基础7-Linux 设备树（Device Tree）（引用）

转自：http://blog.csdn.net/machiner1/article/details/47805069

----------Based on linux 3.10.24 source code

参考/documentation/devicetree/Booting-without-of.txt文档

目录

1. 设备树（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：

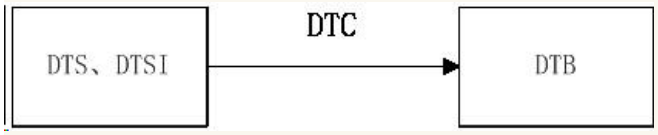


图1-1 DTS、DTC、DTB之间的关系

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-node1”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

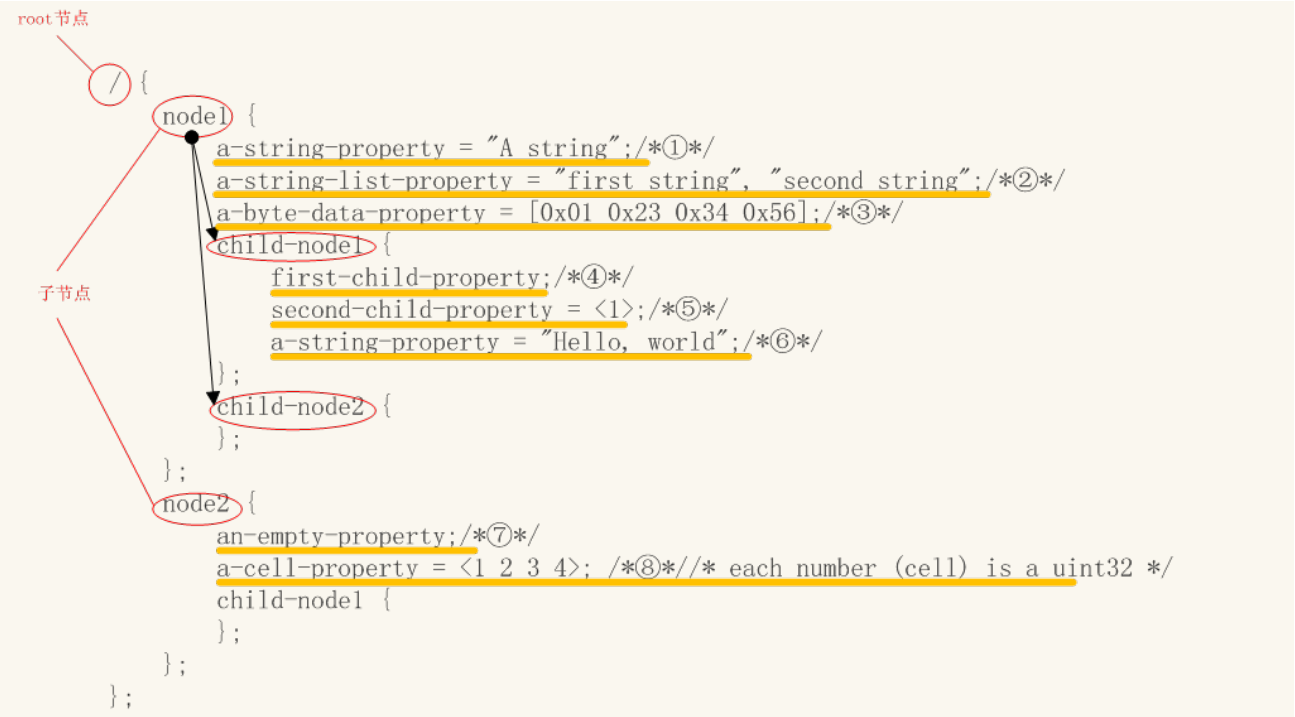


图3-1 DTS的基本语法范例

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。

3.3. 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。

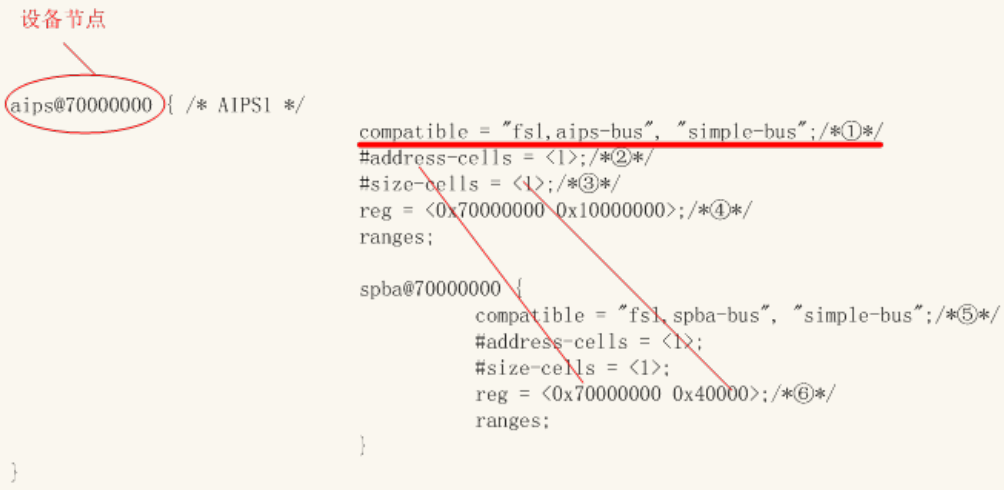
一般而言，在.dts中不对memory进行描述，而是通过bootargs中类似521M@0x00000000的方式传递给内核。

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：



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

```
/ {
        compatible = "nxp,lpc3220";
        interrupt-parent = <&mic>;/*①*/
        #address-cells = <1>;/*⑥*/

        ahb {
                #address-cells = <1>;/*⑦*/
                #size-cells = <1>;/*⑧*/
                compatible = "simple-bus";
                ranges = <0x20000000 0x20000000 0x30000000>;/*⑤*/

                usbd@31020000 {
                        compatible = "nxp,lpc3220-udc";
                        reg = <0x31020000 0x300>;
                        interrupts = <0x3d 0>, <0x3e 0>, <0x3c 0>, <0x3a 0>;/*④*/
                        status = "disabled";
                };
        }

        fab {
                        #address-cells = <1>;
                        #size-cells = <1>;
                        compatible = "simple-bus";
                        ranges = <0x20000000 0x20000000 0x30000000>;

        Mic: interrupt-controller@40008000 {/*②*/
                                compatible = "nxp,lpc3220-mic";
                                interrupt-controller;
                                reg = <0x40008000 0xC000>;
                                #interrupt-cells = <2>;/*③*/
                        };
        };
};
```

