[**Linux 设备驱动开发 —— platform设备驱动应用实例解析**](http://blog.csdn.net/zqixiao_09/article/details/50888795)

标签： [驱动开发](http://www.csdn.net/tag/%e9%a9%b1%e5%8a%a8%e5%bc%80%e5%8f%91)[Linux](http://www.csdn.net/tag/Linux)[platfprm设备驱动](http://www.csdn.net/tag/platfprm%e8%ae%be%e5%a4%87%e9%a9%b1%e5%8a%a8)

2016-03-14 19:27 12284人阅读 [评论](http://blog.csdn.net/zqixiao_09/article/details/50888795#comments)(15) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/zqixiao_09/article/details/50888795#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

Linux 驱动开发进阶（18） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg Linux 字符设备驱动开发（19） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg Linux 系统（41） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

       前面我们已经学习了platform设备的理论知识[Linux 设备驱动开发 —— platform 设备驱动](http://blog.csdn.net/zqixiao_09/article/details/50865480) ，下面将通过一个实例来深入我们的学习。

**一、platform 驱动的工作过程**

        platform模型驱动编程，需要实现**platform\_device(设备)**与**platform\_driver（驱动）**在**platform(虚拟总线)**上的**注册、匹配**，相互绑定，然后再做为一个普通的字符设备进行相应的应用，总之如果编写的是基于字符设备的platform驱动，在遵循并实现platform总线上驱动与设备的特定接口的情况下，最核心的还是字符设备的核心结构：cdev、 file\_operations（他包含的操作函数接口）、dev\_t(设备号)、设备文件（/dev）等，因为用platform机制编写的字符驱动，它的本质是字符驱动。

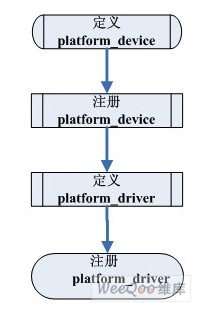
      我们要记住，platform 驱动只是在字符设备驱动外套一层platform\_driver 的外壳。

     在一般情况下，2.6内核中已经初始化并挂载了一条platform总线在sysfs文件系统中。那么我们编写platform模型驱动时，需要完成两个工作：

a -- 实现platform驱动

b -- 实现platform设备

     然而在实现这两个工作的过程中还需要实现其他的很多小工作，在后面介绍。platform模型驱动的实现过程核心架构就很简单，如下所示：



platform驱动模型三个对象：**platform总线**、**platform设备**、**platform驱动**。

platform总线对应的内核结构：struct bus\_type-->它包含的最关键的函数：match() (要注意的是，这块由内核完成，我们不参与)

platform设备对应的内核结构：struct platform\_device-->注册：platform\_device\_register(unregister)

platform驱动对应的内核结构：struct platform\_driver-->注册：platform\_driver\_register(unregister)

那具体platform驱动的工作过程是什么呢：

     设备(或驱动)注册的时候，都会引发总线调用自己的**match函数**来寻找目前platform总线是否挂载有与该设备(或驱动)名字匹配的驱动（或设备），如果存在则将双方绑定；

     如果先注册设备，驱动还没有注册，那么设备在被注册到总线上时，将不会匹配到与自己同名的驱动，然后在驱动注册到总线上时，因为设备已注册，那么总线会立即匹配与绑定这时的同名的设备与驱动，再调用驱动中的**probe函数**等；

