**设备树Device Tree详解**

Table of Contents

[**设备树Device Tree详解** 1](#_Toc133523808)

[1. 设备树（Device  Tree）基本概念及作用 3](#_Toc133523809)

[2. 设备树的组成和使用 3](#_Toc133523810)

[2.1. DTS和DTSI 3](#_Toc133523811)

[2.2. DTC 4](#_Toc133523812)

[2.3. DTB 4](#_Toc133523813)

[2.4. Bootloader 4](#_Toc133523814)

[3. 设备树中dts、dtsi文件的基本语法 5](#_Toc133523815)

[3.1. chosen node 7](#_Toc133523816)

[3.2. aliases node 7](#_Toc133523817)

[3.3. memory node 8](#_Toc133523818)

[3.4.  其他节点 8](#_Toc133523819)

[3.4.1. Reg属性 8](#_Toc133523820)

[3.4.2. compatible属性 9](#_Toc133523821)

[3.4.3. interrupts属性 10](#_Toc133523822)

[3.4.4. ranges属性 10](#_Toc133523823)

[4. DTB相关结构 11](#_Toc133523824)

[4.1. Header 12](#_Toc133523825)

[4.2.device-tree structure 12](#_Toc133523826)

[4.3. 字符串块 13](#_Toc133523827)

[4.4. memory reserve map 13](#_Toc133523828)

[5. 解析DTB的函数及相关数据结构 14](#_Toc133523829)

[5.1. machine\_desc结构 14](#_Toc133523830)

[5.2. 设备节点结构体 15](#_Toc133523831)

[5.3. 属性结构体 15](#_Toc133523832)

[5.4. uboot下的相关结构体 16](#_Toc133523833)

[6. DTB加载及解析过程 16](#_Toc133523834)

[7. OF的API接口 27](#_Toc133523835)

# 1. [设备树](https://so.csdn.net/so/search?q=%E8%AE%BE%E5%A4%87%E6%A0%91&spm=1001.2101.3001.7020)（Device  Tree）基本概念及作用

在内核源码中，存在大量对板级细节信息描述的代码。这些代码充斥在/arch/arm/plat-xxx和/arch/arm/mach-xxx目录，对内核而言这些platform设备、resource、i2c\_board\_info、spi\_board\_info以及各种硬件的platform\_data绝大多数纯属垃圾冗余代码。为了解决这一问题，ARM内核版本3.x之后引入了原先在Power PC等其他体系架构已经使用的Flattened Device Tree。

“A data structure by which bootloaders pass hardware layout to Linux in a device-independent manner, simplifying hardware probing.”开源文档中对设备树的描述是，一种描述硬件资源的数据结构，它通过bootloader将硬件资源传给内核，使得内核和硬件资源描述相对独立(也就是说\*.dtb文件由Bootloader读入内存，之后由内核来解析)。

Device Tree可以描述的信息包括CPU的数量和类别、内存基地址和大小、总线和桥、外设连接、中断控制器和中断使用情况、GPIO控制器和GPIO使用情况、Clock控制器和Clock使用情况。

另外，设备树对于可热插拔的热备不进行具体描述，它只描述用于控制该热插拔设备的控制器。

设备树的主要优势：对于同一SOC的不同主板，只需更换设备树文件.dtb即可实现不同主板的无差异支持，而无需更换内核文件。

注：要使得3.x之后的内核支持使用设备树，除了内核编译时需要打开相对应的选项外，bootloader也需要支持将设备树的数据结构传给内核。

# 2. 设备树的组成和使用

设备树包含DTC（device tree compiler），DTS（device tree source和DTB（device tree blob）。其对应关系如图1-1所示：

A picture containing table

Description automatically generated

图1-1 DTS、DTC、DTB之间的关系

## 2.1. DTS和DTSI

\*.dts文件是一种ASCII文本对Device Tree的描述，放置在内核的/arch/arm/boot/dts目录。一般而言，一个\*.dts文件对应一个ARM的machine。

\*.dtsi文件作用：由于一个SOC可能有多个不同的电路板，而每个电路板拥有一个 \*.dts。这些dts势必会存在许多共同部分，为了减少代码的冗余，设备树将这些共同部分提炼保存在\*.dtsi文件中，供不同的dts共同使用。\*.dtsi的使用方法，类似于C语言的头文件，在dts文件中需要进行include \*.dtsi文件。当然，dtsi本身也支持include 另一个dtsi文件。

