**混杂设备**

**static struct file\_operations myled\_fops={**

**.owner= THIS\_MODULE,**

**.open= myled\_open,**

**.release= myled\_release,**

**.read= myled\_read,**

**};**

**static struct miscdevice myled\_misc={  
 .minor= MISC\_DYNAMIC\_MINOR,**

**.name= “myled”,**

**.fops= &myled\_fops,**

**} ;**

**static int \_\_init myled\_init(void){**

**int ret;**

**ret=misc\_register(&myled\_misc);**

**if(ret<0){**

**printk(KERN\_ERR”misc\_register fail”);**

**goto err\_misc\_register;**

**}**

**}**

**err\_misc\_register:**

**return 0;**

**}**

**static void \_\_exit myled\_exit(void){**

**misc\_deregister(&myled\_misc);**

**}**

**ioctl**

阶段一：

#define LED\_MAGIC ‘L’

#define CMD\_LED\_ON IOW(LED\_MAGIC,0,unsigned long)

#define CMD\_LED\_OFF IOR(LED\_MAGIC,1,unsigned long)

**long myled\_unlocked\_iioctl(struct file\* file,unsigned int cmd,unsigned long args){**

**int n=args-7;**

**switch(cmd){**

**case** CMD\_LED\_ON:

...

**case** CMD\_LED\_OFF**:**

**...**

**}**

**}**

args 参数接收用户传来的 LED 编号，在内核中通过 n = args - 7 进行偏移处理（因为 D7 对应数组下标 0）。

**第二阶段：**

#define CMD\_LED\_ALL\_ON \_IO(LED\_MAGIC, 2)  
#define CMD\_LED\_ALL\_OFF \_IO(LED\_MAGIC, 3)  
#define CMD\_BUF\_W \_IOW(LED\_MAGIC, 4, unsigned long)  
#define CMD\_BUF\_R \_IOW(LED\_MAGIC, 5, unsigned long)

long myled\_unlocked\_ioctl(struct file \*file, unsigned int cmd, unsigned long args){

int n = args - 7;

void \_\_user\* argp=(void \_\_user\*)args;

......

}

使用了 void \_\_user \*argp = (void \_\_user \*)args; 将 args 强转为用户空间指针，这表示用户传入的是指针（间接传值），而不是简单的整数。

用户端：

阶段一：ioctl(fd\_led,CMD\_LED\_ON,i);

直接传值

阶段二：unsigned long args=100;

unsigned long kargs=100;

ioctl(fd\_led,CMD\_BUF\_W,&args);

ioctl(fd\_led,CMD\_BUF\_R,&kargs);

传递指针

**第三阶段：结构体控制 + 数组传递**

#define CMD\_BUF\_W \_IOW(LED\_BUF\_MAGIC, 0, int[4])  
#define CMD\_BUF\_R \_IOW(LED\_BUF\_MAGIC, 1, int[4])  
#define CMD\_STRUCT\_W \_IOW(LED\_STRUCT\_MAGIC, 2, struct led\_cfg)  
#define CMD\_STRUCT\_R \_IOW(LED\_STRUCT\_MAGIC, 3, struct led\_cfg)

struct led\_cfg {  
 int index;  
 int state;  
};

long myled\_unlocked\_ioctl(struct file \*file, unsigned int cmd, unsigned long args){

int arr\_write[4] = {0};  
 int arr\_read[4] = {1, 1, 1, 1};  
 struct led\_cfg cfg;  
 void \_\_user \*argp = (void \_\_user \*)args;  
  
 switch(cmd){  
 case CMD\_BUF\_W:  
 copy\_from\_user(arr\_write, argp, sizeof(arr\_write));  
 for (int i = 0; i < ARRAY\_SIZE(leds\_gpios); i++)  
 gpio\_set\_value(leds\_gpios[i].gpio, arr\_write[i]);  
 break;  
 case CMD\_BUF\_R:  
 copy\_to\_user(argp, arr\_read, sizeof(arr\_read));  
 break;  
 case CMD\_STRUCT\_W:  
 copy\_from\_user(&cfg, argp, sizeof(cfg));  
 if (cfg.index >= 0 && cfg.index < ARRAY\_SIZE(leds\_gpios))  
 gpio\_set\_value(leds\_gpios[cfg.index].gpio, cfg.state);  
 break;  
 case CMD\_STRUCT\_R:  
 cfg.index = 1;  
 cfg.state = gpio\_get\_value(leds\_gpios[cfg.index].gpio);  
 copy\_to\_user(argp, &cfg, sizeof(cfg));  
 break;  
 default:  
 return -ENOIOCTLCMD;  
 }  
 return 0;

