platform 设备驱动全透析

1.1 platform 总线、设备与驱动

在 Linux 2.6 的设备驱动模型中，关心总线、设备和驱动这 3 个实体，总线将设备和驱动绑定。在系统每注册一个设备的时候，会寻找与之匹配的驱动；相反的，在系统每注册一个驱动的时候，会寻找与之匹配的设备，而匹配由总线完成。

一个现实的 Linux 设备和驱动通常都需要挂接在一种总线上，对于本身依附于 PCI、USB、I^2 C、SPI 等的设备而言，这自然不是问题，但是在嵌入式系统里面，SoC 系统中集成的独立的外设控制器、挂接在 SoC 内存空间的外设等确不依附于此类总 线。基于这一背景，Linux 发明了一种虚拟的总线，称为 platform 总线，相应的设备称为 platform_device，而驱动成为 platform_driver。

注意，所谓的 platform_device 并不是与字符设备、块设备和网络设备并列的概念，而是 Linux 系统提供的一种附加手段，例如，在 S3C6410 处理器中，把内部集成的 I^2 C、RTC、SPI、LCD、看门狗等控制器都归纳为 platform_device，而它们本身就是字符设备。

platform_device 结构体的定义如代码清单 1 所示。

代码清单 1 platform_device 结构体

```
1 struct platform_device {
2 const char * name;/* 设备名 */
3 u32 id;
4 struct device dev;
5 u32 num_resources;/* 设备所使用各类资源数量 */
6 struct resource * resource;/* 资源 */
7 };
```

platform_driver 这个结构体中包含 probe()、remove()、shutdown()、suspend()、resume()函数，通常也需要由驱动实现，如代码清单 2。

代码清单 2 platform_driver 结构体

```
1 struct platform_driver {
2 int (*probe)(struct platform_device *);
3 int (*remove)(struct platform_device *);
4 void (*shutdown)(struct platform_device *);
5 int (*suspend)(struct platform_device *, pm_message_t state);
6 int (*suspend_late)(struct platform_device *, pm_message_t state);
7 int (*resume_early)(struct platform_device *);
8 int (*resume)(struct platform_device *);
9 struct pm_ext_ops *pm;
10 struct device_driver driver;
11};
```

系统中为 platform 总线定义了一个 bus_type 的实例 platform_bus_type，其定义如代码清单 15.3。

代码清单 15.3 platform 总线的 bus_type 实例 platform_bus_type

```
1 struct bus_type platform_bus_type = {
2 .name = "platform",
3 .dev_attrs = platform_dev_attrs,
4 .match = platform_match,
5 .uevent = platform_uevent,
```

```
6 .pm = PLATFORM_PM_OPS_PTR,
7 };
8 EXPORT_SYMBOL_GPL(platform_bus_type);
```

这里要重点关注其 match()成员函数，正是此成员表明了 platform_device 和 platform_driver 之间如何匹配，如代码清单 4 所示。

代码清单 4 platform_bus_type 的 match()成员函数

```
1 static int platform_match(struct device *dev, struct device_driver *drv)
2 {
3 struct platform_device *pdev;
4
5 pdev = container_of(dev, struct platform_device, dev);
6 return (strncmp(pdev->name, drv->name, BUS_ID_SIZE) == 0);
7 }
```

从代码清单 4 的第 6 行可以看出，匹配 platform_device 和 platform_driver 主要看二者的 name 字段是否相同。

对 platform_device 的定义通常在 BSP 的板文件中实现，在板文件中，将 platform_device 归纳为一个数组，最终通过 platform_add_devices()函数统一注册。

platform_add_devices()函数可以将平台设备添加到系统中，这个函数的 原型为：

```
int platform_add_devices(struct platform_device **devs, int num);
```

该函数的第一个参数为平台设备数组的指针，第二个参数为平台设备的数量，它内部调用了 platform_device_register()函数用于注册单个的平台设备。

1.2 将 globalfifo 作为 platform 设备

现在我们将前面章节的 globalfifo 驱动挂接到 platform 总线上，要完成 2 个工作:

1. 将 globalfifo 移植为 platform 驱动。

2. 在板文件中添加 globalfifo 这个 platform 设备。

为完成将 globalfifo 移植到 platform 驱动的工作，需要在原始的 globalfifo 字符设备驱动中套一层 platform_driver 的外壳，如代码清单 5。注意进行这一工作后，并没有改变 globalfifo 是字符设备的本质，只是将其挂接到了 platform 总线。

