这是我在网上整理的一些关于 platform 的文章，感觉不错。与大家共赏

----------zys

platform 总线是 kernel 中最近加入的一种虚拟总线，它被用来连接处在仅有最少基本组件的总线上的那些设备.这样的总线包括许多片上系统上的那些用来整合外设的总线，也包括一些"古董" PC 上的连接器；但不包括像 PCI 或 USB 这样的有庞大正规说明的总线.

平台设备

~~~~~~

平台设备通常指的是系统中的自治体，包括老式的基于端口的设备和连接外设总线的北桥(host bridges),以及集成在片上系统中的绝大多数控制器. 它们通常拥有的一个共同特征是直接编址于 CPU 总线上. 即使在某些罕见的情况下，平台设备会通过某段其他类型的总线连入系统，它们的寄存器也会被直接编址.平台设备会分到一个名称(用在驱动绑定中)以及一系列诸如地址和中断请求号(IRQ)之类的资源.

那什么情况可以使用 platform driver 机制编写驱动呢？

我的理解是只要和内核本身运行依赖性不大的外围设备(换句话说只要不在内核运行所需的一个最小系统之内的设备),相对独立的,拥有各自独自的资源(addresses and IRQs)，都可以用 platform_driver 实现。如：lcd,usb,uart 等，都可以用 platfrom_driver 写，而 timer,irq 等最小系统之内的设备则最好不用 platfrom_driver 机制,实际上内核实现也是这样的。下面继续我们的分析过程。

首先要定义一个 platform_device，我们先来看一下 platform_device 结构的定义，如下所示：

```
// include/linux/platform_device.h:
 16struct platform_device {
 17        const char      * name;
 18        u32             id;
 19        struct device   dev;
 20        u32             num_resources;
 21        struct resource * resource;
 22};
```

下面是对应的 FB 设备的变量定义

```
// arch/arm/mach-pxa/generic.c
229static struct platform_device pxafb_device = {
230        .name           = "pxa2xx-fb",
231        .id             = -1,
232        .dev            = {
233                .platform_data  = &pxa_fb_info,
234                .dma_mask       = &fb_dma_mask,
235                .coherent_dma_mask = 0xffffffff,
236        },
237        .num_resources  = ARRAY_SIZE(pxafb_resources),
238        .resource       = pxafb_resources,
239};
```

由上可以看出，name 成员表示设备名，系统正是通过这个名字来与驱动绑定的，所以
~~~~~~

驱动里面相应的设备名必须与该项相符合；id 表示设备编号，id 的值为－1 表示只有一个这样的设备。

该结构中比较重要的一个成员就是 resource, Linux 设计了这个通用的数据结构来描述各种 I/O 资源(如：I/O 端口、外设内存、DMA 和 IRQ 等)。它的定义如下：

// include/linux/ioport.h:

```
16struct resource {
17        const char *name;
18        unsigned long start, end;
19        unsigned long flags;
20        struct resource *parent, *sibling, *child;
21};
```

下面关于这方面的内容，参考了 http://hi.baidu.com/zengzhaonong/blog/item/654c63d92307f0eb39012fff.html

struct resource 是 linux 对挂接在 4G 总线空间上的设备实体的管理方式。

一个独立的挂接在 cpu 总线上的设备单元,一般都需要一段线性的地址空间来描述设备自身,linux 是怎么管理所有的这些外部"物理地址范围段",进而给用户和 linux 自身一个比较好的观察 4G 总线上挂接的一个个设备实体的简洁、统一级联视图的呢?

linux 采用 struct resource 结构体来描述一个挂接在 cpu 总线上的设备实体(32 位 cpu 的总线地址范围是 0~4G):

resource->start　　描述设备实体在 cpu 总线上的线性起始物理地址;

resource->end　　描述设备实体在 cpu 总线上的线性结尾物理地址;

resource->name　　描述这个设备实体的名称,这个名字开发人员可以随意起,但最好贴切;

resource->flag　　描述这个设备实体的一些共性和特性的标志位;

只需要了解一个设备实体的以上 4 项,linux 就能够知晓这个挂接在 cpu 总线的上的设备实体的基本使用情况,也就是 [resource->start, resource->end]这段物理地址现在是空闲着呢,还是被什么设备占用着呢?

linux 会坚决避免将一个已经被一个设备实体使用的总线物理地址区间段[resource->start, resource->end],再分配给另一个后来的也需要这个区间段或者区间段内部分地址的设备实体,进而避免设备之间出现对同一总线物理地址段的重复引用,而造成对唯一物理地址的设备实体二义性.

以上的 4 个属性仅仅用来描述一个设备实体自身,或者是设备实体可以用来自治的单元,但是这不是 linux 所想的,linux 需要管理 4G 物理总线的所有空间,所以挂接到总线上的形形色色的各种设备实体,这就需要链在一起,因此 resource 结构体提供了另外 3 个成员:指针 parent、sibling 和 child：分别为指向父亲、兄弟和子资源的指针，它们的设置是为了以一种树的形式来管理各种 I/O 资源，以 root source 为例，root->child(*pchild)指向 root 所有孩子中地址空间最小的一个；pchild->sibling 是兄弟链表的开头，指向比自己地址空间大的兄弟。

属性 flags 是一个 unsigned long 类型的 32 位标志值，用以描述资源的属性。比如：资源的类型、是否只读、是否可缓存，以及是否已被占用等。下面是一部分常用属性标志位的定义

// include/linux/ioport.h:

```
29/*
```

```
 30 * IO resources have these defined flags.
 31 */
 32#define IORESOURCE_BITS         0x000000ff      /* Bus-specific bits */
 33
 34#define IORESOURCE_IO           0x00000100      /* Resource type */
 35#define IORESOURCE_MEM          0x00000200
 36#define IORESOURCE_IRQ          0x00000400
 37#define IORESOURCE_DMA          0x00000800
 38
 39#define IORESOURCE_PREFETCH     0x00001000      /* No side effects */
 40#define IORESOURCE_READONLY     0x00002000
 41#define IORESOURCE_CACHEABLE    0x00004000
 42#define IORESOURCE_RANGELENGTH  0x00008000
 43#define IORESOURCE_SHADOWABLE   0x00010000
 44#define IORESOURCE_BUS_HAS_VGA  0x00080000
 45
 46#define IORESOURCE_DISABLED     0x10000000
 47#define IORESOURCE_UNSET        0x20000000
 48#define IORESOURCE_AUTO         0x40000000
 49#define IORESOURCE_BUSY         0x80000000      /* Driver has marked
this resource busy */
```

下面来看我们所使用的 LCD 所占用的资源，如下所示：

```
// arch/arm/mach-pxa/generic.c
static struct resource pxafb_resources[] = {
    [0] = {
        .start    = 0x44000000,
        .end    = 0x4400ffff,
        .flags    = IORESOURCE_MEM,
    },
    [1] = {
        .start    = IRQ_LCD,
        .end    = IRQ_LCD,
        .flags    = IORESOURCE_IRQ,
    },
};
```

由上可知 LCD 占用的资源包括两类，一类是 MEM 类型，一类是 IRQ 类型。MEME 类型资源对应的物理地址范围是 0x44000000 - 0x4400ffff;IRQ 类型资源对应的物理地址范围是 IRQ_LCD，查看相应的定义：

```
// include/asm-arm/arch-pxa/irqs.h：
 15#ifdef CONFIG_PXA27x
 16#define PXA_IRQ_SKIP    0
 17#else
 18#define PXA_IRQ_SKIP    7
 19#endif
```

```
 20
 21#define PXA_IRQ(x)      ((x) - PXA_IRQ_SKIP)
 43#define  IRQ_LCD          PXA_IRQ(17)     /*  LCD  Controller  Service
Request */
```

我们所使用的处理器为 PXA255，所以对应的 PXA_IRQ_SKIP 应该为 7，所以 IRQ_LCD = 10,也就是它对应的中断信号线为 10。

设置完了 platform_device 的相关成员后，下一步就是调用 platform_add_devices() 来向系统中添加该设备了，首先来看它的定义：

```
// drivers/base/platform.c:
/**
*     platform_add_devices - add a numbers of platform devices
*     @devs: array of platform devices to add
*     @num: number of platform devices in array
*/
int platform_add_devices(struct platform_device **devs, int num)
{
    int i, ret = 0;
    for (i = 0; i = 0)
                platform_device_unregister(devs);
            break;
        }
    }
    return ret;
}
```

我们目前只关注 LCD 设备，所以不管 for 循环，关键的一句就是 platform_device_register()，该函数用来进行平台设备的注册，首先来看它的定义：

```
// drivers/base/platform.c:
/**
*     platform_device_register - add a platform-level device
*     @pdev:     platform device we're adding
*
*/
int platform_device_register(struct platform_device * pdev)
{
    device_initialize(&pdev->dev);
    return platform_device_add(pdev);
}
```

它首先调用 device_initialize()来初始化该设备，然后调用 platform_device_add() 来添加该设备。关于 device_initialize() 我们暂且不分析，在这里只关注 platform_device_add()

```
// drivers/base/platform.c:
229/**
230 * platform_device_add - add a platform device to device hierarchy
231 * @pdev: platform device we're adding
```

```
232 *
233 * This is part 2 of platform_device_register(), though may be called
234 * separately _iff_ pdev was allocated by platform_device_alloc().
235 */
236int platform_device_add(struct platform_device *pdev)
237{
238       int i, ret = 0;
239
240       if (!pdev)
241              return -EINVAL;
242
243       if (!pdev->dev.parent)
244              pdev->dev.parent = &platform_bus;
245
246       pdev->dev.bus = &platform_bus_type;
247
248       if (pdev->id != -1)
249                     snprintf(pdev->dev.bus_id,   BUS_ID_SIZE,   "%s.%d",
pdev->name,
250                      pdev->id);
251       else
252              strlcpy(pdev->dev.bus_id, pdev->name, BUS_ID_SIZE);
253
254       for (i = 0; i num_resources; i++) {
255              struct resource *p, *r = &pdev->resource;
256
257              if (r->name == NULL)
258                     r->name = pdev->dev.bus_id;
259
260              p = r->parent;
261              if (!p) {
262                     if (r->flags & IORESOURCE_MEM)
263                            p = &iomem_resource;
264                     else if (r->flags & IORESOURCE_IO)
265                            p = &ioport_resource;
266              }
267
268              if (p && insert_resource(p, r)) {
269                     printk(KERN_ERR
270                            "%s: failed to claim resource %d\n",
271                            pdev->dev.bus_id, i);
272                     ret = -EBUSY;
273                     goto failed;
274              }
```

```
275         }
276
277         pr_debug("Registering platform device '%s'. Parent at %s\n",
278                 pdev->dev.bus_id, pdev->dev.parent->bus_id);
279
280         ret = device_add(&pdev->dev);
281         if (ret == 0)
282                 return ret;
283
284 failed:
285         while (--i >= 0)
286                                         if      (pdev->resource.flags      &
(IORESOURCE_MEM|IORESOURCE_IO))
287                         release_resource(&pdev->resource);
288         return ret;
289}
```

先看 243 - 244 两行，如果该设备的父指针为空，则将它的父指针指向 platform_bus，这是一个 device 类型的变量，它的定义如下：

```
// drivers/base/platform.c:
 26struct device platform_bus = {
 27         .bus_id         = "platform",
 28};
```

