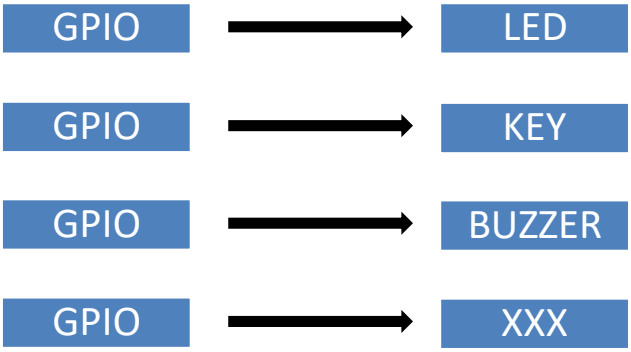
第十二章 platform 平台设备

前面一直都是使用字符设备框架来编写简单的 IO 读写驱动，但是遇到复杂的总线和设备时，仅有字符设备框架是没法应对的。比如，当 SPI 总线挂载了多个设备时，这些设备的驱动该如何实现，SPI 的驱动如何实现，他们之间又如何关联？强行去写也许可以实现功能，但最终的代码会很不环保，复用性很难保证，不符合 Linux 的设计思想。这时就需要用到驱动分离的思想。对此，Linux 内核提供了 platform 设备框架。

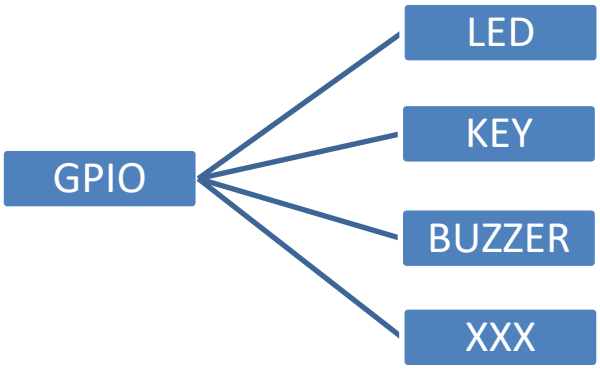
12.1 驱动分离

在说 platform 之前，先简单说说驱动分离的思想。

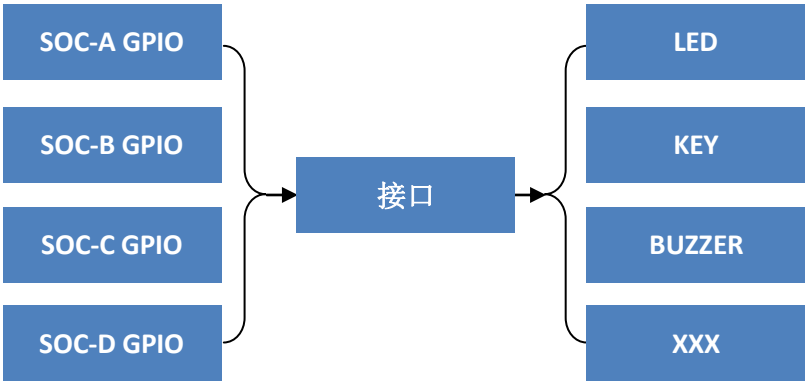
有没有想过这个问题，我们前面写的点亮 LED 实验中的驱动，到底是 IO 的驱动还是 LED 的驱动呢？暂且结合起来看作是 IO_LED 驱动，那么后来我们又用到按键，又有了 IO_KEY 驱动，如果现在要使用蜂鸣器了，就会出现 IO_BUZZER 驱动，每多一个设备，就会多一个 IO_XXX 驱动。



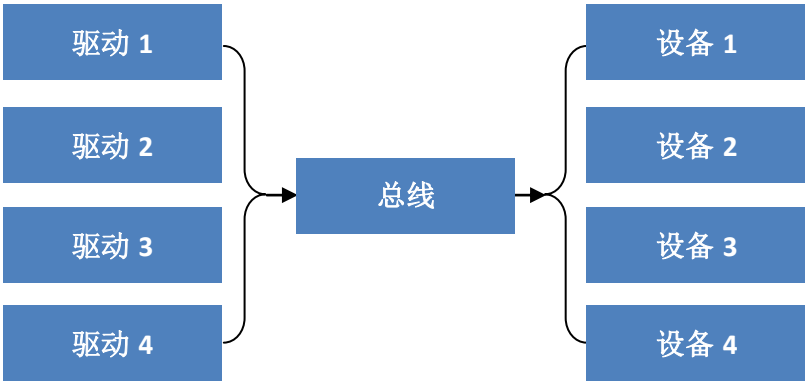
但是这些 IO_XXX 驱动中，IO 的部分几乎是一样的，为了保证代码的简介，把 IO 的部分单独拎出来，写成一个 IO 驱动，然后设备的驱动就只剩下设备自身的内容，只要去调用 IO 驱动中的接口即可。