## 2.2. DTC

DTC为编译工具，它可以将.dts文件编译成.dtb文件。DTC的源码位于内核的scripts/dtc目录，内核选中CONFIG\_OF，编译内核的时候，主机可执行程序DTC就会被编译出来。 即scripts/dtc/Makefile中

hostprogs-y := dtc

always := $(hostprogs-y)

在内核的arch/arm/boot/dts/Makefile中，若选中某种SOC，则与其对应相关的所有dtb文件都将编译出来。在linux下，make dtbs可单独编译dtb。以下截取了TEGRA平台的一部分。

ifeq ($(CONFIG\_OF),y)

dtb-$(CONFIG\_ARCH\_TEGRA) += tegra20-harmony.dtb \

tegra30-beaver.dtb \

tegra114-dalmore.dtb \

tegra124-ardbeg.dtb

## 2.3. DTB

DTC编译\*.dts生成的二进制文件(\*.dtb)，bootloader在引导内核时，会预先读取\*.dtb到内存，进而由内核解析。

## 2.4. Bootloader

Bootloader需要将设备树在内存中的地址传给内核。在ARM中通过bootm或bootz命令来进行传递。bootm [kernel\_addr] [initrd\_address] [dtb\_address]，其中kernel\_addr为内核镜像的地址，initrd为initrd的地址，dtb\_address为dtb所在的地址。若initrd\_address为空，则用“-”来代替。

# 3. 设备树中dts、dtsi文件的基本语法

DTS的基本语法范例，如图3-1 所示。

它包括一系列节点，以及描述节点的属性。

“/”为root节点。在一个.dts文件中，有且仅有一个root节点；在root节点下有“node1”，“node2”子节点，称root为“node1”和“node2”的parent节点，除了root节点外，每个节点有且仅有一个parent；其中子节点node1下还存在子节点“child-nodel1”和“child-node2”。

注：如果看过内核/arch/arm/boot/dts目录的读者看到这可能有一个疑问。在每个.dsti和.dts中都会存在一个“/”根节点，那么如果在一个设备树文件中include一个.dtsi文件，那么岂不是存在多个“/”根节点了么。其实不然，编译器DTC在对.dts进行编译生成dtb时，会对[node](https://so.csdn.net/so/search?q=node&spm=1001.2101.3001.7020)进行合并操作，最终生成的dtb只有一个root node。Dtc会进行合并操作这一点从属性上也可以得到验证。这个稍后做讲解。

在节点的{}里面是描述该节点的属性（property），即设备的特性。它的值是多样化的：

1.它可以是字符串string，如①；也可能是字符串数组string-list，如②

2.它也可以是32 bit unsigned integers，如cell⑧，整形用<>表示

3.它也可以是binary data，如③，十六进制用[]表示

4.它也可能是空，如⑦

Text

Description automatically generated

图3-1  DTS的基本语法范例

在/arch/arm/boot/dts/目录中有一个文件skeleton.dtsi，该文件为各ARM vendor共用的一些硬件定义信息。以下为skeleton.dtsi的全部内容。

/ {

#address-cells = <1>;

#size-cells = <1>;

chosen { };

aliases { };

memory { device\_type = "memory"; reg = <0 0>; };

};

如上，属性# address-cells的值为1，它代表以“/”根节点为parent的子节点中，reg属性中存在一个address值；#size-cells的值为1，它代表以“\” 根节点为parent的子节点中，reg属性中存在一个size值。即父节点的# address-cells和#size-cells决定了子节点的address和size的长度；Reg的组织形式为reg =

下面列举例子，对一些典型节点进行具体描述。

## 3.1. chosen node

chosen {

bootargs = "tegraid=40.0.0.00.00 vmalloc=256M video=tegrafb console=ttyS0,115200n8 earlyprintk";

};

chosen node 主要用来描述由系统指定的runtime parameter，它并没有描述任何硬件设备节点信息。原先通过tag list传递的一些linux kernel运行的参数，可以通过chosen节点来传递。如command line可以通过bootargs这个property来传递。如果存在chosen node，它的parent节点必须为“/”根节点。

## 3.2. aliases node

aliases {

i2c6 = &pca9546\_i2c0;

i2c7 = &pca9546\_i2c1;

i2c8 = &pca9546\_i2c2;

i2c9 = &pca9546\_i2c3;

