LINUX

业界权威机构和专家强力推荐
多年培训、研发经验的总结

设备驱动开发详解

华清远见嵌入式培训中心 宋宝华 编著

Linux

附源代码光盘

人民邮电出版社
POSTS & TELECOM PRESS

第 6 章 字符设备驱动

在 Linux 设备驱动中，字符设备驱动较为基础。本章主要讲解 Linux 字符设备驱动程序的结构及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 节讲解了本章及后续各章所基于的 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 设备的驱动编写方法，对读写函数、seek()函数和 I/O 控制函数等进行了重点分析。该节的最后改造 globalmem 的驱动程序以利用文件私有数据。

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 2.6 内核中使用 cdev 结构体描述字符设备，cdev 结构体的定义如代码清单 6.1 所示。

代码清单 6.1 cdev 结构体

```
1  struct cdev
2  {
3   struct kobject kobj; /* 内嵌的 kobject 对象 */
4   struct module *owner;    /*所属模块*/
5   struct file_operations *ops;  /*文件操作结构体*/
6   struct list_head list;
7   dev_t dev;          /*设备号*/
8   unsigned int count;
9  };
```

cdev 结构体的 dev_t 成员定义了设备号，为 32 位，其中高 12 位为主设备号，低 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 2.6 内核提供了一组函数用于操作 cdev 结构体，如下所示：

```
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   cdev->kobj.ktype = &ktype_cdev_default;
6   kobject_init(&cdev->kobj);
7   cdev->ops = fops;    /*将传入的文件操作结构体指针赋值给 cdev 的 ops*/
8  }
```

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=kmalloc(sizeof(struct cdev),GFP_KERNEL); /*分配 cdev
的内存*/
4   if (p) {
5       memset(p, 0, sizeof(struct cdev));
6       p->kobj.ktype = &ktype_cdev_dynamic;
7       INIT_LIST_HEAD(&p->list);
8       kobject_init(&p->kobj);
9   }
10  return p;
11 }
```

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，这两个函数的原型如下：

```
    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 *name);
```

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()与 register_chrdev_region()对比的优点在于它会自动避开设备号重复的冲突。

相反地，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，这个函数的原型如下：

```
    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

```
1  struct file_operations
2  {
3    struct module *owner;
4      // 拥有该结构的模块的指针，一般为 THIS_MODULES
5    loff_t(*llseek)(struct file *, loff_t, int);
6      // 用来修改文件当前的读写位置
7    ssize_t(*read)(struct file *, char __user *, size_t, loff_t*);
8      // 从设备中同步读取数据
9    ssize_t(*aio_read)(struct kiocb *, char __user *, size_t, loff_t);
10     // 初始化一个异步的读取操作
```

```
    11    ssize_t(*write)(struct file *, const char __user *, size_t,
loff_t*);
    12     // 向设备发送数据
    13   ssize_t(*aio_write)(struct kiocb *, const char __user *, size_t,
loff_t);
    14     // 初始化一个异步的写入操作
    15   int(*readdir)(struct file *, void *, filldir_t);
    16     // 仅用于读取目录，对于设备文件，该字段为 NULL
    17   unsigned int(*poll)(struct file *, struct poll_table_struct*);
    18     // 轮询函数，判断目前是否可以进行非阻塞的读取或写入
    19   int(*ioctl)(struct inode *, struct file *, unsigned int, unsigned
long);
    20     // 执行设备 I/O 控制命令
    21   long(*unlocked_ioctl)(struct file *, unsigned int, unsigned long);
    22     // 不使用 BLK 文件系统，将使用此种函数指针代替 ioctl
    23   long(*compat_ioctl)(struct file *, unsigned int, unsigned long);
    24     // 在 64 位系统上，32 位的 ioctl 调用将使用此函数指针代替
    25   int(*mmap)(struct file *, struct vm_area_struct*);
    26     // 用于请求将设备内存映射到进程地址空间
    27   int(*open)(struct inode *, struct file*);
    28     // 打开
    29   int(*flush)(struct file*);
    30   int(*release)(struct inode *, struct file*);
    31     // 关闭
    32   int(*synch)(struct file *, struct dentry *, int datasync);
    33     // 刷新待处理的数据
    34   int(*aio_fsync)(struct kiocb *, int datasync);
    35     // 异步 fsync
    36   int(*fasync)(int, struct file *, int);
    37     // 通知设备 FASYNC 标志发生变化
    38   int(*lock)(struct file *, int, struct file_lock*);
    39   ssize_t(*readv)(struct file *, const struct iovec *, unsigned long,
loff_t*);
    40   ssize_t(*writev)(struct file *, const struct iovec *, unsigned long,
loff_t*);
    41     // readv 和 writev: 分散/聚集型的读写操作
    42   ssize_t(*sendfile)(struct file *, loff_t *, size_t, read_actor_t,
void*);
    43     // 通常为 NULL
    44    ssize_t(*sendpage)(struct file *, struct page *, int, size_t,
loff_t *, int);
    45     // 通常为 NULL
    46    unsigned long(*get_unmapped_area)(struct file *,unsigned long,
unsigned long,
    47     unsigned long, unsigned long);
    48     // 在进程地址空间找到一个将底层设备中的内存段映射的位置
    49   int(*check_flags)(int);
    50     // 允许模块检查传递给 fcntl(F_SETEL...)调用的标志
    51   int(*dir_notify)(struct file *filp, unsigned long arg);
    52     // 仅对文件系统有效，驱动程序不必实现
    53   int(*flock)(struct file *, int, struct file_lock*);
    54 };
```