问题又来了，Linux 的一大卖点是易与移植，现在需要在其他的硬件平台上运行使用这些设备驱动又会变成什么样子呢。不同厂家 SOC 上的 GPIO 底层的实现都有所差别，那每出现一个新的 SOC，就需要重写一个 GPIO 驱动。当然这是不可避免的，但是这些 GPIO 驱动中，还是有共同的部分，即 IO 和设备之间的接口，把这些 GPIO 中的接口部分再拎出来，单独写成一个驱动。



这样就形成了驱动的分离，一边是 SOC 的硬件资源，另一边是用户设备，他们通过统一的接口来连接。驱动分离的思想带来了很多好处，Linux 长久的发展中，省去了很多冗余的代码，SOC 厂家根据统一的接口提供 SOC 片上外设的驱动，设备厂家也根据结构统一的接口提供设备的驱动，用户只需要选择好 SOC 和外设，就能很方便的关联到一起。这样的思想也延续到了单个 SOC 中，片上驱动(之后称为驱动)和设备驱动(之后称为设备)通过总线协议来关联。



当我们往内核中添加驱动时，总线就会查找对应的设备，添加设备时就去查找对应的驱动。这就是 Linux 中的 bus、driver、device 模型，platform 就是这种模型的实现。

12.2 platform 模型

SPI 等总线设备很容易去对应 bus、driver、device 模型，但是 SOC 上并不是所有资源都有总线的概念，比如 GPIO。为了对应 bus、driver、device 模型，platform 模型定义了 platform_bus 虚拟总线、platform_driver 和 platform_device 分别来对应模型总线、驱动和设备。有一点要说明的是，platform 平台设备模型虽然名字里有”平台设备”，但他并不是独立于 Linux 三种设备之外的新的设备类型，他强调的模型二字，只是一种框架，在使用中，是无法和字符设备、块设备、网络设备脱离关系的。

12.2.1 platform_bus

内核中使用 bus_type 结构体表示总线，定义在文件 include/linux/device.h 中：

```
struct bus_type {
    ......
    int (*match)(struct device *dev, struct device_driver *drv);
    ......
};
```

成员中的 match 函数就是驱动和设备匹配的关键，他的两个输入参数一个是 drv 一个是 dev 也就是驱动和设备，每个总线类型都必须实现这个函数。

platform 虚拟总线也是 bus_type 类型的变量，在文件 drivers/base/platform.c 中定义如下：

```
struct bus_type platform_bus_type = {
    .name       = "platform",
    .dev_groups = platform_dev_groups,
    .match      = platform_match,
    .uevent     = platform_uevent,
    .pm         = &platform_dev_pm_ops,
};
```

看一下其中 match 函数的实现 platform_match 函数，也在这个文件中：

```
static int platform_match(struct device *dev, struct device_driver *drv)
{
    struct platform_device *pdev = to_platform_device(dev);
    struct platform_driver *pdrv = to_platform_driver(drv);

    /* When driver_override is set, only bind to the matching driver */
    if (pdev->driver_override)
        return !strcmp(pdev->driver_override, drv->name);

    /* Attempt an OF style match first */
    if (of_driver_match_device(dev, drv))
        return 1;

    /* Then try ACPI style match */
    if (acpi_driver_match_device(dev, drv))
        return 1;

    /* Then try to match against the id table */
    if (pdrv->id_table)
```

```
20.         return platform_match_id(pdrv->id_table, pdev) != NULL;
21.
22.     /* fall-back to driver name match */
23.     return (strcmp(pdev->name, drv->name) == 0);
24. }
```

platform_match 函数提供了 4 种匹配方式。

11~12 行，设备树下会使用的 OF 匹配表匹配方式，match 的输入参数之一 drv 的数据类型中有一个成员变量 of_match_table，of_match_table 又有一个成员为 compatible，如果这个 compatible 能和设备树种的 compatible 属性相匹配，驱动代码中的 probe 就会被调用。

