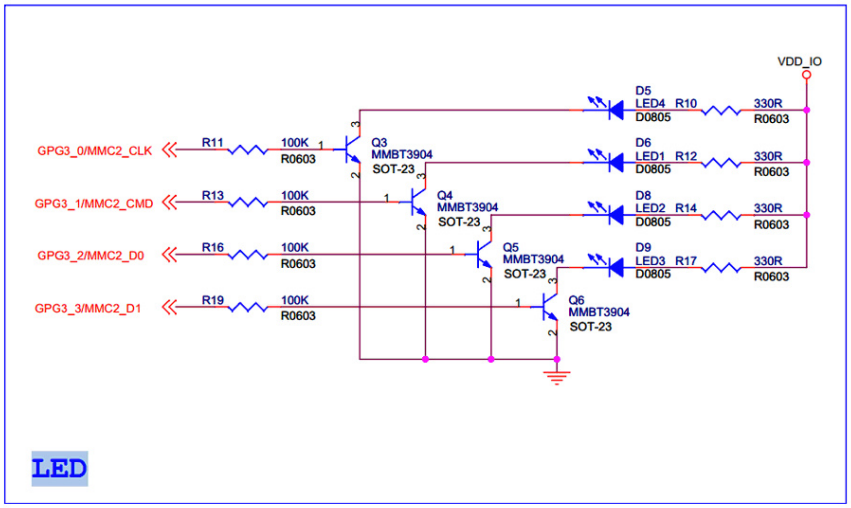
Linux 设备驱动之LED驱动(一)

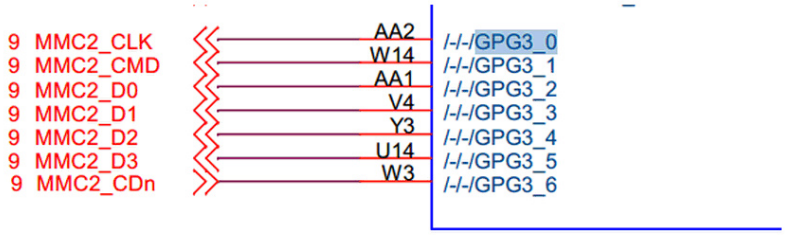
——编写者:草根老师(程姚根)

前面一节我们了解了一下，如何基于platform子系统去写驱动以及驱动和设备之间的匹配关系，但是我们会发现，我们写的驱动并没有实际的功能。这一节我们继续完善我们上一节的LED灯驱动，我们要让我们的灯闪烁起来。

一、FS100开发板的原理图识别



从上面的原理的中我们可以知道，如果想让LED灯亮起来，我们只需要让GPG3_0/MMC2_CLK、GPG3_1/MMC2_CMD、GPG3_2/MMC2_D0、GPG3_3/MMC2_D1对应的四个PIN输出高电平即可。



通过查阅原理图我们知道，这四个PIN分别是:GPG3_0,GPG3_1,GPG3_2,GPG3_3。这些PIN是由GPG3CON和GPG3DAT寄存器进行控制。

二、注册平台设备

通过前面一节我们已经知道，我们基于platform子系统写驱动，应该分别注册平台设备和平台驱动。显然，以上我们从原图查阅的信息应该由平台设备提供,这些信息我们也称为"资源"。这样我们在平台驱动那边只需要获得这些资源就可以控制LED灯了。

不知道大家发现了没，不管是什么开发板，如果想驱动LED设备时，只需要把你的开发板的LED 资源注册到platform子系统，我们写的驱动只要拿到这些资源就可以进行驱动。这就是设备和驱动分离的好处。

好了，接下来我们就来注册LED设备吧。

(1)填充platform_device 结构体

```
#define S5PC100_PA_LED  0xE03001C0
#define SZ_8            0x00000008
```

```
MODULE_LICENSE("GPL");

void led_release(struct device *dev)
{
     return;
}


struct resource s5pc100_led_resources[] = {
    [0] = {
        .start = S5PC100_PA_LED,
        .end   = S5PC100_PA_LED + SZ_8 - 1,
        .flags = IORESOURCE_MEM,
    }
};


struct platform_device s5pc100_device_led = {
    .name = "s5pc100-led",
    .id   = -1,
    .resource = s5pc100_led_resources,
    .num_resources  = ARRAY_SIZE(s5pc100_led_resources),
    .dev = {
        .release = led_release,
    },
};
```

你可能会问，这里的资源到底是什么呀?

这里的资源，就是控制GPIO的GPG3CON和GPG3DAT这两个寄存器的地址哦。我们来看看这两个寄存器在数据手册上的地址:

GPG3CON	0xE030_01C0	R/W	GPG3 Configuration	0x00000000
GPG3DAT	0xE030_01C4	R/W	GPG3 Data	-
GPG3PUD	0xE030_01C8	R/W	GPG3 Pull-up/down	0x1555
GPG3DRV	0xE030_01CC	R/W	GPG3 Drive strength control	0x0000
GPG3PDNCON	0xE030_01D0	R/W	GPG3 Configuration at power down modes	0x00
GPG3PDNPULL	0xE030_01D4	R/W	GPG3 Pulling control at power down modes	0x00

通过手册可以知道，这两个寄存器总共占8个字节，聪明的你可能会问上面的struct resource 结构体成员end的值是不是写错了，应该是:"**S5PC100_PA_LED + SZ_8**"吧!