代码清单 5 为 globalfifo 添加 platform_driver

```
1 static int __devinit globalfifo_probe(struct platform_device *pdev)
2 {
3 int ret;
4 dev_t devno = MKDEV(globalfifo_major, 0);
5
6 /* 申请设备号*/
7 if (globalfifo_major)
8 ret = register_chrdev_region(devno, 1, "globalfifo");
9 else { /* 动态申请设备号 */
10 ret = alloc_chrdev_region(&devno, 0, 1, "globalfifo");
11 globalfifo_major = MAJOR(devno);
12 }
13 if (ret < 0)
14 return ret;
15 /* 动态申请设备结构体的内存*/
```

```
16 globalfifo_devp = kmalloc(sizeof(struct globalfifo_dev), GFP_KERNEL);
17 if (!globalfifo_devp) { /*申请失败*/
18 ret = - ENOMEM;
19 goto fail_malloc;
20 }
21
22 memset(globalfifo_devp, 0, sizeof(struct globalfifo_dev));
23
24 globalfifo_setup_cdev(globalfifo_devp, 0);
25
26 init_MUTEX(&globalfifo_devp->sem); /*初始化信号量*/
27 init_waitqueue_head(&globalfifo_devp->r_wait); /*初始化读等待队列头*/
28 init_waitqueue_head(&globalfifo_devp->w_wait); /*初始化写等待队列头*/
29
30 return 0;
31
32 fail_malloc: unregister_chrdev_region(devno, 1);
33 return ret;
34 }
35
36 static int __devexit globalfifo_remove(struct platform_device *pdev)
37 {
```

```
38 cdev_del(&globalfifo_devp->cdev); /*注销 cdev*/
39 kfree(globalfifo_devp); /*释放设备结构体内存*/
40 unregister_chrdev_region(MKDEV(globalfifo_major, 0), 1); /*释放设备号*/
41 return 0;
42 }
43
44 static struct platform_driver globalfifo_device_driver = {
45 .probe = globalfifo_probe,
46 .remove = __devexit_p(globalfifo_remove),
47 .driver = {
48 .name = "globalfifo",
49 .owner = THIS_MODULE,
50 }
51 };
52
53 static int __init globalfifo_init(void)
54 {
55 return platform_driver_register(&globalfifo_device_driver);
56 }
57
58 static void __exit globalfifo_exit(void)
59 {
```

```
60 platform_driver_unregister(&globalfifo_device_driver);
61 }
62
63 module_init(globalfifo_init);
64 module_exit(globalfifo_exit);
```

在代码清单 5 中，模块加载和卸载函数仅仅通过 platform_driver_register()、platform_driver_unregister()函数进行 platform_driver 的注册与注销，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。代码清单 5 未列出的部分与原始的 globalfifo 驱动相同，都是实现作为字符设备驱动核心的 file_operations 的成员函数。

为了完成在板文件中添加 globalfifo 这个 platform 设备的工作，需要在板文件（对于 LDD6410 而言，为 arch/arm/mach-s3c6410/ mach-ldd6410.c）中添加相应的代码，如代码清单 6。

代码清单 6 globalfifo 对应的 platform_device

```
1 static struct platform_device globalfifo_device = {
2 .name = "globalfifo",
3 .id = -1,
4 };
```

对于 LDD6410 开发板而言，为了完成上述 globalfifo_device 这一 platform_device 的注册，只需要将其地址放入 arch/arm/mach-s3c6410/ mach-ldd6410.c 中定义的 ldd6410_devices 数组，如：

```
static struct platform_device *ldd6410_devices[] __initdata = {
```

```
+ & globalfifo_device,
#ifdef CONFIG_FB_S3C_V2
&s3c_device_fb,
#endif
&s3c_device_hsmmc0,
...
}
```

在加载 LDD6410 驱动后，在 sysfs 中会发现如下结点：

/sys/bus/platform/devices/globalfifo/

/sys/devices/platform/globalfifo/

留意一下代码清单 5 的第 48 行和代码清单 6 的第 2 行，platform_device 和 platform_driver 的 name 一致，这是二者得以匹配的前提。

