**Device Tree详解**

**1.    ARM Device Tree起源**

Linus Torvalds在2011年3月17日的ARM Linux邮件列表宣称“this whole ARM thing is a fucking pain in the ass”，引发ARM Linux社区的地震，随后ARM社区进行了一系列的重大修正。在过去的ARM Linux中，arch/arm/plat-xxx和arch/arm/mach-xxx中充斥着大量的垃圾代码，相当多数的代码只是在描述板级细节，而这些板级细节对于内核来讲，不过是垃圾，如板上的platform设备、resource、i2c\_board\_info、spi\_board\_info以及各种硬件的platform\_data。读者有兴趣可以统计下常见的s3c2410、s3c6410等板级目录，代码量在数万行。社区必须改变这种局面，于是PowerPC等其他体系架构下已经使用的Flattened Device Tree（FDT）进入ARM社区的视野。Device Tree是一种描述硬件的数据结构，它起源于 OpenFirmware (OF)。在Linux 2.6中，ARM架构的板极硬件细节过多地被硬编码在arch/arm/plat-xxx和arch/arm/mach-xxx，采用Device Tree后，许多硬件的细节可以直接透过它传递给Linux，而不再需要在kernel中进行大量的冗余编码。Device Tree由一系列被命名的结点（node）和属性（property）组成，而结点本身可包含子结点。所谓属性，其实就是成对出现的name和value。在Device Tree中，可描述的信息包括（原先这些信息大多被hard code到kernel中）：

* 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

它基本上就是画一棵电路板上CPU、总线、设备组成的树，Bootloader会将这棵树传递给内核，然后内核可以识别这棵树，并根据它展开出Linux内核中的platform\_device、i2c\_client、spi\_device等设备，而这些设备用到的内存、IRQ等资源，也被传递给了内核，内核会将这些资源绑定给展开的相应的设备。

Device Tree是一种用来描述硬件的数据结构，类似板级描述语言，起源于OpenFirmware(OF)。在目前广泛使用的Linux kernel 2.6.x版本中，对于不同平台、不同硬件，往往存在着大量的不同的、移植性差的板级描述代码，以达到对这些不同平台和不同硬件特殊适配的需求。Device Tree的引入给驱动适配带来了很大的方便，一套完整的Device Tree可以将一个PCB摆在你眼前。Device Tree可以描述CPU，可以描述时钟、中断控制器、IO控制器、SPI总线控制器、I2C控制器、存储设备等任何现有驱动单位。对具体器件能够描述到使用哪个中断，内存映射空间是多少等等。

**2.    Device Tree组成和结构**

整个Device Tree牵涉面比较广，即增加了新的用于描述设备硬件信息的文本格式，又增加了编译这一文本的工具，同时Bootloader也需要支持将编译后的Device Tree传递给Linux内核。

**DTS (device tree source)**

.dts文件是一种ASCII 文本格式的Device Tree描述，此文本格式非常人性化，适合人类的阅读习惯。基本上，在ARM Linux在，一个.dts文件对应一个ARM的machine，一般放置在内核的arch/arm/boot/dts/目录。由于一个SoC可能对应多个machine（一个SoC可以对应多个产品和电路板），势必这些.dts文件需包含许多共同的部分，Linux内核为了简化，把SoC公用的部分或者多个machine共同的部分一般提炼为.dtsi，类似于C语言的头文件。其他的machine对应的.dts就include这个.dtsi。譬如，对于VEXPRESS而言，vexpress-v2m.dtsi就被vexpress-v2p-ca9.dts所引用， vexpress-v2p-ca9.dts有如下一行：  
/include/ "vexpress-v2m.dtsi"  
当然，和C语言的头文件类似，.dtsi也可以include其他的.dtsi，譬如几乎所有的ARM SoC的.dtsi都引用了skeleton.dtsi。  
.dts（或者其include的.dtsi）基本元素即为前文所述的结点和属性：

**[plain]** [view plain](http://blog.csdn.net/21cnbao/article/details/8457546" \o "view plain)[copy](http://blog.csdn.net/21cnbao/article/details/8457546" \o "copy)[print](http://blog.csdn.net/21cnbao/article/details/8457546" \o "print)[?](http://blog.csdn.net/21cnbao/article/details/8457546" \o "?)

