Linux Devices tree

2012-11-11 Lipanfeng

目录

1 Overview

The "Open Firmware Device Tree", or simply Device Tree (DT), 是一种描述硬件信息的结构和语言.

确切的说, 它是操作系统可获取的硬件描述信息, 避免在操作系统里以 hard code 的形式描述 machine 的信息.

形式上, 它是一个名字节点和属性的树. 节点又包含属性和(一些)子节点. 属性通过名字来匹配.

bindings 规则描述了如何在设备树里表示一个设备.

2 data model

2.1 Basic Data Format

```
/ {
    node1 {
        a-string-property = "A string";
        a-string-list-property = "first string", "second string";
        a-byte-data-property = [0x01 0x23 0x34 0x56];
        child-node1 {
            first-child-property;
            second-child-property = <1>;
            a-string-property = "Hello, world";
        };
        child-node2 {
        };
    };
    node2 {
        an-empty-property;
        a-cell-property = <1 2 3 4>; /* each number (cell) is a uint32 */
        child-node1 {
        };
    };
};
```

1, "/"表示一个根节点

2, 两个子节点: node1, node2

3, chiled-node1, child-node2 是 node1 的两个子节点

4, 一些 properties

[Note]:

1, 字符串用双引号表示:

a-string-property = "a string"

2, "Cells" 用 32 位无符号整数表示, 用尖括号扩起来:

a-cell-property = <1 2 3 4>;

3, 数据放到方括号里:

a-byte-data-property = [0x01 0x23 0x34 0x56];

4, 可以用“逗号”把不同的几个数据连在一起:

mixed-property = "a string", [0x01 0x23 0x45 0x67], <0x12345678>;

5, "逗号"可以创建字符串表:

string-list = "red fish", "blue fish";

2.2 基本概念: 从一个例子开始(step by step)

machine 是: 一个 ARM 平台的, 由 Acme 制造的, 名为"Coyote's Revenge"的产品.

1, 32 位 ARM 处理器

2, 处理器总线上挂了串口, spi 总线控制器, i2c controller, interrupt controller, and external bus bridge

3, 256MB 的 SDRAM, 从 0 地址开始

4, 2 个串口, 基地址分别是: 0x101f 1000, 0x101f 2000

5, GPIO controller 的基地址: 0x101f 3000

6, SPI controller 基地址为 0x17 0000

SPI 上挂了一个 MMC, 它的 SS pin 接到 GPIO #1

6, 外部总线桥上挂了如下设备:

- SMC SMC91111 以太网卡, 在外部总线桥的基地址是: 0x1010 0000
- i2c controller 基地址是 0x1016 0000.

i2c 上挂了 Maxim DS1338 RTC. slave address 1101000(0x58)

7, 64MB NOR flash 的基地址是 0x3000 0000

2.3 Initial structure

第一步先给 machine 构造一个结构框架(几乎什么都没有).

```
/ {
    compatible = "acme,coyotes-revenge";
};
```

[Note]:

"compatible"指定了这个系统的名称. 它是包含了"<manufacturer>,<model>"的一个字符串.

这个字段要精确的描述这个产品, 而且要保证名字(namespace)不能冲突.

这个字段很重要, 因为操纵系统要根据这个字段来识别唯一的一个 machine.

如果所有的 mechine 描述都是硬编码，OS 只需要在顶层的"compatible"属性里明确地搜索"acme,coyotes-revenge".

2.4 CPUs

第二步描述每个 CPU core. 构建一个"cpus"节点，并加入每个 CPU core 子节点.

(本例是一个双核的 ARM cortex A9)

每个cpu的"compatible"属性是一个精确表示这个cpu的字符串，像顶层一样按照这样的模型组织"<manufacturer>,<model>"

```
/ {
    compatible = "acme,coyotes-revenge";

    cpus {
        cpu@0 {
            compatible = "arm,cortex-a9";
        };
        cpu@1 {
            compatible = "arm,cortex-a9";
        };
    };
};
```

2.5 Node names

每个节点的名字必须是<name>[@<unit-address>]的形式.

- <name>是普通的 ascii 码，最多 31 个字符. 通常, nodes 的名称是根据它所要表示的设备类型来定的. i.e. 一个 3com Ethernet 应该命名为"ethernet"，而不是"3com509".