紧接着，246 行设置设备的总线类型为 platform_bus_type

```
// drivers/base/platform.c:
892struct bus_type platform_bus_type = {
893         .name           = "platform",
894         .dev_attrs      = platform_dev_attrs,
895         .match          = platform_match,
896         .uevent         = platform_uevent,
897         .pm             = PLATFORM_PM_OPS_PTR,
898};
```

248 - 252 行设置设备指向的 dev 结构的 bus_id 成员，由前面可知，我们只有一个 LCD 设备，所以 pdev->id = -1,因而对应的 bus_id = "pxa2xx-fb",关于这个 bus_id，在定义的时候，内核开发者是后面加了一个注释： /* position on parent bus */

254 - 275 行进行资源处理，首先设置资源的名称，如果 name 成员为空的话，就将该成员设置为我们前面已经赋值的 bus_id，也就是"pxa2xx-fb"

260 - 266 行先将 p 指向我们当前处理的资源的 parent 指针成员，如果 p 指向 NULL，也就是我们当前处理的资源的 parent 指针成员指向 NULL 的话，再检测当前处理的资源的类型，如果是 MEM 类型的，则设置 p 指向 iomem_resource ，如果是 IO 类型的，则使 p 指向 ioport_resource，这两个均是 struct resource 类型的变量，它们的定义如下：

```
// kernel/resource.c
 23 struct resource ioport_resource = {
 24         .name       = "PCI IO",
```

```
 25        .start      = 0,
 26        .end        = IO_SPACE_LIMIT,
 27        .flags      = IORESOURCE_IO,
 28};
 29 EXPORT_SYMBOL(ioport_resource);
 30
 31 struct resource iomem_resource = {
 32        .name       = "PCI mem",
 33        .start      = 0,
 34        .end        = -1,
 35        .flags      = IORESOURCE_MEM,
 36};
 37 EXPORT_SYMBOL(iomem_resource);
```

// include/asm/io.h:

#define IO_SPACE_LIMIT 0xffffffff // 这并不是针对 ARM 平台的定义，针对 ARM 平台的定义我没有找到，所以暂且列一个在这里占位

关于这两个 struct resource 类型的变量，在网络上搜到了如下的信息：（ http://hi.baidu.com/zengzhaonong/blog/item/654c63d92307f0eb39012fff.html）

物理内存页面是重要的资源。从另一个角度看，地址空间本身，或者物理存储器在地址空间中的位置，也是一种资源，也要加以管理 -- resource 管理地址空间资源。

内核中有两棵 resource 树，一棵是 iomem_resource，另一棵是 ioport_resource，分别代表着两类不同性质的地址资源。两棵树的根也都是 resource 数据结构，不过这两个数据结构描述的并不是用于具体操作对象的地址资源，而是概念上的整个地址空间。

将主板上的 ROM 空间纳入 iomem_resource 树中；系统固有的 I/O 类资源则纳入 ioport_resource 树

// kernel/resource.c

```
---------------------------------------
struct resource ioport_resource = {
    .name       = "PCI IO",
    .start      = 0,
    .end        = IO_SPACE_LIMIT,
    .flags      = IORESOURCE_IO,
};
struct resource iomem_resource = {
    .name       = "PCI mem",
    .start      = 0,
    .end        = -1,
    .flags      = IORESOURCE_MEM,
};
/usr/src/linux/include/asm-i386/io.h
#define IO_SPACE_LIMIT 0xffff
0 ~ 0xffff     64K
```

继续我们的函数，268 - 276 行将我们当前处理的资源插入到 p 指针指向的 resource 树里面。这里面只有一个关键的函数 insert_resource()

```
// kernel/resource.c
416/**
417 * insert_resource - Inserts a resource in the resource tree
418 * @parent: parent of the new resource
419 * @new: new resource to insert
420 *
421 * Returns 0 on success, -EBUSY if the resource can't be inserted.
422 *
423 * This function is equivalent to request_resource when no conflict
424 * happens. If a conflict happens, and the conflicting resources
425 * entirely fit within the range of the new resource, then the new
426 * resource is inserted and the conflicting resources become children of
427 * the new resource.
428 */
429int insert_resource(struct resource *parent, struct resource *new)
430{
431	struct resource *conflict;
432
433	write_lock(&resource_lock);
434	conflict = __insert_resource(parent, new);
435	write_unlock(&resource_lock);
436	return conflict ? -EBUSY : 0;
437}
```

资源锁 resource_lock 对所有资源树进行读写保护，任何代码段在访问某一颗资源树之前都必须先持有该锁，该锁的定义也在 resource.c 中。锁机制我们暂且不管，该函数里面关键的就是__insert_resource()函数：

```
// kernel/resource.c:
365/*
366 * Insert a resource into the resource tree. If successful, return NULL,
367 * otherwise return the conflicting resource (compare to __request_resource())
368 */
369static struct resource * __insert_resource(struct resource *parent, struct resource *new)
370{
371	struct resource *first, *next;
372
373	for (;; parent = first) {
374		first = __request_resource(parent, new);
375		if (!first)
376			return first;
377
```

```
378             if (first == parent)
379                     return first;
380
381             if ((first->start > new->start) || (first->end end))
382                     break;
383             if ((first->start == new->start) && (first->end == new->end))
384                     break;
385     }
386
387     for (next = first; ; next = next->sibling) {
388             /* Partial overlap? Bad, and unfixable */
389             if (next->start start || next->end > new->end)
390                     return next;
391             if (!next->sibling)
392                     break;
393             if (next->sibling->start > new->end)
394                     break;
395     }
396
397     new->parent = parent;
398     new->sibling = next->sibling;
399     new->child = first;
400
401     next->sibling = NULL;
402     for (next = first; next; next = next->sibling)
403             next->parent = new;
404
405     if (parent->child == first) {
406             parent->child = new;
407     } else {
408             next = parent->child;
409             while (next->sibling != first)
410                     next = next->sibling;
411             next->sibling = new;
412     }
413     return NULL;
414}
```

374 行有个__request_resource()，它完成实际的资源分配工作。如果参数 new 所描述的资源中的一部分或全部已经被其它节点所占用，则函数返回与 new 相冲突的 resource 结构的指针。否则就返回 NULL。该函数的源代码如下：

```
// kernel/resource.c:
142/* Return the conflict entry if you can't request it */
143static struct resource * __request_resource(struct resource *root, struct resource *new)
```

```
{
        resource_size_t start = new->start;
        resource_size_t end = new->end;
        struct resource *tmp, **p;

        if (end start)
                return root;
        if (end > root->end)
                return root;
        p = &root->child;
        for (;;) {
                tmp = *p;
                if (!tmp || tmp->start > end) {
                        new->sibling = tmp;
                        *p = new;
                        new->parent = root;
                        return NULL;
                }
                p = &tmp->sibling;
                if (tmp->end sibling。For 循环体的执行步骤如下：
```

（1）让 tmp 指向当前正被扫描的 resource 结构（tmp＝*p）。

（2）判断 tmp 指针是否为空（tmp 指针为空说明已经遍历完整个 child 链表），或者当前被扫描节点的起始位置 start 是否比 new 的结束位置 end 还要大。只要这两个条件之一成立的话，就说明没有资源冲突，于是就可以把 new 链入 child 链表中：①设置 new 的 sibling 指针指向当前正被扫描的节点 tmp（new->sibling=tmp）；②当前节点 tmp 的前一个兄弟节点的 sibling 指针被修改为指向 new 这个节点（*p=new）；③将 new 的 parent 指针设置为指向 root。然后函数就可以返回了（返回值 NULL 表示没有资源冲突）。

（3）如果上述两个条件都不成立，这说明当前被扫描节点的资源域有可能与 new 相冲突（实际上就是两个闭区间有交集），因此需要进一步判断。为此它首先修改指针 p，让它指向 tmp->sibling，以便于继续扫描 child 链表。然后，判断 tmp->end 是否小于 new->start，如果小于，则说明当前节点 tmp 和 new 没有资源冲突，因此执行 continue 语句，继续向下扫描 child 链表。否则，如果 tmp->end 大于或等于 new->start，则说明 tmp->[start,end]和 new->[start,end]之间有交集。所以返回当前节点的指针 tmp，表示发生资源冲突。

继续回到 platform_device_add()函数里面，如果 insert_resource()成功，下一步就会调用 280 行 device_add()函数来将设备添加到设备树里面。这个函数暂且不做分析。

＋＋＋

下面来看 platform_driver 驱动的注册过程,一般分为三个步骤：

1、定义一个 platform_driver 结构

2、初始化这个结构，指定其 probe、remove 等函数，并初始化其中的 driver 变量

3、实现其 probe、remove 等函数

platform_device 对应的驱动是 struct platform_driver，它的定义如下

```
// include/linux/platform_device.h:
 48struct platform_driver {
 49        int (*probe)(struct platform_device *);
 50        int (*remove)(struct platform_device *);
 51        void (*shutdown)(struct platform_device *);
 52        int (*suspend)(struct platform_device *, pm_message_t state);
 53        int (*suspend_late)(struct platform_device *, pm_message_t state);
 54        int (*resume_early)(struct platform_device *);
 55        int (*resume)(struct platform_device *);
 56        struct device_driver driver;
 57};
```

可见，它包含了设备操作的几个功能函数，同样重要的是，它还包含了一个 device_driver 结构。刚才提到了驱动程序中需要初始化这个变量。下面看一下这个变量的定义，位于 include/linux/device.h 中：

```
// include/linux/device.h:
120struct device_driver {
121        const char              *name;
122        struct bus_type          *bus;
123
124        struct module            *owner;
125        const char              *mod_name;      /* used for built-in modules
*/
126
127        int (*probe) (struct device *dev);
128        int (*remove) (struct device *dev);
129        void (*shutdown) (struct device *dev);
130        int (*suspend) (struct device *dev, pm_message_t state);
131        int (*resume) (struct device *dev);
132        struct attribute_group **groups;
133
134        struct driver_private *p;
135};
```

需要注意这两个变量：name 和 owner。那么的作用主要是为了和相关的 platform_device 关联起来，owner 的作用是说明模块的所有者，驱动程序中一般初始化为 THIS_MODULE。

对于我们的 LCD 设备，它的 platform_driver 变量就是：

```
// drivers/video/pxafb.c:
1384static struct platform_driver pxafb_driver = {
1385        .probe          = pxafb_probe,
1386#ifdef CONFIG_PM
1387        .suspend        = pxafb_suspend,
1388        .resume         = pxafb_resume,
1389#endif
1390        .driver         = {
```

```
1391                .name   = "pxa2xx-fb",
1392        },
1393};
```

由上可知 pxafb_driver.driver.name 的值与前面我们讲过的 platform_device 里面的 name 成员的值是一致的，内核正是通过这个一致性来为驱动程序找到资源，即 platform_device 中的 resource。

上面把驱动程序中涉及到的主要结构都介绍了，下面主要说一下驱动程序中怎样对这些结构进行处理，以使驱动程序能运行。相信大家都知道 module_init（）这个宏。驱动模块加载的时候会调用这个宏。它接收一个函数为参数，作为它的参数的函数将会对上面提到的 platform_driver 进行处理。看我们的实例：这里的 module_init()要接收的参数为 pxafb_init 这个函数，下面是这个函数的定义：

```
// drivers/video/pxafb.c:
1411 int __devinit pxafb_init(void)
1412{
1413#ifndef MODULE
1414        char *option = NULL;
1415
1416        if (fb_get_options("pxafb", &option))
1417                return -ENODEV;
1418        pxafb_setup(option);
1419#endif
1420        return platform_driver_register(&pxafb_driver);
1421}
1422
1423 module_init(pxafb_init);
```