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。

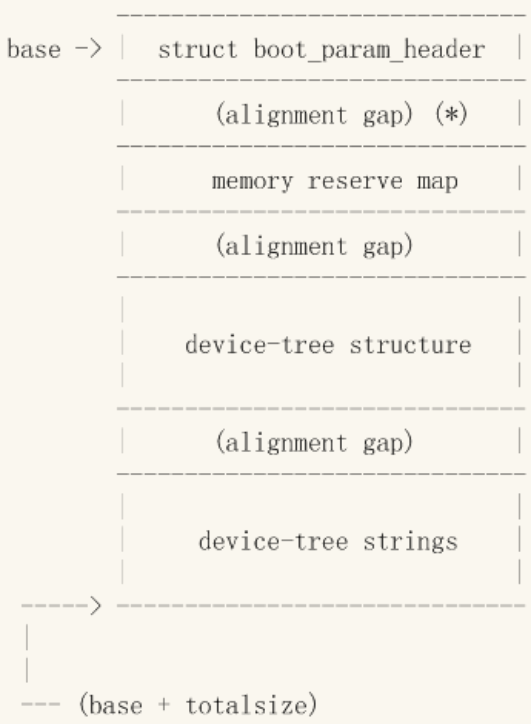
3.4.4. ranges属性

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。



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

```
struct boot_param_header {
        __be32  magic;                  /* 设备树幻数，固定为0xd00dfeed */
        __be32  totalsize;              /* 整个设备树的大小 */
        __be32  off_dt_struct;          /* 保存结构块在整个设备树中的偏移 */
        __be32  off_dt_strings;         /* 保存的字符串块在设备树中的偏移 */
        __be32  off_mem_rsvmap;         /* 保留内存区，该区保留了不能被内核动态分配的内存空间 */
        __be32  version;                /* 设备树版本 */
        __be32  last_comp_version;      /* 向下兼容版本号 */
        __be32  boot_cpuid_phys;        /* 为在多核处理器中用于启动的主cpu的物理id */
        __be32  dt_strings_size;        /* 字符串块大小 */
        __be32  dt_struct_size;         /* 结构块大小 */
};
```

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

```
struct machine_desc {
          unsigned int                    nr;                     /* architecture number          */
          const char              *name;              /* architecture name */
          unsigned long                   atag_offset;            /* tagged list (relative) */
          const char *const       *dt_compat;         /* array of device tree
                                                          * 'compatible' strings         */

          unsigned int                    nr_irqs;   /* number of IRQs */

#ifdef CONFIG_ZONE_DMA
          unsigned long                   dma_zone_size;          /* size of DMA-able area */
#endif

          unsigned int                    video_start;            /* start of video RAM*/
          unsigned int                    video_end; /* end of video RAM  */

          unsigned char                   reserve_lp0 :1;         /* never has lp0      */
          unsigned char                   reserve_lp1 :1;         /* never has lp1      */
          unsigned char                   reserve_lp2 :1;         /* never has lp2      */
          char                            restart_mode;           /* default restart mode         */
          struct smp_operations*smp;              /* SMP operations     */
          void                            (*fixup)(struct tag *, char **,
                                                   struct meminfo *);
          void                            (*reserve)(void);/* reserve mem blocks     */
          void                            (*map_io)(void);/* IO mapping function     */
          void                            (*init_early)(void);
          void                            (*init_irq)(void);
          void                            (*init_time)(void);
          void                            (*init_machine)(void);
          void                            (*init_late)(void);
#ifdef CONFIG_MULTI_IRQ_HANDLER
          void                            (*handle_irq)(struct pt_regs *);
#endif
          void                            (*restart)(char, const char *);
};
```

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.
struct device_node {
        const char *name;/*设备名称*/
        const char *type;/*设备类型*/
        phandle phandle;
        const char *full_name;/*设备全称，包括父设备名*/

        struct  property *properties;/*设备属性链表*/
        struct  property *deadprops;
        struct  device_node *parent;/*指向父节点*/
        struct  device_node *child;/*指向子节点*/
        struct  device_node *sibling;/*指向兄弟节点*/
        struct  device_node *next;      /* next device of same type */
        struct  device_node *allnext;   /* next in list of all nodes */
        struct  proc_dir_entry *pde;    /* this node's proc directory */
        struct  kref kref;
        unsigned long _flags;
        void    *data;
#if defined(CONFIG_SPARC)
        const char *path_component_name;
        unsigned int unique_id;
        struct of_irq_controller *irq_trans;
#endif
};
```

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

