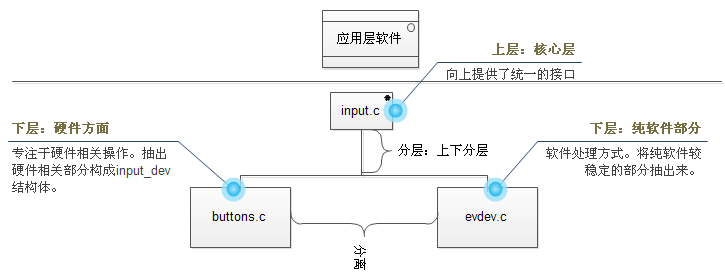
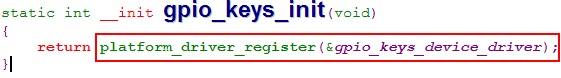
一：分离 分层



在“输入子系统”中有一个实例：gpio\_keys.c.

它的入口函数“gpio\_keys\_init(void)”



注册一个平台driver 结构体。

这个注册的“platform\_driver”我们关心其中的“.probe”函数：

struct platform\_driver gpio\_keys\_device\_driver = {

.probe = gpio\_keys\_probe,

.remove = \_\_devexit\_p(gpio\_keys\_remove),

.driver = {

.name = "gpio-keys",

}

};

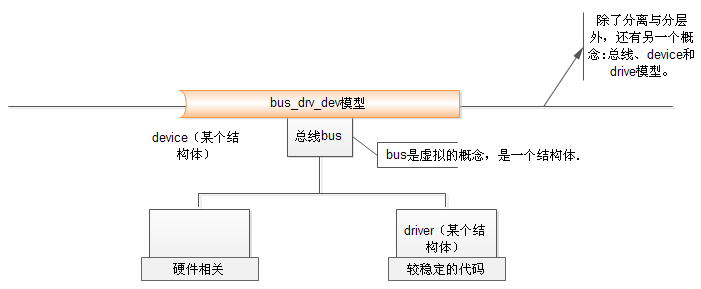
int \_\_devinit gpio\_keys\_probe(struct platform\_device \*pdev)//这里有一个平台设备形参。这个函数就是我们输入子系统框架部分

{  
 struct input\_dev \*input;

input = input\_allocate\_device();分配一个“input\_dev”结构体就是之前的输入子系统框

}

其中的“平台设备”驱动就涉及分离分层的概念:



这个模型的使用方法，可以看“gpio\_keys.c”中的代码：

平台总线是一条虚拟总线：

驱动部分：

gpio\_keys\_init

platform\_driver\_register

drv->driver.bus = &platform\_bus\_type;

return driver\_register(&drv->driver);

// driver\_register会将bus\_drv\_dev模型中的较为稳定的部分driver结构体放到虚拟总线的某个链表drv中

硬件部分：

platform\_device\_add（）

ret = device\_add(&pdev->dev);

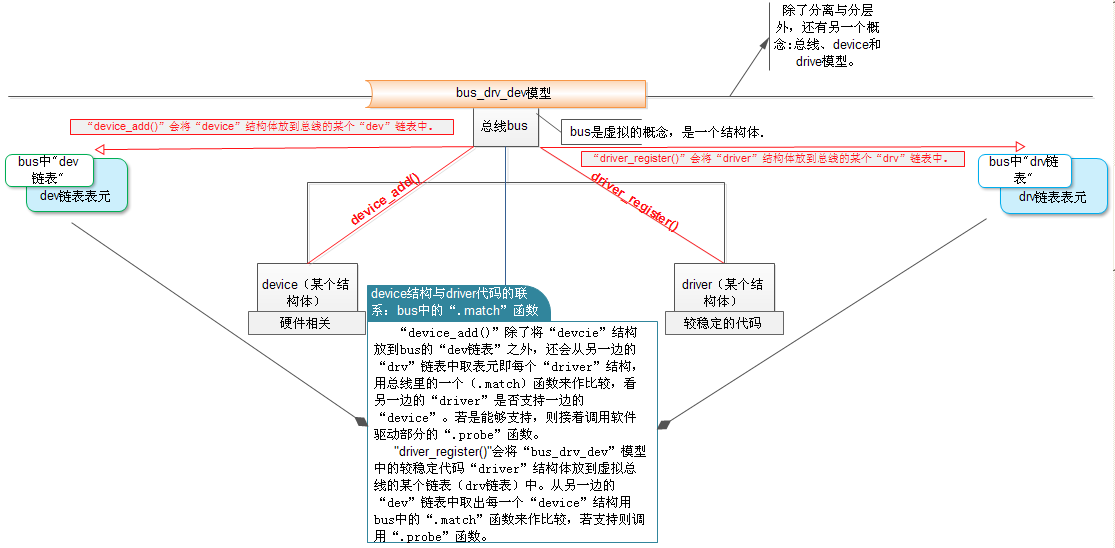
device\_add（）会将bus\_drv\_dev（）模型中的硬件部分device结构体放在虚拟总线的某个链表(dev链表)中。



device 和driver之间的联系：

“device\_add()”除了将“devcie”结构放到bus的“dev链表”之外，还会从另一边的“drv”链表中取表元即某个“driver”结构，用总线里的一个（.match）函数来作比较，看另一边的“driver”是否支持一边的“device”。若是能够支持，则接着调用软件驱动部分的“.probe”函数。