};

aliases node用来定义别名，类似C++中引用。上面是一个在.dtsi中的典型应用，当使用i2c6时，也即使用pca9546\_i2c0，使得引用节点变得简单方便。例：当.dts  include 该.dtsi时，将i2c6的status属性赋值为okay，则表明该主板上的pca9546\_i2c0处于enable状态；反之，status赋值为disabled，则表明该主板上的pca9546\_i2c0处于disenable状态。如下是引用的具体例子：

&i2c6 {--------------这里&i2c6到底是label还是alias???

status = "okay";

};------------------在\*.dtsi中大多默认为设备为disable，然后在\*.dts中将其enable,进行重写使能。

## 3.3. memory node

memory {

device\_type = "memory";

reg = <0x00000000 0x20000000>; /\* 512 MB \*/

};

对于memory node，device\_type必须为memory，由之前的描述可以知道该memory node是以0x00000000为起始地址，以0x20000000为结束地址的512MB的空间。

一般而言，在.dts中不对memory进行描述，而是通过bootargs中类似521M@0x00000000的方式传递给内核。

## 3.4.  其他节点

由于其他设备节点依据属性进行描述，具有类似的形式。接下来的部分主要分析各种属性的含义及作用，并结合相关的例子进行阐述。

### **3.4.1. Reg属性**

在device node 中，reg是描述memory-mapped IO register的offset和length。子节点的reg属性address和length长度取决于父节点对应的#address-cells和#size-cells的值。例：

Shape, arrow

Description automatically generated

在上述的aips节点中，存在子节点spda。spda中的中reg为<0x70000000 0x40000 >，其0x700000000为address，0x40000为size。这一点在图3-1下有作介绍。

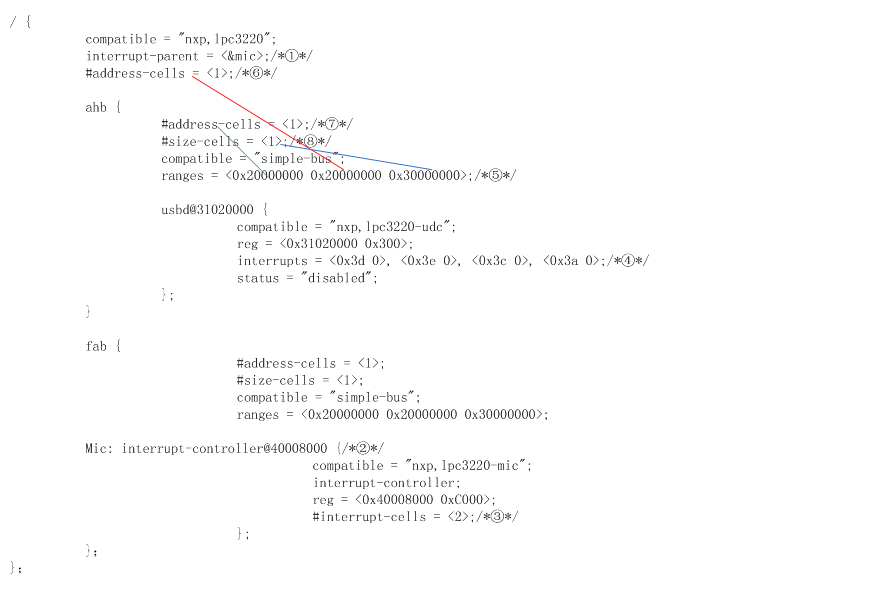
这里补充的一点是：设备节点的名称格式node-name@unit-address，节点名称用node-name唯一标识，为一个ASCII字符串。其中@unit-address为可选项，可以不作描述。unit-address的具体格式和设备挂载在哪个bus上相关。如：cpu的unit-address从0开始编址，以此加1；本例中，aips为0x70000000。

### **3.4.2. compatible属性**

在①中，compatible属性为string list，用来将设备匹配对应的driver驱动，优先级为从左向右。本例中spba的驱动优先考虑“fsl，aips-bus”驱动；若没有“fsl，aips-bus”驱动，则用字符串“simple-bus”来继续寻找合适的驱动。即compatible实现了原先内核版本3.x之前，platform\_device中.name的功能，至于具体的实现方法，本文后面会做讲解。