```
struct property {
    char    *name;/*属性名*/
    int     length;/*属性值长度*/
    void    *value;/*属性值*/
    struct property *next;/*指向下一个属性*/
    unsigned long _flags;/*标志*/
    unsigned int unique_id;
};
```

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

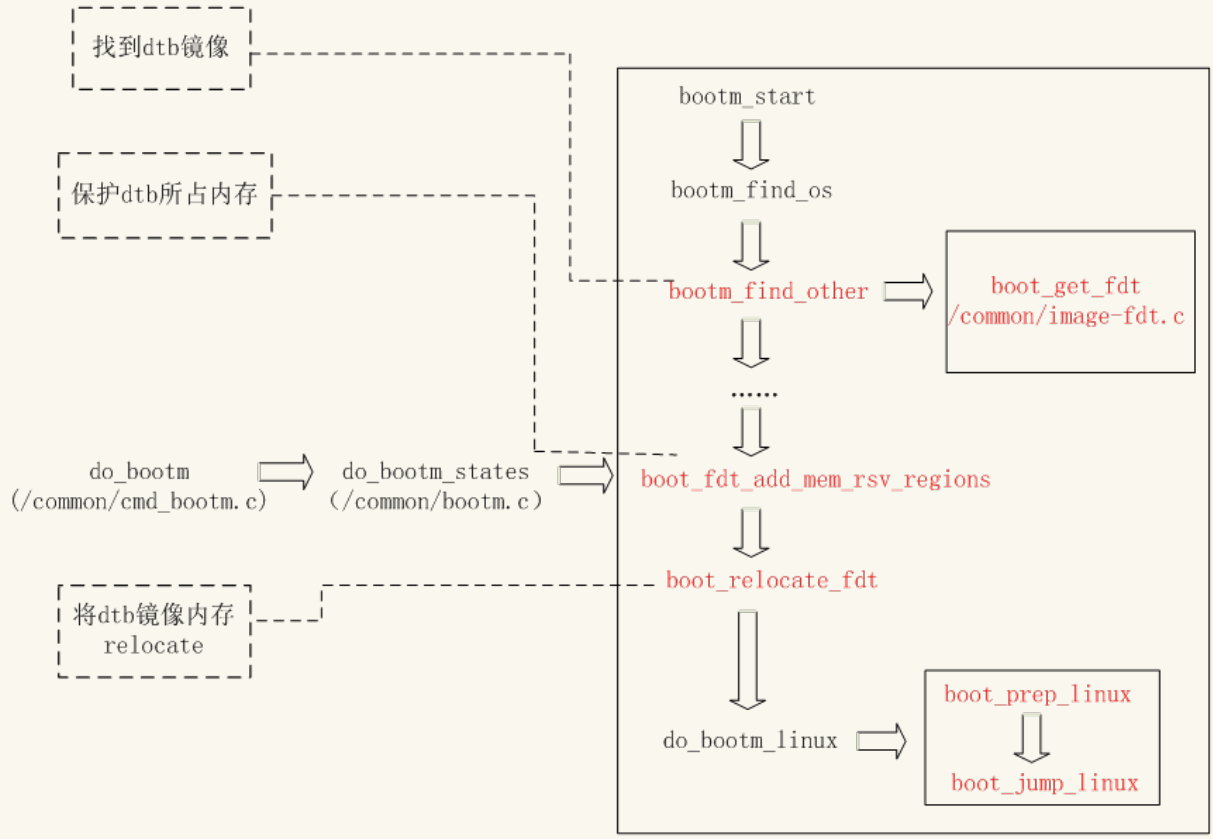
首先我们看下uboot用于记录os、initrd、fdt信息的数据结构bootm_headers，其定义在/include/image.h中，这边截取了其中与dtb相关的一小部分。

```
typedef struct bootm_headers {
......
#if defined(CONFIG_FIT)
        void            *fit_hdr_fdt;     /* FDT blob FIT image header */
        const char      *fit_uname_fdt;   /* FDT blob subimage node unit name */
        int             fit_noffset_fdt;/* FDT blob subimage node offset */
#endif
......
#ifdef CONFIG_LMB
        struct lmb      lmb;              /* for memory mgmt */
#endif
}
```

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



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

```
......
if (IMAGE_ENABLE_OF_LIBFDT && images->ft_len)
        r2 = (unsigned long)images->ft_addr;
else
        r2 = gd->bd->bi_boot_params;
......
kernel_entry(0, machid, r2);
```

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：

