<http://blog.csdn.net/alan445947767/article/details/41146563>

Linux 系统提供了一种分离分层的思想,换句话说,它借用了将复杂问题简单化的思想,总是喜欢把一个驱动拆分开来写。打个比方说,有那么一种设备驱动,假设拆分为 A、B、C 为 3 个部分。A 部分负责管理 B、C 两部分,将 B和 C 进行绑定管理,再假设 B 负责比较稳定的代码,C 负责的是硬件相关的代码。这样一来,如果硬件设备改变了,那么驱动就不用大范围修改,只需要修改C 负责的硬件相关的代码,这就是 Linux 的分离分层的思想。

### platform 总线驱动设备模型

Linux2.6 提出了一种模型,**叫设备驱动模型,它包含总线、设备、驱动三个实体,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;在系统每注册一个驱动的时候,也会寻找与之匹配的设备,**而这个匹配的伟大任务,则由总线这老人家来完成了。Linux 还发明了种虚拟的总线,这种**虚拟的总线叫 platform 总线,设备则称为 platform 设备,相应的驱动则称为 platform 驱动。**在嵌入式系统里,比如,在 S5PV210 处理器中,可以把 LED、BUTTONS、I2C、RTC、SPI、LCD、NAND 等等都可以看做是 platform\_device。这真是,只有想不到没有做不到。

### 三个重要的结构体

为什么又是三个重要的结构体?Linux 内核里,大量的使用了面向对象的思想来编程,虽然使用的语言是 C 语言,但是它却大量的使用了面向对象的思想。不得不感叹一下,Linux 大牛们啊,真牛。

#### 平台总线(platform\_bus\_type)

platform\_bus\_type 是一个 struct bus\_type 型的结构体,其定义如下:

/\* 参考 drivers/base/platform.c \*/

struct bus\_type platform\_bus\_type = {

.name= "platform",/\* 总线名字 \*/

.dev\_attrs= platform\_dev\_attrs, /\* 设备属性 \*/

/\* 设备和驱动使用 match 函数来判断是否匹配 \*/

.match= platform\_match,

.uevent= platform\_uevent,/\* 热拔插操作函数 \*/

.pm= &platform\_dev\_pm\_ops,

};

platform\_bus\_type 中的成员,只需要关心 platform\_match()函数,正是此函数确定了 platform\_device 和 platform\_driver 之间如何匹配,后面再分析。

#### 平台设备(platform\_device)

/\* 参考 include/linux/platform\_device.h \*/

struct platform\_device {

const char \* name;/\* 平台设备名字 \*/

int id;

/\* 一般设置为-1,表示系统自动分配 \*/

bool id\_auto;

struct device dev;/\* 每一个设备都有一个 device 结构体 \*/

u32 num\_resources;/\* 资源数量 \*/

struct resource \* resource; /\* 资源 \*/

/\* platform\_device\_id,里面有个名字列表 \*/

const struct platform\_device\_id \*id\_entry;

...

};

其中有个重要的结构体成员,那就是资源(resource)

struct resource {

resource\_size\_t start;/\* 资源的起始值 \*/

resource\_size\_t end;/\* 资源的结束值 \*/

const char \*name;

/\* 资源的类型,如:

\* IORESOURCE\_IO(IO 资源)、IORESOURCE\_MEM(内存资源)、

\* IORESOURCE\_IRQ(中断资源)、IORESOURCE\_DMA(DMA 资源)

\*/

unsigned long flags;

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

};

resource 里的 start、end 的含义会随着 flags 的变化而变化,比如: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,如果只使用了一个中断,那么开始值和结束值相同。

Linux 提供了一个获取 resource 的接口函数,使用 platform\_get\_resource函数来获取资源。原型如下:

/\*\*

\* platform\_get\_resource - get a resource for a device

\* @dev: platform device

\* @type: resource type

\* @num: resource index

\*/

struct resource \*platform\_get\_resource(struct platform\_device \*dev,

unsigned int type, unsigned int num)