下面对 file_operations 结构体中的主要成员进行讲解。

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。

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。

readdir()函数仅用于目录，设备节点不需要实现它。

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。内核本身识别部分控制命令，而不必调用设备驱动中的 ioctl()。如果设备不提供 ioctl()函数，对于内核不能识别的命令，用户进行 ioctl()系统调用时将获得-EINVAL 返回值。

mmap()函数将设备内存映射到进程内存中，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。

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调用 aio_read()、aio_write()等系统调用进行读写。

6.1.4 Linux 字符设备驱动的组成

在 Linux 系统中，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。

工程师通常习惯将设备定义为一个设备相关的结构体，其包含该设备所涉及的 cdev、私有数据及信号量等信息。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  {
4    struct cdev cdev;
5    ...
6  } xxx_dev;
7  //设备驱动模块加载函数
8  static int __init xxx_init(void)
9  {
10   ...
11   cdev_init(&xxx_dev.cdev, &xxx_fops); //初始化 cdev
12   xxx_dev.cdev.owner = THIS_MODULE;
13   //获取字符设备号
14   if (xxx_major)
15   {
16     register_chrdev_region(xxx_dev_no, 1, DEV_NAME);
```

```
    17   }
    18   else
    19   {
    20     alloc_chrdev_region(&xxx_dev_no, 0, 1, DEV_NAME);
    21   }
    22
    23   ret = cdev_add(&xxx_dev.cdev, xxx_dev_no, 1); //注册设备
    24   ...
    25 }
    26 /*设备驱动模块卸载函数*/
    27 static void __exit xxx_exit(void)
    28 {
    29   unregister_chrdev_region(xxx_dev_no, 1); //释放占用的设备号
    30   cdev_del(&xxx_dev.cdev); //注销设备
    31   ...
    32 }
```

2．字符设备驱动的 file_operations 结构体中成员函数

file_operations 结构体中成员函数是字符设备驱动与内核的接口，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  ssize_t 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  int xxx_ioctl(struct inode *inode, struct file *filp, unsigned int
cmd,
    19    unsigned long arg)
    20  {
    21    ...
    22    switch (cmd)
    23    {
    24      case XXX_CMD1:
    25        ...
    26        break;
    27      case XXX_CMD2:
    28        ...
    29        break;
    30      default:
```

```
    31        /* 不能支持的命令 */
    32        return  - ENOTTY;
    33     }
    34     return 0;
    35  }
```

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能直接读写，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能直接读写，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。

由于内核空间与用户空间的内存不能直接互访，因此借助函数 copy_from_user()完成用户空间到内核空间的复制，函数 copy_to_user()完成内核空间到用户空间的复制。

copy_from_user()和 copy_to_user()的原型如下所示：

```
    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()，如下所示：