如果没有使用设备树，一般会使用 23 行的第四种匹配方式，直接比较驱动和设备中 name 成员。

12.2.2 platform_driver

1) 定义并初始化 platform_driver

platform 驱动用 platform_driver 结构体来表示，在头文件 include/linux/platform_device.h 中：

```
1.  struct platform_driver {
2.      int (*probe)(struct platform_device *);
3.      int (*remove)(struct platform_device *);
4.      void (*shutdown)(struct platform_device *);
5.      int (*suspend)(struct platform_device *, pm_message_t state);
6.      int (*resume)(struct platform_device *);
7.      struct device_driver driver;
8.      const struct platform_device_id *id_table;
9.      bool prevent_deferred_probe;
10. };
```

a) 成员 probe

probe 函数前面提到过，当设备和驱动匹配成功时，就会执行这个函数。在这个函数中调用原先在驱动入口函数中调用的内容，如字符设备中调用的 cdev_init 函数。

b) 成员 remove

remove 函数在驱动或对应设备注销时会执行，与 probe 相对的，这个函数中调用原先在驱动出口函数中调用的内容，如字符设备中的 cdel_del 函数以及其他初始化内容。

c) 成员 driver

driver 是 struct device_driver 类型，device_driver 是基本的设备驱动类型，platform_driver 是在 device_driver 基础上扩展的，所以需要包含这个类型的元素，以使用他的成员。

device_driver 结构体中有个成员变量

```
const struct of_device_id *of_match_table;
```

of_match_table 也就是上面提到的 OF 匹配表匹配方式用到的成员，of_device_id 结构体中有名为 compatible 的成员，设备树中的 compatible 就是和这个成员来比较的。

of_device_id 结构体中还有名为 name 的成员，需要与 platform 设备中的 neme 字段相

同。

d) 成员 id_table

id_table 是用于匹配设备的，是上面介绍的 platform_match 函数提供的第三种匹配方式需要用到的成员。

2) platform_driver 注册和注销

定义好 struct platform_driver 后，需要在驱动入口函数中调用下面的函数来注册 platform 驱动，取代原先的初始化内容：

```
int platform_driver_register (struct platform_driver *driver);
```

注册成功返回 0，失败返回负。

在出口函数中做相应的注销操作取代原先的注销内容：

```
void platform_driver_unregister(struct platform_driver *drv);
```

12.2.3 platform_device

platform 设备用结构体 platform_device 来表示。在支持设备树的内核中，可以使用设备树代替 platform_device，但是 platform_device 仍然保留使用，我们先完整的了解一下 platform 的完整流程，之后再去结合设备树。

1) platform_device 结构体

platform_device 结构体定义在 include/linux/platform_device.h 中，内容如下：

```
1.  struct platform_device {
2.      const char  *name;
3.      int     id;
4.      bool        id_auto;
5.      struct device   dev;
6.      u32     num_resources;
7.      struct resource *resource;
8.
9.      const struct platform_device_id *id_entry;
10.     char *driver_override; /* Driver name to force a match */
11.
12.     /* MFD cell pointer */
13.     struct mfd_cell *mfd_cell;
14.
15.     /* arch specific additions */
16.     struct pdev_archdata    archdata;
17. };
```

成员 name 用于和驱动匹配，需要和 platform_driver 中的 name 相同。

成员 id 表示当前设备在这类设备中的编号，只有一个这种类型的设备是 id 赋值-1。

成员 num_resources 表示资源数量。

成员 resource 是资源的数组，struct resource 定义如下：

```
1.  struct resource {
```

```
2.      resource_size_t start;
3.      resource_size_t end;
4.      const char *name;
5.      unsigned long flags;
6.      struct resource *parent, *sibling, *child;
7.  };
```

start 表示资源的起始地址，end 表示资源的结束地址。name 是资源名称，flags 表示资源类型，资源类型的宏定义在头文件 include/linux/ioport.h 的 29~105 行。

2) platform_device 注册和注销

在声明初始化 platform_device 结构体后，使用下面的方法注册 platform_device：

```
int platform_device_register(struct platform_device *pdev);
```

注销设备时，使用下面的方法注销：

```
void platform_device_unregister(struct platform_device *pdev);
```

3) 获取资源的方法

当 platform 设备设置好资源后，platform 驱动就可以通过下面的函数来获取资源信息：

```
struct resource *platform_get_resource(struct platform_device *dev, unsigned int type, unsigned int num)
```