1. / {
2. node1 {
3. a-string-property = "A string";
4. a-string-list-property = "first string", "second string";
5. a-byte-data-property = [0x01 0x23 0x34 0x56];
6. child-node1 {
7. first-child-property;
8. second-child-property = <1>;
9. a-string-property = "Hello, world";
10. };
11. child-node2 {
12. };
13. };
14. node2 {
15. an-empty-property;
16. a-cell-property = <1 2 3 4>; /\* each number (cell) is a uint32 \*/
17. child-node1 {
18. };
19. };
20. };

/ {

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 {

};

};

};

上述.dts文件并没有什么真实的用途，但它基本表征了一个Device Tree源文件的结构：

1个root结点"/"；

root结点下面含一系列子结点，本例中为"node1" 和 "node2"；  
结点"node1"下又含有一系列子结点，本例中为"child-node1" 和 "child-node2"；  
各结点都有一系列属性。这些属性可能为空，如" an-empty-property"；可能为字符串，如"a-string-property"；可能为字符串数组，如"a-string-list-property"；可能为Cells（由u32整数组成），如"second-child-property"，可能为二进制数，如"a-byte-data-property"。  
下面以一个最简单的machine为例来看如何写一个.dts文件。假设此machine的配置如下：  
1个双核ARM Cortex-A9 32位处理器；  
ARM的local bus上的内存映射区域分布了2个串口（分别位于0x101F1000 和 0x101F2000）、GPIO控制器（位于0x101F3000）、SPI控制器（位于0x10170000）、中断控制器（位于0x10140000）和一个external bus桥；  
External bus桥上又连接了SMC SMC91111 Ethernet（位于0x10100000）、I2C控制器（位于0x10160000）、64MB NOR Flash（位于0x30000000）；  
External bus桥上连接的I2C控制器所对应的I2C总线上又连接了Maxim DS1338实时钟（I2C地址为0x58）。  
其对应的.dts文件为：

**[plain]** [view plain](http://blog.csdn.net/21cnbao/article/details/8457546" \o "view plain)[copy](http://blog.csdn.net/21cnbao/article/details/8457546" \o "copy)[print](http://blog.csdn.net/21cnbao/article/details/8457546" \o "print)[?](http://blog.csdn.net/21cnbao/article/details/8457546" \o "?)

1. / {
2. compatible = "acme,coyotes-revenge";
3. #address-cells = <1>;
4. #size-cells = <1>;
5. interrupt-parent = <&intc>;
7. cpus {
8. #address-cells = <1>;
9. #size-cells = <0>;
10. cpu@0 {
11. compatible = "arm,cortex-a9";
12. reg = <0>;
13. };
14. cpu@1 {
15. compatible = "arm,cortex-a9";
16. reg = <1>;
17. };
18. };
20. serial@101f0000 {
21. compatible = "arm,pl011";
22. reg = <0x101f0000 0x1000 >;
23. interrupts = < 1 0 >;
24. };
26. serial@101f2000 {
27. compatible = "arm,pl011";
28. reg = <0x101f2000 0x1000 >;
29. interrupts = < 2 0 >;
30. };
32. gpio@101f3000 {
33. compatible = "arm,pl061";
34. reg = <0x101f3000 0x1000
35. 0x101f4000 0x0010>;
36. interrupts = < 3 0 >;
37. };
39. intc: interrupt-controller@10140000 {
40. compatible = "arm,pl190";
41. reg = <0x10140000 0x1000 >;
42. interrupt-controller;
43. #interrupt-cells = <2>;
44. };
46. spi@10115000 {
47. compatible = "arm,pl022";
48. reg = <0x10115000 0x1000 >;
49. interrupts = < 4 0 >;
50. };
52. external-bus {
53. #address-cells = <2>
54. #size-cells = <1>;
55. ranges = <0 0  0x10100000   0x10000     // Chipselect 1, Ethernet
56. 1 0  0x10160000   0x10000     // Chipselect 2, i2c controller
57. 2 0  0x30000000   0x1000000>; // Chipselect 3, NOR Flash
59. ethernet@0,0 {
60. compatible = "smc,smc91c111";
61. reg = <0 0 0x1000>;
62. interrupts = < 5 2 >;
63. };
65. i2c@1,0 {
66. compatible = "acme,a1234-i2c-bus";
67. #address-cells = <1>;
68. #size-cells = <0>;
69. reg = <1 0 0x1000>;
70. interrupts = < 6 2 >;
71. rtc@58 {
72. compatible = "maxim,ds1338";
73. reg = <58>;
74. interrupts = < 7 3 >;
75. };
76. };
78. flash@2,0 {
79. compatible = "samsung,k8f1315ebm", "cfi-flash";
80. reg = <2 0 0x4000000>;
81. };
82. };
83. };

/ {

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>;

};

};

};

上述.dts文件中,root结点"/"的compatible 属性compatible = "acme,coyotes-revenge";定义了系统的名称，它的组织形式为：<manufacturer>,<model>。Linux内核透过root结点"/"的compatible 属性即可判断它启动的是什么machine。  
在.dts文件的每个设备，都有一个compatible 属性，compatible属性用户驱动和设备的绑定。compatible 属性是一个字符串的列表，列表中的第一个字符串表征了结点代表的确切设备，形式为"<manufacturer>,<model>"，其后的字符串表征可兼容的其他设备。可以说前面的是特指，后面的则涵盖更广的范围。如在arch/arm/boot/dts/vexpress-v2m.dtsi中的Flash结点：

**[plain]** [view plain](http://blog.csdn.net/21cnbao/article/details/8457546" \o "view plain)[copy](http://blog.csdn.net/21cnbao/article/details/8457546" \o "copy)[print](http://blog.csdn.net/21cnbao/article/details/8457546" \o "print)[?](http://blog.csdn.net/21cnbao/article/details/8457546" \o "?)

