struct module *owner;          /* 所属模块 */  
4     struct file_operations *ops;    /* 文件操作结构体 */  
5     struct list_head list;  
6     dev_t dev;                   /* 设备号 */  
7     unsigned int count;  
8 };
```

cdev 结构体的 dev\_t 成员定义了设备号，为 32 位，其中 12 位为主设备号，20 位为次设

备号。使用下列宏可以从 dev\_t 获得主设备号和次设备号：

```
MAJOR(dev_t dev)
MINOR(dev_t dev)
```

而使用下列宏则可以通过主设备号和次设备号生成 dev\_t：

```
MKDEV(int major, int minor)
```

cdev 结构体的另一个重要成员 file\_operations 定义了字符设备驱动提供给虚拟文件系统的接口函数。

Linux 内核提供了一组函数以用于操作 cdev 结构体：

```
void cdev_init(struct cdev *, struct file_operations *);
struct cdev *cdev_alloc(void);
void cdev_put(struct cdev *p);
int cdev_add(struct cdev *, dev_t, unsigned);
void cdev_del(struct cdev *);
```

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，其源代码如代码清单 6.2 所示。

代码清单 6.2 cdev\_init() 函数

---

```
1 void cdev_init(struct cdev *cdev, struct file_operations *fops)
2 {
3     memset(cdev, 0, sizeof(*cdev));
4     INIT_LIST_HEAD(&cdev->list);
5     kobject_init(&cdev->kobj, &ktype_cdev_default);
6     cdev->ops = fops; /* 将传入的文件操作结构体指针赋值给 cdev 的 ops */
7 }
```

---

cdev\_alloc() 函数用于动态申请一个 cdev 内存，其源代码如代码清单 6.3 所示。

代码清单 6.3 cdev\_alloc() 函数

---

```
1 struct cdev *cdev_alloc(void)
2 {
3     struct cdev *p = kzalloc(sizeof(struct cdev), GFP_KERNEL);
4     if (p) {
5         INIT_LIST_HEAD(&p->list);
6         kobject_init(&p->kobj, &ktype_cdev_dynamic);
7     }
8     return p;
9 }
```

---

cdev\_add() 函数和 cdev\_del() 函数分别向系统添加和删除一个 cdev，完成字符设备的注册和注销。对 cdev\_add() 的调用通常发生在字符设备驱动模块加载函数中，而对 cdev\_del() 函数的调用则通常发生在字符设备驱动模块卸载函数中。

### 6.1.2 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，应首先调用 `register_chrdev_region()` 或 `alloc_chrdev_region()` 函数向系统申请设备号，这两个函数的原型为：

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

`register_chrdev_region()` 函数用于已知起始设备的设备号的情况，而 `alloc_chrdev_region()` 用于设备号未知，向系统动态申请未被占用的设备号的情况，函数调用成功之后，会把得到的设备号放入第一个参数 `dev` 中。`alloc_chrdev_region()` 相比于 `register_chrdev_region()` 的优点在于它会自动避开设备号重复的冲突。

相应地，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，这个函数的原型为：

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

### 6.1.3 file\_operations 结构体

`file_operations` 结构体中的成员函数是字符设备驱动程序设计的主体内容，这些函数实际会在应用程序进行 Linux 的 `open()`、`write()`、`read()`、`close()` 等系统调用时最终被内核调用。`file_operations` 结构体目前已经比较庞大，它的定义如代码清单 6.4 所示。

代码清单 6.4 file\_operations 结构体

```

22     int (*check_flags)(int);
23     int (*flock) (struct file *, int, struct file_lock *);
24     ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *,
25                             size_t, unsigned int);
26     ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *,
27                           size_t, unsigned int);
28     int (*setlease)(struct file *, long, struct file_lock **);
29     long (*fallocate)(struct file *file, int mode, loff_t offset,
30                         loff_t len);
31     int (*show_fdinfo)(struct seq_file *m, struct file *f);
32 };

```

---

下面我们对 `file_operations` 结构体中的主要成员进行分析。

`llseek()` 函数用来修改一个文件的当前读写位置，并将新位置返回，在出错时，这个函数返回一个负值。

`read()` 函数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。它与用户空间应用程序中的 `ssize_t read(int fd, void *buf, size_t count)` 和 `size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)` 对应。

`write()` 函数向设备发送数据，成功时该函数返回写入的字节数。如果此函数未被实现，当用户进行 `write()` 系统调用时，将得到 `-EINVAL` 返回值。它与用户空间应用程序中的 `ssize_t write(int fd, const void *buf, size_t count)` 和 `size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)` 对应。

`read()` 和 `write()` 如果返回 0，则暗示 `end-of-file (EOF)`。

`unlocked_ioctl()` 提供设备相关控制命令的实现（既不是读操作，也不是写操作），当调用成功时，返回给调用程序一个非负值。它与用户空间应用程序调用的 `int fcntl(int fd, int cmd, ... /* arg */)` 和 `int ioctl(int d, int request, ...)` 对应。

`mmap()` 函数将设备内存映射到进程的虚拟地址空间中，如果设备驱动未实现此函数，用户进行 `mmap()` 系统调用时将获得 `-ENODEV` 返回值。这个函数对于帧缓冲等设备特别有意义，帧缓冲被映射到用户空间后，应用程序可以直接访问它而无须在内核和应用间进行内存复制。它与用户空间应用程序中的 `void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)` 函数对应。

当用户空间调用 Linux API 函数 `open()` 打开设备文件时，设备驱动的 `open()` 函数最终被调用。驱动程序可以不实现这个函数，在这种情况下，设备的打开操作永远成功。与 `open()` 函数对应的是 `release()` 函数。

`poll()` 函数一般用于询问设备是否可被非阻塞地立即读写。当询问的条件未触发时，用户空间进行 `select()` 和 `poll()` 系统调用将引起进程的阻塞。

`aio_read()` 和 `aio_write()` 函数分别对与文件描述符对应的设备进行异步读、写操作。设备

实现这两个函数后，用户空间可以对该设备文件描述符执行 SYS\_io\_setup、SYS\_io\_submit、SYS\_io\_getevents、SYS\_io\_destroy 等系统调用进行读写。

### 6.1.4 Linux 字符设备驱动的组成

在 Linux 中，字符设备驱动由如下几个部分组成。

#### 1. 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 的注册，而在卸载函数中应实现设备号的释放和 cdev 的注销。

Linux 内核的编码习惯是为设备定义一个设备相关的结构体，该结构体包含设备所涉及的 cdev、私有数据及锁等信息。常见的设备结构体、模块加载和卸载函数形式如代码清单 6.5 所示。

代码清单 6.5 字符设备驱动模块加载与卸载函数模板

---