注意函数体的最后一行，它调用的是 platform_driver_register 这个函数。这个函数定义于 driver/base/platform.c 中，定义如下：

```
// drivers/base/platform.c
439/**
440 * platform_driver_register
441 * @drv: platform driver structure
442 */
443int platform_driver_register(struct platform_driver *drv)
444{
445        drv->driver.bus = &platform_bus_type;
446        if (drv->probe)
447                drv->driver.probe = platform_drv_probe;
448        if (drv->remove)
449                drv->driver.remove = platform_drv_remove;
450        if (drv->shutdown)
451                drv->driver.shutdown = platform_drv_shutdown;
452        if (drv->suspend)
453                drv->driver.suspend = platform_drv_suspend;
454        if (drv->resume)
```

```
455             drv->driver.resume = platform_drv_resume;
456     if (drv->pm)
457             drv->driver.pm = &drv->pm->base;
458     return driver_register(&drv->driver);
459}
460EXPORT_SYMBOL_GPL(platform_driver_register);
```

由上可知，它的功能先是为上面提到的 plarform_driver 中的 driver 这个结构中的 probe、remove 这些变量指定功能函数，最后调用 driver_register()进行设备驱动的注册。在注册驱动的时候，这个函数会以上面提到的 name 成员的值为搜索内容，搜索系统中注册的 device 中有没有与这个 name 值相一致的 device，如果有的话，那么接着就会执行 platform_driver 里 probe 函数。

到目前为止，内核就已经知道了有这么一个驱动模块。内核启动的时候，就会调用与该驱动相关的 probe 函数。我们来看一下 probe 函数实现了什么功能。

probe 函数的原型为

int xxx_probe(struct platform_device *pdev)

即它的返回类型为 int，接收一个 platform_device 类型的指针作为参数。返回类型就是我们熟悉的错误代码了，而接收的这个参数呢，我们上面已经说过，驱动程序为设备服务，就需要知道设备的信息。而这个参数，就包含了与设备相关的信息。

probe 函数接收到 plarform_device 这个参数后，就需要从中提取出需要的信息。它一般会通过调用内核提供的 platform_get_resource 和 platform_get_irq 等函数来获得相关信息。如通过 platform_get_resource 获得设备的起始地址后，可以对其进行 request_mem_region 和 ioremap 等操作，以便应用程序对其进行操作。通过 platform_get_irq 得到设备的中断号以后，就可以调用 request_irq 函数来向系统申请中断。这些操作在设备驱动程序中一般都要完成。

在完成了上面这些工作和一些其他必须的初始化操作后，就可以向系统注册我们在/dev 目录下能看在的设备文件了。举一个例子，在音频芯片的驱动中，就可以调用 register_sound_dsp 来注册一个 dsp 设备文件，lcd 的驱动中就可以调用 register_framebuffer 来注册 fb 设备文件。这个工作完成以后，系统中就有我们需要的设备文件了。而和设备文件相关的操作都是通过一个 file_operations 来实现的。在调用 register_sound_dsp 等函数的时候，就需要传递一个 file_operations 类型的指针。这个指针就提供了可以供用户空间调用的 write、read 等函数。file_operations 结构的定义位于 include/linux/fs.h 中，列出如下：

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long,
loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long,
loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
```

```
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*aio_fsync) (struct kiocb *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned
long, unsigned long, unsigned long);
    int (*check_flags)(int);
    int (*dir_notify)(struct file *filp, unsigned long arg);
    int (*flock) (struct file *, int, struct file_lock *);
    ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t,
unsigned int);
    ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t,
unsigned int);
    int (*setlease)(struct file *, long, struct file_lock **);
};
```

到目前为止，probe 函数的功能就完成了。

当用户打开一个设备，并调用其 read、write 等函数的时候，就可以通过上面的 file_operations 来找到相关的函数。所以，用户驱动程序还需要实现这些函数，具体实现和相关的设备有密切的关系，这里就不再介绍了。

关于我们所使用的 LCD 设备的 xxx_probe()函数的分析可查看文章 ＜＜pxafb 驱动程序分析＞＞

下面看我们所使用的 LCD 设备的 probe() 函数:

```
// drivers/video/pxafb.c:
1271int __init pxafb_probe(struct platform_device *dev)
1272{
1273        struct pxafb_info *fbi;
1274        struct pxafb_mach_info *inf;
1275        int ret;
1276
1277        dev_dbg(dev, "pxafb_probe\n");
1278
1279        inf = dev->dev.platform_data;
1280        ret = -ENOMEM;
1281        fbi = NULL;
1282        if (!inf)
1283                goto failed;
1284
```

```
1285#ifdef CONFIG_FB_PXA_PARAMETERS
1286       ret = pxafb_parse_options(&dev->dev, g_options);
1287       if (ret lccr0 & LCCR0_INVALID_CONFIG_MASK)
1296                dev_warn(&dev->dev, "machine LCCR0 setting contains
illegal bits: %08x\n",
1297                    inf->lccr0 & LCCR0_INVALID_CONFIG_MASK);
1298       if (inf->lccr3 & LCCR3_INVALID_CONFIG_MASK)
1299                dev_warn(&dev->dev, "machine LCCR3 setting contains
illegal bits: %08x\n",
1300                    inf->lccr3 & LCCR3_INVALID_CONFIG_MASK);
1301       if (inf->lccr0 & LCCR0_DPD &&
1302           ((inf->lccr0 & LCCR0_PAS) != LCCR0_Pas ||
1303            (inf->lccr0 & LCCR0_SDS) != LCCR0_Sngl ||
1304            (inf->lccr0 & LCCR0_CMS) != LCCR0_Mono))
1305              dev_warn(&dev->dev, "Double Pixel Data (DPD) mode is only
valid in passive mono"
1306                       " single panel mode\n");
1307       if ((inf->lccr0 & LCCR0_PAS) == LCCR0_Act &&
1308           (inf->lccr0 & LCCR0_SDS) == LCCR0_Dual)
1309                dev_warn(&dev->dev, "Dual panel only valid in passive
mode\n");
1310       if ((inf->lccr0 & LCCR0_PAS) == LCCR0_Pas &&
1311            (inf->upper_margin || inf->lower_margin))
1312              dev_warn(&dev->dev, "Upper and lower margins must be 0
in passive mode\n");
1313#endif
1314
1315         dev_dbg(&dev->dev, "got a %dx%dx%d LCD\n",inf->xres,
inf->yres, inf->bpp);
1316       if (inf->xres == 0 || inf->yres == 0 || inf->bpp == 0) {
1317              dev_err(&dev->dev, "Invalid resolution or bit depth\n");
1318              ret = -EINVAL;
1319              goto failed;
1320       }
1321       pxafb_backlight_power = inf->pxafb_backlight_power;
1322       pxafb_lcd_power = inf->pxafb_lcd_power;
1323       fbi = pxafb_init_fbinfo(&dev->dev);
1324       if (!fbi) {
1325                  dev_err(&dev->dev, "Failed to initialize framebuffer
device\n");
1326              ret = -ENOMEM; // only reason for pxafb_init_fbinfo to fail is
kmalloc
1327              goto failed;
1328       }
```

```
1329
1330        /* Initialize video memory */
1331        ret = pxafb_map_video_memory(fbi);
1332        if (ret) {
1333                dev_err(&dev->dev, "Failed to allocate video RAM: %d\n",
ret);
1334                ret = -ENOMEM;
1335                goto failed;
1336        }
1337
1338         ret = request_irq(IRQ_LCD, pxafb_handle_irq, SA_INTERRUPT,
"LCD", fbi);
1339        if (ret) {
1340                dev_err(&dev->dev, "request_irq failed: %d\n", ret);
1341                ret = -EBUSY;
1342                goto failed;
1343        }
1344
1345        /*
1346         * This makes sure that our colour bitfield
1347         * descriptors are correctly initialised.
1348         */
1349        pxafb_check_var(&fbi->fb.var, &fbi->fb);
1350        pxafb_set_par(&fbi->fb);
1351
1352        platform_set_drvdata(dev, fbi);
1353
1354        ret = register_framebuffer(&fbi->fb);
1355        if (ret dev, "Failed to register framebuffer device: %d\n", ret);
1357                goto failed;
1358        }
1359
1360#ifdef CONFIG_PM
1361        // TODO
1362#endif
1363
1364#ifdef CONFIG_CPU_FREQ
1365        fbi->freq_transition.notifier_call = pxafb_freq_transition;
1366        fbi->freq_policy.notifier_call = pxafb_freq_policy;
1367                                cpufreq_register_notifier(&fbi->freq_transition,
CPUFREQ_TRANSITION_NOTIFIER);
1368                                    cpufreq_register_notifier(&fbi->freq_policy,
CPUFREQ_POLICY_NOTIFIER);
1369#endif
```

```
1370
1371        /*
1372         * Ok, now enable the LCD controller
1373         */
1374        set_ctrlr_state(fbi, C_ENABLE);
1375
1376        return 0;
1377
1378failed:
1379        platform_set_drvdata(dev, NULL);
1380        kfree(fbi);
1381        return ret;
1382}
```


参考文献：

struct--resource
http://hi.baidu.com/zengzhaonong/blog/item/654c63d92307f0eb39012fff.h

驱动程序模型-platform
http://www.ourkernel.com/bbs/archiver/?tid-67.html

Linux 对 I/O 端口资源的管理
http://www.host01.com/article/server/00070002/0542417251875372.htm

Linux 对 I/O 端口资源的管理(ZZ)
http://hi.baidu.com/zengzhaonong/blog/item/0d6f6909e2aa5dad2fddd444.ht
ml

platform_device 和 platform_driver
http://linux.chinaunix.net/techdoc/net/2008/09/10/1031351.shtml

linux resource, platform_device 和驱动的关系
http://blog.csdn.net/wawuta/archive/2007/03/14/1529621.aspx

由 fb 设备的注册过程来看内核的设备模型

在＜＜platform 设备添加流程 ＞＞一文中，关于 struct device 方面的内容没有详加阐述，现在我们来一一分析。struct platform 结构里面有一个 struct device 类型的成员，我们先来看看该结构的定义：

```
// include/linux/device.h:
305struct device {
306        struct klist            klist_children;
307        struct klist_node       knode_parent;           /* node in sibling list */
308        struct klist_node       knode_driver;
309        struct klist_node       knode_bus;
310        struct device    * parent;
311
312        struct kobject kobj;
313        char    bus_id[BUS_ID_SIZE];    /* position on parent bus */
314        struct device_attribute uevent_attr;
315
316        struct semaphore        sem;    /* semaphore to synchronize calls to
317                                 * its driver.
318                                 */
319
320        struct bus_type * bus;          /* type of bus device is on */
321        struct device_driver *driver;   /* which driver has allocated this
322                                    device */
323        void            *driver_data;   /* data private to the driver */
324        void            *platform_data; /* Platform specific data, device
325                                    core doesn't touch it */
326        void            *firmware_data; /* Firmware specific data (e.g. ACPI,
327                                    BIOS data),reserved for device core*/
328        struct dev_pm_info      power;
329
330        u64             *dma_mask;      /* dma mask (if dma'able device) */
331        u64             coherent_dma_mask;/* Like dma_mask, but for
332                                     alloc_coherent mappings as
333                                     not all hardware supports
334                                     64 bit addresses for consistent
335                                     allocations such descriptors. */
336
337        struct list_head        dma_pools;      /* dma pools (if dma'ble) */
338
339        struct dma_coherent_mem *dma_mem; /* internal for coherent
mem
```