{

int i;

for (i = 0; i < dev->num\_resources; i++) {

struct resource \*r = &dev->resource[i];

if (type == resource\_type(r) && num-- == 0)

return r;

}

return NULL;

}

这个函数的关键点就是二个判断,如果二个判断都符合要求,那么返回符合要求的资源。为了分析这个函数,举个例子。比如你想要 IORESOURCE\_MEM 资源下的第 2 个资源,在数组里即第 1 个资源,因为数组从零算起。就会这样调用:

res = platform\_get\_resource(pdev, IORESOURCE\_MEM,1);

第一个判断是: type == resource\_type(r) , type 就 是 我 们 刚 说 的IORESOURCE\_IO 、 IORESOURCE\_MEM 、 IORESOURCE\_IRQ 、IORESOURCE\_DMA 四种。从平台设备(platform\_dev)获取资源(resource)后,首 先 判 断 是 不 是 这 四 种 当 中 的 一 种 , 因 为 例 子 中 的 type 确 实 是IORESOURCE\_MEM,则继续判断是这种 IORESOURCE\_MEM 类型资源下的,是不是你想要的那个资源,比如有 3 个 IORESOURCE\_MEM 类型的资源,而你只想要第 2 个。如果 type 不是 IORESOURCE\_MEM,则重新从平台设备的下一个资源里获取,直至最后一个。第二个判断是:num-- == 0,这什么意思呢?一开始 num = 1,确实不等于0 , 所 以 会 重 新 从 平 台 设 备 的 下 一 个 资 源 里 获 取 type 类 型 为IORESOURCE\_MEM 类型的资源,这时 num = 0 了,符合要求,就会返回资源。

#### 平台驱动(platform\_drivers)

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 (\*resume)(struct platform\_device \*);

struct device\_driver driver;

const struct platform\_device\_id \*id\_table;

};

platform\_driver 里有个 platform\_device\_id 结构体,它与 platform\_device设 备 里 的 platform\_device\_id 是 一 样 的 。 以 前 总 线 总 是 通 过platform\_device.name 和 platform\_driver.device\_driver.name 来匹配。 新的Linux 版 本 , 也 增 加 了 另 外 一 种 机 制 , 也 就 是 通 过 比 较 二 者 里 的platform\_device\_id 里的 name。说白了,总线匹配最终就是比较二者的名字。

struct device\_driver {

const char

\*name;

struct module

\*owner;

...

const struct of\_device\_id \*of\_match\_table;

...

struct driver\_private \*p;

};

在建有内核工程的 Source Insight 里搜索“of\_match\_table”,就会搜出一大堆示例,你会发现,每一个 device\_driver 实例都会设置上面这三个加深颜色的成员,所以几乎不用关心 device\_driver 的其他成员。

#### 平台总线如何匹配设备和驱动

你前面不是说了吗?总线将设备和驱动进行绑定,在系统每注册一个设备的时候,会寻找与之匹配的驱动;在系统每注册一个驱动的时候,也会寻找与之匹配的设备。那平台总线怎么匹配呢?答:通过 platform\_match()函数。

/\* 参考 drivers/base/platform.c \*/

static int platform\_match(struct device \*dev, struct device\_driver \*drv)

{

struct platform\_device \*pdev = to\_platform\_device(dev);

struct platform\_driver \*pdrv = to\_platform\_driver(drv);

/\* 因为没有配置 CONFIG\_OF\_DEVICE ,这里返回 0,不执行这个 if 分支

\*/

if (of\_driver\_match\_device(dev, drv))

return 1;

/\*这里也是返回 0,不执行这个 if 分支\*/

if (acpi\_driver\_match\_device(dev, drv))

return 1;

/\* 如果 pdrv->id\_table 存在的话,调用 platform\_match\_id 函数

\* 如果匹配成功,则返回 1,否则返回 0

\*/

if (pdrv->id\_table)

return platform\_match\_id(pdrv->id\_table, pdev) != NULL;