- <unit-address> 这个设备的地址(如果有的话). 通常，<unit-address>是访问这个设备的基地址，并 list 到节点的"reg"属性里.(稍后介绍)

- 兄弟(同级的)节点必须有唯一的名字. 但是，兄弟 nodes 使用相同名字的情况也有，只要 address 不同就行, e.g. serial@101f1000 & serial@101f2000.

p.s. [See section 2.2.1 of the ePAPR spec for full details about node naming.]

2.6 Devices

系统中的每个设备都要对应一个设备树中的一个节点. 接下来，填充设备树里的每个设备节点(稍后，在了解了地址范围和中断处理以后再填充).

```
/ {
    compatible = "acme,coyotes-revenge";
```

```
cpus {
	cpu@0 {
		compatible = "arm,cortex-a9";
	};
	cpu@1 {
		compatible = "arm,cortex-a9";
	};
};

serial@101F0000 {
	compatible = "arm,pl011";
};

serial@101F2000 {
	compatible = "arm,pl011";
};

gpio@101F3000 {
	compatible = "arm,pl061";
};

interrupt-controller@10140000 {
	compatible = "arm,pl190";
};

spi@10115000 {
	compatible = "arm,pl022";
};

external-bus {
	ethernet@0,0 {
		compatible = "smc,smc91c111";
	};

	i2c@1,0 {
		compatible = "acme,a1234-i2c-bus";
		rtc@58 {
			compatible = "maxim,ds1338";
		};
	};

	flash@2,0 {
		compatible = "samsung,k8f1315ebm", "cfi-flash";
```

```
            };
        };
    };
```

在这个树里，系统中的每个设备都对应一个 node，并且按层次表现了设备是如何连到系统中的.

i.e. 外部总线上的设备是外部总线的子节点，i2c 总线上的设备是 i2c 总线控制器的子节点.

通常，这个层次结构是站在 CPU 的角度来看的.

[Note]:

-> 每个设备节点都有一个"compatible"属性

-> flash 节点的"compatible"属性里有 2 个字符串. 下个小节将会讨论这个主题.

-> 之前提到，节点的名称反映设备的类型，不是特别的模型. ePAPR spec 的 2.2.2 有一个定义各种用途的 generic node names 的列表.

-- Understanding the "compatible" property --

设备树里每个表示设备的节点都要求有"compatible"属性. "compatible"是操作系统决定为这个设备 bind 哪个设备驱动的关键.

"compatible"是一个字符串列表. 这个列表里的第一个字符串精确地指定"<manufacturer>,<model>"所表示的节点. 其后的字符串表示这个设备所兼容的其他设备.

e.g. Freecale MPC8349 SOC 有一个实现 National Semiconductor ns16550 寄存器接口的串行设备. 那么，MPC8349 串行设备的"compatible"属性应该是：compatible = "fsl,mpc8349-uart", "ns16550". 此时，"fsl,mpc8349-uart"指定了确切的设备，"ns16550"描述:这是一个寄存器级别兼容 National Semiconductor 16550 的 UART.

2.7 How addressing works

可寻址的设备使用下面的属性把地址信息编码到设备树里.

reg

#address-cells

#size-cells

1，每个可寻址的设备都有一个"reg"，一个或多个元素的列表，每个元素表示设备用到的地址范围. 列表的格式如下:

reg = <address1 length1 [address2 length2] [address3 length3] ... >

2，每个地址的值都是一个或者多个叫作"cells"的 32-bit 整形数的 list. 类似地，长度(length)的值也是一个 cells list，后者可以为空(empty).

3, "address"和"length"域有效后，父节点里的"#address-cells"和"#size-cells"属性就可以用来规定每个域里 cells 的数量. 也就是说，正确解析"reg"属性需要父节点的"#address-cells" and "#size-cells"的值.

BTW. 让我们看看这些是如何工作的，先给设备树里的 cpu 添加地址属性.

