**Linux设备模型(8)\_platform设备**

**1. 前言**

在Linux设备模型的抽象中，存在着一类称作“Platform Device”的设备，内核是这样描述它们的（Documentation/driver-model/platform.txt）：

Platform devices are devices that typically appear as autonomous entities in the system. This includes legacy port-based devices and host bridges to peripheral buses, and most controllers integrated into system-on-chip platforms.  What they usually have in common is direct addressing from a CPU bus.  Rarely, a platform\_device will be connected through a segment of some other kind of bus; but its registers will still be directly addressable.

概括来说，Platform设备包括：基于端口的设备（已不推荐使用，保留下来只为兼容旧设备，legacy）；连接物理总线的桥设备；集成在SOC平台上面的控制器；连接在其它bus上的设备（很少见）。等等。

这些设备有一个基本的特征：可以通过CPU bus直接寻址（例如在嵌入式系统常见的“寄存器”）。因此，由于这个共性，内核在设备模型的基础上（device和device\_driver），对这些设备进行了更进一步的封装，抽象出paltform bus、platform device和platform driver，以便驱动开发人员可以方便的开发这类设备的驱动。

可以说，paltform设备对Linux驱动工程师是非常重要的，因为我们编写的大多数设备驱动，都是为了驱动plaftom设备。本文我们就来看看Platform设备在内核中的实现。

**2. Platform模块的软件架构**

内核中Platform设备有关的实现位于include/linux/platform\_device.h和drivers/base/platform.c两个文件中，它的软件架构如下：

[](http://www.wowotech.net/content/uploadfile/201405/8cf2e12b367205a9c1c7888e2dcb818520140507093250.gif)

由图片可知，Platform设备在内核中的实现主要包括三个部分：

Platform Bus，基于底层bus模块，抽象出一个虚拟的Platform bus，用于挂载Platform设备；   
Platform Device，基于底层device模块，抽象出Platform Device，用于表示Platform设备；   
Platform Driver，基于底层device\_driver模块，抽象出Platform Driver，用于驱动Platform设备。

其中Platform Device和Platform Driver会会其它Driver提供封装好的API，具体可参考后面的描述。

**3. Platform模块向其它模块提供的API汇整**

Platform提供的接口包括：Platform Device和Platform Driver两个数据结构，以及它们的操作函数。

**3.1 数据结构**

1. 用于抽象Platform设备的数据结构----“struct platform\_device”：

1: /\* include/linux/platform\_device.h, line 22 \*/

2: struct platform\_device {

3: const char \*name;

4: int id;

5: bool id\_auto;

6: struct device dev;

7: u32 num\_resources;

8: struct resource \*resource;

9:

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

11:

12: /\* MFD cell pointer \*/

13: struct mfd\_cell \*mfd\_cell;

14:

15: /\* arch specific additions \*/

16: struct pdev\_archdata archdata;

17: };

该结构的解释如下：

dev，真正的设备（Platform设备只是一个特殊的设备，因此其核心逻辑还是由底层的模块实现）。

name，设备的名称，和struct device结构中的init\_name（"[Linux设备模型(5)\_device和device driver](http://www.wowotech.net/linux_kenrel/device_and_driver.html)”）意义相同。实际上，该名称在设备注册时，会拷贝到dev.init\_name中。

id，用于标识该设备的ID。   
在“[Linux设备模型(6)\_Bus](http://www.wowotech.net/linux_kenrel/bus.html)”中有提过，内核允许存在多个名称相同的设备。而设备驱动的probe，依赖于名称，Linux采取的策略是：在bus的设备链表中查找device，和对应的device\_driver比对name，如果相同，则查看该设备是否已经绑定了driver（查看其dev->driver指针是否为空），如果已绑定，则不会执行probe动作，如果没有绑定，则以该device的指针为参数，调用driver的probe接口。   
因此，在driver的probe接口中，通过判断设备的ID，可以知道此次驱动的设备是哪个。