/\* 如果前面都失败,则最后比较平台设备里的名字和设备驱动里的名字

\* 即比较 platform\_device.name 与 device\_driver.name

\* 如果匹配成功,则返回 1,否则返回 0

\*/

return (strcmp(pdev->name, drv->name) == 0);

}

下面来看看 platform\_match\_id 函数是怎么匹配的。

/\* 参考 drivers/base/platform.c \*/

static const struct platform\_device\_id \*platform\_match\_id(

const struct platform\_device\_id \*id,struct platform\_device \*pdev)

{

while (id->name[0]) {

if (strcmp(pdev->name, id->name) == 0) {

pdev->id\_entry = id;

return id;

}

id++;

}

return NULL;

}

platform\_match\_id 函数的实现很简单,如果 pdrv->id\_table->name[0]存在的话,就与 dev->name 进行比较,如果相同的话,就把 pdrv->id\_table 赋给pdev->id\_entry,然后返回一个类型为 platform\_device\_id 的 pdrv->id\_table。如果比较不成功,就取出下一个 pdrv->id\_table,然后再比较,直到最后一个pdrv->id\_table,还是不成功,就返回 NULL。

#### 内核如何注册平台总线

与平台驱动和平台设备不一样的是,平台总线并不是以模块的方式加载到内核的,下面以函数的调用流程来看看平台总线到底是怎么被内核调用的。

start\_kernel

-->rest\_init

-->kernel\_thread(kernel\_init,NULL,CLONE\_FS|CLONE\_SIGHAND);

--> kernel\_init\_freeable();

--> do\_basic\_setup();

--> driver\_init();

--> platform\_bus\_init();

-->bus\_register(&platform\_bus\_type);

在这里,除了告诉大家平台总线的调用流程,其实还想引出一个问题。就是Webee 怎么找到这个流程的?其实,很简单,在建有内核的工程里,使用 Source Insight 使用“ctrl + /”搜索对应的函数,就可以找到是哪个函数调用它的了。

### 四个注册/注销函数

#### 平台设备注册函数(platform\_device\_register)

platform\_device\_register 用于注册一个平台设备。

int platform\_device\_register(struct platform\_device \*pdev)

{

/\* 初始化 device 结构体成员 \*/

device\_initialize(&pdev->dev);

arch\_setup\_pdev\_archdata(pdev);

return platform\_device\_add(pdev);

}

/\* 导出函数符号,让其他文件可以调用这个函数 \*/

EXPORT\_SYMBOL\_GPL(platform\_device\_register);

platform\_device\_register 最终使用 platform\_device\_add 函数来注册平台设

备。

/\* 参考 drivers/base/platform.c \*/

int platform\_device\_add(struct platform\_device \*pdev)

{

int i, ret;

if (!pdev)

return -EINVAL;

if (!pdev->dev.parent)

pdev->dev.parent = &platform\_bus;

/\* 设置总线类型为平台总线类型 \*/

pdev->dev.bus = &platform\_bus\_type;

/\* 根据 pdev->id 设置平台设备的名字 \*/

switch (pdev->id) {

default:

dev\_set\_name(&pdev->dev, "%s.%d", pdev->name, pdev->id);

break;

......

}

/\* 遍历取出平台设备里的每一个资源 \*/

for (i = 0; i < pdev->num\_resources; i++) {

struct resource \*p, \*r = &pdev->resource[i];

if (r->name == NULL)

r->name = dev\_name(&pdev->dev);

p = r->parent;

/\* 如果资源不为空,设置资源的类型 \*/

if (!p) {

/\* 分配 IO 内存资源空间\*/

if (resource\_type(r) == IORESOURCE\_MEM)

p = &iomem\_resource;

/\* 分配 IO 端口资源空间\*/

else if (resource\_type(r) == IORESOURCE\_IO)

p = &ioport\_resource;

}

/\* 将新的 resource 插入内核资源树头(resource tree) \*/

if (p && insert\_resource(p, r)) {

printk(KERN\_ERR "%s: failed to claim resource %d\n",dev\_name(&pdev->dev), i);

ret = -EBUSY;

goto failed;

}

}

......