2.8 CPU address

讨论 addressing 时 cpu 节点的情况是最简单的. 每个 CPU 都被分配了一个唯一的 ID, 而且不用表示 size.

```
cpus {
    #address-cells = <1>;
    #size-cells = <0>;
    cpu@0 {
        compatible = "arm,cortex-a9";
        reg = <0>;
    };
    cpu@1 {
        compatible = "arm,cortex-a9";
        reg = <1>;
    };
};
```

在 cpus 的节点里, "#address-cells"被置为"1", "#size-cells"被设置为"0". 这就表示字节点里"reg"的值只有一个 uint32, 用以表示地址, 但没有大小(size)域. 本例里两个 cpu 的地址分别是"0" and "1". 由于每个 cpu 只分配了一个地址, 所以"#size-cells"是"0".

你可能注意到"reg"的值和节点的名字是匹配的(指名字中地址). 通常, 如果一个节点有"reg"属性, 那么 node name 必须包含地址单元(unit-address), 这正好是"reg"属性的第一个地址值.

2.9 Memory mapped devices

不同与 cpu 节点的单地址值, memory mapped device 被分配了它将响应的地址范围.

"#size-cells"用来描述每个子节点的"reg"元素里!长度域!的大小. 下面的例子里, 每个地址值和长度值都是一个 cell(32 bits).(在 32 位系统上这是常见的, 但是在 64 位系统里就要用大小为的"#size-cells"来表示一个 64 位的地址.)

```
/ {
    #address-cells = <1>;
    #size-cells = <1>;

    ...

    serial@101f0000 {
        compatible = "arm,pl011";
        reg = <0x101f0000 0x1000 >;
    };

    serial@101f2000 {
        compatible = "arm,pl011";
```

```
            reg = <0x101f2000 0x1000 >;
        };

        gpio@101f3000 {
            compatible = "arm,pl061";
            reg = <0x101f3000 0x1000
                    0x101f4000 0x0010>;
        };

        interrupt-controller@10140000 {
            compatible = "arm,pl190";
            reg = <0x10140000 0x1000 >;
        };

        spi@10115000 {
            compatible = "arm,pl022";
            reg = <0x10115000 0x1000 >;
        };

        ...

    };
```

每个设备都被分配一个基地址和对应的地址范围大小. GPIO 有两个地址范围:[0x101f 3000 : 0x101f 3fff], [0x101f 4000 : 0x101f400f].

有些设备可能使用总线上不同的地址模式.比如, 某个设备可以通过片选线被连到外部总线上去. 每个父节点为子节点指定了地址域以后, 地址的映射就能更好的描述这个系统. 下面的代码展示了外部总线上地址的分配情况, 包括地址里片选的编码.

```
        external-bus {
            #address-cells = <2>
            #size-cells = <1>;

            ethernet@0,0 {
                compatible = "smc,smc91c111";
                reg = <0 0 0x1000>;
            };

            i2c@1,0 {
                compatible = "acme,a1234-i2c-bus";
                reg = <1 0 0x1000>;
                rtc@58 {
                    compatible = "maxim,ds1338";
                };
            };
```

```
            flash@2,0 {
                compatible = "samsung,k8f1315ebm", "cfi-flash";
                reg = <2 0 0x4000000>;
            };
        };
```

"external-bus"节点使用了 2 个 cells 来表示地址. 一个是片选, 另一个是片选对应的基地址的偏移.

长度字段保持单个 cell,因为只有偏移量需要一个范围. So, 本例中, 每个"reg"包含 3 个 cells: 片选号, offset, 长度.

由于一个节点和其子节点都包含了地址域, 父节点(external-bus)就不用定义任何地址策略. 直接父节点(external-bus)以外的节点和子节点不需要关心当前的地址域, 因为这个地址可能会被映射到不同的位置.

2.10 Non memory mapped devices

其他设备不会被映射到处理器总线上. 它们有地址范围, 但是不能被 CPU 直接访问. 父设备的驱动会间接的完成 cpu 对这些设备的维护访问.

i2c 设备都被分配地址,但是没有长度和范围. 有点像 CPU 的地址(cpu address 小节).

```
        i2c@1,0 {
            compatible = "acme,a1234-i2c-bus";
            #address-cells = <1>;
            #size-cells = <0>;
            reg = <1 0 0x1000>;
            rtc@58 {
                compatible = "maxim,ds1338";
                reg = <58>;
            };
        };
```

2.11 Ranges (Address translation)

我们讨论了设备地址的分配, 但是仅限于设备节点本地的地址. 还没有描述如何将这些地址映射为 CPU 可以使用的地址.