id\_auto，指示在注册设备时，是否自动赋予ID值（不需要人为指定啦，可以懒一点啦）。

num\_resources、resource，该设备的资源描述，由struct resource（include/linux/ioport.h）结构抽象。   
在Linux中，系统资源包括I/O、Memory、Register、IRQ、DMA、Bus等多种类型。这些资源大多具有独占性，不允许多个设备同时使用，因此Linux内核提供了一些API，用于分配、管理这些资源。   
当某个设备需要使用某些资源时，只需利用struct resource组织这些资源（如名称、类型、起始、结束地址等），并保存在该设备的resource指针中即可。然后在设备probe时，设备需求会调用资源管理接口，分配、使用这些资源。而内核的资源管理逻辑，可以判断这些资源是否已被使用、是否可被使用等等。

id\_entry，和内核模块相关的内容，暂不说明。

mfd\_cell，和MFD设备相关的内容，暂不说明。

archdata，一个奇葩的存在！！它的目的是为了保存一些architecture相关的数据，去看看arch/arm/include/asm/device.h中struct pdev\_archdata结构的定义，就知道这种放纵的设计有多么垃圾了。不管它了！！

2. 用于抽象Platform设备驱动的数据结构----“struct platform\_driver”：

1: /\* include/linux/platform\_device.h, line 173 \*/

2: struct platform\_driver {

3: int (\*probe)(struct platform\_device \*);

4: int (\*remove)(struct platform\_device \*);

5: void (\*shutdown)(struct platform\_device \*);

6: int (\*suspend)(struct platform\_device \*, pm\_message\_t state);

7: int (\*resume)(struct platform\_device \*);

8: struct device\_driver driver;

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

10: };

struct platform\_driver结构和struct device\_driver非常类似，无非就是提供probe、remove、suspend、resume等回调函数，这里不再细说。

另外这里有一个id\_table的指针，该指针和"[Linux设备模型(5)\_device和device driver](http://www.wowotech.net/linux_kenrel/device_and_driver.html)”所描述的of\_match\_table、acpi\_match\_table的功能类似：提供其它方式的设备probe。   
我们在"[Linux设备模型(5)\_device和device driver](http://www.wowotech.net/linux_kenrel/device_and_driver.html)”讲过，内核会在合适的时机检查device和device\_driver的名字，如果匹配，则执行probe。其实除了名称之外，还有一些宽泛的匹配方式，例如这里提到的各种match table，具体原理就先不罗嗦了，徒添烦恼！就当没看见，呵呵。

**3.2 Platform Device提供的API**

Platform Device主要提供设备的分配、注册等接口，供其它driver使用，具体包括：

1: /\* include/linux/platform\_device.h \*/

2: extern int platform\_device\_register(struct platform\_device \*);

3: extern void platform\_device\_unregister(struct platform\_device \*);

4:

5: extern void arch\_setup\_pdev\_archdata(struct platform\_device \*);

6: extern struct resource \*platform\_get\_resource(struct platform\_device \*,

7: unsigned int, unsigned int);

8: extern int platform\_get\_irq(struct platform\_device \*, unsigned int);

9: extern struct resource \*platform\_get\_resource\_byname(struct platform\_device \*,

10: unsigned int,

11: const char \*);

12: extern int platform\_get\_irq\_byname(struct platform\_device \*, const char \*);

13: extern int platform\_add\_devices(struct platform\_device \*\*, int);

14:

15: extern struct platform\_device \*platform\_device\_register\_full(

16: const struct platform\_device\_info \*pdevinfo);

17:

18: static inline struct platform\_device \*platform\_device\_register\_resndata(

19: struct device \*parent, const char \*name, int id,

20: const struct resource \*res, unsigned int num,

21: const void \*data, size\_t size)

22:

23: static inline struct platform\_device \*platform\_device\_register\_simple(

24: const char \*name, int id,

25: const struct resource \*res, unsigned int num)

26:

27: static inline struct platform\_device \*platform\_device\_register\_data(

28: struct device \*parent, const char \*name, int id,

29: const void \*data, size\_t size)

30:

31: extern struct platform\_device \*platform\_device\_alloc(const char \*name, int id);

32: extern int platform\_device\_add\_resources(struct platform\_device \*pdev,

33: const struct resource \*res,

34: unsigned int num);

35: extern int platform\_device\_add\_data(struct platform\_device \*pdev,

36: const void \*data, size\_t size);

37: extern int platform\_device\_add(struct platform\_device \*pdev);