```
    int val; //内核空间整型变量
    ...
    get_user(val, (int *) arg); //用户空间到内核空间，arg 是用户空间的地址
    ...
    put_user(val, (int *) arg); //内核空间到用户空间，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，这个宏定义如下：

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

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 是一个设置波特率的命令，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 {
3   .owner = THIS_MODULE,
4   .read = xxx_read,
5   .write = xxx_write,
6   .ioctl = xxx_ioctl,
7   ...
8 };
```

上述 xxx_fops 在代码清单 6.5 第 11 行的 cdev_init（&xxx_dev.cdev, &xxx_fops)的语句中被建立与 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。

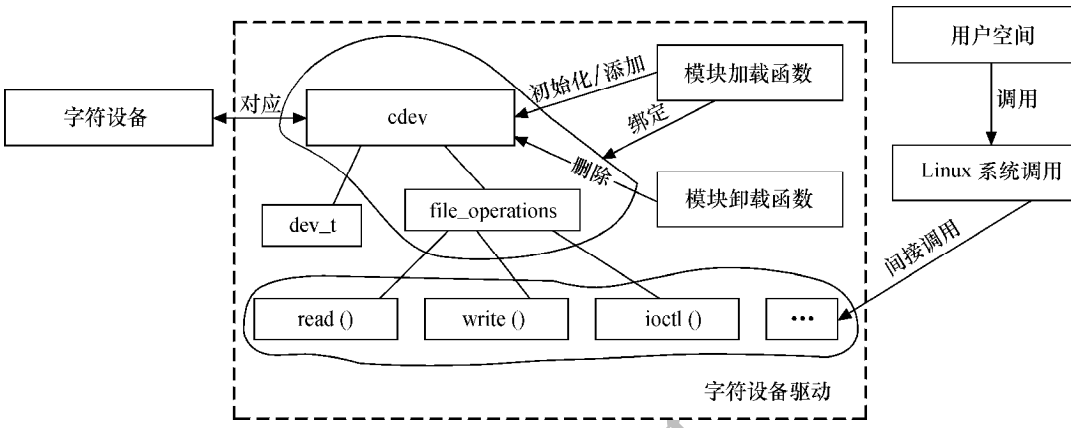

图 6.1　字符设备驱动的结构

6.2 globalmem 虚拟设备实例描述

从本章开始，后续的几章都将基于虚拟的 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 系统调用访问这片内存。

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/types.h>
3  #include <linux/fs.h>
4  #include <linux/errno.h>
5  #include <linux/mm.h>
6  #include <linux/sched.h>
7  #include <linux/init.h>
8  #include <linux/cdev.h>
9  #include <asm/io.h>
10 #include <asm/system.h>
11 #include <asm/uaccess.h>
12
13 #define GLOBALMEM_SIZE  0x1000  /*全局内存大小: 4KB*/
14 #define MEM_CLEAR 0x1  /*清零全局内存*/
15 #define GLOBALMEM_MAJOR 254    /*预设的 globalmem 的主设备号*/
16
17 static globalmem_major = GLOBALMEM_MAJOR;
18 /*globalmem 设备结构体*/
19 struct globalmem_dev
20 {
21   struct cdev cdev; /*cdev 结构体*/
22   unsigned char mem[GLOBALMEM_SIZE]; /*全局内存*/
23 };
24
25 struct globalmem_dev dev; /*设备结构体实例*/
```

从第 19～23 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[GLOBA- LMEM_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，其实现的功能与代码清单 6.5 完全一致，如代码清单 6.9 所示。

代码清单 6.9 globalmem 设备驱动模块加载与卸载函数

```
1  /*globalmem 设备驱动模块加载函数*/
2  int globalmem_init(void)
3  {
4    int result;
5    dev_t devno = MKDEV(globalmem_major, 0);
6
7    /* 申请字符设备驱动区域*/
8    if (globalmem_major)
9      result = register_chrdev_region(devno, 1, "globalmem");
```

```
  10   else
  11    /* 动态获得主设备号 */
  12   {
  13     result = alloc_chrdev_region(&devno, 0, 1, "globalmem");
  14     globalmem_major = MAJOR(devno);
  15   }
  16   if (result < 0)
  17     return result;
  18
  19   globalmem_setup_cdev();
  20   return 0;
  21 }
  22
  23 /*globalmem 设备驱动模块卸载函数*/
  24 void globalmem_exit(void)
  25 {
  26   cdev_del(&dev.cdev); /*删除 cdev 结构*/
  27   unregister_chrdev_region(MKDEV(globalmem_major, 0), 1);/*注销设备
区域*/
  28 }
```