"root"节点总是以 CPU 的视角描述地址空间. 根节点的子节点已经使用了 CPU 的地址空间, 不需要显式的映射.例如, serial@101f 0000 设备给直接分配了地址 0x101f0000.

那些根节点的非直接子节点并不使用 CPU 地址空间. 为了获得地址映射, 设备树必须指定如何从一个地址空间映射到另一个. "ranges"属性就是用于该目的.

下面的例子是填充了"ranges"的设备树.

```
        / {
            compatible = "acme,coyotes-revenge";
            #address-cells = <1>;
```

```
        #size-cells = <1>;
        ...
        external-bus {
            #address-cells = <2>
            #size-cells = <1>;
            ranges = <0 0    0x10100000      0x10000          // Chipselect 1, Ethernet
                      1 0    0x10160000      0x10000          // Chipselect 2, i2c controller
                      2 0    0x30000000      0x1000000>; // Chipselect 3, NOR Flash

            ethernet@0,0 {
                compatible = "smc,smc91c111";
                reg = <0 0 0x1000>;
            };

            i2c@1,0 {
                compatible = "acme,a1234-i2c-bus";
                #address-cells = <1>;
                #size-cells = <0>;
                reg = <1 0 0x1000>;
                rtc@58 {
                    compatible = "maxim,ds1338";
                    reg = <58>;
                };
            };

            flash@2,0 {
                compatible = "samsung,k8f1315ebm", "cfi-flash";
                reg = <2 0 0x4000000>;
            };
        };
    };
```

"ranges"是地址转化的一个列表. 列表里的每个入口都是一个包含子节点地址的元素, 父地址, 子节点地址空间的大小. 每个域的 size 都是由子节点的"#address-cells", 父节点的"#address-cells", 子节点的"#size-cells"决定的. 本例中的外部总线, 其子地址是 2 cells, 父地址是 1 cell, size 是 1 sell. 三个被转化的范围:

-> Offset 0 from chip select 0 is mapped to address range 0x10100000..0x1010ffff
-> Offset 0 from chip select 1 is mapped to address range 0x10160000..0x1016ffff
-> Offset 0 from chip select 2 is mapped to address range 0x30000000..0x10000000

如果父和子地址空间是相同的, 那么节点的"ranges"属性可以为空. 空的"ranges"属性表示子地址空间是按照 1:1 映射到父地址空间的.

你可能会问为何地址的转换总是被使用, 即使地址可以按 1:1 映射. 一些总线(like PCI)有需要对 OS 可见的完全不同的地址空间. 另外, 还有一些总线的 DMA 引擎需要知道总线上

的真是地址. 有时, 设备需要按组共享相同的可编程的物理地址映射. OS 需要很多的信息和硬件设计来决定是否使用 1:1 映射.

你可能也注意到 i2c@1,0 节点里没有"ranges"属性. 原因是它不像外部总线, i2c 上的设备并不会别映射到 CPU 地址空间. instead, CPU 通过 i2c@1,0 间接访问 rtc@58. 没有"ranges"属性意味着这个设备不能不被任何(除其父设备 i2c)设备直接访问.

2.12 How interrupts work

不像根据自然树组织的地址空间转化, 中断信号可以来自或者结束于系统里的任何设备. 不像自然的表示设备树里设备的地址, 中断信号表示为独立于设备树的节点间的链接.

用四个属性来描述一个中断连接:

-> "interrupt-controller": 用一个空属性来表示接受中断信号的设备节点.

-> "interrupt-cells": 这是中断控制器的一个属性. 它表示中断控制器的"interrupt specifier"里有多少个 cells, 类似于#address-cells, #size-cells.

-> "interrupt-parent": A property of a device node containing a phandle to the interrupt controller that it is attached to. 没有 interrupt-parent 属性的节点也可以继承父节点的属性.

-> "interrupts": 包含中断描述符列表的设备节点属性, 每个都对应设备上一个中断输出信号.