38: extern void platform\_device\_del(struct platform\_device \*pdev);

39: extern void platform\_device\_put(struct platform\_device \*pdev);

platform\_device\_register、platform\_device\_unregister，Platform设备的注册/注销接口，和底层的device\_register等接口类似。

arch\_setup\_pdev\_archdata，设置platform\_device变量中的archdata指针。

platform\_get\_resource、platform\_get\_irq、platform\_get\_resource\_byname、platform\_get\_irq\_byname，通过这些接口，可以获取platform\_device变量中的resource信息，以及直接获取IRQ的number等等。

platform\_device\_register\_full、platform\_device\_register\_resndata、platform\_device\_register\_simple、platform\_device\_register\_data，其它形式的设备注册。调用者只需要提供一些必要的信息，如name、ID、resource等，Platform模块就会自动分配一个struct platform\_device变量，填充内容后，注册到内核中。

platform\_device\_alloc，以name和id为参数，动态分配一个struct platform\_device变量。

platform\_device\_add\_resources，向platform device中增加资源描述。

platform\_device\_add\_data，向platform device中添加自定义的数据（保存在pdev->dev.platform\_data指针中）。

platform\_device\_add、platform\_device\_del、platform\_device\_put，其它操作接口。

**3.3 Platform Driver提供的API**

Platform Driver提供struct platform\_driver的分配、注册等功能，具体如下：

1: /\* include/linux/platform\_device.h \*/

2:

3: extern int platform\_driver\_register(struct platform\_driver \*);

4: extern void platform\_driver\_unregister(struct platform\_driver \*);

5:

6: /\* non-hotpluggable platform devices may use this so that probe() and

7: \* its support may live in \_\_init sections, conserving runtime memory.

8: \*/

9: extern int platform\_driver\_probe(struct platform\_driver \*driver,

10: int (\*probe)(struct platform\_device \*));

11:

12: static inline void \*platform\_get\_drvdata(const struct platform\_device \*pdev)

13:

14: static inline void platform\_set\_drvdata(struct platform\_device \*pdev,

15: void \*data)

platform\_driver\_registe、platform\_driver\_unregister，platform driver的注册、注销接口。

platform\_driver\_probe，主动执行probe动作。

platform\_set\_drvdata、platform\_get\_drvdata，设置或者获取driver保存在device变量中的私有数据。

**3.4 懒人API**

又是注册platform device，又是注册platform driver，看着挺啰嗦的。不过内核想到了这点，所以提供一个懒人API，可以同时注册platform driver，并分配一个platform device：

1: extern struct platform\_device \*platform\_create\_bundle(

2: struct platform\_driver \*driver, int (\*probe)(struct platform\_device \*),

3: struct resource \*res, unsigned int n\_res,

4: const void \*data, size\_t size);

只要提供一个platform\_driver（要把driver的probe接口显式的传入），并告知该设备占用的资源信息，platform模块就会帮忙分配资源，并执行probe操作。对于那些不需要热拔插的设备来说，这种方式是最省事的了。

**3.5 Early platform device/driver**

内核启动时，要完成一定的初始化操作之后，才会处理device和driver的注册及probe，因此在这之前，常规的platform设备是无法使用的。但是在Linux中，有些设备需要尽早使用（如在启动过程中充当console输出的serial设备），所以platform模块提供了一种称作Early platform device/driver的机制，允许驱动开发人员，在开发驱动时，向内核注册可在内核早期启动过程中使用的driver。这些机制提供了如下接口：

1: extern int early\_platform\_driver\_register(struct early\_platform\_driver \*epdrv,

2: char \*buf);

3: extern void early\_platform\_add\_devices(struct platform\_device \*\*devs, int num);

4:

5: static inline int is\_early\_platform\_device(struct platform\_device \*pdev)

6: {

7: return !pdev->dev.driver;

8: }

9:

10: extern void early\_platform\_driver\_register\_all(char \*class\_str);

11: extern int early\_platform\_driver\_probe(char \*class\_str,

12: int nr\_probe, int user\_only);