第 19 行调用的 globalmem_setup_cdev()函数完成 cdev 的初始化和添加，如代码清单 6.10 所示。

代码清单 6.10　初始化并添加 cdev 结构体

```
  1  /*初始化并添加 cdev 结构体*/
  2  static void globalmem_setup_cdev()
  3  {
  4    int err, devno = MKDEV(globalmem_major, 0);
  5
  6    cdev_init(&dev.cdev, &globalmem_fops);
  7    dev.cdev.owner = THIS_MODULE;
  8    dev.cdev.ops = &globalmem_fops;
  9    err = cdev_add(&dev.cdev, devno, 1);
  10   if (err)
  11     printk(KERN_NOTICE "Error %d adding globalmem", err);
  12 }
```

在 cdev_init()函数中，与 globalmem 的 cdev 关联的 file_operations 结构体如代码清单 6.11 所示。

代码清单 6.11　globalmem 设备驱动文件操作结构体

```
  1  static const struct file_operations globalmem_fops =
  2  {
  3    .owner = THIS_MODULE,
  4    .llseek = globalmem_llseek,
  5    .read = globalmem_read,
  6    .write = globalmem_write,
  7    .ioctl = globalmem_ioctl,
  8  };
```

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返回用户的文件读写偏移位置。读和写函数的实现分别如代码清单 6.12 和代码清单 6.13 所示。

代码清单 6.12　globalmem 设备驱动的读函数

```
1  static ssize_t globalmem_read(struct file *filp, char __user *buf,
size_t count,
2    loff_t *ppos)
3  {
4    unsigned long p =  *ppos;
5    int ret = 0;
6
7    /*分析和获取有效的读长度*/
8    if (p >= GLOBALMEM_SIZE) //要读的偏移位置越界
9      return count ?  - ENXIO: 0;
10   if (count > GLOBALMEM_SIZE - p)//要读的字节数太大
11     count = GLOBALMEM_SIZE - p;
12
13   /*内核空间→用户空间*/
14   if (copy_to_user(buf, (void*)(dev.mem + p), count))
15   {
16     ret =  - EFAULT;
17   }
18   else
19   {
20     *ppos += count;
21     ret = count;
22
23     printk(KERN_INFO "read %d bytes(s) from %d\n", count, p);
24   }
25
26   return ret;
27 }
```

代码清单 6.13　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 count, loff_t *ppos)
3  {
4    unsigned long p =  *ppos;
5    int ret = 0;
6
```

```
    /*分析和获取有效的写长度*/
    if (p >= GLOBALMEM_SIZE)   //要写的偏移位置越界
      return count ?  - ENXIO: 0;
    if (count > GLOBALMEM_SIZE - p) //要写的字节数太多
      count = GLOBALMEM_SIZE - p;

    /*用户空间→内核空间*/
    if (copy_from_user(dev->mem + p, buf, count))
      ret =  - EFAULT;
    else
    {
      *ppos += count;
      ret = count;

      printk(KERN_INFO "written %d bytes(s) from %d\n", count, p);
    }

    return ret;
}
```

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），globalmem 支持从文件开头和当前位置相对偏移。

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，合法时返回文件的当前位置，如代码清单 6.14 所示。

代码清单 6.14　globalmem 设备驱动的 seek()函数

```
static loff_t globalmem_llseek(struct file *filp, loff_t offset, int
orig)
{
  loff_t ret;
  switch (orig)
  {
    case 0:   /*从文件开头开始偏移*/
      if (offset < 0)
      {
        ret =  - EINVAL;
        break;
      }
      if ((unsigned int)offset > GLOBALMEM_SIZE) //偏移越界
```