```
/*
 * r1 = machine no, r2 = atags or dtb,
 * r8 = phys_offset, r9 = cpuid, r10 =
procinfo
*/      bl      __vet_atags
```

_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。

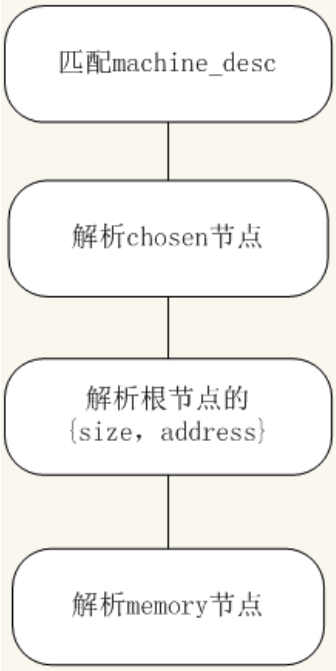
```
__vet_atags:
        tst     r2, #0x3                @ aligned?
        bne     1f

        ldr     r5, [r2, #0]
#ifdef CONFIG_OF_FLATTREE
        ldr     r6, =OF_DT_MAGIC        @ is it a DTB?
        cmp     r5, r6
        beq     2f
#endif
        cmp     r5, #ATAG_CORE_SIZE             @ is first tag ATAG_CORE?
        cmpne   r5, #ATAG_CORE_SIZE_EMPTY
        bne     1f
        ldr     r5, [r2, #4]
        ldr     r6, =ATAG_CORE
        cmp     r5, r6
        bne     1f

2:      mov     pc, lr                  @ atag/dtb pointer is ok

1:      mov     r2, #0
        mov     pc, lr
ENDPROC(__vet_atags)
```

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。



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

```
void __init unflatten_device_tree(void)
{
    //解析设备树，将所有的设备节点链入全局链表of_allnodes中

__unflatten_device_tree(initial_boot_params,&of_allnodes,early_init_dt_alloc_memory_arch);

    //设置内核输出终端，以及遍历“/aliases”节点下的所有的属性，挂入相应链表
    of_alias_scan(early_init_dt_alloc_memory_arch);
}
```

```
static void __unflatten_device_tree(struct boot_param_header*blob,
                                    struct device_node **mynodes,
                                    void *(*dt_alloc)(u64 size, u64 align))
{
    unsigned long size;
    void *start,*mem;
    struct device_node **allnextp= mynodes;

    pr_debug(" -> unflatten_device_tree()\n");

    if (!blob){
        pr_debug("No device tree pointer\n");
        return;
    }

    pr_debug("Unflattening device tree:\n");
    pr_debug("magic: %08x\n", be32_to_cpu(blob->magic));
    pr_debug("size: %08x\n", be32_to_cpu(blob->totalsize));
    pr_debug("version: %08x\n", be32_to_cpu(blob->version));

    //检查设备树magic
    if (be32_to_cpu(blob->magic)!= OF_DT_HEADER){
        pr_err("Invalid device tree blob header\n");
        return;
    }

    //找到设备树的设备节点起始地址
    start = ((void*)blob)+ be32_to_cpu(blob->off_dt_struct);
    //第一次调用mem传0，allnextpp传NULL，实际上是为了计算整个设备树所要的空间
    size = (unsigned long)unflatten_dt_node(blob, 0,&start, NULL, NULL, 0);
    size = ALIGN(size, 4);//4字节对齐

    pr_debug(" size is %lx, allocating...\n", size);

    //调用early_init_dt_alloc_memory_arch函数，为设备树分配内存空间
    mem = dt_alloc(size+ 4, __alignof__(struct device_node));
    memset(mem, 0, size);