13: extern void early\_platform\_cleanup(void);

early\_platform\_driver\_register，注册一个用于Early device的driver。

early\_platform\_add\_devices，添加一个Early device。

is\_early\_platform\_device，判断指定的device是否是Early device。

early\_platform\_driver\_register\_all，将指定class的所有driver注册为Early device driver。

early\_platform\_driver\_probe，probe指定class的Early device。

early\_platform\_cleanup，清除所有的Early device/driver。

**4. Platform模块的内部动作解析**

**4.1 Platform模块的初始化**

Platform模块的初始化是由drivers/base/platform.c中platform\_bus\_init接口完成的，该接口的实现和动作如下：

1: int \_\_init platform\_bus\_init(void)

2: {

3: int error;

4:

5: early\_platform\_cleanup();

6:

7: error = device\_register(&platform\_bus);

8: if (error)

9: return error;

10: error = bus\_register(&platform\_bus\_type);

11: if (error)

12: device\_unregister(&platform\_bus);

13: return error;

14: }

early\_platform\_cleanup，清除所有和Early device/driver相关的代码。因为执行到这里的时候，证明系统已经完成了Early阶段的启动，转而进行正常的设备初始化、启动操作，所以不再需要Early Platform相关的东西。

device\_register，注册一个名称为platform\_bus的设备，该设备的定义非常简单，只包含init\_name（为“platform”）。该步骤会在sysfs中创建“/sys/devices/platform/”目录，所有的Platform设备，都会包含在此目录下。

bus\_register，注册一个名称为platform\_bus\_type的bus，该bus的定义如下：   
struct bus\_type platform\_bus\_type = {   
        .name           = "platform",   
        .dev\_attrs      = platform\_dev\_attrs,   
        .match          = platform\_match,   
        .uevent         = platform\_uevent,   
        .pm             = &platform\_dev\_pm\_ops,   
};   
该步骤会在sysfs中创建“/sys/bus/platform/”目录，同时，结合“[Linux设备模型(6)\_Bus](http://www.wowotech.net/linux_kenrel/bus.html)”的描述，会在“/sys/bus/platform/”目录下，创建uevent attribute（/sys/bus/platform/uevent）、devices目录、drivers目录、drivers\_probe和drivers\_autoprobe两个attribute（/sys/bus/platform/drivers\_probe和/sys/bus/platform/drivers\_autoprobe）。

**4.2 platform device和platform driver的注册**

结合第3章的描述，platform device和platform driver的注册，由platform\_device\_add和platform\_driver\_register两个接口实际实现。其内部动作分别如下。

platform\_device\_add的内部动作：

如果该设备没有指定父设备，将其父设备设置为platform\_bus，即“/sys/devices/platform/”所代表的设备，这时该设备的sysfs目录即为“/sys/devices/platform/xxx\_device”。

将该设备的bus指定为platform\_bus\_type（pdev->dev.bus = &platform\_bus\_type）。

根据设备的ID，修改或者设置设备的名称。对于多个同名的设备，可以使用ID区分，在这里将实际名称修改为“name.id”的形式。

调用resource模块的insert\_resource接口，将该设备需要使用的resource统一管理起来（我们知道，在这之前，只是声明了本设备需要使用哪些resource，但resource模块并不知情，也就无从管理，因此需要告知）。

调用device\_add接口，将内嵌的struct device变量添加到内核中。

platform\_driver\_register的内部动作：

将该driver的bus指定为platform\_bus\_type（drv->driver.bus = &platform\_bus\_type）。

如果该platform driver提供了probe、remove、shutdown等回调函数，将该它内嵌的struct driver变量的probe、remove、shutdown等指针，设置为platform模块提供函数，包括platform\_drv\_probe、platform\_drv\_remove和platform\_drv\_shutdown。因为probe等动作会从struct driver变量开始，经过platform\_drv\_xxx等接口的转接，就可以到达platform diver自身的回调函数中。

调用driver\_register接口，将内嵌的struct driver变量添加到内核中。

**4.3 platform设备的probe**

我们在“[Linux设备模型(6)\_Bus](http://www.wowotech.net/linux_kenrel/bus.html)”中讲过，设备的probe，都发生在向指定的bus添加device或者device\_driver时，由bus模块的bus\_probe\_device，或者device\_driver模块driver\_attach接口触发。这里就不再详细描述了。