    如果是驱动先注册，同设备驱动一样先会匹配失败，匹配失败将导致它的probe函数暂不调用，而是要等到设备注册成功并与自己匹配绑定后才会调用。

**二、实现platform 驱动与设备的详细过程**

**1、思考问题？**

      在分析platform 之前，可以先思考一下下面的问题：

**a -- 为什么要用 platform 驱动？不用platform驱动可以吗？**

**b -- 设备驱动中引入platform 概念有什么好处？**

        现在先不回答，看完下面的分析就明白了，后面会附上总结。

**2、platform\_device 结构体 VS platform\_driver 结构体**

      这两个结构体分别描述了设备和驱动，二者有什么关系呢？先看一下具体结构体对比

|  |  |
| --- | --- |
| 设备（硬件部分）：中断号，寄存器，DMA等                     platform\_device 结构体 | 驱动（软件部分）                           platform\_driver 结构体 |
| struct platform\_device {  **const char    \*name;       名字**      int        id;      bool        id\_auto;      struct device    **dev**;   硬件模块必须包含该结构体      u32        **num\_resources**;        资源个数  **struct resource    \*resource;**资源  人脉      const struct platform\_device\_id    \***id\_entry**;      /\* arch specific additions \*/      struct pdev\_archdata    archdata;  }; | struct platform\_driver {      int (\***probe**)(struct platform\_device \*);  硬件和软件匹配成功之后调用该函数      int (\***remove**)(struct platform\_device \*);  硬件卸载了调用该函数      void (\*shutdown)(struct platform\_device \*);      int (\*suspend)(struct platform\_device \*, pm\_message\_t state);      int (\*resume)(struct platform\_device \*);      struct device\_driver **driver**;内核里所有的驱动程序必须包含该结构体      const struct platform\_device\_id \***id\_table**;  八字  }; |
| 设备实例：  static struct platform\_device hello\_device=  {      .name = "bigbang",      .id = -1,      .dev.release = hello\_release,  }; | 驱动实例：  static struct platform\_driver hello\_driver=  {      .driver.name = "bigbang",      .probe = hello\_probe,      .remove = hello\_remove,  }; |

       前面提到，实现platform模型的过程就是总线对设备和驱动的匹配过程 。打个比方，就好比相亲，总线是红娘，设备是男方，驱动是女方：

a -- 红娘（总线）负责男方（设备）和女方（驱动）的撮合；

b -- 男方（女方）找到红娘，说我来登记一下，看有没有合适的姑娘（汉子）—— **设备或驱动的注册**；

c -- 红娘这时候就需要看看有没有八字（**二者的name 字段**）匹配的姑娘（汉子）——**match 函数**进行匹配，看name是否相同；

d -- 如果八字不合，就告诉男方（女方）没有合适的对象，先等着，别急着乱做事 —— 设备和驱动会等待，直到匹配成功；

e -- 终于遇到八字匹配的了，那就结婚呗！接完婚，男方就向女方交代，我有多少存款，我的房子在哪，钱放在哪等等（ **struct resource    \*resource**），女方说好啊，于是去房子里拿钱，去给男方买菜啦，给自己买衣服、化妆品、首饰啊等等（**int (\*probe)(struct platform\_device \***) 匹配成功后驱动执行的第一个函数），当然如果男的跟小三跑了（**设备卸载**），女方也不会继续待下去的（**int (\*remove)(struct platform\_device \*)**）。

**3、设备资源结构体**

      在struct platform\_device 结构体中有一重要成员 struct resource \*resource

**[cpp]** [view plain](http://blog.csdn.net/zqixiao_09/article/details/50888795) [copy](http://blog.csdn.net/zqixiao_09/article/details/50888795)

1. **struct** resource {
2. resource\_size\_t start;  资源起始地址
3. resource\_size\_t end;   资源结束地址
4. **const** **char** \*name;
5. unsigned **long** flags;   区分是资源什么类型的
6. **struct** resource \*parent, \*sibling, \*child;
7. };
9. #define IORESOURCE\_MEM        0x00000200
10. #define IORESOURCE\_IRQ        0x00000400

       flags 指资源类型，我们常用的是 IORESOURCE\_MEM、IORESOURCE\_IRQ  这两种。start 和 end 的含义会随着 flags而变更，如

a -- flags为**IORESOURCE\_MEM** 时，start 、end 分别表示该platform\_device**占据的内存的开始地址和结束值**；

b -- flags为**IORESOURCE\_IRQ**   时，start 、end 分别表示该platform\_device**使用的中断号的开始地址和结束值**；

下面看一个实例：

**[cpp]** [view plain](http://blog.csdn.net/zqixiao_09/article/details/50888795) [copy](http://blog.csdn.net/zqixiao_09/article/details/50888795)

1. **static** **struct**  resource beep\_resource[] =
2. {
3. [0] = {
4. .start = 0x114000a0,
5. .end = 0x114000a0+0x4,
6. .flags = IORESOURCE\_MEM,
7. },
9. [1] = {
10. .start = 0x139D0000,
11. .end = 0x139D0000+0x14,
12. .flags = IORESOURCE\_MEM,
13. },
14. };

**4、将字符设备添加到 platform的driver中**

      前面我们提到platform 驱动只是在字符设备驱动外套一层platform\_driver 的外壳，下面我们看一下添加的过程：

**[cpp]** [view plain](http://blog.csdn.net/zqixiao_09/article/details/50888795) [copy](http://blog.csdn.net/zqixiao_09/article/details/50888795)