    //设备树结束处赋值0xdeadbeef，为了后边检查是否有数据溢出
    *(__be32*)(mem+ size) = cpu_to_be32(0xdeadbeef);
    pr_debug(" unflattening %p...\n", mem);

    //再次获取设备树的设备节点起始地址
    start = ((void*)blob)+ be32_to_cpu(blob->off_dt_struct);
    //mem为设备树分配的内存空间，allnextp指向全局变量of_allnodes，生成整个设备树
    unflatten_dt_node(blob, mem,&start, NULL, &allnextp, 0);
    if (be32_to_cpup(start)!= OF_DT_END)
        pr_warning("Weird tag at end of tree: %08x\n", be32_to_cpup(start));
    if (be32_to_cpup(mem+ size) != 0xdeadbeef)
        pr_warning("End of tree marker overwritten: %08x\n",be32_to_cpup(mem+ size));
    *allnextp = NULL;

    pr_debug(" <- unflatten_device_tree()\n");
}
```

```
static void * unflatten_dt_node(struct boot_param_header*blob,
                                void *mem,void**p,
                                struct device_node *dad,
                                struct device_node ***allnextpp,
                                unsigned long fpsize)
{
    struct device_node *np;
    struct property *pp, **prev_pp= NULL;
    char *pathp;
    u32 tag;
    unsigned int l, allocl;
    int has_name = 0;
    int new_format = 0;

    //*p指向设备树的设备块起始地址
    tag = be32_to_cpup(*p);
    //每个有孩子的设备节点，其tag一定是OF_DT_BEGIN_NODE
    if (tag!= OF_DT_BEGIN_NODE){
        pr_err("Weird tag at start of node: %x\n", tag);
        return mem;
    }

    *p += 4;//地址+4，跳过tag，这样指向节点的名称或者节点路径名
    pathp = *p;//获得节点名或者节点路径名
    l = allocl = strlen(pathp)+ 1;//该节点名称的长度
    *p = PTR_ALIGN(*p+ l, 4);//地址对齐后，*p指向该节点属性的地址

    //如果是节点名则进入，若是节点路径名则(*pathp)== '/'
    if ((*pathp)!= '/'){
        new_format = 1;
        if (fpsize== 0){//fpsize=0
            fpsize = 1;
            allocl = 2;
            l = 1;
            *pathp = '\0';
        } else{
            fpsize += 1;//代分配的长度=本节点名称长度+父亲节点绝对路径的长度
            allocl = fpsize;
        }
    }

    //分配一个设备节点device_node结构，*mem记录分配了多大空间，最终会累加计算出该设备树总共分
配的空间大小
    np = unflatten_dt_alloc(&mem, sizeof(struct device_node)+ allocl,__alignof__(struct
device_node));

    //第一次调用unflatten_dt_node时，allnextpp=NULL
    //第一次调用unflatten_dt_node时，allnextpp指向全局变量of_allnodes的地址
    if (allnextpp){
        char *fn;
        //full_name保存完整的节点名，即包括各级父节点的名称
        np->full_name= fn = ((char *)np)+ sizeof(*np);
        //若new_format=1，表示pathp保存的是节点名，而不是节点路径名，所以需要加上父节点的name
        if (new_format){
            if (dad && dad->parent){
                strcpy(fn, dad->full_name);//把父亲节点绝对路径先拷贝
                fn += strlen(fn);
            }
            *(fn++)= '/';
        }
        memcpy(fn, pathp, l);//拷贝本节点的名称
```

```
    //prev_pp指向节点的属性链表
    prev_pp = &np->properties;

    //当前节点插入全局链表of_allnodes
    **allnextpp= np;
    *allnextpp = &np->allnext;

    //若父亲节点不为空，则设置该节点的parent
    if (dad!= NULL) {
        np->parent= dad;//指向父亲节点
        if (dad->next== NULL)//第一个孩子
            dad->child= np;//child指向第一个孩子
        else
            dad->next->sibling= np;//把np插入next，这样孩子节点形成链表
        dad->next= np;
    }
    kref_init(&np->kref);
}