/\* 调用 device\_add 函数注册一个设备(device) \*/

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

if (ret == 0)

return ret;

......

}

platform\_device\_add 最终调用device\_add 来完成平台的注册 。device\_add 就比较复杂了,这里就不分析了,但是给你们总结一下 device\_add函数主要做了些什么工作:

第一、把device嵌入bus的dev链表。

第二、从bus的drv 链表遍历取出每一个 drv,使用bus的match函数来判断drv能不能支持这个dev。

第三、如果支持,调用drv的probe函数。

#### 平台驱动注册函数(platform\_driver\_register)

platform\_driver\_register 用于注册一个平台驱动。

int platform\_driver\_register(struct platform\_driver \*drv)

{

/\* 先初始化 platform\_driver 里的 driver,该 driver 的类型为

\* device\_driver,设置 driver 的 bus 为 platform\_bus\_type

\*/

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

/\* 继续设置 driver 的 probe、remove、shutdown 成员

\* 为平台驱动下的 probe、remove、shutdown 函数

\*/

if (drv->probe)

drv->driver.probe = platform\_drv\_probe;

if (drv->remove)

drv->driver.remove = platform\_drv\_remove;

if (drv->shutdown)

drv->driver.shutdown = platform\_drv\_shutdown;

/\* 最终调用 driver\_register 函数注册一个驱动 \*/

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

}

driver\_register 函数主要工作与device\_add类似,这里也不分析源码,只总结 driver\_register 函数的主要工作:

第一、把drv嵌入bus的drv链表。

第二、从bus的dev链表遍历取出每一个dev,使用 bus 的 match 函数来判断dev 能不能支持这个drv。

第三、如果支持,调用 drv 的 probe 函数。

#### 平台设备注销函数(platform\_device\_unregister)

想要从内核里卸载平台设备,就使用 platform\_device\_unregister 函数。

void platform\_device\_unregister(struct platform\_device \*pdev)

{

/\* 删除 device 结构,释放资源 \*/

platform\_device\_del(pdev);

/\* 调用 put\_device 函数将 device 从 bus 链表里删除 \*/

platform\_device\_put(pdev);

}

/\* 参考 drivers/base/platform.c \*/

void platform\_device\_del(struct platform\_device \*pdev)

{

int i;

/\* 删除 device \*/

if (pdev) {

device\_del(&pdev->dev);

......

/\* 遍历资源,调用 release\_resource 函数释放资源 \*/

for (i = 0; i < pdev->num\_resources; i++) {

struct resource \*r = &pdev->resource[i];

unsigned long type = resource\_type(r);

if (type == IORESOURCE\_MEM || type == IORESOURCE\_IO)

release\_resource(r);

}

}

}

EXPORT\_SYMBOL\_GPL(platform\_device\_del);

#### 平台驱动注销函数(platform\_driver\_unregister)

平台驱动的注销使用 platform\_driver\_unregister 函数。

void platform\_driver\_unregister(struct platform\_driver \*drv)  
{

/\* 将 drv 从 bus 链表里删除 \*/

driver\_unregister(&drv->driver);

}

### 平台总线驱动设备驱动实例

平台总线驱动设备模型的驱动,一般分为两部分:一部分写 platform\_device相关的,另一部分写 platform\_driver 相关的。一般来说,platform\_device 相关的代码编写,可以写到 BSP 里,这样一来,就实现了隔离 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,而在驱动里,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。还记得7 章,第我们移植了 gpio\_key驱动吗,那里我们主要修改的就是 BSP 板相关代码。但是,为了让大家更好的学习,Webee 将本实例分开二个文件来编写,并没有添加到 BSP 板里去。

#### 平台设备驱动实例(platform\_device)

本实例其实就是将 webee210\_drivers\2th\_led\led.c 拆开来编写。平台设备驱动源码路径为:webee210\_drivers\7th\_bus\_drv\_devc\led\_dev.c

##### 入口函数分析

static int \_\_init led\_dev\_init(void)