参数说明：

dev：目标 platform 设备。

type：也就是上面说的 platform_device 结构体成员 resource 结构体的 flags 成员。

num：指定 type 的资源的下标。

返回值：资源的信息，成功时返回 resource 结构体类型指针，失败时返回 NULL。

12.3 实验

这一章我们用 platform 架构来实现简单的点亮 led 实验。

12.3.1 原理图

和第一章 1.3.1 节的内容相同。

12.3.2 设备树

这一章的实验使用 platform_device 来表示设备，不用设备树。

12.3.3 驱动程序

驱动程序分为驱动和设备两个部分。

1) 先完成驱动的代码，使用 petalinux 新建名为”ax-platform-drv”的驱动程序，并执行 petalinux-config -c rootfs 命令选上新增的驱动程序。

在 ax-platform-drv.c 文件中输入下面的代码：

```c
#include <linux/types.h>
#include <linux/kernel.h>
#include <linux/delay.h>
#include <linux/ide.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/errno.h>
#include <linux/gpio.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/of_gpio.h>
#include <linux/semaphore.h>
#include <linux/timer.h>
#include <linux/irq.h>
#include <linux/wait.h>
#include <linux/poll.h>
#include <linux/fs.h>
#include <linux/fcntl.h>
#include <linux/platform_device.h>
#include <asm/mach/map.h>
#include <asm/uaccess.h>
#include <asm/io.h>

/* 设备节点名称 */
#define DEVICE_NAME       "gpio_leds"
/* 设备号个数 */
#define DEVID_COUNT       1
/* 驱动个数 */
#define DRIVE_COUNT       1
/* 主设备号 */
#define MAJOR
/* 次设备号 */
#define MINOR             0

/* gpio 寄存器虚拟地址 */
static u32 *GPIO_DIRM_0;
/* gpio 使能寄存器 */
static u32 *GPIO_OEN_0;
/* gpio 控制寄存器 */
static u32 *GPIO_DATA_0;
/* AMBA 外设时钟使能寄存器 */
static u32 *APER_CLK_CTRL;

/* 把驱动代码中会用到的数据打包进设备结构体 */
```

```
45. struct alinx_char_dev{
46.     dev_t              devid;      //设备号
47.     struct cdev        cdev;       //字符设备
48.     struct class       *class;     //类
49.     struct device      *device;    //设备
50. };
51. /* 声明设备结构体 */
52. static struct alinx_char_dev alinx_char = {
53.     .cdev = {
54.         .owner = THIS_MODULE,
55.     },
56. };
57. 
58. /* open 函数实现，对应到 Linux 系统调用函数的 open 函数 */
59. static int gpio_leds_open(struct inode *inode_p, struct file *file_p)
60. {
61.     /* 设置私有数据 */
62.     file_p->private_data = &alinx_char;
63. 
64.     return 0;
65. }
66. 
67. /* write 函数实现，对应到 Linux 系统调用函数的 write 函数 */
68. static ssize_t gpio_leds_write(struct file *file_p, const char __user *buf, size_t len, loff_t *lo
    ff_t_p)
69. {
70.     int rst;
71.     char writeBuf[5] = {0};
72. 
73.     rst = copy_from_user(writeBuf, buf, len);
74.     if(0 != rst)
75.     {
76.         return -1;
77.     }
78. 
79.     if(1 != len)
80.     {
81.         printk("gpio_test len err\n");
82.         return -2;
83.     }
84.     if(1 == writeBuf[0])
85.     {
86.         *GPIO_DATA_0 &= 0xFFFFFFFE;
87.     }
```

```
    else if(0 == writeBuf[0])
    {
        *GPIO_DATA_0 |= 0x00000001;
    }
    else
    {
        printk("gpio_test para err\n");
        return -3;
    }

    return 0;
}

/* release 函数实现，对应到 Linux 系统调用函数的 close 函数 */
static int gpio_leds_release(struct inode *inode_p, struct file *file_p)
{
    return 0;
}

/* file_operations 结构体声明，是上面 open、write 实现函数与系统调用函数对应的关键 */
static struct file_operations ax_char_fops = {
    .owner   = THIS_MODULE,
    .open    = gpio_leds_open,
    .write   = gpio_leds_write,
    .release = gpio_leds_release,
};

/* probe 函数实现，驱动和设备匹配时会被调用 */
static int gpio_leds_probe(struct platform_device *dev)
{
    /* 资源大小 */
    int regsize[4];
    /* 资源信息 */
    struct resource *led_source[4];

    int i;
    for(i = 0; i < 4; i ++)
    {