```

1  /* 设备结构体
2  struct xxx_dev_t {
3      struct cdev cdev;
4      ...
5  } xxx_dev;
6  /* 设备驱动模块加载函数
7  static int __init xxx_init(void)
8  {
9      ...
10     cdev_init(&xxx_dev.cdev, &xxx_fops);           /* 初始化 cdev */
11     xxx_dev.cdev.owner = THIS_MODULE;
12     /* 获取字符设备号 */
13     if (xxx_major) {
14         register_chrdev_region(xxx_dev_no, 1, DEV_NAME);
15     } else {
16         alloc_chrdev_region(&xxx_dev_no, 0, 1, DEV_NAME);
17     }
18
19     ret = cdev_add(&xxx_dev.cdev, xxx_dev_no, 1);    /* 注册设备 */
20     ...
21 }
22 /* 设备驱动模块卸载函数 */
23 static void __exit xxx_exit(void)
24 {
25     unregister_chrdev_region(xxx_dev_no, 1);          /* 释放占用的设备号 */
26     cdev_del(&xxx_dev.cdev);                         /* 注销设备 */
27     ...
28 }
```

---

#### 2. 字符设备驱动的 file\_operations 结构体中的成员函数

file\_operations 结构体中的成员函数是字符设备驱动与内核虚拟文件系统的接口，是用

户空间对 Linux 进行系统调用最终的落实者。大多数字符设备驱动会实现 read()、write() 和 ioctl() 函数，常见的字符设备驱动的这 3 个函数的形式如代码清单 6.6 所示。

代码清单 6.6 字符设备驱动读、写、I/O 控制函数模板

---

```

1  /* 读设备 */
2  ssize_t xxx_read(struct file *filp, char __user *buf, size_t count,
3      loff_t *f_pos)
4  {
5      ...
6      copy_to_user(buf, ..., ...);
7      ...
8  }
9  /* 写设备 */
10 long xxx_write(struct file *filp, const char __user *buf, size_t count,
11     loff_t *f_pos)
12 {
13     ...
14     copy_from_user(..., buf, ...);
15     ...
16 }
17 /* ioctl 函数 */
18 long xxx_ioctl(struct file *filp, unsigned int cmd,
19     unsigned long arg)
20 {
21     ...
22     switch (cmd) {
23         case XXX_CMD1:
24             ...
25             break;
26         case XXX_CMD2:
27             ...
28             break;
29         default:
30             /* 不能支持的命令 */
31             return -ENOTTY;
32     }
33     return 0;
34 }
```

---

设备驱动的读函数中，filp 是文件结构体指针，buf 是用户空间内存的地址，该地址在内核空间不宜直接读写，count 是要读的字节数，f\_pos 是读的位置相对于文件开头的偏移。

设备驱动的写函数中，filp 是文件结构体指针，buf 是用户空间内存的地址，该地址在内核空间不宜直接读写，count 是要写的字节数，f\_pos 是写的位置相对于文件开头的偏移。

由于用户空间不能直接访问内核空间的内存，因此借助了函数 copy\_from\_user() 完成用户空间缓冲区到内核空间的复制，以及 copy\_to\_user() 完成内核空间到用户空间缓冲区的复制，见代码第 6 行和第 14 行。

完成内核空间和用户空间内存复制的 copy\_from\_user() 和 copy\_to\_user() 的原型分别为：

```
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);
```

上述函数均返回不能被复制的字节数，因此，如果完全复制成功，返回值为 0。如果复制失败，则返回负值。

如果要复制的内存是简单类型，如 char、int、long 等，则可以使用简单的 put\_user() 和 get\_user()，如：

```
int val;                                /* 内核空间整型变量
...
get_user(val, (int *) arg);      /* 用户→内核，arg 是用户空间的地址 */
...
put_user(val, (int *) arg);      /* 内核→用户，arg 是用户空间的地址 */
```

读和写函数中的 \_\_user 是一个宏，表明其后的指针指向用户空间，实际上更多地充当了代码自注释的功能。这个宏定义为：

```
#ifdef __CHECKER__
#define __user __attribute__((noderef, address_space(1)))
#else
#define __user
#endif
```

内核空间虽然可以访问用户空间的缓冲区，但是在访问之前，一般需要先检查其合法性，通过 access\_ok(type, addr, size) 进行判断，以确定传入的缓冲区的确属于用户空间，例如：