注：对于“/”root节点，它也存在compatible属性，用来匹配machine type。具体说明将在后面给出。

### **3.4.3. interrupts属性**



设备节点通过interrupt-parent来指定它所依附的中断控制器，当节点没有指定interrupt-parent时，则从parent节点中继承。上面例子中，root节点的interrupt-parent = <&mic>。这里使用了引用，即mic引用了②中的inrerrupt-controller @40008000；root节点的子节点并没有指定interrupt-controller，如ahb、fab，它们均使用从根节点继承过来的mic，即位于0x40008000的中断控制器。

若子节点使用到中断(中断号、触发方法等等)，则需用interrupt属性来指定，该属性的数值长度受中断控制器中#inrerrupt-controller值③控制，即interrupt属性<>中数值的个数为#inrerrupt-controller的值；本例中#inrerrupt-controller=<2>，因而④中interrupts的值为<0x3d 0>形式，具体每个数值的含义由驱动实现决定。

### **3.4.4. ranges属性**

ranges属性为地址转换表，这在pcie中使用较为常见，它表明了该设备在到parent节点中所对用的地址映射关系。ranges格式长度受当前节点#address-cell、parent节点#address-cells、当前节点#size-cell所控制。顺序为ranges=<前节点#address-cell, parent节点#address-cells , 当前节点#size-cell。在本例中，当前节点#address-cell=<1>,对应于⑤中的第一个0x20000000；parent节点#address-cells=<1>，对应于⑤中的第二个0x20000000；当前节点#size-cell=<1>，对应于⑤中的0x30000000。即ahb0节点所占空间从0x20000000地址开始，对应于父节点的0x20000000地址开始的0x30000000地址空间大小。

注：对于相同名称的节点，dtc会根据定义的先后顺序进行合并，其相同属性，取后定义的那个。

# 4. DTB相关结构

本节讲下.dts编译生成的dtb文件，其布局结构。

Shape

Description automatically generated with low confidence

DTB由三部分组成：头（Header）、结构块（device-tree structure）、字符串块（string block）。下面将详细介绍这三部分的内容。

## 4.1. Header

在\kernel\include\linux\of\_fdt.h文件中有相关定义

Shape

Description automatically generated with medium confidence

## 4.2.****device-tree structure****

设备树结构块是一个线性化的结构体，是设备树的主体，以节点的形式保存了主板上的设备信息。

在结构块中，以宏OF\_DT\_BEGIN\_NODE标志一个节点的开始，以宏OF\_DT\_END\_NODE标识一个节点的结束，整个结构块以宏OF\_DT\_END (0x00000009)结束。在\kernel\include\linux\of\_fdt.h中有相关定义，我们把这些宏称之为token。

（1）FDT\_BEGIN\_NODE (0x00000001)。该token描述了一个node的开始位置，紧挨着该token的就是node name（包括unit address）

（2）FDT\_END\_NODE (0x00000002)。该token描述了一个node的结束位置。

（3）FDT\_PROP (0x00000003)。该token描述了一个property的开始位置，该token之后是两个u32的数据，分别是length和name offset。length表示该property value data的size。name offset表示该属性字符串在device tree strings block的偏移值。length和name offset之后就是长度为length具体的属性值数据。

（4）FDT\_NOP (0x00000004)。

（5）FDT\_END (0x00000009)。该token标识了一个DTB的结束位置。

一个节点的结构如下：

(1)节点开始标志：一般为OF\_DT\_BEGIN\_NODE（0x00000001）。

(2)节点路径或者节点的单元名(version<3以节点路径表示，version>=0x10以节点单元名表示)

(3)填充字段（对齐到四字节）

(4)节点属性。每个属性以宏OF\_DT\_PROP(0x00000003)开始，后面依次为属性值的字节长度(4字节)、属性名称在字符串块中的偏移量(4字节)、属性值和填充（对齐到四字节）。

(5)如果存在子节点，则定义子节点。

(6)节点结束标志OF\_DT\_END\_NODE(0x00000002)。

## 4.3. 字符串块

通过节点的定义知道节点都有若干属性，而不同的节点的属性又有大量相同的属性名称，因此将这些属性名称提取出一张表，当节点需要应用某个属性名称时，直接在属性名字段保存该属性名称在字符串块中的偏移量。