注意哦，这里的end是最后一个字节的地址哦。

(2)注册/注销平台设备

```
int __init led_device_init(void)
{
    int ret;

    printk("Register led device to platfrom bus!\n");

    ret = platform_device_register(&s5pc100_device_led);

    return ret;
}

void __exit led_device_exit(void)
{
    printk("Remove led device to platfrom bus!\n");

    platform_device_unregister(&s5pc100_device_led);

    return;
}

module_init(led_device_init);
```

```
module_exit(led_device_exit);
```

嗯，在模块加载的时候，注册平台设备，卸载模块的时候注销平台设备。

三、平台驱动编写流程

可以发现注册平台设备很简单，只需要把我们的资源填充到对应的结构体中,然后用platform_device_register函数进行注册就可以了。

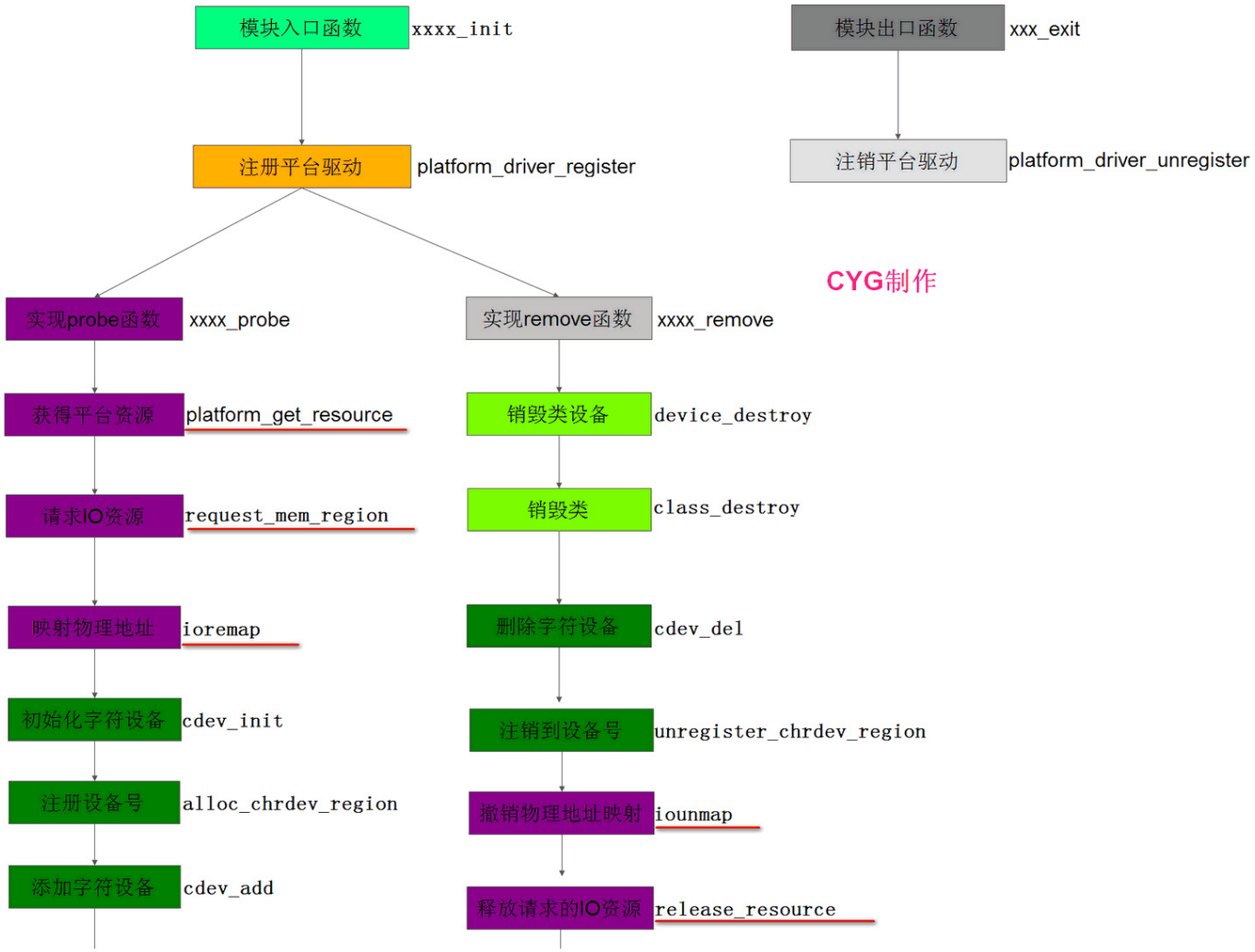
那我们的驱动又该如何去写呢?