中断描述符是一个或者多个数据 cells(由#interrupt-cells 指定)决定了这个设备接到那个中断输入. 大多数设备只有一个中断输出, 但是也有多个中断输出的设备. 中断描述符的含义完全取决于中断控制器的 binding. 每个中断控制器都可以决定它需要多少个 cells 来定义唯一的一个中断输入.

填充了中断连接的设备树:

```
/ {
    compatible = "acme,coyotes-revenge";
    #address-cells = <1>;
    #size-cells = <1>;
    interrupt-parent = <&intc>;

    cpus {
        #address-cells = <1>;
        #size-cells = <0>;
        cpu@0 {
            compatible = "arm,cortex-a9";
            reg = <0>;
        };
        cpu@1 {
            compatible = "arm,cortex-a9";
            reg = <1>;
        };
    };
```

```
serial@101f0000 {
        compatible = "arm,pl011";
        reg = <0x101f0000 0x1000 >;
        interrupts = < 1 0 >;
};

serial@101f2000 {
        compatible = "arm,pl011";
        reg = <0x101f2000 0x1000 >;
        interrupts = < 2 0 >;
};

gpio@101f3000 {
        compatible = "arm,pl061";
        reg = <0x101f3000 0x1000
                0x101f4000 0x0010>;
        interrupts = < 3 0 >;
};

intc: interrupt-controller@10140000 {
        compatible = "arm,pl190";
        reg = <0x10140000 0x1000 >;
        interrupt-controller;
        #interrupt-cells = <2>;
};

spi@10115000 {
        compatible = "arm,pl022";
        reg = <0x10115000 0x1000 >;
        interrupts = < 4 0 >;
};

external-bus {
        #address-cells = <2>
        #size-cells = <1>;
        ranges = <0 0    0x10100000      0x10000         // Chipselect 1, Ethernet
                  1 0    0x10160000      0x10000         // Chipselect 2, i2c controller
                  2 0    0x30000000      0x1000000>; // Chipselect 3, NOR Flash

        ethernet@0,0 {
                compatible = "smc,smc91c111";
                reg = <0 0 0x1000>;
                interrupts = < 5 2 >;
```

```
            };

            i2c@1,0 {
                compatible = "acme,a1234-i2c-bus";
                #address-cells = <1>;
                #size-cells = <0>;
                reg = <1 0 0x1000>;
                interrupts = < 6 2 >;
                rtc@58 {
                    compatible = "maxim,ds1338";
                    reg = <58>;
                    interrupts = < 7 3 >;
                };
            };

            flash@2,0 {
                compatible = "samsung,k8f1315ebm", "cfi-flash";
                reg = <2 0 0x4000000>;
            };
        };
    };
```

[Notice]:

1, 这个系统有一个中断控制器, interrupt-controller@10140000

2, "intc:"标签已经被加到 interrupt controller 节点, 这个节点用来给根节点的 interrupt-parent 分配一个 phandle. 这个 interrupt-parent 值将作为系统的默认值, 因为所有的子节点都继承了它(除非显式的 overridden).

3, 各个设备使用 interrupt 属性指定不同的中断输入信号

4, #interrupt-cells 是 2, 因而每个中断描述符有 2 个 cells.本例使用通用模式通过第一个 cell 编码中断号, 用第二个 cell 编码中断的 flags, e.g. 电平触发, 高电平触发, 低电平触发, 边沿触发等. 对任何一个指定的中断控制器, 可以参考 controller 的 binging 文档来学习如何编码中断描述符.

2.13 Device specific data

通用属性以外的杂项属性和子节点也可以被添加到某个节点. 任何 OS 需要的数据都可以添加进来, 只要有合适的规则.

1, 新的设备特有的属性名称应该以厂商为前缀, 以避免与通用属性的冲突.

2, 属性的含义和子节点必须形成 binding 文档, 这样驱动的作者就知道如何解析数据. binding 文档是一个特殊的兼容途径, 属性+子节点+设备的表示都要有这样文档. 每个唯一的"compatible"值应该有自己的 binding.