## 4.4. memory reserve map

这个区域包括了若干的reserve memory描述符。每个reserve memory描述符是由address和size组成。其中address和size都是用U64来描述。

有些系统，我们也许会保留一些memory有特殊用途（例如DTB或者initrd image），或者在有些DSP+ARM的SOC platform上，有些memory被保留用于ARM和DSP进行信息交互。这些保留内存不会进入内存管理系统。

# 5. 解析DTB的函数及相关数据结构

## 5.1. machine\_desc结构

Shape

Description automatically generated

内核将机器信息记录为machine\_desc结构体（该定义在/arch/arm/include/asm/mach/arch.h），并保存在\_arch\_info\_begin到\_arch\_info\_end之间（\_arch\_info\_begin，\_arch\_info\_end为虚拟地址，是编译内核时指定的，此时mmu还未进行初始化。它其实通过汇编完成地址偏移操作）

machine\_desc结构体用宏MACHINE\_START进行定义，一般在/arch/arm/子目录，与板级相关的文件中进行成员函数及变量的赋值。由linker将machine\_desc聚集在.arch.info.init节区形成列表。

bootloader引导内核时，ARM寄存器r2会将.dtb的首地址传给内核，内核根据该地址，解析.dtb中根节点的compatible属性，将该属性与内核中预先定义machine\_desc结构体的dt\_compat成员做匹配，得到最匹配的一个machine\_desc。

在代码中，内核通过在start\_kernel->setup\_arch中调用setup\_machine\_fdt来实现上述功能，该函数的具体实现可参见/arch/arm/kernel/devtree.c。

## 5.2. 设备节点结构体

1.Shape

Description automatically generated

记录节点信息的结构体。.dtb经过解析之后将以device\_node列表的形式存储节点信息。

## 5.3. 属性结构体

Shape

Description automatically generated

device\_node结构体中的成员结构体，用于描述节点属性信息。

## 5.4. uboot下的相关结构体

首先我们看下uboot用于记录os、initrd、fdt信息的数据结构bootm\_headers，其定义在/include/image.h中，这边截取了其中与dtb相关的一小部分。

Shape, rectangle

Description automatically generated

fit\_hdr\_fdt指向DTB设备树镜像的头。

lmb为uboot下的一种内存管理机制，全称为logical memory blocks。用于管理镜像的内存。lmb所记录的内存信息最终会传递给kernel。这里对lmb不做展开描述。在/include/lmb.h和/lib/lmb.c中有对lmb的接口和定义的具体描述。有兴趣的读者可以看下，所包含的代码量不多。

# 6. DTB加载及解析过程

Text

Description automatically generated

先从uboot里的do\_bootm出发，根据之前描述，DTB在内存中的地址通过bootm命令进行传递。在bootm中，它会根据所传进来的DTB地址，对DTB所在内存做一系列操作，为内核解析DTB提供保证。上图为对应的函数调用关系图。

在do\_bootm中，主要调用函数为do\_bootm\_states，第四个参数为bootm所要处理的阶段和状态。

在do\_bootm\_states中，bootm\_start会对lmb进行初始化操作，lmb所管理的物理内存块有三种方式获取。起始地址，优先级从上往下：

1. 环境变量“bootm\_low”

2. 宏CONFIG\_SYS\_SDRAM\_BASE（在tegra124中为0x80000000）

3. gd->bd->bi\_dram[0].start

大小：

1. 环境变量“bootm\_size”

2. gd->bd->bi\_dram[0].size

经过初始化之后，这块内存就归lmb所管辖。接着，调用bootm\_find\_os进行kernel镜像的相关操作，这里不具体阐述。

还记得之前讲过bootm的三个参数么，第一个参数内核地址已经被bootm\_find\_os处理，而接下来的两个参数会在bootm\_find\_other中执行操作。

首先，bootm\_find\_other根据第二个参数找到ramdisk的地址，得到ramdisk的镜像；然后根据第三个参数得到DTB镜像，同检查kernel和ramdisk镜像一样，检查DTB镜像也会进行一系列的校验工作，如果校验错误，将无法正常启动内核。另外，uboot在确认DTB镜像无误之后，会将该地址保存在环境变量“fdtaddr”中。