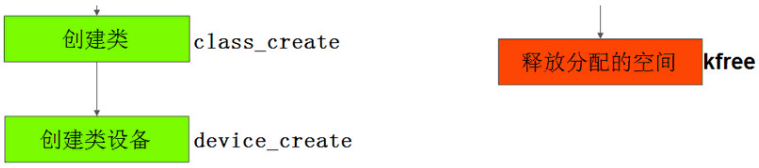
在这里一定要明白我们写驱动的时候，提供的是机制而不是策略。

机制：需要提供什么功能
策略：如何使用这些功能

也就是说我们在写驱动的时候，尽可能向应用层提供操作设备的接口 ,关于如何使用这些接口那是应用层的事。
例如:
我们在驱动层提供一个接口来控制LED灯的亮和灭。具体控制LED灯是亮还是灭，由应用层来决定。应用层可以调用这个接口来点灯也可以调用这个接口来灭灯。

明白了这些以后，我们就来看看平台驱动编写流程:





我们来总结一下，上面图看起来内容挺多的，其实就是在前面我们学的字符驱动框架上加了一个马甲,具体可以分成以下几步:
第一步 ：实现模块的加载和卸载函数
第二步 ：注册平台驱动
第三步 ：实现probe函数和remove函数

其中，第一步和第二步我们在上一节的时候已经实现了。在这里我们重点来看看第三步怎么实现。
在上图中有很多函数接口，大部分函数接口在前面的时候已经讲解过了，未讲解的函数接口我用红色标示出来了，接下来我们先看看这些函数
接口怎么使用。

(1)获得平台资源

```
/**
 * platform_get_resource - get a resource for a device
 * @dev: platform device
 * @type: resource type
 * @num: resource index
 */
struct resource *platform_get_resource(struct platform_device *dev,
                                       unsigned int type, unsigned int num)
```

参数:
@dev platform device
@type 资源类型，常用的有:IORESOURCE_MEM,IORESOURCE_IRQ
@num 同类型资源的编号(从0开始)
返回值:
成功返回资源所在的存储空间首地址，失败返回NULL

例如:
在平台设备注册的时候，注册的资源如下:

```
static struct resource s3c_wdt_resource[] = {
    [0] = {
        .start  = S3C_PA_WDT,
        .end    = S3C_PA_WDT + SZ_1M - 1,
        .flags  = IORESOURCE_MEM,
    },

    [1] = {
        .start  = IRQ_WDT,
        .end    = IRQ_WDT,
        .flags  = IORESOURCE_IRQ,
    }

    [2] = {
        .start  = S3C_PA_WDT1,
        .end    = S3C_PA_WDT1 + SZ_1M - 1,
        .flags  = IORESOURCE_MEM,
    },
};
```

如果此时想获得资源数组的第二个IORESOURCE_MEM资源，则需要按如下方式申请:

```
res = platform_get_resource(pdev,IORESOURCE_MEM,1);
```

注意:num不是指资源数组的下标，而是指同类型资源的编号

(2)请求占用IO占用资源

```
/**
 * request_mem_region - create a new busy resource region
 * @start: resource start address
 * @n: resource region size
 * @name: reserving caller's ID string
 */
```

```
struct resource * request_mem_region(resource_size_t start, resource_size_t n,
                                     const char *name)
```

参数：
@start IO资源的起始地址
@n   资源的大小
@name   一般写平台设备的名字
返回值：
如果申请的IO内存没有被别人占用，则请求成功,返回请求的IO内存首地址
如果对应的IO内存已经被别人申请过，则请求失败，返回NULL

注意：
由于一个IO管脚可能被多个设备共用，如果随意访问必定会出问题，所以每个驱动程序在使用IO内存的时候，都应该先请求，如果请求成功则是使用,反之则不使用。

(3)释放请求的资源

```
/**
 *release_resource - release a previously reserved resource
 *@old: resource pointer
 */
int release_resource(struct resource *old)
```

(4)映射IO物理内存

```
void __iomem *ioremap(unsigned long phy_start,unsigned long size);
```

参数：
@phy_start 起始的物理地址
@size   映射的物理内存大小
返回值：
成功返回映射后的虚拟地址，失败返回NULL

(5)撤销IO内存映射

```
/**
 * iounmap - Free a IO remapping
 * @addr: virtual address from ioremap_*
 *
 * Caller must ensure there is only one unmapping for the same pointer.
 */
void iounmap(volatile void __iomem *addr)
```

(6)读/写IO内存的内容

```
unsigned int readl(unsigned int addr);
```

功能：从指定的地址中读取四个字节

```
void writel(unsigned int value,unsigned int addr)
```

功能：将value写到指定的地址addr

四、实现平台驱动


led2.zip
2014/06/23 15:02, 7.49KB

这里我们写的驱动提供给应用层的接口是一次性操作所有的LED 灯,不能单个操作。大家可以修改驱动程序，完成每次对单个LED灯进行操作。