//分析该节点的属性
while (1){
    u32 sz, noff;
    char *pname;

    //前边已经将*p移到指向节点属性的地址处，取出属性标识
    tag = be32_to_cpup(*p);
    //空属性，则跳过
    if (tag== OF_DT_NOP){
        *p += 4;
        continue;
    }
    //tag不是属性则退出，对于有孩子节点退出时为OF_DT_BEGIN_NODE，对于叶子节点退出时为OF_DT_END_NODE
    if (tag!= OF_DT_PROP)
        break;
    //地址加4，跳过tag
    *p += 4;
    //获得属性值的大小，是以为占多少整形指针计算的
    sz = be32_to_cpup(*p);
    //获取属性名称在节点的字符串块中的偏移
    noff = be32_to_cpup(*p+ 4);
    //地址加8，跳过属性值的大小和属性名称在节点的字符串块中的偏移
    *p += 8;
    //地址对齐后，*P指向属性值所在的地址
    if (be32_to_cpu(blob->version)< 0x10)
        *p = PTR_ALIGN(*p, sz>= 8 ? 8 : 4);

    //从节点的字符串块中noff偏移处，得到该属性的name
    pname = of_fdt_get_string(blob, noff);
    if (pname== NULL) {
        pr_info("Can't find property name in list !\n");
        break;
    }

    //如果有名称为name的属性，表示变量has_name为1
    if (strcmp(pname,"name") == 0)
        has_name = 1;
    //计算该属性name的大小
    l = strlen(pname)+ 1;
```

```
    //为该属性分配一个属性结构，即struct property，
    //*mem记录分配了多大空间，最终会累加计算出该设备树总共分配的空间大小
    pp = unflatten_dt_alloc(&mem, sizeof(structproperty),__alignof__(structproperty));

    //第一次调用unflatten_dt_node时，allnextpp=NULL
    //第一次调用unflatten_dt_node时，allnextpp指向全局变量of_allnodes的地址
    if (allnextpp){
        if ((strcmp(pname,"phandle") == 0)|| (strcmp(pname,"linux,phandle")== 0)){
            if (np->phandle== 0)
                np->phandle= be32_to_cpup((__be32*)*p);
        }
        if (strcmp(pname,"ibm,phandle")== 0)
            np->phandle= be32_to_cpup((__be32*)*p);
        pp->name= pname;//属性名
        pp->length= sz;//属性值长度
        pp->value= *p;//属性值

        //属性插入该节点的属性链表np->properties
        *prev_pp = pp;
        prev_pp = &pp->next;
    }
    *p = PTR_ALIGN((*p)+ sz, 4);//指向下一个属性
}
//至此遍历完该节点的所有属性

//如果该节点没有"name"的属性，则为该节点生成一个name属性，插入该节点的属性链表
if (!has_name){
    char *p1 = pathp, *ps= pathp, *pa = NULL;
    int sz;

    while (*p1){
        if ((*p1)== '@')
            pa = p1;
        if ((*p1)== '/')
            ps = p1 + 1;
            p1++;
    }
    if (pa< ps)
        pa = p1;
    sz = (pa- ps) + 1;
    pp = unflatten_dt_alloc(&mem, sizeof(structproperty) + sz,__alignof__(structproperty));
    if (allnextpp){
        pp->name= "name";
        pp->length= sz;
        pp->value= pp + 1;
        *prev_pp = pp;
        prev_pp = &pp->next;
        memcpy(pp->value, ps, sz- 1);
        ((char*)pp->value)[sz- 1] = 0;
        pr_debug("fixed up name for %s -> %s\n", pathp,(char*)pp->value);
    }
}
```