```
340                                     override */
341
342         void    (*release)(struct device * dev);
343};
```

下面来看一下各个成员的函义：

g_list:　　全局设备列表中的节点.

node:　　父设备的孩子列表中的节点.

bus_list:　　设备从属的总线所属的设备列表中的节点.

driver_list:　　设备对应的驱动所属的设备列表中的节点.

intf_list:　　intf_data 列表. 对每个该设备所支持的接口, 程序会分配一个结构体.

children:　　子设备列表.

parent:　　*** 待修正 ***

name:　　设备描述(用 ASCII 码).例: " 3Com Corporation 3c905 100BaseTX [Boomerang]"

bus_id:　　设备总线的位置描述(用 ASCII 码). 此设备从属的总线下的所有设备应对应唯一的描述符.

例: PCI 总线 bus_id 的格式为

::

系统中所有 PCI 设备的 bus_id 都对应上述同一个名字.

lock:　　设备的自旋锁(spinlock).

(译注: 在自旋锁中, 线程简单地循环等待并检查, 直到锁可用.)(xlp 补注: 在 UP（uni processor）情况下,spinlock 就是简单的 cli(清中断)和 seiL(设中断).

在 SMP 情况下,spinlock_lock 是循环检测锁,可用后获取锁,spinlock_unlock 是直接释放锁).

refcount:　　设备引用的数量.

bus:　　指向设备所属总线的 bus_type 结构体的指针.

dir:　　设备的 sysfs 目录.(译注: sysfs 是 Linux 2.6 提供的虚拟文件系统. sysfs 从内核设备模型中向用户空间导出设备及驱动的信息. 它也被用来进行配置.)

class_num:　　设备的 Class-enumerated 值.

driver:　　指向设备驱动程序结构体 device_driver 的指针.

driver_data:　　驱动相关的数据.

platform_data:　　设备所在平台的数据.例: 对嵌入式或片上系统(SOC)这类用户自定义板上的设备, Linux 常使用 platform_data 指向一个针对板的结构体, 来描述设备以及设备间

的连线.这样一个结构体中可能包含可用的端口, 芯片参数, GPIO 针扮演的额外角色等等. 它可以缩小"板支持包"(Board Support Packages, BSP)的体积并减少

驱动中针对板的#ifdef 的数量.

current_state:　　设备当前电源状态.

saved_state:　　指向设备已保存的状态的指针. 驱动可以利用它来控制设备,

release:　　当所有设备引用都撤销后用来释放设备的回调函数. 它应由为设备分配空间的程序(也就是发现这个设备的总线驱动)来设置.

现在再来看一下 pxafb_device 的定义

// arch/arm/mach-pxa/generic.c:

```
229static struct platform_device pxafb_device = {
230       .name          = "pxa2xx-fb",
231       .id            = -1,
232       .dev           = {
233              .platform_data  = &pxa_fb_info,
234              .dma_mask       = &fb_dma_mask,
235              .coherent_dma_mask = 0xffffffff,
236       },
237       .num_resources  = ARRAY_SIZE(pxafb_resources),
238       .resource       = pxafb_resources,
239};
```

也就是说只初始化了 struct device 结构的 platform_data、dma_mask 和 coherent_dma_mask 成员，相关的定义如下：

```
// arch/arm/mach-pxa/generic.c:
207 static struct pxafb_mach_info pxa_fb_info;
227 static u64 fb_dma_mask = ~(u64)0;
```

那其它成员是什么时候赋值呢？记得当初我们在 platform_device_register()曾经遇到过 device_initialize()，当初直接跳过了，我们现在再来看它

```
// drivers/base/platform.c:
/**
*     platform_device_register - add a platform-level device
*     @pdev:     platform device we're adding
*
*/
int platform_device_register(struct platform_device * pdev)
{
    device_initialize(&pdev->dev);
    return platform_device_add(pdev);
}
// drivers/base/core.c:
216/**
217 *      device_initialize - init device structure.
218 *      @dev:    device.
219 *
220 *      This prepares the device for use by other layers,
221 *      including adding it to the device hierarchy.
222 *      It is the first half of device_register(), if called by
223 *      that, though it can also be called separately, so one
224 *      may use @dev's fields (e.g. the refcount).
225 */
226
227void device_initialize(struct device *dev)
228{
229       kobj_set_kset_s(dev, devices_subsys);
```

```
230        kobject_init(&dev->kobj);
231        klist_init(&dev->klist_children, klist_children_get,
232                   klist_children_put);
233        INIT_LIST_HEAD(&dev->dma_pools);
234        init_MUTEX(&dev->sem);
235        device_init_wakeup(dev, 0);
236}
```

这里的 devices_subsys 定义于 drivers/base/core.c,如下所示：

```
// drivers/base/core.c:
164/*
165 *      devices_subsys - structure to be registered with kobject core.
166 */
167
168decl_subsys(devices, &ktype_device, &device_uevent_ops);
```

其中 decl_subsys 是一个宏，定义于 include/linux/kobject.h 中，如下所示：

```
// include/linux/kobject.h:
165struct subsystem {
166        struct kset             kset;
167        struct rw_semaphore     rwsem;
168};
169
170#define decl_subsys(_name,_type,_uevent_ops) \
171struct subsystem _name##_subsys = { \
172        .kset = { \
173                .kobj = { .name = __stringify(_name) }, \
174                .ktype = _type, \
175                .uevent_ops =_uevent_ops, \
176        } \
177}
```

其中的__stringify()也是一个宏，定义于 include/linux/stringify.h,如下所示：

```
#ifndef __LINUX_STRINGIFY_H
#define __LINUX_STRINGIFY_H
/* Indirect stringification.  Doing two levels allows the parameter to be a
* macro itself.  For example, compile with -DFOO=bar, __stringify(FOO)
* converts to "bar".
*/
#define __stringify_1(x)    #x
#define __stringify(x)      __stringify_1(x)
#endif    /* !__LINUX_STRINGIFY_H */
```

"#"的作用是把宏参数转换成字符串,因此 __stringify(x) "x"

需要注意的是，这里用了两层，原因可以查看博客上面 C/C++里面>

把它代入即得到

```
struct subsystem devices_subsys = {
    .kset = {
```

```
        .kobj = { .name = "devices" },
        .ktype = &ktype_device,
        .uevent_ops = &device_uevent_ops,
    }
}
```

其中 ktype_device 和 device_uevent_ops 以及对应的 dev_sysfs_ops 均位于 drivers/base/core.c,里面主要是指定了一些回调函数，如下所示：

```
// drivers/base/core.c:
 87static struct kobj_type ktype_device = {
 88        .release        = device_release,
 89        .sysfs_ops      = &dev_sysfs_ops,
 90};
151static struct kset_uevent_ops device_uevent_ops = {
152        .filter =       dev_uevent_filter,
153        .name =         dev_uevent_name,
154        .uevent =       dev_uevent,
155};
 59static struct sysfs_ops dev_sysfs_ops = {
 60        .show   = dev_attr_show,
 61        .store  = dev_attr_store,
 62};
```

实际上就是为一些函数指针指定了对应的调用函数。

定义了这个变量之后，但相当于定义了一个子系统，名字就是 devices，对应于/sys 下面的 devices 目录。需要注意的是，在 2.6 内核的后期版本中，去掉了 struct subsystem 结构，直接用一个 struct kset 结构来代替，这是因为本质上二者其实是一个东西，去掉这么一个名词会让人更容易理解。之后，我们便可以调用 subsystem_register()来对该子系统进行注册，但目前我们关注的重点不在这里，所以对 subsystem 的讨论暂且到这里。我们只需要记住，我们已经在内核中注册进了一个子系统，子系统的名字就是"devices",对应的变量为 devices_subsys。

现在继续回到我们的 device_initialize()函数， 229 行的 kobj_set_kset_s() 是一个宏，如下所示：

```
// include/linux/kobject.h:
197/**
198 *      kobj_set_kset_s(obj,subsys) - set kset for embedded kobject.
199 *      @obj:           ptr to some object type.
200 *      @subsys:        a subsystem object (not a ptr).
201 *
202 *      Can be used for any object type with an embedded ->kobj.
203 */
204
205#define kobj_set_kset_s(obj,subsys) \
206        (obj)->kobj.kset = &(subsys).kset
116struct kset {
117        struct subsystem        * subsys;
```

```
118        struct kobj_type        * ktype;
119        struct list_head        list;
120        spinlock_t              list_lock;
121        struct kobject          kobj;
122        struct kset_uevent_ops  * uevent_ops;
123};
```

因此， kobj_set_kset_s(dev, devices_subsys)的作用实际上就是指定 dev 所指向的设备所属的 kobject 的 kset 为 devices_subsys 这个子系统（其实就是一个 kset）。

紧接着 230 行初始化 struct device 的 kobj 成员，相关定义如下：

```
// include/linux/kobject.h:
 50 struct kobject {
 51        const char              * k_name;
 52        char                    name[KOBJ_NAME_LEN];
 53        struct kref             kref;
 54        struct list_head        entry;
 55        struct kobject          * parent;
 56        struct kset             * kset;
 57        struct kobj_type        * ktype;
 58        struct dentry           * dentry;
 59};
 85 struct kobj_type {
 86        void (*release)(struct kobject *);
 87        struct sysfs_ops        * sysfs_ops;
 88        struct attribute        ** default_attrs;
 89};
//  lib/kobject.c
123/**
124 *      kobject_init - initialize object.
125 *      @kobj:  object in question.
126 */
127 void kobject_init(struct kobject * kobj)
128{
129        kref_init(&kobj->kref);
130        INIT_LIST_HEAD(&kobj->entry);
131        kobj->kset = kset_get(kobj->kset);
132}
329/**
330 *      kobject_get - increment refcount for object.
331 *      @kobj:  object.
332 */
333
334struct kobject * kobject_get(struct kobject * kobj)
335{
336        if (kobj)
```

```
337             kref_get(&kobj->kref);
338     return kobj;
339}
```

struct kref 是内核中用来计数的一个结构，它的操作具有原子性。kref_init()将对应的 object 的引用计数设置为 1。INIT_LIST_HEAD(&kobj->entry)将 kobj 的 entry 的 prev 和 next 指针都指向 entry，最后来设置 kobj 的 kset 成员，首先来看用到的几个函数，如下所示：

```
// include/linux/kobject.h:
131static inline struct kset * to_kset(struct kobject * kobj)
132{
133     return kobj ? container_of(kobj,struct kset,kobj) : NULL;
134}
135
136static inline struct kset * kset_get(struct kset * k)
137{
138     return k ? to_kset(kobject_get(&k->kobj)) : NULL;
139}
```

由上可知，LCD 对应的 struct device 中的 kobj 成员所指向的 kset 在之前已经设置为指向 devices_subsys 子系统对应的 kset。所以将会执行 kset_get()中的 to_kset(kobject_get(&k->kobj))

这个函数首先使 kset 对应的 kobject 的引用计数增 1，也就是 devices_subsys 子系统对应的 kset 所对应的 kobject 的引用计数增 1，然后再将我们的 LCD 对应的 struct device 中的 kobj 成员所指向的 kset 指向刚才的 kset,绕了这么多，实际上也很简单，就是将 kset 的引用计数增 1。

再来看 device_initialize() 里面的 231 - 232 行，即 klist_init(&dev->klist_children, klist_children_get,klist_children_put);

struct klist 是对 struct list_head 的一个包装，功能更强大一些：