```
    {
      ret =  - EINVAL;
      break;
    }
    filp->f_pos = (unsigned int)offset;
    ret = filp->f_pos;
    break;
  case 1:    /*从当前位置开始偏移*/
    if ((filp->f_pos + offset) > GLOBALMEM_SIZE) //偏移越界
    {
      ret =  - EINVAL;
      break;
    }
    if ((filp->f_pos + offset) < 0)
    {
      ret =  - EINVAL;
      break;
    }
    filp->f_pos += offset;
    ret = filp->f_pos;
    break;
  default:
    ret =  - EINVAL;
  }
  return ret;
}
```

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零，对于设备不支持的命令，ioctl()函数应该返回-EINVAL，如代码清单 6.15 所示。

代码清单 6.15　globalmem 设备驱动的 ioctl()函数

```
static int globalmem_ioctl(struct inode *inodep, struct file *filp,
unsigned
  int cmd, unsigned long arg)
{
  switch (cmd)
  {
    case MEM_CLEAR:
      //清除全局内存
      memset(dev->mem, 0, GLOBALMEM_SIZE);
```

```
    9       printk(KERN_INFO "globalmem is set to zero\n");
    10      break;
    11
    12    default:
    13      return  - EINVAL; //其他不支持的命令
    14   }
    15   return 0;
    16 }
```

在上述程序中，MEM_CLEAR 被宏定义为 0x01，实际上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，假设用户本身希望给 A 发 0x1 命令，可是不经意间发给了 B，这个时候 B 因为支持该命令，它就会执行该命令。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 系统建议以如图 6.2 所示的方式定义 ioctl()的命令码。

设备类型	序列号	方向	数据尺寸
8bit	8bit	2bit	13/14bit

图 6.2　ioctl()命令码的组成

命令码的设备类型字段为一个“幻数”，可以是 0～0xff 之间的值，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 位或者 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 6.16 所示。

代码清单 6.16　_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_SIZESHIFT))
```

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）和 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，因此它可以定义如下：

```
#define GLOBALMEM_MAGIC …
#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，这些命令会被当作预定义命令被内核处理而不是被设备驱动处理，预定义命令有如下 4 种。

- 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。
- FIOQSIZE：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、FIOQSIZE 和 FIONBIO 这些宏的定义如下：

```
#define FIONCLEX        0x5450
#define FIOCLEX     0x5451
#define FIOQSIZE        0x5460
#define FIONBIO     0x5421
```

由以上定义可以看出，FIOCLEX、FIONCLEX、FIOQSIZE 和 FIONBIO 的幻数为“T”。

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 雏形，在其代码中，为 globalmem 设备结构体 globalmem_dev 定义了全局实例 dev(见代码清单 6.8 第 25 行)，而 globalmem 的驱动中 read()、write()、ioctl()、llseek()函数都针对 dev 进行操作。

实际上，大多数 Linux 驱动工程师都将文件的私有数据 private_data 指向设备结构体，read()、write()、ioctl()、llseek()等函数通过 private_data 访问设备结构体。

下面对各函数进行少量的修改，使读者了解字符设备驱动的全貌，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。

代码清单 6.17　使用文件私有数据的 globalmem 的设备驱动

```
1   #include <linux/module.h>
2   #include <linux/types.h>
3   #include <linux/fs.h>
4   #include <linux/errno.h>
5   #include <linux/mm.h>
6   #include <linux/sched.h>
```