3, 在这里查看新设备 binding, 及 binding 文档的书写格式.

http://devicetree.org/Main_Page

2.14 Special nodes

2.14.1 aliases Node

通常用 full path 来引用一个确切的节点, like /external-bus/ethernet@0,0; 但是对一个用户真正想知道的事情来说, 这样太麻烦了. "aliases"节点用来给 full path 设备分配一个短的别名. e.g.

```
aliases {
        ethernet0 = &eth0;
        serial0 = &serial0;
};
```

通过 aliases 来分配一个设备的识别符使 OS 更加友好.

你可能发现这里用了一个新的语法. "preperty = &label;" 这个语法通过一个标签把 full path 节点引用为一个字符串属性. 这有别于在 cell 里插入一个 phandle 的格式"phandle = < &label >".

2.14.2 chosen Node

"chosen"节点并不表示一个真正的设备, 但是提供从 firmware 向 OS 传递参数的途径, 就像启动参数. "chosen"节点里的数据不表示硬件. 典型地, *.dts 文件里的"chosen"节点被设置为空.

在本例的系统中, firmware 可以添加这样的"chosen"节点:

```
chosen {
        bootargs = "root=/dev/nfs rw nfsroot=192.168.1.1 console=ttyS0,115200";
};
```

2.15 include

像 C 语言的#include 一样包含别的 dts 文件, 格式如下:

/include/ "msm8974-iommu.dtsi"

2.16 Advanced Topics

1, Advanced sample machine

我们已经讨论了基础的定义, 现在来添加一些硬件到例子 machine 用以讨论更加复杂的

case.

高级例子 machine 添加了一个 PCI host bridge, 它的控制寄存器被映射到 0x10180000, BARs 被编程到从 0x8000 0000 到更高的地址空间.

p.s. 后面 PCI 的部分稍后添加

3 High level view

理解 DT 的重点在于: 它是一种简单的, 描述硬件的数据结构. DT 所做的就是提供一种语言, 用以减少 board 硬件配置和内核中设备驱动的耦合性.

4 How to package DT with kernel

1, kenrel config USE_OF ?

"Flattened Device Tree support"

2, kernel/AndroidKernel.mk. 用 dtc 脚本处理*tds 文件后直接 cat 到 zImage 的结尾.

| kenrel image | DT |

- DTS_NAME ?= $(MSM_ARCH)

- 展开所有*.dts 文件

DTS_FILES = $(wildcard $(TOP)/kernel/arch/arm/boot/dts/$(DTS_NAME)*.dts)

- $(subst FROM,TO,TEXT),即将 TEXT 中的东西从 FROM 变为 TO, lastword 取字符串中的最后一个单词

DTS_FILE = $(lastword $(subst /, ,$(1)))

- 将.dts 替换为.dtb, 再加上前缀

DTB_FILE = $(addprefix $(KERNEL_OUT)/arch/arm/boot/,$(patsubst %.dts,%.dtb,$(call DTS_FILE,$(1))))

-

ZIMG_FILE = $(addprefix $(KERNEL_OUT)/arch/arm/boot/,$(patsubst %.dts,%-zImage,$(call DTS_FILE,$(1))))

KERNEL_ZIMG = $(KERNEL_OUT)/arch/arm/boot/zImage

DTC = $(KERNEL_OUT)/scripts/dtc/dtc

- foreach(var,list,func)循环将 list 的内容赋值给 var, 并执行后面 func 的动作.

```
mkdir -p $(KERNEL_OUT)/arch/arm/boot;\
$(foreach DTS_NAME, $(DTS_NAMES), \
    $(foreach d, $(DTS_FILES), \
        $(DTC) -p 1024 -O dtb -o $(call DTB_FILE,$(d)) $(d); \
        cat $(KERNEL_ZIMG) $(call DTB_FILE,$(d)) > $(call ZIMG_FILE,$(d));))
```

- One by one 用 dtc 处理*.dts 文件, 并 cat 到 zimage 尾部.