```
// include/linux/klist.h
 21struct klist {
 22     spinlock_t            k_lock;
 23     struct list_head      k_list;
 24     void                  (*get)(struct klist_node *);
 25     void                  (*put)(struct klist_node *);
 26};
 32struct klist_node {
 33     struct klist          * n_klist;
 34     struct list_head      n_node;
 35     struct kref           n_ref;
 36     struct completion     n_removed;
 37};
// lib/klist.c
 42/**
 43 *     klist_init - Initialize a klist structure.
 44 *     @k:     The klist we're initializing.
```

```
 45 *      @get:    The get function for the embedding object (NULL if none)
 46 *      @put:    The put function for the embedding object (NULL if none)
 47 *
 48 * Initialises the klist structure.  If the klist_node structures are
 49 * going to be embedded in refcounted objects (necessary for safe
 50 * deletion) then the get/put arguments are used to initialise
 51 * functions that take and release references on the embedding
 52 * objects.
 53 */
 54
 55void klist_init(struct klist * k, void (*get)(struct klist_node *),
 56               void (*put)(struct klist_node *))
 57{
 58       INIT_LIST_HEAD(&k->k_list);
 59       spin_lock_init(&k->k_lock);
 60       k->get = get;
 61       k->put = put;
 62}
```

到此，我们可以看出 231 -232 行的功能其实也很简单，就是初始化 struct device 里面的 klist_children 的 k_list 链表，设置 klist_children 的 get 函数为 klist_children_get(),put 函数为 klist_children_put(),而这两个函数也很简单，如下

```
// drivers/base/core.c
201static void klist_children_get(struct klist_node *n)
202{
203       struct device *dev = container_of(n, struct device, knode_parent);
204
205       get_device(dev);
206}
207
208static void klist_children_put(struct klist_node *n)
209{
210       struct device *dev = container_of(n, struct device, knode_parent);
211
212       put_device(dev);
213}
322/**
323 *      get_device - increment reference count for device.
324 *      @dev:    device.
325 *
326 *      This simply forwards the call to kobject_get(), though
327 *      we do take care to provide for the case that we get a NULL
328 *      pointer passed in.
329 */
330
```

```
331struct device * get_device(struct device * dev)
332{
333       return dev ? to_dev(kobject_get(&dev->kobj)) : NULL;
334}
335
336
337/**
338 *      put_device - decrement reference count.
339 *      @dev:    device in question.
340 */
341void put_device(struct device * dev)
342{
343       if (dev)
344              kobject_put(&dev->kobj);
345}
 31#define to_dev(obj) container_of(obj, struct device, kobj)
```

从上可以看出，put 和 get 函数也只是简单地减少或增加设备对就的 kobject 的引用计数。

再回到 device_initialize()函数里面，233 行初始化设备的 dma_pools 链表，234 行初始化结构体里面包含的信号量，235 行 device_init_wakeup(dev, 0)实际上是一个宏：

```
// include/linux/pm.h
190#ifdef CONFIG_PM
...
193#define device_set_wakeup_enable(dev,val) \
194       ((dev)->power.should_wakeup = !!(val))
........
201#else /* !CONFIG_PM */
........
208#define device_set_wakeup_enable(dev,val)       do{}while(0)
........
221#endif
223/* changes to device_may_wakeup take effect on the next pm state change.
224 * by default, devices should wakeup if they can.
225 */
226#define device_can_wakeup(dev) \
227       ((dev)->power.can_wakeup)
228#define device_init_wakeup(dev,val) \
229       do { \
230              device_can_wakeup(dev) = !!(val); \
231              device_set_wakeup_enable(dev,val); \
232       } while(0)
233
```

因此，如果有电源管理单元的话，!! 作用不清楚，暂且跳过。至此，device_initialize()函数总算分析完毕，这还只是 device_register()的 top-half，OH GOD！

在接下来的 platform_device_add()函数里面，将 struct device 的 parent 成员指针指向 platform_bus,这是一个 device 类型的变量，它的定义如下：

```
// drivers/base/platform.c:
 26struct device platform_bus = {
 27        .bus_id         = "platform",
 28};
```

紧接着，设置设备的总线类型,将 struct device 的 bus 成员指针指向 platform_bus_type

```
// drivers/base/platform.c:
892struct bus_type platform_bus_type = {
893        .name           = "platform",
894        .dev_attrs      = platform_dev_attrs,
895        .match          = platform_match,
896        .uevent         = platform_uevent,
897        .pm             = PLATFORM_PM_OPS_PTR,
898};
```

紧接着，设置设备的 bus_id，为"pxa2xx-fb"，在申请了相应的资源后，调用 device_add() 来向系统中添加设备，这应该算是 device_register() 的 bottom-half,^-^!!下面来看它的定义：

```
// drivers/base/core.c
238/**
239 *      device_add - add device to device hierarchy.
240 *      @dev:   device.
241 *
242 *      This is part 2 of device_register(), though may be called
243 *      separately _iff_ device_initialize() has been called separately.
244 *
245 *      This adds it to the kobject hierarchy via kobject_add(), adds it
246 *      to the global and sibling lists for the device, then
247 *      adds it to the other relevant subsystems of the driver model.
248 */
249int device_add(struct device *dev)
250{
251        struct device *parent = NULL;
252        int error = -EINVAL;
253
254        dev = get_device(dev);
255        if (!dev || !strlen(dev->bus_id))
256                goto Error;
257
258        parent = get_device(dev->parent);
259
260        pr_debug("DEV: registering device: ID = '%s'\n", dev->bus_id);
261
```

```
262       /* first, register with generic layer. */
263       kobject_set_name(&dev->kobj, "%s", dev->bus_id);
264       if (parent)
265             dev->kobj.parent = &parent->kobj;
266
267       if ((error = kobject_add(&dev->kobj)))
268             goto Error;
269
270       dev->uevent_attr.attr.name = "uevent";
271       dev->uevent_attr.attr.mode = S_IWUSR;
272       if (dev->driver)
273             dev->uevent_attr.attr.owner = dev->driver->owner;
274       dev->uevent_attr.store = store_uevent;
275       device_create_file(dev, &dev->uevent_attr);
276
277       kobject_uevent(&dev->kobj, KOBJ_ADD);
278       if ((error = device_pm_add(dev)))
279             goto PMError;
280       if ((error = bus_add_device(dev)))
281             goto BusError;
282       if (parent)
283             klist_add_tail(&dev->knode_parent, &parent->klist_children);
284
285       /* notify platform of device entry */
286       if (platform_notify)
287             platform_notify(dev);
288 Done:
289       put_device(dev);
290       return error;
291 BusError:
292       device_pm_remove(dev);
293 PMError:
294       kobject_uevent(&dev->kobj, KOBJ_REMOVE);
295       kobject_del(&dev->kobj);
296 Error:
297       if (parent)
298             put_device(parent);
299       goto Done;
300}
```

254 行的 get_device()前面我们已经见过，就是增加 device 对应的 kobject 的引用计数。

258 行增加它的父 device 对应的 kobject 的引用计数。

263 行设置 device 对应的 kobject 的名字为"pxa2xx-fb"

264 - 265 行将 device 对应的 kobject 的 parent 指针指向父 device 对应的 kobject

267 - 268 行调用 kobject_add()来向系统中添加 device 对应的 kobject

270 - 274 行设置 device 的 uevent_attr 成员，这是一个 struct device_attribute 结构，它的定义如下：

```
// include/linux/device.h:
291/* interface for exporting device attributes */
292 struct device_attribute {
293       struct attribute       attr;
294       ssize_t (*show)(struct device *dev, struct device_attribute *attr,
295                       char *buf);
296       ssize_t (*store)(struct device *dev, struct device_attribute *attr,
297                        const char *buf, size_t count);
298};
// include/linux/sysfs.h:
 18struct attribute {
 19       const char             * name;
 20       struct module           * owner;
 21       mode_t                 mode;
 22};
// drivers/base/core.c
157static ssize_t store_uevent(struct device *dev, struct device_attribute *attr,
158                            const char *buf, size_t count)
159{
160       kobject_uevent(&dev->kobj, KOBJ_ADD);
161       return count;
162}
```

275 行用来在 sys 中创建相应的文件和目录，如下所示：

```
// drivers/base/core.c
171/**
172 *      device_create_file - create sysfs attribute file for device.
173 *      @dev:    device.
174 *      @attr:  device attribute descriptor.
175 */
176
177int device_create_file(struct device * dev, struct device_attribute * attr)
178{
179       int error = 0;
180       if (get_device(dev)) {
181              error = sysfs_create_file(&dev->kobj, &attr->attr);
182              put_device(dev);
183       }
184       return error;
185}
```

277 行用 kobject_uevent()来添加设备。

278 - 279 行调用 device_pm_add()，它是关于设备的电源管理方面的，在此只需要

知道它的用途就行了，详细情况我们以后再考虑。

280 - 281 行调用 bus_add_device(),它的定义如下：

```
// drivers/base/bus.c:
355/**
356 *      bus_add_device - add device to bus
357 *      @dev:    device being added
358 *
359 *      - Add the device to its bus's list of devices.
360 *      - Try to attach to driver.
361 *      - Create link to device's physical location.
362 */
363int bus_add_device(struct device * dev)
364{
365       struct bus_type * bus = get_bus(dev->bus);
366       int error = 0;
367
368       if (bus) {
369                pr_debug("bus %s: add device %s\n", bus->name,
dev->bus_id);
370             device_attach(dev);
371             klist_add_tail(&dev->knode_bus, &bus->klist_devices);
372             error = device_add_attrs(bus, dev);
373             if (!error) {
374                      sysfs_create_link(&bus->devices.kobj, &dev->kobj,
dev->bus_id);
375                                              sysfs_create_link(&dev->kobj,
&dev->bus->subsys.kset.kobj, "bus");
376             }
377       }
378       return error;
379}
```

365 行中的 get_bus()定义如下：

```
// drivers/base/bus.c:
540struct bus_type * get_bus(struct bus_type * bus)
541{
542         return bus ? container_of(subsys_get(&bus->subsys), struct
bus_type, subsys) : NULL;
543}
544
```

由上可知，它的功能实际上是先将 bus 所指向的总线所属的 subsys 引用计数增 1，然后再返回指向 bus 的指针。

由前面的分析可知，在 platform_device_add()函数中，我们已经将 device 的 bus 成员指向了 platform_bus_type，所以 bus 指针不为 NULL。

下面看 370 行的 device_attach(),它的定义如下：