}

**应用层实现要点**：

ioctl(fd, CMD\_STRUCT\_R, &cfg)

ioctl(fd, CMD\_STRUCT\_W, &cfg)

搭配驱动层的copy\_to\_user和copy\_from\_user

**第四阶段：**

* **新增命令宏**：

#define CMD\_KEY\_R \_IOR(LED\_KEY\_MAGIC, 4, unsigned long)

* **myled\_unlocked\_ioctl 实现要点**：

long myled\_unlocked\_ioctl(struct file \*file, unsigned int cmd, unsigned long args){  
 void \_\_user \*argp = (void \_\_user \*)args;  
 unsigned int key\_val = 0;  
  
 if (\_IOC\_TYPE(cmd) != LED\_MAGIC && \_IOC\_TYPE(cmd) != LED\_BUF\_MAGIC && \_IOC\_TYPE(cmd) != LED\_KEY\_MAGIC)  
 return -ENOIOCTLCMD;  
  
 if (cmd == CMD\_KEY\_R) {  
 for (int i = 0; i < ARRAY\_SIZE(key\_gpios); i++) {  
 if (gpio\_get\_value(key\_gpios[i].gpio) == 0)  
 key\_val |= (1 << i);  
 }  
 return copy\_to\_user(argp, &key\_val, sizeof(key\_val));  
 }

.......

}

**应用层实现要点**：

ioctl(fd\_led, CMD\_KEY\_R, &key\_val)

传递指针

**重点解释下在传指针时期会出现的void \_\_user \*argp = (void \_\_user \*)args;这个涉及到了copy以及应用层的ioctl。因为应用层会用ioctl传一个参数，在驱动层的体现就是args，所以要在驱动层用copy，就需要把它当作应用层来的东西。**

**GPIO**

1. **驱动初始化和卸载：**

**myled\_init:**

1. **分配设备号：**

**alloc\_chardev\_region(&led\_dev\_num,0,1,”myled”);**

**&led\_dev\_num:设备号指针**

**0：次设备号的起始值，通常设置为 0**

**1：想分配的次设备号的数量**

**myled：设备的名称，用于注册该设备。**

**（可以替换为手动注册：**

**led\_dev\_num=MKDEV(240,0);**

**register\_chrdev\_region(led\_dev\_num,1,"myled");**

**）**

1. **cdev\_init 和 cdev\_add初始化并注册字符设备（led\_cdev）与设备号的关联**

**cdev\_init(&led\_cdev, &led\_fops); //初始化 cdev，绑定文件操作函数**

**cdev\_add(&led\_cdev, dev\_num, 1); // 向内核注册 cdev 和设备号的对应关 系**

1. **class\_create 和 device\_create：创建设备类和设备文件 /dev/myled，使得用 户空间可以访问该设备**

**需要全局变量led\_class led\_device**

**static struct class \*led\_class = NULL;**

**static struct device \*led\_device = NULL;**

**led\_class = class\_create(THIS\_MODULE,"led\_class"); //创建一个设备类 led\_class，会在 /sys/class/led\_class/ 生成对应目录。**

**THIS\_MODULE：表示当前正在加载的内核模块**

**led\_class：是设备类的名称**

**led\_device = device\_create(led\_class,NULL,led\_dev\_num,NULL,"myled"); // /dev/myled**

**led\_class：这是之前通过 class\_create 创建的设备类的指针，表示设备所属的 设备类。**

**NULL：这是一个父设备指针，一般情况下，如果没有父设备，可以传递 NULL**

**led\_dev\_num：设备号**

**myled：指定设备文件的名称**

1. **gpio\_request 和 gpio\_direction\_output：请求和配置 GPIO 引脚，设置为输出 模式，并初始化为高电平（LED 亮）。**

**rt = gpio\_request(GPIOD7,"led\_D7"); // 申请 GPIO**

**rt = gpio\_direction\_output(GPIOD7,1); // 设置为输出模式，高电平（LED 亮）**

**需要#define GPIOD7 (PAD\_GPIO\_E + 13)**

**myled\_exit：释放 GPIO 引脚，并清理设备、字符设备、设备号等资源。**

1. **gpio\_free(GPIOD7)：释放之前申请的 GPIO，引脚不再使用。**
2. **device\_destroy(led\_class, dev\_num)：销毁设备文件**
3. **class\_destroy(led\_class)：清理设备类**
4. **unregister\_chrdev\_region(dev\_num,1)：清理字符设备**