{

/\* 注册一个平台设备 \*/

return platform\_device\_register(&led\_device);

}

入口函数很简单,注册了一个 led\_device 平台设备。那 led\_device 的定义是怎么样的呢?

static struct platform\_device led\_device = {

.id= -1,

/\* 必须与 led\_driver 的 name 一致 \*/

.name= "webee210\_led",

.resource= led\_resources,

.num\_resources = ARRAY\_SIZE(led\_resources),

.dev={

.release= led\_release,

},

};

led\_device 有个 led\_resources 资源,来,继续看看它的定义。

/\* Webee210 开发板上的 LED1,LED2,LED3,LED4

\* 对应 GPJ2\_0、GPJ2\_1、GPJ2\_2、GPJ2\_3 引脚

\*/

static struct resource led\_resources[] = {

[0] = {

.start = 0xE0200280,

.end= 0xE0200280 + 8 -1,

.flags = IORESOURCE\_MEM,

},

[1] = {

.start = 0,

/\* LED1 \*/

.end= 0,

.flags = IORESOURCE\_IRQ,

},

};

这个就是硬件相关的代码了,如果有不同的设备,那么一般只需要修改这些硬件相关的代码,而 platform\_driver 部分就不需要修改了。

##### 出口函数分析

static void \_\_exit led\_dev\_exit(void)

{

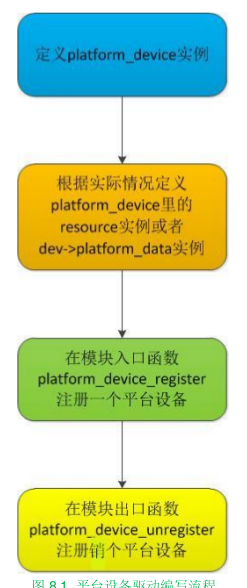
/\* 注销一个平台设备 \*/

platform\_device\_unregister(&led\_device);

}

还是那句,出口函数干了些什么事情,出口函数就是要把它“消灭”掉。驱动里,很多函数都是对称的。

##### 图说平台设备驱动的编写流程



#### 平台驱动实例(platform\_driver)

##### 入口函数分析

static int \_\_init led\_drv\_init(void)

{

/\* 注册一个平台驱动 \*/

return platform\_driver\_register(&led\_driver);

}

入口函数很简单,注册了一个 led\_driver 平台驱动。那 led\_driver 的定义是怎么样的呢?

static struct platform\_driver led\_driver = {

.probe= led\_probe,

.remove= led\_remove,

.driver={

.name = "webee210\_led",

.owner = THIS\_MODULE,

}

};

注意了,这里的 name 必须和平台设备里的 name 要保持一致,因为总线的match 函数就是通过比较它们的名字,来判断驱动能不能支持该设备。如果支持,就会调用 platform\_driver 里的 probe 函数,在这里是调用 led\_probe 函数。

##### probe 函数分析

static int led\_probe(struct platform\_device \*pdev)

{

struct resource \*res;

printk("led\_probe, found led\n");

/\* 根据 platform\_device 的资源进行 ioremap \*/

res = platform\_get\_resource(pdev, IORESOURCE\_MEM, 0);

gpj2con = ioremap(res->start, res->end - res->start + 1);

gpj2dat = gpj2con + 1;

res = platform\_get\_resource(pdev, IORESOURCE\_IRQ, 0);

pin = res->start;

/\* 注册字符设备 \*/

major = register\_chrdev(0, "led\_drv", &led\_fops);

/\* 创建 led\_drv 类 \*/

led\_class = class\_create(THIS\_MODULE,"led\_drv");

/\* 在 led\_drv 类下创建/dev/LED 设备,供应用程序打开设备\*/

device\_create(led\_class,NULL,MKDEV(major,0),NULL,DEVICE\_NAME);

return 0;