```
// drivers/base/dd.c
123/**
124 *      device_attach - try to attach device to a driver.
125 *      @dev:   device.
126 *
127 *      Walk the list of drivers that the bus has and call
128 *      driver_probe_device() for each pair. If a compatible
129 *      pair is found, break out and return.
130 *
131 *      Returns 1 if the device was bound to a driver;
132 *      0 if no matching device was found; error code otherwise.
133 *
134 *      When called for a USB interface, @dev->parent->sem must be held.
135 */
136int device_attach(struct device * dev)
137{
138       int ret = 0;
139
140       down(&dev->sem);
141       if (dev->driver) {
142              device_bind_driver(dev);
143              ret = 1;
144       } else
145                     ret   =   bus_for_each_drv(dev->bus,   NULL,   dev,
__device_attach);
146       up(&dev->sem);
147       return ret;
148}
```

回顾整个过程，我们还没有给 device 的 driver 成员赋值，所以 driver 指针应该指向 NULL ， 因 此 这 里 会 调 用 bus_for_each_drv(dev->bus, NULL, dev, __device_attach)，它的定义如下：

```
// drivers/base/bus.c:
279static struct device_driver * next_driver(struct klist_iter * i)
280{
281       struct klist_node * n = klist_next(i);
282       return n ? container_of(n, struct device_driver, knode_bus) : NULL;
283}
284
285/**
286 *      bus_for_each_drv - driver iterator
287 *      @bus:   bus we're dealing with.
288 *      @start: driver to start iterating on.
289 *      @data:  data to pass to the callback.
```

```
290 *      @fn:     function to call for each driver.
291 *
292 *      This is nearly identical to the device iterator above.
293 *      We iterate over each driver that belongs to @bus, and call
294 *      @fn for each. If @fn returns anything but 0, we break out
295 *      and return it. If @start is not NULL, we use it as the head
296 *      of the list.
297 *
298 *      NOTE: we don't return the driver that returns a non-zero
299 *      value, nor do we leave the reference count incremented for that
300 *      driver. If the caller needs to know that info, it must set it
301 *      in the callback. It must also be sure to increment the refcount
302 *      so it doesn't disappear before returning to the caller.
303 */
304
305int bus_for_each_drv(struct bus_type * bus, struct device_driver * start,
306                     void * data, int (*fn)(struct device_driver *, void *))
307{
308       struct klist_iter i;
309       struct device_driver * drv;
310       int error = 0;
311
312       if (!bus)
313              return -EINVAL;
314
315       klist_iter_init_node(&bus->klist_drivers, &i,
316                            start ? &start->knode_bus : NULL);
317       while ((drv = next_driver(&i)) && !error)
318              error = fn(drv, data);
319       klist_iter_exit(&i);
320       return error;
321}
```

315 - 316 行初始化一个 klist_iter 结构

317 - 318 行依次从上面的驱动链表中取出一个 driver 然后调用相应的回调函数。

结合我们当前的情景，实际上就是从 platform_bus_type 下面的驱动链中依次取出一个 driver，然后调用__device_attach()函数，而该函数的定义如下：

```
// drivers/base/dd.c
 54/**
 55 *      driver_probe_device - attempt to bind device & driver.
 56 *      @drv:    driver.
 57 *      @dev:    device.
 58 *
 59 *      First, we call the bus's match function, if one present, which
 60 *      should compare the device IDs the driver supports with the
```

```
 61 *      device IDs of the device. Note we don't do this ourselves
 62 *      because we don't know the format of the ID structures, nor what
 63 *      is to be considered a match and what is not.
 64 *
 65 *      This function returns 1 if a match is found, an error if one
 66 *      occurs (that is not -ENODEV or -ENXIO), and 0 otherwise.
 67 *
 68 *      This function must be called with @dev->sem held.  When called
 69 *      for a USB interface, @dev->parent->sem must be held as well.
 70 */
 71int driver_probe_device(struct device_driver * drv, struct device * dev)
 72{
 73       int ret = 0;
 74
 75       if (drv->bus->match && !drv->bus->match(dev, drv))
 76              goto Done;
 77
 78       pr_debug("%s: Matched Device %s with Driver %s\n",
 79              drv->bus->name, dev->bus_id, drv->name);
 80       dev->driver = drv;
 81       if (dev->bus->probe) {
 82              ret = dev->bus->probe(dev);
 83              if (ret) {
 84                     dev->driver = NULL;
 85                     goto ProbeFailed;
 86              }
 87       } else if (drv->probe) {
 88              ret = drv->probe(dev);
 89              if (ret) {
 90                     dev->driver = NULL;
 91                     goto ProbeFailed;
 92              }
 93       }
 94       device_bind_driver(dev);
 95       ret = 1;
 96       pr_debug("%s: Bound Device %s to Driver %s\n",
 97              drv->bus->name, dev->bus_id, drv->name);
 98       goto Done;
 99
100 ProbeFailed:
101       if (ret == -ENODEV || ret == -ENXIO) {
102              /* Driver matched, but didn't support device
103               * or device not found.
104               * Not an error; keep going.
```

```
105                 */
106                 ret = 0;
107         } else {
108                 /* driver matched but the probe failed */
109                 printk(KERN_WARNING
110                       "%s: probe of %s failed with error %d\n",
111                       drv->name, dev->bus_id, ret);
112         }
113 Done:
114         return ret;
115}
116
117static int __device_attach(struct device_driver * drv, void * data)
118{
119         struct device * dev = data;
120         return driver_probe_device(drv, dev);
121}
```

根据函数开头那部分的说明，我们首先检测 bus 的 match 函数是否存在，如果存在的话则调用它来检测驱动支持的设备 ID 与设备的 ID 是否匹配。说明里面还提到我们不能自己进行比较，因为我们自己不清楚 ID 结构的格式等，也不知道什么算是匹配，什么算是不匹配。

如果找到匹配的设备与驱动的话，将 device 的 driver 指针指向该驱动。

81 - 86 行检测 bus 的 probe（）函数是否存在，如果存在的话，则调用它来检测我们的驱动是否合适，如果不合适的话则报错

87 - 93 行 如果 bus 的 probe()函数不存在，则检测 driver 中的 probe()函数是否存在，如果存在的话，则调用它来检测设备，同样如果不符号的话将报错。

94 行调用 device_bind_driver()函数进行设备与驱动的绑定。下面来看它的定义：

```
// drivers/base/dd.c
 27/**
 28 *      device_bind_driver - bind a driver to one device.
 29 *      @dev:    device.
 30 *
 31 *      Allow manual attachment of a driver to a device.
 32 *      Caller must have already set @dev->driver.
 33 *
 34 *      Note that this does not modify the bus reference count
 35 *      nor take the bus's rwsem. Please verify those are accounted
 36 *      for before calling this. (It is ok to call with no other effort
 37 *      from a driver's probe() method.)
 38 *
 39 *      This function must be called with @dev->sem held.
 40 */
 41void device_bind_driver(struct device * dev)
 42{
```

```
 43        if (klist_node_attached(&dev->knode_driver))
 44                return;
 45
 46        pr_debug("bound device '%s' to driver '%s'\n",
 47                 dev->bus_id, dev->driver->name);
 48        klist_add_tail(&dev->knode_driver, &dev->driver->klist_devices);
 49        sysfs_create_link(&dev->driver->kobj, &dev->kobj,
 50                          kobject_name(&dev->kobj));
 51        sysfs_create_link(&dev->kobj, &dev->driver->kobj, "driver");
 52}
// lib/klist.c:
174/**
175 *      klist_node_attached - Say whether a node is bound to a list or not.
176 *      @n:     Node that we're testing.
177 */
178
179int klist_node_attached(struct klist_node * n)
180{
181        return (n->n_klist != NULL);
182}
```

43 - 44 行，如果我们的设备的驱动链表不为空的话，则返回

48 行，将 dev->knode_driver 加到设备的驱动支持的设备链表中。

49 - 50 行，调用 sysfs_create_link()函数在 dev->driver->kobj 对应的目录里面（pxa2xx-fb 目录？）创建链接，指向 dev->kobj 对应的目录（pxa2xx-fb 目录？）,链接名为

51 行，调用 sysfs_create_link()函数在 dev->kobj 对应的目录里面（pxa2xx-fb 目录？）创建链接，指向 dev->driver->kobj 对应的目录（pxa2xx-fb 目录？）,链接名为"driver"

至些，设备与驱动的绑定 device_attach()完毕。我们再回到 bus_add_device()函数里面，我们再把该函数列一下：

```
// drivers/base/bus.c:
355/**
356 *      bus_add_device - add device to bus
357 *      @dev:   device being added
358 *
359 *      - Add the device to its bus's list of devices.
360 *      - Try to attach to driver.
361 *      - Create link to device's physical location.
362 */
363int bus_add_device(struct device * dev)
364{
365        struct bus_type * bus = get_bus(dev->bus);
366        int error = 0;
367
```

```
368       if (bus) {
369                 pr_debug("bus %s: add device %s\n", bus->name,
dev->bus_id);
370             device_attach(dev);
371             klist_add_tail(&dev->knode_bus, &bus->klist_devices);
372             error = device_add_attrs(bus, dev);
373             if (!error) {
374                       sysfs_create_link(&bus->devices.kobj, &dev->kobj,
dev->bus_id);
375                                               sysfs_create_link(&dev->kobj,
&dev->bus->subsys.kset.kobj, "bus");
376             }
377       }
378       return error;
379}
```

371 行，将 dev->knode_bus，添加到总线支持的设备列表 bus->klist_deives 中。

372 行，调用 device_add_attrs()函数添加设备属性，其定义如下所示：

```
// drivers/base/bus.c:
323static int device_add_attrs(struct bus_type * bus, struct device * dev)
324{
325       int error = 0;
326       int i;
327
328       if (bus->dev_attrs) {
329             for (i = 0; attr_name(bus->dev_attrs); i++) {
330                   error = device_create_file(dev,&bus->dev_attrs);
331                   if (error)
332                         goto Err;
333             }
334       }
335 Done:
336       return error;
337 Err:
338       while (--i >= 0)
339             device_remove_file(dev,&bus->dev_attrs);
340       goto Done;
341}
```

实际上就是将设备的属性一一创建一个属性文件反映在 sys 中。

374 行，在 bus->devices.kobj 对应的目录里面（platform 目录？）创建链接，指向 dev->kobj 对应的目录（pxa2xx-fb 目录？），链接名为 dev->bus_id，即"pxa2xx-fb"。

372 行，在 dev->kobj 对应的目录（pxa2xx-fb 目录？）里面创建链接，指向 dev->bus->subsys.kset.kobj 对应的目录（platform 目录？），链接名为"bus"。

再回到 device_add()函数里面，

```
// drivers/base/core.c
282        if (parent)
283                klist_add_tail(&dev->knode_parent, &parent->klist_children);
285        /* notify platform of device entry */
286        if (platform_notify)
287                platform_notify(dev);
```

282 - 283 行，如果你指针不为空的话，则将 dev->knode_parent 加到 parent->klist_children。

286 - 287 行，如果 platform_notify 指针不为空的话，则调用它，它的定义如下：

```
// drivers/base/core.c
int (*platform_notify)(struct device * dev) = NULL;
```

至此，device_add()函数分析完毕，device_initialize()和 device_add()函数组合成 device_register()，它们分别被称为上半部与下半部。

驱动程序模型-device

http://www.ourkernel.com/bbs/archiver/?tid-55.html

platform_device_register()注册过程

——

```
/* arch/arm/mach-s3c2410/mach-smdk2410.c */
struct platform_device s3c_device_i2c = {
	.name		= "s3c2410-i2c",
	.id		= -1,
	.num_resources	= ARRAY_SIZE(s3c_i2c_resource),
	.resource	= s3c_i2c_resource,
};