        /* 获取 dev 中的 IORESOURCE_MEM 资源 */
        led_source[i] = platform_get_resource(dev, IORESOURCE_MEM, i);
        /* 返回 NULL 获取资源失败 */
        if(!led_source[i])
        {
            dev_err(&dev->dev, "get resource failed\r\n");
```

```
132.             return -ENXIO;
133.         }
134.         /* 获取当前资源大小 */
135.         regsize[i] = resource_size(led_source[i]);
136.     }
137.
138.     /* 把需要修改的物理地址映射到虚拟地址 */
139.     GPIO_DIRM_0   = ioremap(led_source[0]->start, regsize[0]);
140.     GPIO_OEN_0    = ioremap(led_source[1]->start, regsize[1]);
141.     GPIO_DATA_0   = ioremap(led_source[2]->start, regsize[2]);
142.     APER_CLK_CTRL = ioremap(led_source[3]->start, regsize[3]);
143.
144.     /* MIO_0时钟使能 */
145.     *APER_CLK_CTRL |= 0x00400000;
146.     /* MIO_0设置成输出 */
147.     *GPIO_DIRM_0 |= 0x00000001;
148.     /* MIO_0使能 */
149.     *GPIO_OEN_0 |= 0x00000001;
150.
151.     /* 注册设备号 */
152.     alloc_chrdev_region(&alinx_char.devid, MINOR, DEVID_COUNT, DEVICE_NAME);
153.
154.     /* 初始化字符设备结构体 */
155.     cdev_init(&alinx_char.cdev, &ax_char_fops);
156.
157.     /* 注册字符设备 */
158.     cdev_add(&alinx_char.cdev, alinx_char.devid, DRIVE_COUNT);
159.
160.     /* 创建类 */
161.     alinx_char.class = class_create(THIS_MODULE, DEVICE_NAME);
162.     if(IS_ERR(alinx_char.class))
163.     {
164.         return PTR_ERR(alinx_char.class);
165.     }
166.
167.     /* 创建设备节点 */
168.     alinx_char.device = device_create(alinx_char.class, NULL,
169.                                       alinx_char.devid, NULL,
170.                                       DEVICE_NAME);
171.     if (IS_ERR(alinx_char.device))
172.     {
173.         return PTR_ERR(alinx_char.device);
174.     }
175.
```

```
176.     return 0;
177. }
178. 
179. static int gpio_leds_remove(struct platform_device *dev)
180. {
181.     /* 注销字符设备 */
182.     cdev_del(&alinx_char.cdev);
183. 
184.     /* 注销设备号 */
185.     unregister_chrdev_region(alinx_char.devid, DEVID_COUNT);
186. 
187.     /* 删除设备节点 */
188.     device_destroy(alinx_char.class, alinx_char.devid);
189. 
190.     /* 删除类 */
191.     class_destroy(alinx_char.class);
192. 
193.     /* 释放对虚拟地址的占用 */
194.     iounmap(GPIO_DIRM_0);
195.     iounmap(GPIO_OEN_0);
196.     iounmap(GPIO_DATA_0);
197.     return 0;
198. }
199. 
200. /* 声明并初始化 platform 驱动 */
201. static struct platform_driver led_driver = {
202.     .driver = {
203.         /* 将会用 name 字段和设备匹配，这里 name 命名为 alinx-led */
204.         .name = "alinx-led",
205.     },
206.     .probe  = gpio_leds_probe,
207.     .remove = gpio_leds_remove,
208. };
209. 
210. /* 驱动入口函数 */
211. static int __init gpio_led_drv_init(void)
212. {
213.     /* 在入口函数中调用 platform_driver_register，注册 platform 驱动 */
214.     return platform_driver_register(&led_driver);
215. }
216. 
217. /* 驱动出口函数 */
218. static void __exit gpio_led_dev_exit(void)
219. {
```

```
220.     /* 在出口函数中调用 platform_driver_register，卸载 platform 驱动 */
221.     platform_driver_unregister(&led_driver);
222. }
223.
224. /* 标记加载、卸载函数 */
225. module_init(gpio_led_drv_init);
226. module_exit(gpio_led_dev_exit);
227.
228. /* 驱动描述信息 */
229. MODULE_AUTHOR("Alinx");
230. MODULE_ALIAS("gpio_led");
231. MODULE_DESCRIPTION("PLATFORM LED driver");
232. MODULE_VERSION("v1.0");
233. MODULE_LICENSE("GPL");
```