```
static ssize_t read_port(struct file *file, char __user *buf,
                        size_t count, loff_t *ppos)
{
    unsigned long i = *ppos;
    char __user *tmp = buf;

    if (!access_ok(VIEWER_WRITE, buf, count))
        return -EFAULT;
    while (count-- > 0 && i < 65536) {
        if (__put_user(inb(i), tmp) < 0)
            return -EFAULT;
        i++;
        tmp++;
    }
    *ppos = i;
    return tmp - buf;
}
```

上述代码中引用的 \_\_put\_user() 与前文讲解的 put\_user() 的区别在于前者不进行类似 access\_ok() 的检查，而后者会进行这一检查。在本例中，不使用 put\_user() 而使用 \_\_put\_

user() 的原因是在 \_\_put\_user() 调用之前，已经手动检查了用户空间缓冲区（buf 指向的大小为 count 的内存）的合法性。get\_user() 和 \_\_get\_user() 的区别也相似。

特别要提醒读者注意的是：在内核空间与用户空间的界面处，内核检查用户空间缓冲区的合法性显得尤其必要，Linux 内核的许多安全漏洞都是因为遗忘了这一检查造成的，非法侵入者可以伪造一片内核空间的缓冲区地址传入系统调用的接口，让内核对这个 evil 指针指向的内核空间填充数据。有兴趣的读者可以从 <http://www.cvedetails.com/> 网站查阅 Linux CVE(Common Vulnerabilities and Exposures) 列表。

其实 copy\_from\_user()、copy\_to\_user() 内部也进行了这样的检查：

```
static inline unsigned long __must_check copy_from_user(void *to, const void __user
*from, unsigned long n)
{
    if (access_okVERIFY_READ, from, n))
        n = __copy_from_user(to, from, n);
    else /* security hole - plug it */
        memset(to, 0, n);
    return n;
}

static inline unsigned long __must_check copy_to_user(void __user *to, const void
*from, unsigned long n)
{
    if (access_okVERIFY_WRITE, to, n))
        n = __copy_to_user(to, from, n);
    return n;
}
```

I/O 控制函数的 cmd 参数为事先定义的 I/O 控制命令，而 arg 为对应于该命令的参数。例如对于串行设备，如果 SET\_BAUDRATE 是一道设置波特率的命令，那后面的 arg 就应该是波特率值。

在字符设备驱动中，需要定义一个 file\_operations 的实例，并将具体设备驱动的函数赋值给 file\_operations 的成员，如代码清单 6.7 所示。

代码清单 6.7 字符设备驱动文件操作结构体模板

---

```
1 struct file_operations xxx_fops = {
2     .owner = THIS_MODULE,
3     .read = xxx_read,
4     .write = xxx_write,
5     .unlocked_ioctl= xxx_ioctl,
6     ...
7 };
```

---

上述 xxx\_fops 在代码清单 6.5 第 10 行的 cdev\_init(&xxx\_dev.cdev, &xxx\_fops) 的语句中建立与 cdev 的连接。

图 6.1 所示为字符设备驱动的结构、字符设备驱动与字符设备以及字符设备驱动与用户空间访问该设备的程序之间的关系。

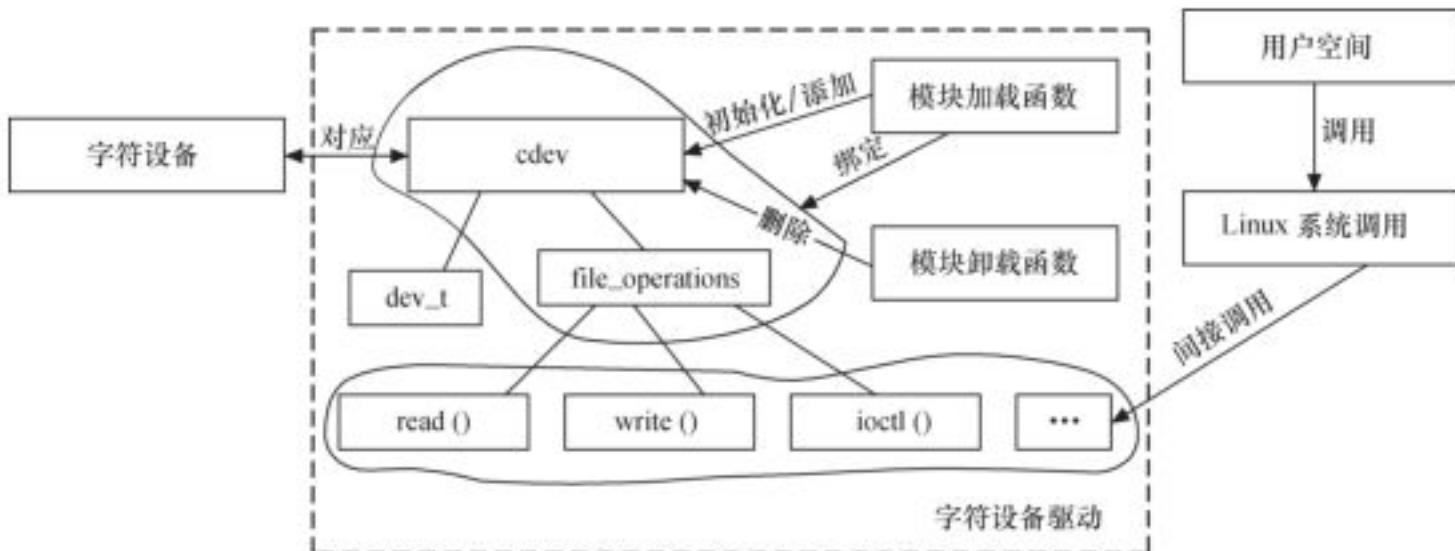


图 6.1 字符设备驱动的结构

## 6.2 globalmem 虚拟设备实例描述

从本章开始，后续的数章都将基于虚拟的 globalmem 设备进行字符设备驱动的讲解。globalmem 意味着“全局内存”，在 globalmem 字符设备驱动中会分配一片大小为 GLOBALMEM\_SIZE (4KB) 的内存空间，并在驱动中提供针对该片内存的读写、控制和定位函数，以供用户空间的进程能通过 Linux 系统调用获取或设置这片内存的内容。

实际上，这个虚拟的 globalmem 设备几乎没有任何实用价值，仅仅是一种为了讲解问题的方便而凭空制造的设备。

本章将给出 globalmem 设备驱动的雏形，而后续章节会在这个雏形的基础上添加并发与同步控制等复杂功能。

## 6.3 globalmem 设备驱动

### 6.3.1 头文件、宏及设备结构体

在 globalmem 字符设备驱动中，应包含它要使用的头文件，并定义 globalmem 设备结构体及相关宏。

代码清单 6.8 globalmem 设备结构体和宏

---