}

led\_probe 函数,通过 platform\_get\_resource 函数来获取 platform\_device里设置的 resource。获取到资源后,使用 ioremap 函数将资源映射到内核空间,这样驱动就可以使用这些资源了。接下来,又回到了以前我们熟悉的字符设备驱动了,其中,定义了一个 file\_operations 型的 led\_fops 实例。

static const struct file\_operations led\_fops = {

.owner= THIS\_MODULE,

.open= webee210\_led\_open,

.write= webee210\_led\_write,

};

##### webee210\_led\_open

webee210\_led\_open 函数很简单,只是将 LED 对应的 GPIO 引脚功能设置为输入功能。

#define S5PV210\_OUTP (0x1<<(pin\*2))

static int webee210\_led\_open(struct inode \* inode, struct file \* filp)

{

/\* 设置 LED 对应 GPIO 引脚功能为输出功能 \*/

\*gpj2con |= S5PV210\_OUTP;

return 0;

}

##### webee210\_led\_write

webee210\_led\_write 函数首先通过 copy\_from\_user 将用户应用程序传进来的数据存放到 val 里,然后根据 val 的值写入 gpj2dat 寄存器,这样一来就实现了数据从应用空间到内核空间,最后到硬件设备。

static int webee210\_led\_write(struct file \* file, const char \_\_user \*buffer, size\_t count, loff\_t \* ppos)

{

int val;

if(copy\_from\_user(&val, buffer, count))

return -EFAULT;

if (val == 1)

{

/\* 点灯 \*/

\*gpj2dat &= ~(1<<pin);

}

else

{

/\* 灭灯 \*/

\*gpj2dat |= (1<<pin);

}

return 0;

}

##### led\_remove 函数分析

remove 函数什么时候被调用呢?当驱动被卸载的时候,就会调用 remove函数。它的工作主要是将 probe 函数干的事情“消灭”掉。

static int led\_remove(struct platform\_device \*pdev)

{

printk("led\_remove, remove led\n");

device\_destroy(led\_class, MKDEV(major, 0));

class\_destroy(led\_class);

unregister\_chrdev(major, "led\_drv");

iounmap(gpj2con);

return 0;

}

##### 出口函数分析

static void \_\_exit led\_drv\_exit(void)

{

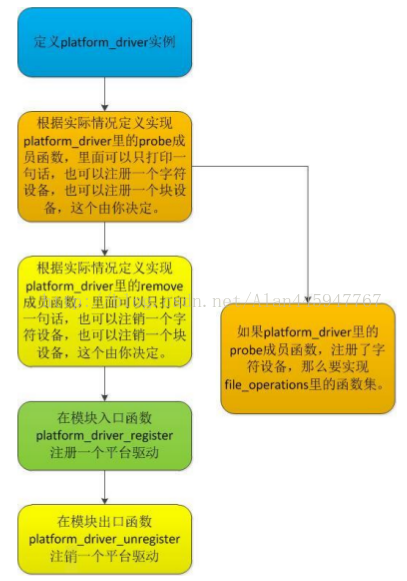
/\* 注销一个平台驱动 \*/

platform\_driver\_unregister(&led\_driver);

}

有入口函数,自然就有出口函数。入口函数是注册平台驱动,出口函数自然是注销平台驱动。其实,Linux 在告诉大家一个道理:做人要有始有终。

##### 图说平台驱动的编写流程



### 测试程序

测 试 程 序 和 以 前 的 LED 测 试 程 序 是 一 样 的 , 源 码 路 径 为 :webee210\_drivers\7th\_bus\_drv\_devc\led\_test.c

#include <stdio.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <unistd.h>

#include <string.h>

/\* led\_test on

\* led\_test off

\*/

int main(int argc ,char \*argv[])

{

int fd;

int val = 0;

fd = open("/dev/LED",O\_RDWR);

if (fd < 0)

{

printf("open error\n");

}

if (argc != 2)

{

printf("Usage:\n");

printf("%s <on|off>\n",argv[0]);

return 0;

}

if(strncmp(argv[1],"on",2) == 0)

{

val = 1;

}

else if (strncmp(argv[1],"off",3) == 0)

{

val = 0;

}

write(fd,&val,4);

return 0;

}