```
7  #include <linux/init.h>
8  #include <linux/cdev.h>
9  #include <asm/io.h>
10 #include <asm/system.h>
11 #include <asm/uaccess.h>
12
13 #define GLOBALMEM_SIZE  0x1000  /*全局内存最大 4KB*/
14 #define MEM_CLEAR 0x1  /*清零全局内存*/
15 #define GLOBALMEM_MAJOR 254   /*预设的 globalmem 的主设备号*/
16
17 static globalmem_major = GLOBALMEM_MAJOR;
18 /*globalmem 设备结构体*/
19 struct globalmem_dev
20 {
21   struct cdev cdev; /*cdev 结构体*/
22   unsigned char mem[GLOBALMEM_SIZE]; /*全局内存*/
23 };
24
25 struct globalmem_dev *globalmem_devp; /*设备结构体指针*/
26 /*文件打开函数*/
27 int globalmem_open(struct inode *inode, struct file *filp)
28 {
29   /*将设备结构体指针赋值给文件私有数据指针*/
30   filp->private_data = globalmem_devp;
31   return 0;
32 }
33 /*文件释放函数*/
34 int globalmem_release(struct inode *inode, struct file *filp)
35 {
36   return 0;
37 }
38
39 /* ioctl 设备控制函数 */
40 static int globalmem_ioctl(struct inode *inodep, struct file *filp,
unsigned
41   int cmd, unsigned long arg)
42 {
43   struct globalmem_dev *dev = filp->private_data;/*获得设备结构体指
针*/
44
```

```
45   switch (cmd)
46   {
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   return 0;
56 }
57
58 /*读函数*/
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; /*获得设备结构体
指针*/
66
67   /*分析和获取有效的写长度*/
68   if (p >= GLOBALMEM_SIZE)
69     return count ?  - ENXIO: 0;
70   if (count > GLOBALMEM_SIZE - p)
71     count = GLOBALMEM_SIZE - p;
72
73   /*内核空间→用户空间*/
74   if (copy_to_user(buf, (void*)(dev->mem + p), count))
75   {
76     ret =  - EFAULT;
77   }
78   else
79   {
80     *ppos += count;
81     ret = count;
82
```

```
    printk(KERN_INFO "read %d bytes(s) from %d\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; /*获得设备结构体
指针*/

  /*分析和获取有效的写长度*/
  if (p >= GLOBALMEM_SIZE)
    return count ?  - ENXIO: 0;
  if (count > GLOBALMEM_SIZE - p)
    count = GLOBALMEM_SIZE - p;

  /*用户空间→内核空间*/
  if (copy_from_user(dev->mem + p, buf, count))
    ret =  - EFAULT;
  else
  {
    *ppos += count;
    ret = count;

    printk(KERN_INFO "written %d bytes(s) from %d\n", count, p);
  }

  return ret;
}

/* seek 文件定位函数 */
static loff_t globalmem_llseek(struct file *filp, loff_t offset,
int orig)
```

```
120 {
121   loff_t ret = 0;
122   switch (orig)
123   {
124     case 0:   /*相对文件开始位置偏移*/
125       if (offset < 0)
126       {
127         ret =  - EINVAL;
128         break;
129       }
130       if ((unsigned int)offset > GLOBALMEM_SIZE)
131       {
132         ret =  - EINVAL;
133         break;
134       }
135       filp->f_pos = (unsigned int)offset;
136       ret = filp->f_pos;
137       break;
138     case 1:   /*相对文件当前位置偏移*/
139       if ((filp->f_pos + offset) > GLOBALMEM_SIZE)
140       {
141         ret =  - EINVAL;
142         break;
143       }
144       if ((filp->f_pos + offset) < 0)
145       {
146         ret =  - EINVAL;
147         break;
148       }
149       filp->f_pos += offset;
150       ret = filp->f_pos;
151       break;
152     default:
153       ret =  - EINVAL;
154       break;
155   }
156   return ret;
157 }
158
159 /*文件操作结构体*/
```

```
160 static const struct file_operations globalmem_fops =
161 {
162   .owner = THIS_MODULE,
163   .llseek = globalmem_llseek,
164   .read = globalmem_read,
165   .write = globalmem_write,
166   .ioctl = globalmem_ioctl,
167   .open = globalmem_open,
168   .release = globalmem_release,
169 };
170
171 /*初始化并注册 cdev*/
172 static void globalmem_setup_cdev(struct globalmem_dev *dev, int
index)
173 {
174   int err, devno = MKDEV(globalmem_major, index);
175
176   cdev_init(&dev->cdev, &globalmem_fops);
177   dev->cdev.owner = THIS_MODULE;
178   dev->cdev.ops = &globalmem_fops;
179   err = cdev_add(&dev->cdev, devno, 1);
180   if (err)
181     printk(KERN_NOTICE "Error %d adding LED%d", err, index);
182 }
183
184 /*设备驱动模块加载函数*/
185 int globalmem_init(void)
186 {
187   int result;
188   dev_t devno = MKDEV(globalmem_major, 0);
189
190   /* 申请设备号*/
191   if (globalmem_major)
192     result = register_chrdev_region(devno, 1, "globalmem");
193   else  /* 动态申请设备号 */
194   {
195     result = alloc_chrdev_region(&devno, 0, 1, "globalmem");
196     globalmem_major = MAJOR(devno);
197   }
198   if (result < 0)
```