3, dtc usage (kernel/scripts/dtc/)

Usage:

dtc [options] <input file>

Options:

-h

This help text

-q

Quiet: -q suppress warnings, -qq errors, -qqq all

-I <input format>

Input formats are:

dts - device tree source text

dtb - device tree blob

fs - /proc/device-tree style directory

-o <output file>

-O <output format>

Output formats are:

dts - device tree source text

dtb - device tree blob

asm - assembler source

-V <output version>

Blob version to produce, defaults to ? (relevant for dtb and asm output only)

-d <output dependency file>

-R <number>

Make space for <number> reserve map entries (relevant for dtb and asm output only)

-S <bytes>

Make the blob at least <bytes> long (extra space)

-p <bytes>

Add padding to the blob of <bytes> long (extra space)

-b <number>

Set the physical boot cpu

-f

Force - try to produce output even if the input tree has errors

-s

Sort nodes and properties before outputting (only useful for comparing trees)

-v

Print DTC version and exit

-H <phandle format>

phandle formats are:

legacy - "linux,phandle" properties only

epapr - "phandle" properties only

both - Both "linux,phandle" and "phandle" properties

5 How to pass

entrys in image-> | kernel header | kernel | ramdisk | DT |
DT-> | DT header | dt_entry... | actual entry... |
1, lk/app/aboot/aboot.c-> boot_linux_from_mmc()
- dt_table_offset = ((uint32_t)image_addr + page_size + kernel_actual + ramdisk_actual + second_actual)
  DT 存在(boot.img + sizeof(boot_img_hdr) + sizeof(kernel) + sizeof(ramdisk) + 0)这个位置
- 检验 DT, #define DEV_TREE_MAGIC 0x54444351 /* "QCDT" */
- 跳过 DEV_TREE_HEADER_SIZE(12), 获取平台的 entry.
  dev_tree_get_entry_ptr()
- 把 DT 存入 hdr->tags_addr
  memmove((void *)hdr->tags_addr, (char *)dt_table_offset + dt_entry_ptr->offset, dt_entry_ptr->size);
- boot_linux((void *)hdr->kernel_addr, (unsigned *) hdr->tags_addr,
  (const char *)cmdline, board_machtype(),
  (void *)hdr->ramdisk_addr, hdr->ramdisk_size);
  void (*entry)(unsigned, unsigned, unsigned*) = kernel;
  entry(0, machtype, tags);
  (ATPCS) r0 r1 r2

2, arch/arm/kernel/head-common.S
- 把 r2 的里的地址赋值给标号__atags_pointer, 后续通过这个标号访问.
  str r2, [r6] @ Save atags pointer
  .long __atags_pointer @ r6 .globl
- start_kernel()-> setup_arch()

6 Platform Identification

内核通过 DT 识别特定的 machine.

大多数情况下, machine 的识别是不能通用的. 内核根据 machine 的 CPU 或者 SoC 来选择配置代码.

ARM 体系结构里:

arch/arm/kernel/setup.c-> setup_arch() 调用 arch/arm/kernel/devicetree.c-> setup_machine_fdt()

在 machine_desc{} table 里搜索与 DT 数据最匹配的 machine.

设备树根节点的 compatible 属性跟 machine_desc{}.dt_compat 表(数组)相比较决定最匹配的 machine.

找到最匹配的 machine 后 setup_machine_fdt()返回对应的 machine_desc{}的基地址, 否

则返回 NULL.

1, arch/arm/mach-msm/board-8974.c

```
static const struct machine_desc __mach_desc_MSM8974_DT_name
        __used __attribute__((__section__(".arch.info.init"))) = {
        .nr          = ~0,
        .name       = "Qualcomm MSM 8974 (Flattened Device Tree)",
        .map_io = msm8974_map_io,
        .init_irq = msm_dt_init_irq,
        .init_machine = msm8974_init,
        .handle_irq = gic_handle_irq,
        .timer = &msm_dt_timer,
        .dt_compat = msm8974_dt_match,
        .reserve = msm_reserve,
        .init_very_early = msm8974_init_very_early,
        .restart = msm_restart,
};
```

2,

```
static const char *msm8974_dt_match[] __initconst = {
        "qcom,msm8974",
        NULL
};
```

7 Runtime configuration

```
chosen {
        bootargs = "console=ttyS0,115200 loglevel=8";
        initrd-start = <0xc8000000>;
        initrd-end = <0xc8200000>;
};
```

setup_machine_fdt()会多次调用 of_scan_flat_dt()(with different callbacks).