可以和第二章设备树下的驱动代码做比较，字符设备的部分几乎是一样的。open 函数、write 函数、release 函数都是熟悉的字符设备驱动写法。

116 行实现 probe 函数，把第二章驱动代码驱动入口函数中的内容复制过来，修改资源的获取方式。第二张是从设备树获取资源信息的，修改成从 platform 设备中获取。使用 platform_get_resource 函数，可以通过 resource_size 函数获取资源大小。

179 行实现 remove 函数，把第二章驱动代码驱动出口函数中的内容复制过来即可，甚至不需要修改。

201 行定义 platform_driver 并初始化。

204 行的 name 命名为 alinx-led，之后在实现 platform_device 时，要保持一致。

214 行在驱动入口函数中注册 platform_driver。

221 行在出口函数中注销 platform_driver。

2) 再完成设备的部分，使用 petalinux 新建名为”ax-platform-dev”的驱动程序，并执行 petalinux-config -c rootfs 命令选上新增的程序。

在 ax-platform-dev.c 文件中输入下面的代码：

```
1.  #include <linux/init.h>
2.  #include <linux/module.h>
3.  #include <linux/errno.h>
4.  #include <linux/gpio.h>
5.  #include <linux/cdev.h>
6.  #include <linux/device.h>
7.  #include <linux/of_gpio.h>
8.  #include <linux/semaphore.h>
9.  #include <linux/timer.h>
10. #include <linux/irq.h>
11. #include <linux/wait.h>
12. #include <linux/poll.h>
13. #include <linux/fs.h>
14. #include <linux/fcntl.h>
15. #include <linux/platform_device.h>
```

```
16. #include <asm/mach/map.h>
17. #include <asm/uaccess.h>
18. #include <asm/io.h>
19.
20. /* 寄存器首地址 */
21. /* gpio 方向寄存器 */
22. #define GPIO_DIRM_0       0xE000A204
23. /* gpio 使能寄存器 */
24. #define GPIO_OEN_0        0xE000A208
25. /* gpio 控制寄存器 */
26. #define GPIO_DATA_0       0xE000A040
27. /* AMBA 外设时钟使能寄存器 */
28. #define APER_CLK_CTRL     0xF800012C
29. /* 寄存器大小 */
30. #define REGISTER_LENGTH   4
31.
32. /* 删除设备时会执行此函数 */
33. static void led_release(struct device *dev)
34. {
35.     printk("led device released\r\n");
36. }
37.
38. /* 初始化 LED 的设备信息，即寄存器信息 */
39. static struct resource led_resources[] =
40. {
41.     {
42.         .start = GPIO_DIRM_0,
43.         .end   = GPIO_DIRM_0 + REGISTER_LENGTH - 1,
44.         /* 寄存器当作内存处理 */
45.         .flags = IORESOURCE_MEM,
46.     },
47.     {
48.         .start = GPIO_OEN_0,
49.         .end   = GPIO_OEN_0 + REGISTER_LENGTH - 1,
50.         .flags = IORESOURCE_MEM,
51.     },
52.     {
53.         .start = GPIO_DATA_0,
54.         .end   = GPIO_DATA_0 + REGISTER_LENGTH - 1,
55.         .flags = IORESOURCE_MEM,
56.     },
57.     {
58.         .start = APER_CLK_CTRL,
59.         .end   = APER_CLK_CTRL + REGISTER_LENGTH - 1,
```

```
        .flags = IORESOURCE_MEM,
    },
};

/* 声明并初始化 platform_device */
static struct platform_device led_device =
{
    /* 名字和 driver 中的 name 一致 */
    .name = "alinx-led",
    /* 只有一个设备 */
    .id = -1,
    .dev = {
        /* 设置 release 函数 */
        .release = &led_release,
     },
    /* 设置资源个数 */
    .num_resources = ARRAY_SIZE(led_resources),
    /* 设置资源信息 */
    .resource = led_resources,
};

/* 入口函数 */
static int __init led_device_init(void)
{
    /* 在入口函数中调用 platform_driver_register，注册 platform 驱动 */
    return platform_device_register(&led_device);
}

/* 出口函数 */
static void __exit led_device_exit(void)
{
    /* 在出口函数中调用 platform_driver_register，卸载 platform 驱动 */
    platform_device_unregister(&led_device);
}

/* 标记加载、卸载函数 */
module_init(led_device_init);
module_exit(led_device_exit);

/* 驱动描述信息 */
MODULE_AUTHOR("Alinx");
MODULE_ALIAS("gpio_led");
MODULE_DESCRIPTION("PLATFORM LED device");
MODULE_VERSION("v1.0");
```

```
104. MODULE_LICENSE("GPL");
```