```
    199    return result;
    200
    201  /* 动态申请设备结构体的内存*/
    202       globalmem_devp   =   kmalloc(sizeof(struct   globalmem_dev),
GFP_KERNEL);
    203  if (!globalmem_devp)    /*申请失败*/
    204  {
    205    result =  - ENOMEM;
    206    goto fail_malloc;
    207  }
    208  memset(globalmem_devp, 0, sizeof(struct globalmem_dev));
    209
    210  globalmem_setup_cdev(globalmem_devp, 0);
    211  return 0;
    212
    213  fail_malloc: unregister_chrdev_region(devno, 1);
    214  return result;
    215 }
    216
    217 /*模块卸载函数*/
    218 void globalmem_exit(void)
    219 {
    220  cdev_del(&globalmem_devp->cdev);   /*注销 cdev*/
    221  kfree(globalmem_devp);     /*释放设备结构体内存*/
    222  unregister_chrdev_region(MKDEV(globalmem_major, 0), 1); /*释放设
备号*/
    223 }
    224
    225 MODULE_AUTHOR("Song Baohua");
    226 MODULE_LICENSE("Dual BSD/GPL");
    227
    228 module_param(globalmem_major, int, S_IRUGO);
    229
    230 module_init(globalmem_init);
    231 module_exit(globalmem_exit);
```

除了在 globalmem_open()函数中通过 filp->private_data = globalmem_devp 语句（见第 31 行）将设备结构体指针赋值给文件私有数据指针并在 globalmem_read()、globalmem_write()、globalmem_llseek() 和 globalmem_ioctl() 函数中通过 struct globalmem_dev *dev = filp->private_data 语句获得设备结构体指针并使用该指针操作设备结构体外，代码清单 6.17 与代码清单 6.8～代码清单 6.15 的程序基本相同。

现在翻回到本书的第 1 章，再次阅读代码清单 1.4，即 Linux 下 LED 的设备驱动，是否豁然开朗？

代码清单 6.17 仅仅作为使用 private_data 的范例，直接访问全局变量 globalmem_devp 会更加结构清晰。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 驱动中包含两个同样的设备（次设备号分别为 0 和 1），如代码清单 6.18 所示。

代码清单 6.18　支持两个 globalmem 设备的 globalmem 驱动

```
/*文件打开函数*/
int globalmem_open(struct inode *inode, struct file *filp)
{
  /*将设备结构体指针赋值给文件私有数据指针*/
  struct globalmem_dev *dev;

  dev = container_of(inode->i_cdev,struct globalmem_dev,cdev);
  filp->private_data = dev;
  return 0;
}

/*设备驱动模块加载函数*/
int globalmem_init(void)
{
  int result;
  dev_t devno = MKDEV(globalmem_major, 0);

  /* 申请设备号*/
  if (globalmem_major)
    result = register_chrdev_region(devno, 2, "globalmem");
  else  /* 动态申请设备号 */
  {
    result = alloc_chrdev_region(&devno, 0, 2, "globalmem");
    globalmem_major = MAJOR(devno);
  }
  if (result < 0)
    return result;

  /* 动态申请两个设备结构体的内存*/
    globalmem_devp  =  kmalloc(2*sizeof(struct  globalmem_dev),
GFP_KERNEL);
  if (!globalmem_devp)    /*申请失败*/
  {
    result =  - ENOMEM;
    goto fail_malloc;
  }
  memset(globalmem_devp, 0, 2*sizeof(struct globalmem_dev));

  globalmem_setup_cdev(globalmem_devp[0], 0);
  globalmem_setup_cdev(&globalmem_devp[1], 1);
  return 0;

  fail_malloc: unregister_chrdev_region(devno, 1);
  return result;
}

/*模块卸载函数*/
void globalmem_exit(void)
{
  cdev_del(&(global mem_devp[0].cdev));
  cdev_del(&(global mem_devp[1].cdev)); /*注销 cdev*/
```

