为了使驱动和设备分离，linux提供了很多中间层以使接口统一，这样就可以使SoC以统一的方式将驱动进行注册，设备端则可以根据统一的接口进行使用，这样做的好处就是Soc可以不去关心用户到底要接什么设备，而设备端也不用去关心到底要接到什么Soc，用户可以自由的将两者关联。

目前的linux设备驱动模型都秉承着总线、设备、驱动三个部分，总线负责设备和驱动的绑定，在往系统注册驱动时，总线会自动去匹配对应的设备，同样，在往系统注册设备时，总线会自动取匹配对应的驱动。但是不像I2C、SPI等这种本身就是总线的概念，dma、看门狗等并没有总线的概念，linux提供了一种虚拟总线platform\_bus，相应的设备与驱动为platform\_driver、platform\_device。

下面先介绍一下这三个结构：

对于platform\_bus，它使用的是内核用于描述总线的结构bus\_type，定义如下：

struct bus\_type platform\_bus\_type = {

.name = "platform",

.dev\_groups = platform\_dev\_groups,

.match = platform\_match,

.uevent = platform\_uevent,

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

};

platform\_driver定义如下：

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;

bool prevent\_deferred\_probe;

};

platform\_device定义如下：

struct platform\_device {

const char \*name;

int id;//此设备在这类设备中的编号，如果设备类型只有一种则为-1

bool id\_auto;

struct device dev;

u32 num\_resources;//资源数量，包括内存/dma/irq等

struct resource \*resource;//资源数组

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

char \*driver\_override; /\* Driver name to force a match \*/

/\* MFD cell pointer \*/

struct mfd\_cell \*mfd\_cell;

/\* arch specific additions \*/

struct pdev\_archdata archdata;

};

# platform\_driver和platform\_device的匹配

## 匹配的方法

driver和device的匹配使用的是driver\_match\_device，最终调用的是drv->bus->match(dev, drv)，也就是上面platform\_bus提到的platform\_match，下面来着重分析一下：

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);

/\* When driver\_override is set, only bind to the matching driver \*/

if (pdev->driver\_override)

return !strcmp(pdev->driver\_override, drv->name);

/\* Attempt an OF style match first \*/

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

return 1;

/\* Then try ACPI style match \*/

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

return 1;

/\* Then try to match against the id table \*/

if (pdrv->id\_table)

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

/\* fall-back to driver name match \*/

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

}

可以看到内核提供了5种driver和device匹配的方法：

1. 比较platform\_device. driver\_override 和 platform\_driver.driver.name，可以设置driver\_override，使其匹配与此名字相同的驱动，driver\_override可以在对应设备目录进行设置，如gpio的driver\_override路径为/sys/bus/platform/devices/44e07000.gpio/driver\_override。
2. 基于设备树的匹配，of\_driver\_match\_device(dev, drv)的过程如下：

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

of\_match\_device(drv->of\_match\_table, dev) != NULL;

of\_match\_device(drv->of\_match\_table, dev) != NULL;

最终会遍历platform\_driver的device\_driver的of\_match\_table表，将它的每一个“compatible”字段与设备节点的“compatible"进行对比。

1. 基于ACPI的也就是高级配置和电源管理的匹配，这部分没捋过。
2. 如果设置了platform\_driver的id\_table，则会遍历id\_table，如果设备的名称与id\_table中的任意一个匹配则成功。
3. 直接对比驱动和设备的名称，这也是在设备树出现之前最常使用的匹配方式。

## 匹配的调用

前面提到过驱动可以去匹配设备，设备也可以去匹配驱动，那么就可以理解为无论是注

册设备还是注册驱动肯定都执行了上面的driver\_match\_device，platform的设备注册和驱动注册分别为platform\_device\_add()和platform\_driver\_register()，两个函数执行到driver\_match\_device的过程如下：

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

int device\_add(struct device \*dev)

void bus\_probe\_device(struct device \*dev)

if (bus->p->drivers\_autoprobe) // cat /sys/bus/platform/drivers\_autoprobe

device\_initial\_probe(dev);

int \_\_device\_attach(struct device \*dev, bool allow\_async)

int \_\_device\_attach\_driver(struct device\_driver \*drv, void \*\_data)

ret = driver\_match\_device(drv, dev);

int platform\_driver\_register(struct platform\_driver \*drv, struct module \*owner)

int driver\_register(struct device\_driver \*drv)

int bus\_add\_driver(struct device\_driver \*drv)

int driver\_attach(struct device\_driver \*drv)

int \_\_driver\_attach(struct device \*dev, void \*data)

ret = driver\_match\_device(drv, dev);

# 设备树展开platform\_device过程

//使用设备树数据填充platform\_devices

int of\_platform\_populate(struct device\_node \*root,const struct of\_device\_id \*matches,const struct of\_dev\_auxdata \*lookup,struct device \*parent)

//遍历所有根节点下的子节点，为这个子节点以及这个子节点的子节点创建设备

for\_each\_child\_of\_node(root, child) {

rc = of\_platform\_bus\_create(child, matches, lookup, parent, true);

if (rc) {

of\_node\_put(child);

break;

}

}

设备的申请、初始化与注册都是在of\_platform\_bus\_create中完成的：

of\_platform\_bus\_create

dev = of\_platform\_device\_create\_pdata(bus, bus\_id, platform\_data, parent);

//申请和注册设备

dev = of\_device\_alloc(np, bus\_id, parent);

// platform\_device的申请

dev = platform\_device\_alloc("", -1);

//资源的填充

dev->num\_resources = num\_reg + num\_irq;

dev->resource = res;

//设备的添加

of\_device\_add(dev)

那接下来的问题就是谁来调用of\_platform\_populate呢，这就回到了讲解设备树compatible 属性时提到的DT\_MACHINE\_START，还是以am33xx为例，其包含.init\_machine= omap\_generic\_init，而omap\_generic\_init中就包含了of\_platform\_populate，即对设备树的展开：

void \_\_init omap\_generic\_init(void)

r = of\_platform\_populate(node, NULL, NULL, &pdev->dev);

# 测试程序说明

我们在介绍设备树的测试程序上稍加修改，设备树内容不用更改，可以看到改动并不大，只是套了个platform driver的壳子，以前在init/exit完成的初始化与反初始化过程可以转移到probe/remove中完成。

还有注意到设备树节点不用我们再自己查找了，因为已经在设备树展开的时候的of\_device\_alloc()时被赋值，可以直接使用。