platform_device 也是使用驱动入口出口的方式，当时候加载设备的方式也和驱动一样使用 insmod 命令。

platform_device 两个关键的地方有两点。

一是 39 行开始的 struct resource 结构体类型的数组，这里面是我们需要的设备信息，每个元素都需要初始化的三个成员变量时 start、end、flags，当我们在 drv 中调用 resource_size 函数时，会根据 start 和 end 返回资源大小，flags 是 platform_get_resource 函数获取资源信息的依据之一。

二是 65 行的 platform_device 结构体的实现，关键就是 name 字段要和 drv 中的一致。num_resource 成员的值可以通过宏 ARRAY_SIZE 来获取资源的个数。

33 行实现 release 函数，删除设备时会执行这个函数。

85 行在入口函数中注册 platform_device 设备。

92 行在出口函数中注销 platform_device 设备。

12.3.4 测试程序

测试 APP 和第一章 1.3.4 节内容一致，可以使用第一章的测试程序。

12.3.5 运行测试

测试步骤如下：

```
mount -t nfs -o nolock 192.168.1.107:/home/alinx/work /mnt
cd /mnt
mkdir /tmp/qt
mount qt_lib.img /tmp/qt
cd /tmp/qt
source ./qt_env_set.sh
cd /mnt
insmod ax-platform-dev.ko
insmod ax-platform-drv.ko
cd ./build-axleddev_test-ZYNQ-Debug/
./axleddev_test /dev/gpio_leds on
```

IP 和路径根据实际情况调整。挑键的现象不上一章相同。

现象如下，板子上的 ps_led1 会被点亮火熄灭：

```
root@ax_peta:~# mount -t nfs -o nolock 192.168.1.107:/home/alinx/work /mnt
root@ax_peta:~# cd /mnt
root@ax_peta:/mnt# mkdir /tmp/qt
root@ax_peta:/mnt# mount qt_lib.img /tmp/qt
EXT4-fs (loop0): recovery complete
EXT4-fs (loop0): mounted filesystem with ordered data mode. Opts: (null)
root@ax_peta:/mnt# cd /tmp/qt
root@ax_peta:/tmp/qt# source ./qt_env_set.sh
/tmp/qt
root@ax_peta:/tmp/qt# cd /mnt
```

```
root@ax_peta:/mnt# insmod ax-platform-dev.ko
root@ax_peta:/mnt# insmod ax-platform-drv.ko
[drm] load() is defered & will be called again
```

```
root@ax_peta:/mnt# cd ./build-axleddev_test-ZYNQ-Debug/
root@ax_peta:/mnt/build-axleddev_test-ZYNQ-Debug# ./axleddev_test /dev/gpio_leds on
ps_led1 on
root@ax_peta:/mnt/build-axleddev_test-ZYNQ-Debug#
```

如果我们把设备删除，设备文件也就不见了：

```
root@ax_peta:/mnt/build-axleddev_test-ZYNQ-Debug# rmmod ax_platform_dev
led device released
root@ax_peta:/mnt/build-axleddev_test-ZYNQ-Debug# ./axleddev_test /dev/gpio_leds on
Can't open file /dev/gpio_leds
```

再重新加载，又存在了，说明 probe 函数就是在 dev 和 drv 相匹配的时候执行的

```
root@ax_peta:/mnt/build-axleddev_test-ZYNQ-Debug# insmod ../ax-platform-dev.ko
root@ax_peta:/mnt/build-axleddev_test-ZYNQ-Debug# [drm] load() is defered & will be called again

root@ax_peta:/mnt/build-axleddev_test-ZYNQ-Debug# ./axleddev_test /dev/gpio_leds on
ps_led1 on
root@ax_peta:/mnt/build-axleddev_test-ZYNQ-Debug#
```