Linux Devices tree

2012-11-11 Lipanfeng

目录

[1 Overview 3](#_Toc340328109)

[2 data model 3](#_Toc340328110)

[2.1 Basic Data Format 3](#_Toc340328111)

[2.2 基本概念: 从一个例子开始(step by step) 4](#_Toc340328112)

[2.3 Initial structure 4](#_Toc340328113)

[2.4 CPUs 5](#_Toc340328114)

[2.5 Node names 5](#_Toc340328115)

[2.6 Devices 5](#_Toc340328116)

[2.7 How addressing works 7](#_Toc340328117)

[2.8 CPU address 8](#_Toc340328118)

[2.9 Memory mapped devices 8](#_Toc340328119)

[2.10 Non memory mapped devices 10](#_Toc340328120)

[2.11 Ranges (Address translation) 10](#_Toc340328121)

[2.12 How interrupts work 12](#_Toc340328122)

[2.13 Device specific data 14](#_Toc340328123)

[2.14 Special nodes 15](#_Toc340328124)

[2.14.1 aliases Node 15](#_Toc340328125)

[2.14.2 chosen Node 15](#_Toc340328126)

[2.15 include 15](#_Toc340328127)

[2.16 Advanced Topics 15](#_Toc340328128)

[3 High level view 16](#_Toc340328129)

[4 How to package DT with kernel 16](#_Toc340328130)

[5 How to pass 16](#_Toc340328131)

[6 Platform Identification 17](#_Toc340328132)

[7 Runtime configuration 18](#_Toc340328133)

[8 Device population 18](#_Toc340328134)

[9 References 21](#_Toc340328135)

# 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/