专题11-LED驱动程序设计

一、字符设备控制技术

1.1、设备控制理论

1.1.1、作用

大部分驱动程序除了需要提供读写 设备的能力外,还需要具备控制设备的能力。比如: 改变波特率。

1.1.2、应用程序接口

在用户空间，使用ioctl系统调用来控制设备，原型 如下:

int ioctl(int fd,unsigned long cmd,...)

fd: 要控制的设备文件描述符

cmd: 发送给设备的控制命令

...: 第3个参数是可选的参数，存在与否是依赖于控 制命令(第 2 个参数)。

1.1.3、设备驱动方法

当应用程序使用ioctl系统调用时，驱动程序将由如 下函数来响应：

1: 2.6.36 之前的内核

long (*ioctl) (struct inode* node ,struct file* filp, unsigned int cmd,unsigned long arg)

2：2.6.36之后的内核

long (*unlocked_ioctl) (struct file *filp, unsigned int cmd, unsigned long arg)

参数cmd: 通过应用函数ioctl传递下来的命令

1.2、设备控制实现

1.2.1、定义命令

命令从其实质而言就是一个整数,但为了让这个整数具备更好的 可读性，我们通常会把这个整数分为几个段：类型（8位），序号， 参数传送方向，参数长度。

Type(类型/幻数): 表明这是属于哪个设备的命令。

Number(序号) ，用来区分同一设备的不同命令

Direction：参数传送的方向，可能的值是 _IOC_NONE(没有 数据传输), _IOC_READ, _IOC_WRITE（向设备写入参数）

Size： 参数长度

Linux系统提供了下面的宏来帮助定义命令:

_IO(type,nr)：不带参数的命令

_IOR(type,nr,datatype)：从设备中读参数的命令

_IOW(type,nr,datatype)：向设备写入参数的命令

例：

```
#define MEM_MAGIC  'm'  //定义幻数
#define MEM_SET _IOW(MEM_MAGIC, 0, int)
```

1.2.2、实现操作

unlocked_ioctl函数的实现通常是根据命令执行的一个 switch语句。但是，当命令号不能匹配任何一个设备所 支持的命令时，返回-EINVAL.

编程模型：

```
Switch cmd
   Case 命令A：
      //执行A对应的操作
   Case 命令B：
      //执行B对应的操作
   Default:
      // return -EINVAL
```

1.3、驱动实现

memdev.h:

```
#define MEM_MAGIC 'm'
#define MEM_RESTART _IO(MEM_MAGIC,0)
#define MEM_SET _IOW(MEM_MAGIC,1,int)
```

memdev.c:

```
long mem_ioctl (struct file *filp, unsigned int cmd, unsigned long arg)
{
 switch (cmd) {
```

```
  case MEM_RESTART:
   printk("restart device!\n");
   return 0;

  case MEM_SET:
   printk("ARG IS %d\n", arg);
   return 0;

  default:
   return -EINVAL;
 }
}

struct file_operations memfops =
{
 .llseek = mem_lseek,
 .read = mem_read,
 .write = mem_write,
 .open = mem_open,
 .release = mem_close,
 .unlocked_ioctl = mem_ioctl,
};
```

mem_ctl.c:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/fcntl.h>
#include <sys/ioctl.h>
#include "memdev.h"

int main(void)
{
 int fd;
 fd = open("/dev/memdev0", O_RDWR);
 ioctl(fd, MEM_SET, 115200);
 ioctl(fd, MEM_RESTART);
 return 0;
}
```

二、LED驱动程序设计(OK6410)

led.h:

```
#define GPMCON        0x7f008820
#define GPMDAT        0x7f008824

unsigned int * led_config;
unsigned int * led_data;

#define LED_MAGIC    'l'
#define LED_ON        _IO(LED_MAGIC, 0)
#define LED_OFF       _IO(LED_MAGIC, 1)
```

led.c:

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/cdev.h>
#include <linux/fs.h>
#include <linux/io.h>

#include "led.h"

struct cdev led_dev;
dev_t devno;

int led_open (struct inode * node, struct file *filp)
{
    led_config = ioremap(GPMCON,4);
    writel(0x00001111,led_config);
```

```
    led_data = ioremap(GPMDAT,4);
    return 0;
}

long led_ioctl (struct file *filp, unsigned int cmd, unsigned long arg)
{
    switch (cmd) {
        case LED_ON:
            writel(0xf0,led_data);
            return 0;

        case LED_OFF:
            writel(0xff,led_data);
            return 0;

        default:
            return -EINVAL;
    }
}

static struct file_operations led_fops =
{
    .open = led_open,
    .unlocked_ioctl = led_ioctl,
};

static int led_init(void)
{
    cdev_init(&led_dev, &led_fops);

    alloc_chrdev_region(&devno, 0, 1, "myled");

    cdev_add(&led_dev, devno, 1);
    return 0;
}


static void led_exit(void)
{
    cdev_del(&led_dev);
    unregister_chrdev_region(devno, 1);
}

module_init(led_init);
module_exit(led_exit);

MODULE_AUTHOR("JOHNSON");
MODULE_LICENSE("Dual BSD/GPL");
```

led_app.c:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include "led.h"

int main(int argc, char *argv[])
{
    int fd;
    int cmd;

    if (argc < 2) {
        printf("please enter the second parameter!\n");
        return 0;
    }

    cmd = atoi(argv[1]);
```

```
    fd = open("/dev/myled", O_RDWR);

    if (cmd == 1)
        ioctl(fd, LED_ON);
    else
        ioctl(fd, LED_OFF);

    return 0;
}
```