1. **static** **struct** file\_operations hello\_ops=
2. {
3. .open = hello\_open,
4. .release = hello\_release,
5. .unlocked\_ioctl = hello\_ioctl,
6. };
8. **static** **int** hello\_remove(**struct** platform\_device \*pdev)
9. {
10. 注销分配的各种资源
11. }
13. **static** **int** hello\_probe(**struct** platform\_device \*pdev)
14. {
15. 1.申请设备号
16. 2.cdev初始化注册,&hello\_ops
17. 3.从pdev读出硬件资源
18. 4.对硬件资源初始化，ioremap，request\_irq( )
19. }
21. **static** **int** hello\_init(**void**)
22. {
23. 只注册 platform\_driver
24. }
26. **static** **void** hello\_exit(**void**)
27. {
28. 只注销 platform\_driver
29. }

      可以看到，**模块加载和卸载函数仅仅通过paltform\_driver\_register()、paltform\_driver\_unregister() 函数进行 platform\_driver 的注册和注销，而原先注册和注销字符设备的工作已经被移交到 platform\_driver 的 probe() 和 remove() 成员函数中**。

**5、platform是如何匹配device和driver**

      这时就该总线出场了，系统为platform总线定义了一个bus\_type 的实例platform\_bus\_type，其定义如下：

**[cpp]** [view plain](http://blog.csdn.net/zqixiao_09/article/details/50888795) [copy](http://blog.csdn.net/zqixiao_09/article/details/50888795)

1. **struct** bus\_type platform\_bus\_type = {
2. .name        = "platform",
3. .dev\_groups    = platform\_dev\_groups,
4. .match        = platform\_match,
5. .uevent        = platform\_uevent,
6. .pm        = &platform\_dev\_pm\_ops,
7. };

      其又是怎样工作的呢？在platform.c (e:\linux-3.14-fs4412\drivers\base)    31577    2014/3/31 中可以看到

**[cpp]** [view plain](http://blog.csdn.net/zqixiao_09/article/details/50888795) [copy](http://blog.csdn.net/zqixiao_09/article/details/50888795)

1. \_\_platform\_driver\_register（）
2. {
3. drv->driver.bus = &platform\_bus\_type;     536行
4. }

    在 platform\_bus\_type 中调用 了platform\_match：

**[cpp]** [view plain](http://blog.csdn.net/zqixiao_09/article/details/50888795) [copy](http://blog.csdn.net/zqixiao_09/article/details/50888795)

1. **static** **int** platform\_match(**struct** device \*dev, **struct** device\_driver \*drv)
2. {
3. **struct** platform\_device \*pdev = to\_platform\_device(dev);
4. **struct** platform\_driver \*pdrv = to\_platform\_driver(drv);
6. 匹配设备树信息，如果有设备树，就调用 of\_driver\_match\_device() 函数进行匹配
7. **if** (of\_driver\_match\_device(dev, drv))
8. **return** 1;

11. 匹配id\_table
12. **if** (pdrv->id\_table)
13. **return** platform\_match\_id(pdrv->id\_table, pdev) != NULL;
15. 最基本匹配规则
16. **return** (strcmp(pdev->name, drv->name) == 0);
17. }

**6、解决问题**

      现在可以回答这两个问题了

a -- 为什么要用 platform 驱动？不用platform驱动可以吗？

b -- 设备驱动中引入platform 概念有什么好处？

**引入platform模型符合Linux 设备模型 —— 总线、设备、驱动，设备模型中配套的sysfs节点都可以用，方便我们的开发；当然你也可以选择不用，不过就失去了一些platform带来的便利；**

**设备驱动中引入platform 概念，隔离BSP和驱动。在BSP中定义platform设备和设备使用的资源、设备的具体匹配信息，而在驱动中，只需要通过API去获取资源和数据，做到了板相关代码和驱动代码的分离，使得驱动具有更好的可扩展性和跨平台性。**

**三、实例**

这是一个蜂鸣器的驱动，其实前面已经有解析 [Linux 字符设备驱动开发基础（二）—— 编写简单 PWM 设备驱动](http://blog.csdn.net/zqixiao_09/article/details/50858776)， 下面来看一下，套上platform 外壳后的程序：