```

1 #include <linux/module.h>
2 #include <linux/fs.h>
3 #include <linux/init.h>
```

```

4 #include <linux/cdev.h>
5 #include <linux/slab.h>
6 #include <linux/uaccess.h>
7
8 #define GLOBALMEM_SIZE 0x1000
9 #define MEM_CLEAR 0x1
10 #define GLOBALMEM_MAJOR 230
11
12 static int globalmem_major = GLOBALMEM_MAJOR;
13 module_param(globalmem_major, int, S_IRUGO);
14
15 struct globalmem_dev {
16     struct cdev cdev;
17     unsigned char mem[GLOBALMEM_SIZE];
18 };
19
20 struct globalmem_dev *globalmem_devp;

```

从第 15 ~ 18 行代码可以看出，定义的 globalmem\_dev 设备结构体包含了对应于 globalmem 字符设备的 cdev、使用的内存 mem[GLOBALMEM\_SIZE]。当然，程序中并不一定要把 mem[GLOBALMEM\_SIZE] 和 cdev 包含在一个设备结构体中，但这样定义的好处在于，它借用了面向对象程序设计中“封装”的思想，体现了一种良好的编程习惯。

### 6.3.2 加载与卸载设备驱动

globalmem 设备驱动的模块加载和卸载函数遵循代码清单 6.5 的类似模板，其实现的工作与代码清单 6.5 完全一致，如代码清单 6.9 所示。

代码清单 6.9 globalmem 设备驱动模块的加载与卸载函数

```

1 static void globalmem_setup_cdev(struct globalmem_dev *dev, int index)
2 {
3     int err, devno = MKDEV(globalmem_major, index);
4
5     cdev_init(&dev->cdev, &globalmem_fops);
6     dev->cdev.owner = THIS_MODULE;
7     err = cdev_add(&dev->cdev, devno, 1);
8     if (err)
9         printk(KERN_NOTICE "Error %d adding globalmem%d", err, index);
10 }
11
12 static int __init globalmem_init(void)
13 {
14     int ret;
15     dev_t devno = MKDEV(globalmem_major, 0);
16
17     if (globalmem_major)
18         ret = register_chrdev_region(devno, 1, "globalmem");
19     else {

```

```

20         ret = alloc_chrdev_region(&devno, 0, 1, "globalmem");
21         globalmem_major = MAJOR(devno);
22     }
23     if (ret < 0)
24         return ret;
25
26     globalmem_devp = kzalloc(sizeof(struct globalmem_dev), GFP_KERNEL);
27     if (!globalmem_devp) {
28         ret = -ENOMEM;
29         goto fail_malloc;
30     }
31
32     globalmem_setup_cdev(globalmem_devp, 0);
33     return 0;
34
35 fail_malloc:
36     unregister_chrdev_region(devno, 1);
37     return ret;
38 }
39 module_init(globalmem_init);

```

第 1~10 行的 globalmem\_setup\_cdev() 函数完成 cdev 的初始化和添加，17 ~ 22 行完成了设备号的申请，第 26 行调用 kzalloc() 申请了一份 globalmem\_dev 结构体的内存并清 0。在 cdev\_init() 函数中，与 globalmem 的 cdev 关联的 file\_operations 结构体如代码清单 6.10 所示。

代码清单 6.10 globalmem 设备驱动的文件操作结构体

```

1 static const struct file_operations globalmem_fops = {
2     .owner = THIS_MODULE,
3     .llseek = globalmem_llseek,
4     .read = globalmem_read,
5     .write = globalmem_write,
6     .unlocked_ioctl = globalmem_ioctl,
7     .open = globalmem_open,
8     .release = globalmem_release,
9 };

```

### 6.3.3 读写函数

globalmem 设备驱动的读写函数主要是让设备结构体的 mem[] 数组与用户空间交互数据，并随着访问的字节数变更更新文件读写偏移位置。读和写函数的实现分别如代码清单 6.11 和 6.12 所示。

代码清单 6.11 globalmem 设备驱动的读函数

```

1 static ssize_t globalmem_read(struct file *filp, char __user *buf, size_t size,
2                               loff_t *ppos)
3 {

```

```

4  unsigned long p = *ppos;
5  unsigned int count = size;
6  int ret = 0;
7  struct globalmem_dev *dev = filp->private_data;
8
9  if (p >= GLOBALMEM_SIZE)
10     return 0;
11  if (count > GLOBALMEM_SIZE - p)
12      count = GLOBALMEM_SIZE - p;
13
14  if (copy_to_user(buf, dev->mem + p, count)) {
15      ret = -EFAULT;
16  } else {
17      *ppos += count;
18      ret = count;
19
20      printk(KERN_INFO "read %u bytes(s) from %lu\n", count, p);
21  }
22
23  return ret;
24 }

```

---

\*ppos 是要读的位置相对于文件开头的偏移，如果该偏移大于或等于 GLOBALMEM\_SIZE，意味着已经到达文件末尾，所以返回 0 (EOF)。

**代码清单 6.12 globalmem 设备驱动的写函数**