1.3 platform 设备资源和数据

留意一下代码清单 1 中 platform_device 结构体定义的第 5~6 行，描述了 platform_device 的资源，资源本身由 resource 结构体描述，其定义如代码清单 7。

代码清单 7 resouce 结构体定义

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

7 };

我们通常关心 start、end 和 flags 这 3 个字段，分别标明资源的开始值、结束值和类型，flags 可以为 IORESOURCE_IO、 IORESOURCE_MEM、IORESOURCE_IRQ、IORESOURCE_DMA 等。start、end 的含义会随着 flags 而变更，如当 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地址；当 flags 为 IORESOURCE_IRQ 时，start、end 分别表示该 platform_device 使用的中断号的开始值和结束值，如果只使用了 1 个中断号，开始和结束值相同。对于同种类型的资源而言，可以有多份，譬如说某设备占据了 2 个内存区域，则可以定义 2 个 IORESOURCE_MEM 资 源。

对 resource 的定义也通常在 BSP 的板文件中进行，而在具体的设备驱动中透过 platform_get_resource()这样的 API 来获取，此 API 的原型为：

```
struct resource *platform_get_resource(struct platform_device *, unsigned int,
unsigned int);
```

譬如在 LDD6410 开发板的板文件中为 DM9000 网卡定义了如下 resouce：

```
static struct resource ldd6410_dm9000_resource[] = {
[0] = {
.start = 0x18000000,
.end = 0x18000000 + 3,
.flags = IORESOURCE_MEM
},
[1] = {
.start = 0x18000000 + 0x4,
.end = 0x18000000 + 0x7,
```

```
.flags = IORESOURCE_MEM
},
[2] = {
.start = IRQ_EINT(7),
.end = IRQ_EINT(7),
.flags = IORESOURCE_IRQ | IORESOURCE_IRQ_HIGHLEVEL,
}
};
```

在 DM9000 网卡的驱动中则是通过如下办法拿到这 3 份资源：

```
db->addr_res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
db->data_res = platform_get_resource(pdev, IORESOURCE_MEM, 1);
db->irq_res = platform_get_resource(pdev, IORESOURCE_IRQ, 0);
```

对于 IRQ 而言，platform_get_resource()还有一个进行了封装的变体 platform_get_irq()，其原型为：

```
int platform_get_irq(struct platform_device *dev, unsigned int num);
```

它实际上调用了“platform_get_resource(dev, IORESOURCE_IRQ, num);”。

设备除了可以在 BSP 中定义资源以外，还可以附加一些数据信息，因为对设备的硬件描述除了中断、内存、DMA 通道以外，可能还会有一些配置信息，而 这些配置信息也依赖于板，不适宜直接放置在设备驱动本身，因此，platform 也提供了 platform_data 的支持。

platform_data 的形式是自定义的，如对于 DM9000 网卡而言，platform_data 为一个 dm9000_plat_data 结构体，我们就可以将 MAC 地址、总 线宽度、有无 EEPROM 信息放入 platform_data：

```
static struct dm9000_plat_data ldd6410_dm9000_platdata = {
.flags = DM9000_PLATF_16BITONLY | DM9000_PLATF_NO_EEPROM,
.dev_addr = { 0x0, 0x16, 0xd4, 0x9f, 0xed, 0xa4 },
};
static struct platform_device ldd6410_dm9000 = {
.name = "dm9000",
.id = 0,
.num_resources = ARRAY_SIZE(ldd6410_dm9000_resource),
.resource = ldd6410_dm9000_resource,
.dev = {
.platform_data = &ldd6410_dm9000_platdata,
}
};
```

而在 DM9000 网卡的驱动中，通过如下方式就拿到了 platform_data：

```
struct dm9000_plat_data *pdata = pdev->dev.platform_data;
```

其中，pdev 为 platform_device 的指针。

由以上分析可知，设备驱动中引入 platform 的概念至少有如下 2 大好处：

1. 使得设备被挂接在一个总线上，因此，符合 Linux 2.6 的设备模型。其结果是，配套的 sysfs 结点、设备电源管理都成为可能。

2. 隔离 BSP 和驱动。在 BSP 中定义 platform 设备和设备使用的资源、设备的具体配置信息，而在驱动中，只需要通过通用 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。