1、device.c

**[cpp]** [view plain](http://blog.csdn.net/zqixiao_09/article/details/50888795) [copy](http://blog.csdn.net/zqixiao_09/article/details/50888795)

1. #include <linux/module.h>
2. #include <linux/device.h>
3. #include <linux/platform\_device.h>
4. #include <linux/ioport.h>
6. **static** **struct** resource beep\_resource[] =
7. {
8. [0] ={
9. .start = 0x114000a0,
10. .end =  0x114000a0 + 0x4,
11. .flags = IORESOURCE\_MEM,
12. },
14. [1] ={
15. .start = 0x139D0000,
16. .end =  0x139D0000 + 0x14,
17. .flags = IORESOURCE\_MEM,
18. }
19. };
21. **static** **void** hello\_release(**struct** device \*dev)
22. {
23. printk("hello\_release\n");
24. **return** ;
25. }


29. **static** **struct** platform\_device hello\_device=
30. {
31. .name = "bigbang",
32. .id = -1,
33. .dev.release = hello\_release,
34. .num\_resources = ARRAY\_SIZE(beep\_resource),
35. .resource = beep\_resource,
36. };
38. **static** **int** hello\_init(**void**)
39. {
40. printk("hello\_init");
41. **return** platform\_device\_register(&hello\_device);
42. }
44. **static** **void** hello\_exit(**void**)
45. {
46. printk("hello\_exit");
47. platform\_device\_unregister(&hello\_device);
48. **return**;
49. }
51. MODULE\_LICENSE("GPL");
52. module\_init(hello\_init);
53. module\_exit(hello\_exit);

2、driver.c

**[cpp]** [view plain](http://blog.csdn.net/zqixiao_09/article/details/50888795) [copy](http://blog.csdn.net/zqixiao_09/article/details/50888795)

1. #include <linux/module.h>
2. #include <linux/fs.h>
3. #include <linux/cdev.h>
4. #include <linux/device.h>
5. #include <linux/platform\_device.h>
6. #include <asm/io.h>
8. **static** **int** major = 250;
9. **static** **int** minor=0;
10. **static** dev\_t devno;
11. **static** **struct** **class** \*cls;
12. **static** **struct** device \*test\_device;
14. #define TCFG0         0x0000
15. #define TCFG1         0x0004
16. #define TCON          0x0008
17. #define TCNTB0        0x000C
18. #define TCMPB0        0x0010
20. **static** unsigned **int** \*gpd0con;
21. **static** **void** \*timer\_base;
23. #define  MAGIC\_NUMBER    'k'
24. #define  BEEP\_ON    \_IO(MAGIC\_NUMBER    ,0)
25. #define  BEEP\_OFF   \_IO(MAGIC\_NUMBER    ,1)
26. #define  BEEP\_FREQ   \_IO(MAGIC\_NUMBER   ,2)
28. **static** **void** fs4412\_beep\_init(**void**)
29. {
30. writel ((readl(gpd0con)&~(0xf<<0)) | (0x2<<0),gpd0con);
31. writel ((readl(timer\_base +TCFG0  )&~(0xff<<0)) | (0xff <<0),timer\_base +TCFG0);
32. writel ((readl(timer\_base +TCFG1 )&~(0xf<<0)) | (0x2 <<0),timer\_base +TCFG1 );
34. writel (500, timer\_base +TCNTB0  );
35. writel (250, timer\_base +TCMPB0 );
36. writel ((readl(timer\_base +TCON )&~(0xf<<0)) | (0x2 <<0),timer\_base +TCON );
37. }
39. **void** fs4412\_beep\_on(**void**)
40. {
41. writel ((readl(timer\_base +TCON )&~(0xf<<0)) | (0x9 <<0),timer\_base +TCON );
42. }
44. **void** fs4412\_beep\_off(**void**)
45. {
46. writel ((readl(timer\_base +TCON )&~(0xf<<0)) | (0x0 <<0),timer\_base +TCON );
47. }
49. **static** **void** beep\_unmap(**void**)
50. {
51. iounmap(gpd0con);
52. iounmap(timer\_base);
53. }
55. **static** **int** beep\_open (**struct** inode \*inode, **struct** file \*filep)
56. {
57. fs4412\_beep\_on();
58. **return** 0;
59. }
61. **static** **int** beep\_release(**struct** inode \*inode, **struct** file \*filep)
62. {
63. fs4412\_beep\_off();
64. **return** 0;
65. }
67. #define BEPP\_IN\_FREQ 100000
68. **static** **void** beep\_freq(unsigned **long** arg)
69. {
70. writel(BEPP\_IN\_FREQ/arg, timer\_base +TCNTB0  );
71. writel(BEPP\_IN\_FREQ/(2\*arg), timer\_base +TCMPB0 );
73. }
75. **static** **long** beep\_ioctl(**struct** file \*filep, unsigned **int** cmd, unsigned **long** arg)
76. {
77. **switch**(cmd)
78. {
79. **case** BEEP\_ON:
80. fs4412\_beep\_on();
81. **break**;
82. **case** BEEP\_OFF:
83. fs4412\_beep\_off();
84. **break**;
85. **case** BEEP\_FREQ:
86. beep\_freq( arg );
87. **break**;
88. **default** :
89. **return** -EINVAL;
90. }
91. **return** 0;
92. }
94. **static** **struct** file\_operations beep\_ops=
95. {
96. .open     = beep\_open,
97. .release = beep\_release,
98. .unlocked\_ioctl      = beep\_ioctl,
99. };
101. **static** **int** beep\_probe(**struct** platform\_device \*pdev)
102. {
103. **int** ret;
104. printk("match ok!");
106. gpd0con = ioremap(pdev->resource[0].start,pdev->resource[0].end - pdev->resource[0].start);
107. timer\_base = ioremap(pdev->resource[1].start, pdev->resource[1].end - pdev->resource[1].start);
109. devno = MKDEV(major,minor);
110. ret = register\_chrdev(major,"beep",&beep\_ops);
112. cls = class\_create(THIS\_MODULE, "myclass");
113. **if**(IS\_ERR(cls))
114. {
115. unregister\_chrdev(major,"beep");
116. **return** -EBUSY;
117. }
119. test\_device = device\_create(cls,NULL,devno,NULL,"beep");//mknod /dev/hello
120. **if**(IS\_ERR(test\_device))
121. {
122. class\_destroy(cls);
123. unregister\_chrdev(major,"beep");
124. **return** -EBUSY;
125. }
127. fs4412\_beep\_init();
129. **return** 0;
130. }
132. **static** **int** beep\_remove(**struct** platform\_device \*pdev)
133. {
134. beep\_unmap();
135. device\_destroy(cls,devno);
136. class\_destroy(cls);
137. unregister\_chrdev(major,"beep");
139. **return** 0;
140. }