```

1  static ssize_t globalmem_write(struct file *filp, const char __user * buf,
2                                size_t size, loff_t * ppos)
3  {
4      unsigned long p = *ppos;
5      unsigned int count = size;
6      int ret = 0;
7      struct globalmem_dev *dev = filp->private_data;
8
9      if (p >= GLOBALMEM_SIZE)
10         return 0;
11      if (count > GLOBALMEM_SIZE - p)
12          count = GLOBALMEM_SIZE - p;
13
14      if (copy_from_user(dev->mem + p, buf, count))
15          ret = -EFAULT;
16      else {
17          *ppos += count;
18          ret = count;
19
20          printk(KERN_INFO "written %u bytes(s) from %lu\n", count, p);
21      }
22
23      return ret;
24 }

```

---

### 6.3.4 seek 函数

seek() 函数对文件定位的起始地址可以是文件开头 (SEEK\_SET, 0)、当前位置 (SEEK\_CUR, 1) 和文件尾 (SEEK\_END, 2)，假设 globalmem 支持从文件开头和当前位置的相对偏移。

在定位的时候，应该检查用户请求的合法性，若不合法，函数返回 -EINVAL，合法时更新文件的当前位置并返回该位置，如代码清单 6.13 所示。

代码清单 6.13 globalmem 设备驱动的 seek() 函数

---

```

1 static loff_t globalmem_llseek(struct file *filp, loff_t offset, int orig)
2 {
3     loff_t ret = 0;
4     switch (orig) {
5         case 0: /* 从文件开头位置 seek */
6             if (offset < 0) {
7                 ret = -EINVAL;
8                 break;
9             }
10            if ((unsigned int)offset > GLOBALMEM_SIZE) {
11                ret = -EINVAL;
12                break;
13            }
14            filp->f_pos = (unsigned int)offset;
15            ret = filp->f_pos;
16            break;
17        case 1: /* 从文件当前位置开始 seek */
18            if ((filp->f_pos + offset) > GLOBALMEM_SIZE) {
19                ret = -EINVAL;
20                break;
21            }
22            if ((filp->f_pos + offset) < 0) {
23                ret = -EINVAL;
24                break;
25            }
26            filp->f_pos += offset;
27            ret = filp->f_pos;
28            break;
29        default:
30            ret = -EINVAL;
31            break;
32    }
33    return ret;
34 }
```

---

### 6.3.5 ioctl 函数

#### 1. globalmem 设备驱动的 ioctl() 函数

globalmem 设备驱动的 ioctl() 函数接受 MEM\_CLEAR 命令，这个命令会将全局内存的有

效数据长度清 0，对于设备不支持的命令，`ioctl()` 函数应该返回 -EINVAL，如代码清单 6.14 所示。

代码清单 6.14 globalmem 设备驱动的 I/O 控制函数

---

```

1 static long globalmem_ioctl(struct file *filp, unsigned int cmd,
2                             unsigned long arg)
3 {
4     struct globalmem_dev *dev = filp->private_data;
5
6     switch (cmd) {
7         case MEM_CLEAR:
8             memset(dev->mem, 0, GLOBALMEM_SIZE);
9             printk(KERN_INFO "globalmem is set to zero\n");
10            break;
11
12        default:
13            return -EINVAL;
14    }
15
16    return 0;
17 }
```

---

在上述程序中，`MEM_CLEAR` 被宏定义为 0x01，实际上这并不是一种值得推荐的方法，简单地对命令定义为 0x0、0x1、0x2 等类似值会导致不同的设备驱动拥有相同的命令号。如果设备 A、B 都支持 0x0、0x1、0x2 这样的命令，就会造成命令码的污染。因此，Linux 内核推荐采用一套统一的 `ioctl()` 命令生成方式。

## 2. `ioctl()` 命令

Linux 建议以如图 6.2 所示的方式定义 `ioctl()` 的命令。

设备类型	序列号	方向	数据尺寸
8位	8位	2位	13/14位

图 6.2 I/O 控制命令的组成

命令码的设备类型字段为一个“幻数”，可以是 0 ~ 0xff 的值，内核中的 `ioctl-number.txt` 给出了一些推荐的和已经被使用的“幻数”，新设备驱动定义“幻数”的时候要避免与其冲突。

命令码的序列号也是 8 位宽。

命令码的方向字段为 2 位，该字段表示数据传送的方向，可能的值是 `_IOC_NONE`（无数据传输）、`_IOC_READ`（读）、`_IOC_WRITE`（写）和 `_IOC_READ|_IOC_WRITE`（双向）。数据传送的方向是从应用程序的角度来看的。

命令码的数据长度字段表示涉及的用户数据的大小，这个成员的宽度依赖于体系结构，通常是 13 或者 14 位。

内核还定义了 `_IO()`、`_IOR()`、`_IOW()` 和 `_IOWR()` 这 4 个宏来辅助生成命令，这 4 个宏

的通用定义如代码清单 6.15 所示。

代码清单 6.15 `_IO()`、`_IOR()`、`_IOW()` 和 `_IOWR()` 宏定义

---

```

1 #define _IO(type,nr)           _IOC(_IOC_NONE,(type),(nr),0)
2 #define _IOR(type,nr,size)    _IOC(_IOC_READ,(type),(nr),\
3                                (_IOC_TYPECHECK(size)))
4 #define _IOW(type,nr,size)    _IOC(_IOC_WRITE,(type),(nr),\
5                                (_IOC_TYPECHECK(size)))
6 #define _IOWR(type,nr,size)   _IOC(_IOC_READ|_IOC_WRITE,(type),(nr), \
7                                (_IOC_TYPECHECK(size)))
8 /* _IO、_IOR 等使用的_IOC 宏 */
9 #define _IOC(dir,type,nr,size) \
10    (((dir) << _IOC_DIRSHIFT) | \
11     ((type) << _IOC_TYPESHIFT) | \
12     ((nr) << _IOC_NRSHIFT) | \
13     ((size) << _IOC_SIZESSHIFT))

```

---

由此可见，这几个宏的作用是根据传入的 type（设备类型字段）、nr（序列号字段）、size（数据长度字段）和宏名隐含的方向字段移位组合生成命令码。

由于 globalmem 的 MEM\_CLEAR 命令不涉及数据传输，所以它可以定义为：