/*
* platform_device_register - add a platform-level device
* @pdev: platform device we're adding
*
*/
int platform_device_register(struct platform_device * pdev)
{
	device_initialize(&pdev->dev);		//初始化设备结构
	return platform_device_add(pdev); //添加一个片上的设备到设备层
}


/**
 * platform_device_add - add a platform device to device hierarchy
 * @pdev: platform device we're adding
 *
 * This is part 2 of platform_device_register(), though may be called
 * separately _iff_ pdev was allocated by platform_device_alloc().
 */
int platform_device_add(struct platform_device *pdev)
{
	int i, ret = 0;

	if (!pdev)
		return -EINVAL;

	if (!pdev->dev.parent)
		pdev->dev.parent = &platform_bus;

	pdev->dev.bus = &platform_bus_type;
```

```
    if (pdev->id != -1)
        snprintf(pdev->dev.bus_id, BUS_ID_SIZE, "%s.%d", pdev->name,
                 pdev->id);    /* 若支持同类多个设备，则用 pdev->name 和 pdev->id 在总线上标识该设备  */
    else
        strlcpy(pdev->dev.bus_id, pdev->name, BUS_ID_SIZE);    /* 否则，用 pdev->name(如"s3c2410-i2c")在总线上标识该设备 */

    for (i = 0; i < pdev->num_resources; i++) {  /*   遍历资源数，并为各自在总线地址空间请求分配 */
        struct resource *p, *r = &pdev->resource[i];

        if (r->name == NULL)
            r->name = pdev->dev.bus_id;

        p = r->parent;
        if (!p) {
            if (r->flags & IORESOURCE_MEM)
                p = &iomem_resource;  /*   作为 IO 内存资源分配    */
            else if (r->flags & IORESOURCE_IO)
                p = &ioport_resource;   /*   作为 IO Port 资源分配    */
        }

        if (p && insert_resource(p, r)) {   /*    将新的 resource 插入内核 resource tree */
            printk(KERN_ERR
                   "%s: failed to claim resource %d\n",
                   pdev->dev.bus_id, i);
            ret = -EBUSY;
            goto failed;
        }
    }

    pr_debug("Registering platform device '%s'. Parent at %s\n",
             pdev->dev.bus_id, pdev->dev.parent->bus_id);

    ret = device_add(&pdev->dev);
    if (ret == 0)
        return ret;

 failed:
    while (--i >= 0)
        if (pdev->resource[i].flags & (IORESOURCE_MEM|IORESOURCE_IO))
            release_resource(&pdev->resource[i]);
    return ret;
```

}

这里发现，添加 device 到内核最终还是调用的 device_add 函数。Platform_device_add 和 device_add 最主要的区别是多了一步 insert_resource(p, r)即将 platform 资源(resource)添加进内核，由内核统一管理。

platform_driver_register()注册过程

－－

```
static struct platform_driver s3c2410_i2c_driver = {
        .probe          = s3c24xx_i2c_probe,
        .remove         = s3c24xx_i2c_remove,
        .resume         = s3c24xx_i2c_resume,
        .driver         = {
                .owner  = THIS_MODULE,
                .name   = "s3c2410-i2c",
        },
};
```

platform_driver_register(&s3c2410fb_driver)----->
driver_register(&drv->driver)----->
bus_add_driver(drv)----->
driver_attach(drv)----->
bus_for_each_dev(drv->bus, NULL, drv, __driver_attach)----->
__driver_attach(struct device * dev, void * data)----->
driver_probe_device(drv, dev)----->
really_probe(dev, drv)----->
在 really_probe()中：为设备指派管理该设备的驱动：dev->driver = drv, 调用 probe()函数初始化设备：drv->probe(dev)

注：Platform_device和Platform_driver的使用请参考这篇文章:
《Linux Platform Device and Driver》
http://blog.chinaunix.net/u2/60011/showart.php?id=1018502

从 Linux 2.6 起引入了一套新的驱动管理和注册机制:Platform_device 和 Platform_driver。
Linux 中大部分的设备驱动，都可以使用这套机制，设备用 Platform_device 表示，驱动用 Platform_driver 进行注册。

Linux platform driver 机制和传统的 device driver 机制(通过 driver_register 函数进行注册)相比，一个十分明显的优势在于 platform 机制将设备本身的资源注册进内核，由内核统一管理，在驱动程序中使用这些资源时通过 platform device 提供的标准接口进行申请并使用。这样提高了驱动和资源管理的独立性，并且拥有较好的可移植性和安全性(这些标准接口是安全的)。

Platform 机制的本身使用并不复杂，由两部分组成：platform_device 和 platfrom_driver。
通过 Platform 机制开发发底层驱动的大致流程为： 定义 platform_device → 注册 platform_device → 定义 platform_driver → 注册 platform_driver。

首先要确认的就是设备的资源信息，例如设备的地址，中断号等。
在 2.6 内核中 platform 设备用结构体 platform_device 来描述，该结构体定义在 kernel\include\linux\platform_device.h 中，

```
struct platform_device {
 const char * name;
 u32  id;
 struct device dev;
 u32  num_resources;
 struct resource * resource;
};
```

该结构一个重要的元素是 resource，该元素存入了最为重要的设备资源信息，定义在 kernel\include\linux\ioport.h 中，

```
struct resource {
 const char *name;
 unsigned long start, end;
 unsigned long flags;
 struct resource *parent, *sibling, *child;
};
```

下面举 s3c2410 平台的 i2c 驱动作为例子来说明：

```
/* arch/arm/mach-s3c2410/devs.c */
/* I2C */
```

```
static struct resource s3c_i2c_resource[] = {
        [0] = {
                .start = S3C24XX_PA_IIC,
                .end = S3C24XX_PA_IIC + S3C24XX_SZ_IIC - 1,
                .flags = IORESOURCE_MEM,
        },
        [1] = {
                .start = IRQ_IIC, //S3C2410_IRQ(27)
                .end = IRQ_IIC,
                .flags = IORESOURCE_IRQ,
        }
};
```

这里定义了两组 resource，它描述了一个 I2C 设备的资源，第 1 组描述了这个 I2C 设备所占用的总线地址范围，IORESOURCE_MEM 表示第 1 组描述的是内存类型的资源信息，第 2 组描述了这个 I2C 设备的中断号，IORESOURCE_IRQ 表示第 2 组描述的是中断资源信息。设备驱动会根据 flags 来获取相应的资源信息。

有了 resource 信息，就可以定义 platform_device 了：

```
struct platform_device s3c_device_i2c = {
        .name = "s3c2410-i2c",
        .id = -1,
        .num_resources = ARRAY_SIZE(s3c_i2c_resource),
        .resource = s3c_i2c_resource,
};
```

定义好了 platform_device 结构体后就可以调用函数 platform_add_devices 向系统中添加该设备了，之后可以调用 platform_driver_register()进行设备注册。要注意的是，这里的 platform_device 设备的注册过程必须在相应设备驱动加载之前被调用，即执行 platform_driver_register 之前，原因是因为驱动注册时需要匹配内核中所以已注册的设备名。

s3c2410-i2c 的 platform_device 是在系统启动时，在 cpu.c 里的 s3c_arch_init()函数里进行注册的，这个函数申明为 arch_initcall(s3c_arch_init);会在系统初始化阶段被调用。arch_initcall 的优先级高于 module_init。所以会在 Platform 驱动注册之前调用。(详细参考 include/linux/init.h)

s3c_arch_init 函数如下：

```
/* arch/arm/mach-3sc2410/cpu.c */
static int __init s3c_arch_init(void)
{
    int ret;
    ……
/* 这里 board 指针指向在 mach-smdk2410.c 里的定义的 smdk2410_board，
里面包含了预先定义的 I2C Platform_device 等. */
    if (board != NULL) {
        struct platform_device **ptr = board->devices;
        int i;

        for (i = 0; i < board->devices_count; i++, ptr++) {
            ret = platform_device_register(*ptr);    //在这里进行
注册

            if (ret) {
                printk(KERN_ERR "s3c24xx: failed to add board
device %s (%d) @%p\n", (*ptr)->name,
ret, *ptr);
            }
        }
        /* mask any error, we may not need all these board
         * devices */
        ret = 0;
```

```
    }
    return ret;
}
```

同时被注册还有很多其他平台的 platform_device，详细查看 arch/arm/mach-s3c2410/mach-smdk2410.c 里的 smdk2410_devices 结构体。

驱动程序需要实现结构体 struct platform_driver，参考 drivers/i2c/busses

```
/* device driver for platform bus bits */


static struct platform_driver s3c2410_i2c_driver = {
        .probe = s3c24xx_i2c_probe,
        .remove = s3c24xx_i2c_remove,
        .resume = s3c24xx_i2c_resume,
        .driver = {
                .owner = THIS_MODULE,
                .name = "s3c2410-i2c",
        },
};
```

在驱动初始化函数中调用函数 platform_driver_register()注册 platform_driver，需要注意的是 s3c_device_i2c 结构中 name 元素和 s3c2410_i2c_driver 结构中 driver.name 必须是相同的，这样在 platform_driver_register()注册时会对所有已注册的所有 platform_device 中的 name 和当前注册的 platform_driver 的 driver.name 进行比较，只有找到相同的名称的 platfomr_device 才能注册成功，当注册成功时会调用 platform_driver 结构元素 probe 函数指针，这里就是 s3c24xx_i2c_probe，当进入 probe 函数后，需要获取设备的资源信息，常用获取资源的函数主要是：
struct resource * platform_get_resource(struct platform_device *dev, unsigned int type, unsigned int num);
根据参数 type 所指定类型，例如 IORESOURCE_MEM，来获取指定的资源。

struct int platform_get_irq(struct platform_device *dev, unsigned int num);
获取资源中的中断号。

下面举 s3c24xx_i2c_probe 函数分析,看看这些接口是怎么用的。
前面已经讲了，s3c2410_i2c_driver 注册成功后会调用 s3c24xx_i2c_probe 执行，下面看代码：

```
/* drivers/i2c/busses/i2c-s3c2410.c */

static int s3c24xx_i2c_probe(struct platform_device *pdev)
{
    struct s3c24xx_i2c *i2c = &s3c24xx_i2c;
    struct resource *res;
    int ret;

    /* find the clock and enable it */

    i2c->dev = &pdev->dev;
    i2c->clk = clk_get(&pdev->dev, "i2c");
    if (IS_ERR(i2c->clk)) {
     dev_err(&pdev->dev, "cannot get clock\n");
     ret = -ENOENT;
     goto out;
    }

    dev_dbg(&pdev->dev, "clock source %p\n", i2c->clk);
    clk_enable(i2c->clk);

    /* map the registers */
    res = platform_get_resource(pdev, IORESOURCE_MEM, 0); /* 获取
设备的 IO 资源地址 */
```

```
    if (res == NULL) {
     dev_err(&pdev->dev, "cannot find IO resource\n");
     ret = -ENOENT;
     goto out;
    }

    i2c->ioarea = request_mem_region(res->start,
(res->end-res->start)+1, pdev->name); /* 申请这块 IO Region */

    if (i2c->ioarea == NULL) {
     dev_err(&pdev->dev, "cannot request IO\n");
     ret = -ENXIO;
     goto out;
    }

    i2c->regs = ioremap(res->start, (res->end-res->start)+1); /*
映射至内核虚拟空间 */

    if (i2c->regs == NULL) {
     dev_err(&pdev->dev, "cannot map IO\n");
     ret = -ENXIO;
     goto out;
    }

    dev_dbg(&pdev->dev, "registers %p (%p, %p)\n", i2c->regs,
i2c->ioarea, res);