143. **static** **struct** platform\_driver beep\_driver=
144. {
145. .driver.name = "bigbang",
146. .probe = beep\_probe,
147. .remove = beep\_remove,
148. };

151. **static** **int** beep\_init(**void**)
152. {
153. printk("beep\_init");
155. **return** platform\_driver\_register(&beep\_driver);
156. }
158. **static** **void** beep\_exit(**void**)
159. {
160. printk("beep\_exit");
161. platform\_driver\_unregister(&beep\_driver);
163. **return**;
164. }

167. MODULE\_LICENSE("GPL");
168. module\_init(beep\_init);
169. module\_exit(beep\_exit);

3、makefile

**[cpp]** [view plain](http://blog.csdn.net/zqixiao_09/article/details/50888795) [copy](http://blog.csdn.net/zqixiao_09/article/details/50888795)

1. ifneq  ($(KERNELRELEASE),)
2. obj-m:=device.o driver.o
3. $(info "2nd")
4. **else**
5. #KDIR := /lib/modules/$(shell uname -r)/build
6. KDIR := /home/fs/linux/linux-3.14-fs4412
7. PWD:=$(shell pwd)
8. all:
9. $(info "1st")
10. make -C $(KDIR) M=$(PWD) modules
11. clean:
12. rm -f \*.ko \*.o \*.symvers \*.mod.c \*.mod.o \*.order
13. endif

4、test.c

**[cpp]** [view plain](http://blog.csdn.net/zqixiao_09/article/details/50888795) [copy](http://blog.csdn.net/zqixiao_09/article/details/50888795)

1. #include <sys/types.h>
2. #include <sys/stat.h>
3. #include <fcntl.h>
4. #include <stdio.h>
6. main()
7. {
8. **int** fd,i,lednum;
10. fd = open("/dev/beep",O\_RDWR);
11. **if**(fd<0)
12. {
13. perror("open fail \n");
14. **return** ;
15. }
17. sleep(10);
18. close(fd);
19. }