```
#define GLOBALMEM_MAGIC 'g'
#define MEM_CLEAR _IO(GLOBALMEM_MAGIC,0)
```

### 3. 预定义命令

内核中预定义了一些 I/O 控制命令，如果某设备驱动中包含了与预定义命令一样的命令码，这些命令会作为预定义命令被内核处理而不是被设备驱动处理，下面列举一些常用的预定义命令。

**FIOCLEX**：即 File IOctl Close on Exec，对文件设置专用标志，通知内核当 exec() 系统调用发生时自动关闭打开的文件。

**FIONCLEX**：即 File IOctl Not Close on Exec，与 FIOCLEX 标志相反，清除由 FIOCLEX 命令设置的标志。

**FIOQSIZEx**：获得一个文件或者目录的大小，当用于设备文件时，返回一个 ENOTTY 错误。

**FIONBIO**：即 File IOctl Non-Blocking I/O，这个调用修改在 filp->f\_flags 中的 O\_NONBLOCK 标志。

FIOCLEX、FIONCLEX、FIOQSIZEx 和 FIONBIO 这些宏定义在内核的 include/uapi/asm-generic/ioctls.h 文件中。

### 6.3.6 使用文件私有数据

6.3.1 ~ 6.3.5 节给出的代码完整地实现了预期的 globalmem 锚形，代码清单 6.11 的第 7 行，代码清单 6.12 的第 7 行，代码清单 6.14 的第 4 行，都使用了 struct globalmem\_dev \*dev = filp->private\_data 获取 globalmem\_dev 的实例指针。实际上，大多数 Linux 驱动遵循一个“潜规则”，那就是将文件的私有数据 private\_data 指向设备结构体，再用 read()、write()、

ioctl()、llseek() 等函数通过 private\_data 访问设备结构体。私有数据的概念在 Linux 驱动的各个子系统中广泛存在，实际上体现了 Linux 的面向对象的设计思想。对于 globalmem 驱动而言，私有数据的设置是在 globalmem\_open() 中完成的，如代码清单 6.16 所示。

代码清单 6.16 globalmem 设备驱动的 open() 函数

---

```

1 static int globalmem_open(struct inode *inode, struct file *filp)
2 {
3     filp->private_data = globalmem_devp;
4     return 0;
5 }
```

---

为了让读者建立字符设备驱动的全貌视图，代码清单 6.17 列出了完整的使用文件私有数据的 globalmem 的设备驱动，本程序位于本书配套虚拟机代码的 /kernel/drivers/globalmem/ch6 目录下。

代码清单 6.17 使用文件私有数据的 globalmem 的设备驱动

---

```

1 /*
2  * a simple char device driver: globalmem without mutex
3  *
4  * Copyright (C) 2014 Barry Song (baohua@kernel.org)
5  *
6  * Licensed under GPLv2 or later.
7  */
8
9 #include <linux/module.h>
10 #include <linux/fs.h>
11 #include <linux/init.h>
12 #include <linux/cdev.h>
13 #include <linux/slab.h>
14 #include <linux/uaccess.h>
15
16 #define GLOBALMEM_SIZE 0x1000
17 #define MEM_CLEAR 0x1
18 #define GLOBALMEM_MAJOR 230
19
20 static int globalmem_major = GLOBALMEM_MAJOR;
21 module_param(globalmem_major, int, S_IRUGO);
22
23 struct globalmem_dev {
24     struct cdev cdev;
25     unsigned char mem[GLOBALMEM_SIZE];
26 };
27
28 struct globalmem_dev *globalmem_devp;
29
30 static int globalmem_open(struct inode *inode, struct file *filp)
31 {
```

---

```
32     filp->private_data = globalmem_devp;
33     return 0;
34 }
35
36 static int globalmem_release(struct inode *inode, struct file *filp)
37 {
38     return 0;
39 }
40
41 static long globalmem_ioctl(struct file *filp, unsigned int cmd,
42                             unsigned long arg)
43 {
44     struct globalmem_dev *dev = filp->private_data;
45
46     switch (cmd) {
47     case MEM_CLEAR:
48         memset(dev->mem, 0, GLOBALMEM_SIZE);
49         printk(KERN_INFO "globalmem is set to zero\n");
50         break;
51
52     default:
53         return -EINVAL;
54     }
55
56     return 0;
57 }
58
59 static ssize_t globalmem_read(struct file *filp, char __user *buf, size_t size,
60                             loff_t *ppos)
61 {
62     unsigned long p = *ppos;
63     unsigned int count = size;
64     int ret = 0;
65     struct globalmem_dev *dev = filp->private_data;
66
67     if (p >= GLOBALMEM_SIZE)
68         return 0;
69     if (count > GLOBALMEM_SIZE - p)
70         count = GLOBALMEM_SIZE - p;
71
72     if (copy_to_user(buf, dev->mem + p, count)) {
73         ret = -EFAULT;
74     } else {
75         *ppos += count;
76         ret = count;
77
78         printk(KERN_INFO "read %u bytes(s) from %lu\n", count, p);
79     }
80
81     return ret;
82 }
```

```
83
84 static ssize_t globalmem_write(struct file *filp, const char __user * buf,
85                               size_t size, loff_t * ppos)
86 {
87     unsigned long p = *ppos;
88     unsigned int count = size;
89     int ret = 0;
90     struct globalmem_dev *dev = filp->private_data;
91
92     if (p >= GLOBALMEM_SIZE)
93         return 0;
94     if (count > GLOBALMEM_SIZE - p)
95         count = GLOBALMEM_SIZE - p;
96
97     if (copy_from_user(dev->mem + p, buf, count))
98         ret = -EFAULT;
99     else {
100         *ppos += count;
101         ret = count;
102
103         printk(KERN_INFO "written %u bytes(s) from %lu\n", count, p);
104     }
105
106     return ret;
107 }
108
109 static loff_t globalmem_llseek(struct file *filp, loff_t offset, int orig)
110 {
111     loff_t ret = 0;
112     switch (orig) {
113     case 0:
114         if (offset < 0) {
115             ret = -EINVAL;
116             break;
117         }
118         if ((unsigned int)offset > GLOBALMEM_SIZE) {
119             ret = -EINVAL;
120             break;
121         }
122         filp->f_pos = (unsigned int)offset;
123         ret = filp->f_pos;
124         break;
125     case 1:
126         if ((filp->f_pos + offset) > GLOBALMEM_SIZE) {
127             ret = -EINVAL;
128             break;
129         }
130         if ((filp->f_pos + offset) < 0) {
131             ret = -EINVAL;
132             break;
133         }
```