    /* setup info block for the i2c core */
    i2c->adap.algo_data = i2c;
```

```
    i2c->adap.dev.parent = &pdev->dev;

    /* initialise the i2c controller */
    ret = s3c24xx_i2c_init(i2c);
    if (ret != 0)
     goto out;

    /* find the IRQ for this unit (note, this relies on the init call
to ensure no current IRQs pending */

    res = platform_get_resource(pdev, IORESOURCE_IRQ, 0); /* 获取
设备 IRQ 中断号 */

    if (res == NULL) {
     dev_err(&pdev->dev, "cannot find IRQ\n");
     ret = -ENOENT;
     goto out;
    }

    ret = request_irq(res->start, s3c24xx_i2c_irq, IRQF_DISABLED,
/* 申请 IRQ */
     pdev->name, i2c);

    ......

    return ret;

}
```

小思考：

那什么情况可以使用 platform driver 机制编写驱动呢？
我的理解是只要和内核本身运行依赖性不大的外围设备(换句话说只要不在内核运行所需的一个最小系统之内的设备),相对独立的,拥有各自独自的资源(addresses and IRQs)，都可以用 platform_driver 实现。如：lcd,usb,uart 等，都可以用 platfrom_driver 写，而 timer,irq 等最小系统之内的设备则最好不用 platfrom_driver 机制，实际上内核实现也是这样的。

参考资料：
linux-2.6.24/Documentation/driver-model/platform.txt

首先介绍一下注册一个驱动的步骤：

1、定义一个 platform_driver 结构

2、初始化这个结构，指定其 probe、remove 等函数，并初始化其中的 driver 变量

3、实现其 probe、remove 等函数

看 platform_driver 结构，定义于 include/linux/platform_device.h 文件中：

```
struct platform_driver {
    int (*probe)(struct platform_device *);
    int (*remove)(struct platform_device *);
    void (*shutdown)(struct platform_device *);
    int (*suspend)(struct platform_device *, pm_message_t state);
    int (*suspend_late)(struct platform_device *, pm_message_t state);
    int (*resume_early)(struct platform_device *);
    int (*resume)(struct platform_device *);
    struct device_driver driver;
};
```

可见，它包含了设备操作的几个功能函数，同样重要的是，它还包含了一个 device_driver 结构。刚才提到了驱动程序中需要初始化这个变量。下面看一下这个变量的定义，位于 include/linux/device.h 中：

```
struct device_driver {
    const char        * name;
    struct bus_type        * bus;
    struct kobject        kobj;
    struct klist        klist_devices;
    struct klist_node     knode_bus;
    struct module        * owner;
    const char         * mod_name;  /* used for built-in modules */
    struct module_kobject     * mkobj;
    int     (*probe)     (struct device * dev);
    int     (*remove)     (struct device * dev);
    void     (*shutdown)     (struct device * dev);
    int     (*suspend)     (struct device * dev, pm_message_t state);
    int     (*resume)     (struct device * dev);
};
```

需要注意这两个变量：name 和 owner。那么的作用主要是为了和相关的 platform_device 关联起来，owner 的作用是说明模块的所有者，驱动程序中一般初始化为 THIS_MODULE。

下面是一个 platform_driver 的初始化实例：

```
static struct platform_driver s3c2410iis_driver = {
  .probe = s3c2410iis_probe,
```

```
  .remove = s3c2410iis_remove,
  .driver = {
    .name = "s3c2410-iis",
    .owner = THIS_MODULE,
  },
};
```

上面的初始化是一个音频驱动的实例。注意其中的 driver 这个结构体，只初始化了其 name 和 owner 两个量。接着看一下和 driver 相关的另一个结构，定义如下：

```
struct platform_device {
    const char     * name;
    int        id;
    struct device     dev;
    u32        num_resources;
    struct resource     * resource;
};
```

该结构中也有一个 name 变量。platform_driver 从字面上来看就知道是设备驱动。设备驱动是为谁服务的呢？当然是设备了。platform_device 就描述了设备对象。下面是一个具体的实例：

```
struct platform_device s3c_device_iis = {
    .name         = "s3c2410-iis",
    .id         = -1,
    .num_resources     = ARRAY_SIZE(s3c_iis_resource),
    .resource     = s3c_iis_resource,
    .dev = {
        .dma_mask = &s3c_device_iis_dmamask,
        .coherent_dma_mask = 0xffffffffUL
    }
};
```

它的 name 变量和刚才上面的 platform_driver 的 name 变量是一致的，内核正是通过这个一致性来为驱动程序找到资源，即 platform_device 中的 resource。这个结构的定义如下，位于 include/linux/ioport.h 中：

```
struct resource {
    resource_size_t start;
    resource_size_t end;
    const char *name;
    unsigned long flags;
    struct resource *parent, *sibling, *child;
};
```

下面是一个具体的实例：

```
static struct resource s3c_iis_resource[] = {
    [0] = {
        .start = S3C24XX_PA_IIS,
        .end = S3C24XX_PA_IIS + S3C24XX_SZ_IIS -1,
        .flags = IORESOURCE_MEM,
    }
```

};

这个结构的作用就是告诉驱动程序设备的起始地址和终止地址和设备的端口类型。这里的地址指的是物理地址。

另外还需要注意 platform_device 中的 device 结构，它详细描述了设备的情况，定义如下：

```
struct device {
    struct klist        klist_children;
    struct klist_node     knode_parent;        /* node in sibling list */
    struct klist_node     knode_driver;
    struct klist_node     knode_bus;
    struct device        *parent;
    struct kobject kobj;
    char     bus_id[BUS_ID_SIZE];     /* position on parent bus */
    struct device_type     *type;
    unsigned        is_registered:1;
    unsigned        uevent_suppress:1;
    struct semaphore     sem;     /* semaphore to synchronize calls to
                 * its driver.
                 */
    struct bus_type     * bus;        /* type of bus device is on */
    struct device_driver *driver;     /* which driver has allocated this
                 device */
    void        *driver_data;     /* data private to the driver */
    void        *platform_data;     /* Platform specific data, device
                 core doesn't touch it */
    struct dev_pm_info     power;
#ifdef CONFIG_NUMA
    int        numa_node;     /* NUMA node this device is close to */
#endif
    u64        *dma_mask;     /* dma mask (if dma'able device) */
    u64        coherent_dma_mask;/* Like dma_mask, but for
                 alloc_coherent mappings as
                 not all hardware supports
                 64 bit addresses for consistent
                 allocations such descriptors. */
    struct list_head     dma_pools;     /* dma pools (if dma'ble) */
    struct dma_coherent_mem     *dma_mem; /* internal for coherent mem
                 override */
    /* arch specific additions */
    struct dev_archdata     archdata;
    spinlock_t        devres_lock;
    struct list_head     devres_head;
    /* class_device migration path */
    struct list_head     node;
    struct class        *class;
```

```
    dev_t           devt;       /* dev_t, creates the sysfs "dev" */
    struct attribute_group    **groups;    /* optional groups */
    void    (*release)(struct device * dev);
};
```

上面把驱动程序中涉及到的主要结构都介绍了，下面主要说一下驱动程序中怎样对这个结构进行处理，以使驱动程序能运行。

相信大家都知道 module_init（）这个宏。驱动模块加载的时候会调用这个宏。它接收一个函数为参数，作为它的参数的函数将会对上面提到的 platform_driver 进行处理。看一个实例：假如这里 module_init 要接收的参数为 s3c2410_uda1341_init 这个函数，下面是这个函数的定义：

```
static int __init s3c2410_uda1341_init(void) {
 memzero(&input_stream, sizeof(audio_stream_t));
 memzero(&output_stream, sizeof(audio_stream_t));
 return platform_driver_register(&s3c2410iis_driver);
}
```

注意函数体的最后一行，它调用的是 platform_driver_register 这个函数。这个函数定义于 driver/base/platform.c 中，原型如下：

int platform_driver_register(struct platform_driver *drv)

它的功能就是为上面提到的 plarform_driver 中的 driver 这个结构中的 probe、remove 这些变量指定功能函数。

到目前为止，内核就已经知道了有这么一个驱动模块。内核启动的时候，就会调用与该驱动相关的 probe 函数。我们来看一下 probe 函数实现了什么功能。

probe 函数的原型为

int xxx_probe(struct platform_device *pdev)

即它的返回类型为 int，接收一个 platform_device 类型的指针作为参数。返回类型就是我们熟悉的错误代码了，而接收的这个参数呢，我们上面已经说过，驱动程序为设备服务，就需要知道设备的信息。而这个参数，就包含了与设备相关的信息。

probe 函数接收到 plarform_device 这个参数后，就需要从中提取出需要的信息。它一般会通过调用内核提供的 platform_get_resource 和 platform_get_irq 等函数来获得相关信息。如通过 platform_get_resource 获得设备的起始地址后，可以对其进行 request_mem_region 和 ioremap 等操作，以便应用程序对其进行操作。通过 platform_get_irq 得到设备的中断号以后，就可以调用 request_irq 函数来向系统申请中断。这些操作在设备驱动程序中一般都要完成。

在完成了上面这些工作和一些其他必须的初始化操作后，就可以向系统注册我们在/dev 目录下能看在的设备文件了。举一个例子，在音频芯片的驱动中，就可以调用 register_sound_dsp 来注册一个 dsp 设备文件，lcd 的驱动中就可以调用 register_framebuffer 来注册 fb 设备文件。这个工作完成以后，系统中就有我们需要的设备文件了。而和设备文件相关的操作都是通过一个 file_operations 来实现的。在调用 register_sound_dsp 等函数的时候，就需要传递一个 file_operations 类型的指针。这个指针就提供了可以供用户空间调用的 write、read 等函数。file_operations 结构的定义位于 include/linux/fs.h 中，列出如下：

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t
*);
    ssize_t (*write) (struct file *, const char __user *, size_t,
loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *,
unsigned long, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *,
unsigned long, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct
*);
    int (*ioctl) (struct inode *, struct file *, unsigned int,
unsigned long);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned
long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned
long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
```

```
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*aio_fsync) (struct kiocb *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t,
loff_t *, int);
    unsigned long (*get_unmapped_area)(struct file *, unsigned
long, unsigned long, unsigned long, unsigned long);
    int (*check_flags)(int);
    int (*dir_notify)(struct file *filp, unsigned long arg);
    int (*flock) (struct file *, int, struct file_lock *);
    ssize_t (*splice_write)(struct pipe_inode_info *, struct file
*, loff_t *, size_t, unsigned int);
    ssize_t (*splice_read)(struct file *, loff_t *, struct
pipe_inode_info *, size_t, unsigned int);
    int (*setlease)(struct file *, long, struct file_lock **);
};
```

到目前为止，probe 函数的功能就完成了。

当用户打开一个设备，并调用其 read、write 等函数的时候，就可以通过上面的 file_operations 来找到相关的函数。所以，用户驱动程序还需要实现这些函数，具体实现和相关的设备有密切的关系，这里就不再介绍了。