"driver\_register()"会将“bus\_drv\_dev”模型中的较稳定代码“driver”结构体放到虚拟总线的某个链表（drv链表）中。从另一边的“dev”链表中取出每一个“device”结构用bus中的“.match”函数来作比较，若支持则调用“.probe”函数。



“只不过”左右两个注册建立起来的一种机制。在“.probe”函数中做的事件由自已决定，打印一句话，或注册一个字符设备，再或注册一个“input\_dev”结构体等等都是由自已决定。

强制的把一个驱动程序分为左右两边这种机制而已，可以把这套东西放在任何地方，这里的“driver”只是个结构体不要被这个名字迷惑，“device”也只是个结构体，里面放什么内容都是由自已决定的。

int \_\_init gpio\_keys\_init(void)

-->platform\_driver\_register(&gpio\_keys\_device\_driver);注册一个平台driver。

-->drv->driver.bus = &platform\_bus\_type; 总线是平台总线（虚拟出的总线，有一个函数）。

-->.match= platform\_match, (看左右device与driver结构两边是否匹配)

-->struct platform\_device \*pdev = container\_of(dev, struct platform\_device, dev);

//平台dev的名字和drv的名字相同了就认为它们能匹配。

-->return (strncmp(pdev->name, drv->name, BUS\_ID\_SIZE) == 0);

匹配后就会调用platform\_driver gpio\_keys\_device\_driver结构中的.probe = gpio\_keys\_probe函数,若想“.probe”函数能够调用，左边“device”硬件结构部分应该有一个同名的“.name= "gpio-keys"”平台device.

可以搜索这个平台“device”名字“gpio-keys”，但Linux-2.6.22源码中并没有，所以这个“gpio\_keys.c”只不过是个示例程序而已。

编写代码：

现在只想点一个LED灯，这里强制的把代码分成左右两边：

想写一个驱动，想达到一个目的：

左边“device”表示某一个LED灯。要想修改是哪个LED灯，就只需要修改左边这个led\_platform\_dev.c即可。而

右边那个“led\_platform\_drv.c”保持稳定不变。

Led\_platform\_dev.c

模仿：这是打过补丁后的“linux-2.6.22.6\arch\arm\mach-s3c2440\Mach-smdk2440.c"中的代码。

static struct platform\_device s3c2440\_device\_sdi = {

.name = "s3c2440-sdi",// 名字。

.id = -1,

.num\_resources = ARRAY\_SIZE(s3c2440\_sdi\_resource),

.resource = s3c2440\_sdi\_resource,

};

平台资源结构体：

struct resource {

resource\_size\_t start; //资源开始地址。

resource\_size\_t end; //资源结束地址。

const char \*name; //资源名字。

unsigned long flags; //flags表示是哪一类资源。

struct resource \*parent, \*sibling, \*child;

};

“flags”表示是指哪一类资源，也只是定义而已，也是一些结构体：



查看LED1是哪个引脚：\*gpfdat &= ~((1<<4) | (1<<5) | (1<<6));



static struct resource s3c2440\_sdi\_resource[] = {

[0] = { 寄存器的物理地址

.start = S3C2410\_PA\_SDI,

.end = S3C2410\_PA\_SDI + S3C24XX\_SZ\_SDI - 1,

.flags = IORESOURCE\_MEM,

},

[1] = {

.start = IRQ\_SDI,

.end = IRQ\_SDI,

.flags = IORESOURCE\_IRQ,

}

};

仿照上面的示例写：



还有硬件地址，看LED灯的寄存器：

初始物理地址：0x56000050

结束地址：0x56000050 + 8 -1;

驱动程序里定义了一个平台设备：

struct platform\_device led\_dev，这个平台设备里还有一些所谓的平台资源。

资源里有它的寄存器地址：

[0] = {

.start = 0x56000050, //LED的寄存器起始地址:gpfcon = (volatile unsigned long \*)ioremap(0x56000050, 16);

.end = 0x56000050 + 8 -1, //LED寄存器结束地址:gpfdat = gpfcon + 1;

.flags = IORESOURCE\_MEM,

},

要修改寄存器，就直接修改上面的寄存器起始地址就可以。

资源里还有哪一个引脚的信息：

[1] = {

.start = 4, //LED1引脚4是"4,5,6"中的第一个LED灯。

.end = 4,

.flags = IORESOURCE\_IRQ,

}

以后想换另一个LED灯，就只需要换这里的引脚就好，如第2个LED灯引脚是5：

1] = {

.start = 5, //LED2引脚5是"4,5,6"中的第二个LED灯。

.end = 5,

.flags = IORESOURCE\_IRQ,

}

写“平台设备”：

可以参考自己的代码

接着写“平台驱动”：

可以参考自己的代码