```
134         filp->f_pos += offset;
135         ret = filp->f_pos;
136         break;
137     default:
138         ret = -EINVAL;
139         break;
140     }
141     return ret;
142 }
143
144 static const struct file_operations globalmem_fops = {
145     .owner = THIS_MODULE,
146     .llseek = globalmem_llseek,
147     .read = globalmem_read,
148     .write = globalmem_write,
149     .unlocked_ioctl = globalmem_ioctl,
150     .open = globalmem_open,
151     .release = globalmem_release,
152 };
153
154 static void globalmem_setup_cdev(struct globalmem_dev *dev, int index)
155 {
156     int err, devno = MKDEV(globalmem_major, index);
157
158     cdev_init(&dev->cdev, &globalmem_fops);
159     dev->cdev.owner = THIS_MODULE;
160     err = cdev_add(&dev->cdev, devno, 1);
161     if (err)
162         printk(KERN_NOTICE "Error %d adding globalmem%d", err, index);
163 }
164
165 static int __init globalmem_init(void)
166 {
167     int ret;
168     dev_t devno = MKDEV(globalmem_major, 0);
169
170     if (globalmem_major)
171         ret = register_chrdev_region(devno, 1, "globalmem");
172     else {
173         ret = alloc_chrdev_region(&devno, 0, 1, "globalmem");
174         globalmem_major = MAJOR(devno);
175     }
176     if (ret < 0)
177         return ret;
178
179     globalmem_devp = kzalloc(sizeof(struct globalmem_dev), GFP_KERNEL);
180     if (!globalmem_devp) {
181         ret = -ENOMEM;
182         goto fail_malloc;
183     }
184 }
```

```

185     globalmem_setup_cdev(globalmem_devp, 0);
186     return 0;
187
188 fail_malloc:
189     unregister_chrdev_region(devno, 1);
190     return ret;
191 }
192 module_init(globalmem_init);
193
194 static void __exit globalmem_exit(void)
195 {
196     cdev_del(&globalmem_devp->cdev);
197     kfree(globalmem_devp);
198     unregister_chrdev_region(MKDEV(globalmem_major, 0), 1);
199 }
200 module_exit(globalmem_exit);
201
202 MODULE_AUTHOR("Barry Song <baohua@kernel.org>");
203 MODULE_LICENSE("GPL v2");

```

---



读者朋友们，这个时候，请您翻到本书的第1章，再次阅读代码清单1.4，即Linux下LED的设备驱动，是否会豁然开朗？

如果globalmem不只包括一个设备，而是同时包括两个或两个以上的设备，采用private\_data的优势就会集中显现出来。在不对代码清单6.17中的globalmem\_read()、globalmem\_write()、globalmem\_ioctl()等重要函数及globalmem\_fops结构体等数据结构进行任何修改的前提下，只是简单地修改globalmem\_init()、globalmem\_exit()和globalmem\_open()，就可以轻松地让globalmem驱动中包含N个同样的设备（次设备号分为0~N），如代码清单6.18列出了支持多个实例的globalmem和支持单实例的globalmem驱动的差异部分。

代码清单6.18 支持N个globalmem设备的globalmem驱动

```

1 #define GLOBALMEM_SIZE    0x1000
2 #define MEM_CLEAR 0x1
3 #define GLOBALMEM_MAJOR 230
4 #define DEVICE_NUM    10
5
6 static int globalmem_open(struct inode *inode, struct file *filp)
7 {
8     struct globalmem_dev *dev = container_of(inode->i_cdev,
9                     struct globalmem_dev, cdev);
10    filp->private_data = dev;
11    return 0;
12 }
13
14 static int __init globalmem_init(void)
15 {

```

```

16     int ret;
17     int i;
18     dev_t devno = MKDEV(globalmem_major, 0);
19
20     if (globalmem_major)
21         ret = register_chrdev_region(devno, DEVICE_NUM, "globalmem");
22     else {
23         ret = alloc_chrdev_region(&devno, 0, DEVICE_NUM, "globalmem");
24         globalmem_major = MAJOR(devno);
25     }
26     if (ret < 0)
27         return ret;
28
29     globalmem_devp = kzalloc(sizeof(struct globalmem_dev) * DEVICE_NUM, GFP_KERNEL);
30     if (!globalmem_devp) {
31         ret = -ENOMEM;
32         goto fail_malloc;
33     }
34
35     for (i = 0; i < DEVICE_NUM; i++)
36         globalmem_setup_cdev(globalmem_devp + i, i);
37
38     return 0;
39
40 fail_malloc:
41     unregister_chrdev_region(devno, DEVICE_NUM);
42     return ret;
43 }
44 module_init(globalmem_init);
45
46 static void __exit globalmem_exit(void)
47 {
48     int i;
49     for (i = 0; i < DEVICE_NUM; i++)
50         cdev_del(&(globalmem_devp + i)->cdev);
51     kfree(globalmem_devp);
52     unregister_chrdev_region(MKDEV(globalmem_major, 0), DEVICE_NUM);
53 }
54 module_exit(globalmem_exit);

```

---

代码清单 6.18 第 8 行调用的 `container_of()` 的作用是通过结构体成员的指针找到对应结构体的指针，这个技巧在 Linux 内核编程中十分常用。在 `container_of(inode->i_cdev, struct globalmem_dev, cdev)` 语句中，传给 `container_of()` 的第 1 个参数是结构体成员的指针，第 2 个参数为整个结构体的类型，第 3 个参数为传入的第 1 个参数即结构体成员的类型，`container_of()` 返回值为整个结构体的指针。

从代码清单 6.18 可以看出，我们仅仅进行了极其少量的更改就使得 `globalmem` 驱动支持多个实例，这一点可以看出私有数据的魔力。完整的代码位于 `kernel/drivers/globalmem/ch6/multi_globalmem.c` 下。高亮 `globalmem.c` 和 `multi_globalmem.c`(以“-”和“+”开头的代码)的区别如下：