```
//若设置了allnextpp指针
  if (allnextpp){
      *prev_pp = NULL;
      //设置节点的名称
      np->name= of_get_property(np,"name", NULL);
      //设置该节点对应的设备类型
      np->type= of_get_property(np,"device_type",NULL);

      if (!np->name)
          np->name= "<NULL>";
      if (!np->type)
          np->type= "<NULL>";
  }

  //前边在遍历属性时，tag不是属性则退出
  //对于有孩子节点退出时tag为OF_DT_BEGIN_NODE，对于叶子节点退出时tag为OF_DT_END_NODE
  while (tag== OF_DT_BEGIN_NODE|| tag == OF_DT_NOP){
      //空属性则指向下个属性
      if (tag== OF_DT_NOP)
          *p += 4;
      else
          //OF_DT_BEGIN_NODE则表明其还有子节点，所以递归分析其子节点
          mem = unflatten_dt_node(blob, mem, p, np, allnextpp,fpsize);
      tag = be32_to_cpup(*p);
  }

  //对于叶子节点或者分析完成
  if (tag!= OF_DT_END_NODE){
      pr_err("Weird tag at end of node: %x\n", tag);
      return mem;
  }
  *p += 4;
  //mem返回整个设备树所分配的内存大小，即设备树占的内存空间
  return mem;
                                          }
```

```
void of_alias_scan(void * (*dt_alloc)(u64 size, u64 align))
{
    struct property *pp;

    //根据全局的device_node结构的链表of_allnodes,查找节点名为“/chosen”或者“/chosen@0”的节点，赋值给全局变量
of_chosen
    of_chosen = of_find_node_by_path("/chosen");
    if (of_chosen== NULL)
        of_chosen = of_find_node_by_path("/chosen@0");

    //找到的话，则在该节点查找"linux,stdout-path"    属性
    //"linux,stdout-path"的属性值，常常为标准终端设备的节点路径名，内核会以此作为默认终端
    if (of_chosen) {
        const char *name;
        //返回属性"linux,stdout-path"的属性值
        name = of_get_property(of_chosen, "linux,stdout-path", NULL);
        //根据属性值查找设备节点device_node，即内核默认终端的设备节点,赋值给全局变量of_stdout
        if (name)
            of_stdout = of_find_node_by_path(name);
    }

    //据全局链表of_allnodes,查找节点名为“/aliases”的节点，赋值给全局变量of_aliases
    of_aliases = of_find_node_by_path("/aliases");
    if (!of_aliases)
        return;

    //遍历“/aliases”节点下的所有的属性
    for_each_property_of_node(of_aliases, pp) {
        const char *start = pp->name;//属性名
        const char *end = start + strlen(start);//属性名结尾
        struct device_node *np;
        struct alias_prop *ap;
        int id, len;

        //跳过"name"、"phandle"和"linux,phandle"的属性
        if (!strcmp(pp->name, "name") ||
            !strcmp(pp->name, "phandle") ||
            !strcmp(pp->name, "linux,phandle"))
            continue;

            //根据属性值找到对应的设备节点
            np = of_find_node_by_path(pp->value);
            if (!np)
                continue;

            //去除属性名中结尾的数字，即设备id
            while (isdigit(*(end-1)) && end > start)
                end--;
            //len为属性名去掉结尾数字序号的长度
            len = end - start;

            //此时end指向属性名中结尾的数字，即开始时start指向“&uart0”，end指向字符串结尾。
            //经过上步操作，start仍指向“&uart0”字符串开始处，而end指向字符‘0’。
            //将end字符串转化为10进制数，赋值给id，作为设备的id号
            if (kstrtoint(end, 10, &id) < 0)
                continue;

            //分配alias_prop结构
            ap = dt_alloc(sizeof(*ap) + len + 1, 4);
            if (!ap)
                continue;
            memset(ap, 0, sizeof(*ap) + len + 1);
            ap->alias = start;
            //将该设备的aliases指向对应的device_node,并且链入aliases_lookup链表中
            of_alias_add(ap, np, id, start, len);
        }
}
```

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)