**Linux IOCTL 驱动开发文档 —— 多阶段LED驱动实现总结**

### 一、项目背景

本系列驱动以控制 GEC6818 板上的多个 LED 灯为核心功能，逐步引入了 IOCTL 命令、缓冲区读写、结构体传递以及按键状态读取。

### 二、代码演进过程

#### ✅ 阶段一：基本 IOCTL 控制

* **命令宏定义**：

#define LED\_MAGIC 'L'  
#define CMD\_LED\_ON \_IOW(LED\_MAGIC, 0, unsigned long)  
#define CMD\_LED\_OFF \_IOW(LED\_MAGIC, 1, unsigned long)

* **myled\_unlocked\_ioctl 实现**：

long myled\_unlocked\_ioctl(struct file \*file, unsigned int cmd, unsigned long args){  
 int n = args - 7;  
 if (\_IOC\_TYPE(cmd) != LED\_MAGIC)  
 return -ENOIOCTLCMD;  
  
 switch (cmd) {  
 case CMD\_LED\_ON:  
 gpio\_set\_value(leds\_gpios[n].gpio, 0);  
 break;  
 case CMD\_LED\_OFF:  
 gpio\_set\_value(leds\_gpios[n].gpio, 1);  
 break;  
 default:  
 return -ENOIOCTLCMD;  
 }  
 return 0;  
}

* **要点总结**：
  + 使用 \_IOW 传递 LED 编号，表示用户空间将编号写入内核。
  + 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）。
  + 魔术字符 LED\_MAGIC 做类型校验，避免命令混淆。

#### ✅ 阶段二：增加全部 LED 控制 + 缓冲区读写

* **新增命令宏**：

#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)

* **myled\_unlocked\_ioctl 实现要点**：

long myled\_unlocked\_ioctl(struct file \*file, unsigned int cmd, unsigned long args){  
 int n = args - 7;  
 unsigned int value;  
 void \_\_user \*argp = (void \_\_user \*)args;  
 int rt = 0;  
  
 if (\_IOC\_TYPE(cmd) != LED\_MAGIC)  
 return -ENOIOCTLCMD;  
  
 switch(cmd){  
 case CMD\_LED\_ON:  
 gpio\_set\_value(leds\_gpios[n].gpio, 0);  
 break;  
 case CMD\_LED\_OFF:  
 gpio\_set\_value(leds\_gpios[n].gpio, 1);  
 break;  
 case CMD\_LED\_ALL\_ON:  
 for (int i = 0; i < ARRAY\_SIZE(leds\_gpios); i++)  
 gpio\_set\_value(leds\_gpios[i].gpio, 0);  
 break;  
 case CMD\_LED\_ALL\_OFF:  
 for (int i = 0; i < ARRAY\_SIZE(leds\_gpios); i++)  
 gpio\_set\_value(leds\_gpios[i].gpio, 1);  
 break;  
 case CMD\_BUF\_W:  
 rt = copy\_from\_user(&value, argp, sizeof(value));  
 break;  
 case CMD\_BUF\_R:  
 value = 666;  
 rt = copy\_to\_user(argp, &value, sizeof(value));  
 break;  
 default:  
 return -ENOIOCTLCMD;  
 }  
 return rt;  
}

* **要点总结**：
  + 增加了多个命令，不再只控制单个 LED，而是支持“一键全开/全关”。
  + 引入了 copy\_from\_user / copy\_to\_user 实现缓冲区传输。
  + 所以使用了 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。
  + 用户层直接用ioctl(fd,CMD\_BUF\_W,write\_data)来对应copy..;

#### ✅ 阶段三：结构体控制 + 数组传递

* **命令宏定义**：

#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)

* **结构体定义**：

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

* **myled\_unlocked\_ioctl 实现要点**：

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

#### ✅ 阶段四：整合 LED 与按键功能

* **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));  
 }  
  
 if (cmd == CMD\_LED\_ON) {  
 if (args < ARRAY\_SIZE(leds\_gpios)){  
 gpio\_set\_value(leds\_gpios[args].gpio, 0);  
 }   
 else if (cmd == CMD\_LED\_OFF) {  
 if (args < ARRAY\_SIZE(leds\_gpios))  
 gpio\_set\_value(leds\_gpios[args].gpio, 1);  
 }  
  
 return 0;  
}

### 三、流程图

+-----------------+  
| 用户空间调用 |  
+--------+--------+  
 |  
 v  
 ioctl(fd, cmd, arg)  
 |  
 v  
+--------+-------------+  
| myled\_unlocked\_ioctl |  
+--------+-------------+  
 |  
 v  
 判断 MAGIC 类型  
 |  
 v  
 switch(cmd)  
 / | \  
 LED\_ON LED\_OFF KEY\_R

### 四、重要知识点总结

| 项目 | 说明 |
| --- | --- |
| \_IO, \_IOR, \_IOW | ioctl 命令编码宏 |
| copy\_from\_user | 拷贝用户空间数据到内核空间 |
| copy\_to\_user | 拷贝内核空间数据到用户空间 |
| MAGIC 区分 | 为每类设备设置不同的 MAGIC，便于识别命令 |
| GPIO 控制 | 使用 gpio\_set\_value, gpio\_get\_value |

### 五、面试高频问题与答案

1. **\_IO/\_IOR/\_IOW/\_IOWR 有什么区别？**
   * \_IO：无数据传输，仅命令传达。
   * \_IOR：从内核读数据到用户空间（Read）。
   * \_IOW：用户空间写数据到内核（Write）。
   * \_IOWR：双向传输（如设置参数又希望返回实际值）。
2. **为什么要使用 MAGIC 码？如何防止冲突？**
   * MAGIC 字节用于唯一标识设备类型，防止命令号混用。
   * 通常选择字符，如 'L'、'K'、'B'，避免与已有 ioctl 冲突。
3. **ioctl 有什么优势？**
   * 可灵活传递命令和参数。
   * 支持结构体传递和复杂命令集，优于 read/write 单一方式。
4. **copy\_to\_user 有哪些注意事项？**
   * 不能直接操作用户指针。
   * 要做 NULL 判断、返回值检查，防止内核崩溃。

if (user\_buf == NULL) {

return -EINVAL; // 或者适当的错误码

}

* + 返回值是没有传递的字节数

1. **如何支持多设备 ioctl？**
   * 通过 MAGIC 区分。
   * 可以在同一驱动中支持多种功能设备（LED + KEY）。
2. **结构体作为 ioctl 参数安全吗？**
   * 需要 copy\_from\_user/copy\_to\_user 安全拷贝。
   * 不可直接解引用用户指针。
   * 可封装成小模块，提升代码规范性。