```
    51   kfree(globalmem_devp);     /*释放设备结构体内存*/
    52   unregister_chrdev_region(MKDEV(globalmem_major, 0), 2); /*释放设
备号*/
    53 }
    /* 其他代码同代码清单 6.16 */
```

代码清单 6.18 第 7 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。

下面是代码清单 6.18 针对 6.17 的 patch：

```
    36c38,41
    <   filp->private_data = globalmem_devp;
    ---
    >   struct globalmem_dev *dev;
    >
    >   dev = container_of(inode->i_cdev,struct globalmem_dev,cdev);
    >   filp->private_data = dev;
    198c203
    <    result = register_chrdev_region(devno, 1, "globalmem");
    ---
    >    result = register_chrdev_region(devno, 2, "globalmem");
    201c206
    <    result = alloc_chrdev_region(&devno, 0, 1, "globalmem");
    ---
    >    result = alloc_chrdev_region(&devno, 0, 2, "globalmem");
    207,208c212,213
    <   /* 动态申请设备结构体的内存*/
    <       globalmem_devp   =   kmalloc(sizeof(struct   globalmem_dev),
GFP_KERNEL);
    ---
    >   /* 动态申请两个设备结构体的内存*/
    >      globalmem_devp   =   kmalloc(2*sizeof(struct   globalmem_dev),
GFP_KERNEL);
    214c219
    <   memset(globalmem_devp, 0, sizeof(struct globalmem_dev));
    ---
    >   memset(globalmem_devp, 0, 2*sizeof(struct globalmem_dev));
    216c221,222
    <   globalmem_setup_cdev(globalmem_devp, 0);
    ---
    >   globalmem_setup_cdev(&globalmem_devp[0], 0);
    >   globalmem_setup_cdev(&globalmem_devp[1], 1);
    226c232,233
    <   cdev_del(&globalmem_devp->cdev);   /*注销 cdev*/
    ---
    >   cdev_del(&(globalmem_devp[0].cdev));
    >   cdev_del(&(globalmem_devp[1].cdev));   /*注销 cdev*/
    228c235
    <   unregister_chrdev_region(MKDEV(globalmem_major, 0), 1); /*释放设备
号*/
    ---
    >   unregister_chrdev_region(MKDEV(globalmem_major, 0), 2); /*释放设备
号*/
```

6.4 globalmem 驱动在用户空间的验证

编译 globalmem 的驱动，得到 globalmem.ko 文件。运行“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 254 的“globalmem”字符设备驱动，如下所示：

```
[root@localhost driver_study]# cat /proc/devices
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 /dev/vc/0
  4 tty
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
 10 misc
 13 input
 21 sg
 29 fb
128 ptm
136 pts
171 ieee1394
180 usb
189 usb_device
254 globalmem
```

接下来，通过“mknod /dev/globalmem c 254 0”命令创建“/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 globalmem 字符设备：

```
[root@localhost driver_study]# echo 'hello world' > /dev/globalmem
[root@localhost driver_study]# 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，该目录下的树型结构如下所示：

```
|-- refcnt
'-- sections
      |-- .bss
      |-- .data
      |-- .gnu.linkonce.this_module
      |-- .rodata
      |-- .rodata.str1.1
      |-- .strtab
      |-- .symtab
```

```
        |-- .text
        '-- __versions
```

refcnt 记录了 globalmem 模块的引用计数，sections 下包含的多个文件则给出了 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 给出的支持两个 globalmem 设备的驱动，在加载模块后需创建两个设备节点，/dev/globalmem0 对应主设备号 globalmem_major，次设备号 0，/dev/globalmem1 对应主设备号 globalmem_major，次设备号 1。分别读写/dev/globalmem0 和/dev/globalmem1，发现都读写到了正确的对应设备。

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）中较简单的一类设备，其驱动程序中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。