接着，uboot会把DTB镜像reload一次，使得DTB镜像所在的物理内存归lmb所管理：①boot\_fdt\_add\_mem\_rsv\_regions会将原先的内存DTB镜像所在的内存置为reserve，保证该段内存不会被其他非法使用，保证接下来的reload数据是正确的；②boot\_relocate\_fdt会在bootmap区域中申请一块未被使用的内存，接着将DTB镜像内容复制到这块区域（即归lmb所管理的区域）

注：若环境变量中，指定“fdt\_high”参数，则会根据该值，调用lmb\_alloc\_base函数来分配DTB镜像reload的地址空间。若分配失败，则会停止bootm操作。因而，不建议设置fdt\_high参数。

接下来，do\_bootm会根据内核的类型调用对应的启动函数。与linux对应的是do\_bootm\_linux。

① boot\_prep\_linux

为启动后的kernel准备参数

② boot\_jump\_linux

Shape, rectangle

Description automatically generated

以上是boot\_jump\_linux的片段代码，可以看出：若使用DTB，则原先用来存储ATAG的寄存器R2，将会用来存储.dtb镜像地址。

boot\_jump\_linux最后将调用kernel\_entry，将.dtb镜像地址传给内核。

下面我们来看下内核的处理部分：

在arch/arm/kernel/head.S中，有这样一段：

Shape, rectangle

Description automatically generated

\_vet\_atags定义在/arch/arm/kernel/head-common.S中，它主要对DTB镜像做了一个简单的校验。

Shape

Description automatically generated

真正解析处理dbt的开始部分，是setup\_arch->setup\_machine\_fdt。这部分的处理在第五部分的machine\_mdesc中有提及。

Graphical user interface, text, application, chat or text message

Description automatically generated

如图，是setup\_machine\_fdt中的解析过程。

解析chosen节点将对boot\_command\_line进行初始化。

解析根节点的{size，address}将对dt\_root\_size\_cells，dt\_root\_addr\_cells进行初始化。为之后解析memory等其他节点提供依据。

解析memory节点，将会把节点中描述的内存，加入memory的bank。为之后的内存初始化提供条件。

解析设备树在函数unflatten\_device\_tree中完成，它将.dtb解析成device\_node结构（第五部分有其定义），并构成单项链表，以供OF的API接口使用。

下面主要结合代码分析：/drivers/of/fdt.c

Shape, rectangle

Description automatically generated

Shape

Description automatically generated with medium confidence

Shape

Description automatically generated with medium confidence

Shape

Description automatically generated with low confidence

A picture containing shape

Description automatically generated

A picture containing shape

Description automatically generated

**总的归纳为**：

① kernel入口处获取到uboot传过来的.dtb镜像的基地址

② 通过early\_init\_dt\_scan()函数来获取kernel初始化时需要的bootargs和cmd\_line等系统引导参数。

③ 调用unflatten\_device\_tree函数来解析dtb文件，构建一个由device\_node结构连接而成的单向链表，并使用全局变量of\_allnodes保存这个链表的头指针。

④ 内核调用OF的API接口，获取of\_allnodes链表信息来初始化内核其他子系统、设备等。

# 7. OF的API接口

OF的接口函数在/drivers/of/目录下，有of\_i2c.c、of\_mdio.c、of\_mtd.c、Adress.c等等

这里将列出几个常用的API接口。

1. 用来查找在dtb中的根节点

unsigned long \_\_init of\_get\_flat\_dt\_root(void)

2. 根据deice\_node结构的full\_name参数，在全局链表of\_allnodes中，查找合适的device\_node

struct device\_node \*of\_find\_node\_by\_path(const char \*path)

例如：

struct device\_node \*cpus;

cpus=of\_find\_node\_by\_path("/cpus");

3. 若from=NULL，则在全局链表of\_allnodes中根据name查找合适的device\_node

struct device\_node \*of\_find\_node\_by\_name(struct device\_node \*from,const char \*name)

例如：

struct device\_node \*np;

np = of\_find\_node\_by\_name(NULL,"firewire");

4. 根据设备类型查找相应的device\_node

struct device\_node \*of\_find\_node\_by\_type(struct device\_node \*from,const char \*type)

例如：

struct device\_node \*tsi\_pci;

tsi\_pci= of\_find\_node\_by\_type(NULL,"pci");

5. 根据compatible字符串查找device\_node

struct device\_node \*of\_find\_compatible\_node(struct device\_node \*from,const char \*type, const char \*compatible)