```
@@ -29,7 +30,9 @@ struct globalmem_dev *globalmem_devp;

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

@@ -165,37 +168,42 @@ static void globalmem_setup_cdev(struct globalmem_dev *dev,
     int index)
 static int __init globalmem_init(void)
 {
     int ret;
+    int i;
     dev_t devno = MKDEV(globalmem_major, 0);

     if (globalmem_major)
-        ret = register_chrdev_region(devno, 1, "globalmem");
+        ret = register_chrdev_region(devno, DEVICE_NUM, "globalmem");
     else {
-        ret = alloc_chrdev_region(&devno, 0, 1, "globalmem");
+        ret = alloc_chrdev_region(&devno, 0, DEVICE_NUM, "globalmem");
         globalmem_major = MAJOR(devno);
     }
     if (ret < 0)
         return ret;

-    globalmem_devp = kzalloc(sizeof(struct globalmem_dev), GFP_KERNEL);
+    globalmem_devp = kzalloc(sizeof(struct globalmem_dev) * DEVICE_NUM, GFP_KERNEL);
     if (!globalmem_devp) {
         ret = -ENOMEM;
         goto fail_malloc;
     }

-    globalmem_setup_cdev(globalmem_devp, 0);
+    for (i = 0; i < DEVICE_NUM; i++)
+        globalmem_setup_cdev(globalmem_devp + i, i);
+
     return 0;

- fail_malloc:
-     unregister_chrdev_region(devno, 1);
+fail_malloc:
+    unregister_chrdev_region(devno, DEVICE_NUM);
     return ret;
 }

module_init(globalmem_init);

static void __exit globalmem_exit(void)
{
```

```

-    cdev_del(&globalmem_devp->cdev);
+    int i;
+    for (i = 0; i < DEVICE_NUM; i++)
+        cdev_del(&(globalmem_devp + i)->cdev);
    kfree(globalmem_devp);
-    unregister_chrdev_region(MKDEV(globalmem_major, 0), 1);
+    unregister_chrdev_region(MKDEV(globalmem_major, 0), DEVICE_NUM);
}
module_exit(globalmem_exit);

```

## 6.4 globalmem 驱动在用户空间中的验证

在 globalmem 的源代码目录通过“make”命令编译 globalmem 的驱动，得到 globalmem.ko 文件。运行

```
baohua@baohua-VirtualBox:~/develop/training/kernel/drivers/globalmem/ch6$ sudo
insmod globalmem.ko
```

命令加载模块，通过“lsmod”命令，发现 globalmem 模块已被加载。再通过“cat /proc/devices”命令查看，发现多出了主设备号为 230 的“globalmem”字符设备驱动：

```
$ 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
 21 sg
 29 fb
116 alsa
128 ptm
136 pts
180 usb
189 usb_device
202 cpu/msr
203 cpu/cpuid
226 drm
230 globalmem
249 hidraw
250 usbmon
251 bsg
252 ptp
253 pps
254 rtc
```

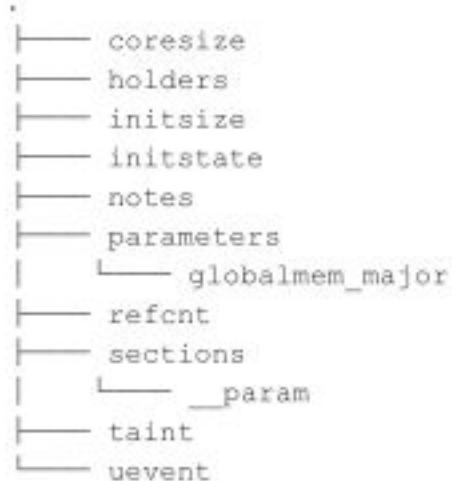
接下来，通过命令

```
#mknod /dev/globalmem c 230 0
```

创建 “/dev/globalmem” 设备节点，并通过 “echo ‘hello world’ > /dev/globalmem” 命令和 “cat /dev/globalmem” 命令分别验证设备的写和读，结果证明 “hello world” 字符串被正确地写入了 globalmem 字符设备：

```
# echo "hello world" > /dev/globalmem
# cat /dev/globalmem
hello world
```

如果启用了 sysfs 文件系统，将发现多出了 /sys/module/globalmem 目录，该目录下的树形结构为：



refcnt 记录了 globalmem 模块的引用计数，sections 下包含的几个文件则给出了 globalmem 所包含的 BSS、数据段和代码段等的地址及其他信息。

对于代码清单 6.18 给出的支持  $N$  个 globalmem 设备的驱动，在加载模块后需创建多个设备节点，如运行 mknod /dev/globalmem0 c 230 0 使得 /dev/globalmem0 对应主设备号为 globalmem\_major、次设备号为 0 的设备，运行 mknod /dev/globalmem1 c 230 1 使得 /dev/globalmem1 对应主设备号为 globalmem\_major、次设备号为 1 的设备。分别读写 /dev/globalmem0 和 /dev/globalmem1，发现都读写到了正确的对应的设备。

## 6.5 总结

字符设备是 3 大类设备（字符设备、块设备和网络设备）中的一类，其驱动程序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\_operations 结构体中的操作函数，实现 file\_operations 结构体中的 read()、write() 和 ioctl() 等函数是驱动设计的主体工作。