- early_init_dt_scan_chosen()解析 chosen 属性;
- early_init_dt_scan_root()初始化 DT 地址空间;
- early_init_dt_scan_memory()初始化可用的 RAM.

8 Device population

```
/{
        compatible = "nvidia,harmony", "nvidia,tegra20";
        #address-cells = <1>;
        #size-cells = <1>;

        memory {
                device_type = "memory";
                reg = <0x00000000 0x40000000>;
        };

        soc {
                compatible = "nvidia,tegra20-soc", "simple-bus";
                #address-cells = <1>;
                #size-cells = <1>;
                ranges;

                serial@70006300 {
                        compatible = "nvidia,tegra20-uart";
                        reg = <0x70006300 0x100>;
                        interrupts = <122>;
                };

                i2c@7000c000 {
                        compatible = "nvidia,tegra20-i2c";
                        #address-cells = <1>;
                        #size-cells = <0>;
                        reg = <0x7000c000 0x100>;
                        interrupts = <70>;

                        wm8903: codec@1a {
                                compatible = "wlf,wm8903";
                                reg = <0x1a>;
                                interrupts = <347>;
                        };
                };
        };

        sound {
                compatible = "nvidia,harmony-sound";
                i2s-controller = <&i2s1>;
                i2s-codec = <&wm8903>;
```

};
};

1, 初期初始化完成后, unflatten_device_tree()用来把设备树数据转化为更有效率的表示方式.

2, 这时通常也会调用 machine_desc{} 里的.init_very_early(), .init_early(), .init_irq()， .init_machine()等 hooks.

3, 在.init_machine()的时候, kernel 要创建设备树里的设备, 但是设备树里并没有指定设备的类型. 如何创建?

- 从 root 节点开始搜索所用包含 compatible 属性的节点, 并假设这些节点都表示的是设备
- 内核假设他们或者是直接连到处理器总线的设备,或者是无法用其他方式表示的杂项设备
- 将这些设备全部注册为 platform_device, 后面绑定 platform_driver

4, board code(init_machine())调用 of_platform_populate()寻找根设备节点, 注册 platform devices 并 bind driver.

p.s init_machine()可以什么都不做, 只调用 of_platform_populate()

[Note]:

1, 为什么选择 platform model ?

在 linux 设备模型里, 所有的 bus_type 假设它的所有设备是连接在 bus controller 上的子设备.

e.g. i2c_client 是 i2c_master 的子设备; spi_device 是 SPI bus 的子设备; USB, PCI 等都是类似的.

DT 也采用这样的层次结构. 所有 i2c 设备节点都只出现在 i2c 总线节点里. 同理, SPI, USB, PCI...

2, 这么多种总线类型如何用同一种模型来表示? (子设备的概念都是相同的, 所以现在只考虑总线)

唯一没有特定父设备类型的模型就是 platform_device.

3, 只有根设备节点被注册, 跟设备节点的子设备怎么办?(前面那个 DT 的例子只能找出 SoC 和 sound 节点.)

Linux 对 DT 的支持, 在这种情况下, 通常会在父设备的 probe()里完成对其各个子设备的注册(platform_devices).

4, 内核还支持一种形式:

of_platform_populate()的第二个参数是一个 of_device_id{} table. 任何设备树里被这个 table 匹配到的节点的子节点将被注册(不用 bus 的 probe).

```
void __init msm8974_init(void)
{
    /*...*/
    of_platform_populate(NULL, of_default_bus_match_table, adata, NULL);

    msm8974_add_devices();
    msm8974_add_drivers();
}
```

```
const struct of_device_id of_default_bus_match_table[] = {
    { .compatible = "simple-bus", },
#ifdef CONFIG_ARM_AMBA
    { .compatible = "arm,amba-bus", },
#endif /* CONFIG_ARM_AMBA */
    {} /* Empty terminated list */
};
```

"simple-bus"定义在 ePAPR 1.0 协议里，它表示一个简单内存映射的属性. 所以，设计 of_platform_populate()时可以假设 simple-bus 兼容的设备节点总是可以通过.

9 References

[1] usage-> http://devicetree.org/Device_Tree_Usage
[2] spec-> http://www.power.org/documentation/epapr-version-1-1/