6. 根据节点属性的name查找device\_node

struct device\_node \*of\_find\_node\_with\_property(struct device\_node \*from,const char \*prop\_name)

7. 根据phandle查找device\_node

struct device\_node \*of\_find\_node\_by\_phandle(phandle handle)

8. 根据alias的name获得设备id号

int of\_alias\_get\_id(struct device\_node \*np, const char \*stem)

9. device node计数增加/减少

struct device\_node \*of\_node\_get(struct device\_node \*node)

void of\_node\_put(struct device\_node \*node)

10. 根据property结构的name参数，在指定的device node中查找合适的property

struct property \*of\_find\_property(const struct device\_node \*np,const char \*name,int \*lenp)

11. 根据property结构的name参数，返回该属性的属性值

const void \*of\_get\_property(const struct device\_node \*np, const char \*name,int \*lenp)

12. 根据compat参数与device node的compatible匹配，返回匹配度

int of\_device\_is\_compatible(const struct device\_node \*device,const char \*compat)

13. 获得父节点的device node

struct device\_node \*of\_get\_parent(const struct device\_node \*node)

14. 将matches数组中of\_device\_id结构的name和type与device node的compatible和type匹配，返回匹配度最高的of\_device\_id结构

const struct of\_device\_id \*of\_match\_node(const struct of\_device\_id \*matches,const struct device\_node \*node)

15. 根据属性名propname，读出属性值中的第index个u32数值给out\_value

int of\_property\_read\_u32\_index(const struct device\_node \*np,const char \*propname,u32 index, u32 \*out\_value)

16. 根据属性名propname，读出该属性的数组中sz个属性值给out\_values

int of\_property\_read\_u8\_array(const struct device\_node \*np,const char \*propname, u8 \*out\_values, size\_t sz)

int of\_property\_read\_u16\_array(const struct device\_node \*np,const char \*propname, u16 \*out\_values, size\_t sz)

int of\_property\_read\_u32\_array(const struct device\_node \*np,const char \*propname, u32 \*out\_values,size\_t sz)

17. 根据属性名propname，读出该属性的u64属性值

int of\_property\_read\_u64(const struct device\_node \*np, const char \*propname,u64 \*out\_value)

18. 根据属性名propname，读出该属性的字符串属性值

int of\_property\_read\_string(struct device\_node \*np, const char \*propname,const char \*\*out\_string)

19. 根据属性名propname，读出该字符串属性值数组中的第index个字符串

int of\_property\_read\_string\_index(struct device\_node \*np, const char \*propname,int index, const char \*\*output)

20. 读取属性名propname中，字符串属性值的个数

int of\_property\_count\_strings(struct device\_node \*np, const char \*propname)

21. 读取该设备的第index个irq号

unsigned int irq\_of\_parse\_and\_map(struct device\_node \*dev, int index)

22. 读取该设备的第index个irq号，并填充一个irq资源结构体

int of\_irq\_to\_resource(struct device\_node \*dev, int index, struct resource \*r)

23. 获取该设备的irq个数

int of\_irq\_count(struct device\_node \*dev)

24. 获取设备寄存器地址，并填充寄存器资源结构体

int of\_address\_to\_resource(struct device\_node \*dev, int index,struct resource \*r)

const \_\_be32 \*of\_get\_address(struct device\_node \*dev, int index, u64 \*size,unsigned int \*flags)

25. 获取经过映射的寄存器虚拟地址

void \_\_iomem \*of\_iomap(struct device\_node \*np, int index)

24. 根据device\_node查找返回该设备对应的platform\_device结构

struct platform\_device \*of\_find\_device\_by\_node(struct device\_node \*np)

25. 根据device node，bus id以及父节点创建该设备的platform\_device结构

struct platform\_device \*of\_device\_alloc(struct device\_node \*np,const char \*bus\_id,struct device \*parent)

static struct platform\_device \*of\_platform\_device\_create\_pdata(struct device\_node \*np,const char \*bus\_id,

void \*platform\_data,struct device \*parent)

26. 遍历of\_allnodes中的节点挂接到of\_platform\_bus\_type总线上,由于此时of\_platform\_bus\_type总线上还没有驱动,所以此时不进行匹配

int of\_platform\_bus\_probe(struct device\_node \*root,const struct of\_device\_id \*matches,struct device \*parent)