1. flash@0,00000000 {
2. compatible = "arm,vexpress-flash", "cfi-flash";
3. reg = <0 0x00000000 0x04000000>,
4. <1 0x00000000 0x04000000>;
5. bank-width = <4>;
6. };

flash@0,00000000 {

compatible = "arm,vexpress-flash", "cfi-flash";

reg = <0 0x00000000 0x04000000>,

<1 0x00000000 0x04000000>;

bank-width = <4>;

};

compatible属性的第2个字符串"cfi-flash"明显比第1个字符串"arm,vexpress-flash"涵盖的范围更广。  
再比如，Freescale MPC8349 SoC含一个串口设备，它实现了国家半导体（National Semiconductor）的ns16550 寄存器接口。则MPC8349串口设备的compatible属性为compatible = "fsl,mpc8349-uart", "ns16550"。其中，fsl,mpc8349-uart指代了确切的设备， ns16550代表该设备与National Semiconductor 的16550 UART保持了寄存器兼容。  
接下来root结点"/"的cpus子结点下面又包含2个cpu子结点，描述了此machine上的2个CPU，并且二者的compatible 属性为"arm,cortex-a9"。  
注意cpus和cpus的2个cpu子结点的命名，它们遵循的组织形式为：<name>[@<unit-address>]，<>中的内容是必选项，[]中的则为可选项。name是一个ASCII字符串，用于描述结点对应的设备类型，如3com Ethernet适配器对应的结点name宜为ethernet，而不是3com509。如果一个结点描述的设备有地址，则应该给出@unit-address。多个相同类型设备结点的name可以一样，只要unit-address不同即可，如本例中含有cpu@0、cpu@1以及serial@101f0000与serial@101f2000这样的同名结点。设备的unit-address地址也经常在其对应结点的reg属性中给出。ePAPR标准给出了结点命名的规范。  
可寻址的设备使用如下信息来在Device Tree中编码地址信息：

* reg
* #address-cells
* #size-cells

其中reg的组织形式为reg = <address1 length1 [address2 length2] [address3 length3] ... >，其中的每一组address length表明了设备使用的一个地址范围。address为1个或多个32位的整型（即cell），而length则为cell的列表或者为空（若#size-cells = 0）。address 和 length 字段是可变长的，父结点的#address-cells和#size-cells分别决定了子结点的reg属性的address和length字段的长度。在本例中，root结点的#address-cells = <1>;和#size-cells = <1>;决定了serial、gpio、spi等结点的address和length字段的长度分别为1。cpus 结点的#address-cells = <1>;和#size-cells = <0>;决定了2个cpu子结点的address为1，而length为空，于是形成了2个cpu的reg = <0>;和reg = <1>;。external-bus结点的#address-cells = <2>和#size-cells = <1>;决定了其下的ethernet、i2c、flash的reg字段形如reg = <0 0 0x1000>;、reg = <1 0 0x1000>;和reg = <2 0 0x4000000>;。其中，address字段长度为0，开始的第一个cell（0、1、2）是对应的片选，第2个cell（0，0，0）是相对该片选的基地址，第3个cell（0x1000、0x1000、0x4000000）为length。特别要留意的是i2c结点中定义的 #address-cells = <1>;和#size-cells = <0>;又作用到了I2C总线上连接的RTC，它的address字段为0x58，是设备的I2C地址。  
root结点的子结点描述的是CPU的视图，因此root子结点的address区域就直接位于CPU的memory区域。但是，经过总线桥后的address往往需要经过转换才能对应的CPU的memory映射。external-bus的ranges属性定义了经过external-bus桥后的地址范围如何映射到CPU的memory区域。

**[plain]** [view plain](http://blog.csdn.net/21cnbao/article/details/8457546" \o "view plain)[copy](http://blog.csdn.net/21cnbao/article/details/8457546" \o "copy)[print](http://blog.csdn.net/21cnbao/article/details/8457546" \o "print)[?](http://blog.csdn.net/21cnbao/article/details/8457546" \o "?)

1. ranges = <0 0  0x10100000   0x10000     // Chipselect 1, Ethernet
2. 1 0  0x10160000   0x10000     // Chipselect 2, i2c controller
3. 2 0  0x30000000   0x1000000>; // Chipselect 3, NOR Flash

ranges = <0 0 0x10100000 0x10000 // Chipselect 1, Ethernet

1 0 0x10160000 0x10000 // Chipselect 2, i2c controller

2 0 0x30000000 0x1000000>; // Chipselect 3, NOR Flash

ranges是地址转换表，其中的每个项目是一个子地址、父地址以及在子地址空间的大小的映射。映射表中的子地址、父地址分别采用子地址空间的#address-cells和父地址空间的#address-cells大小。对于本例而言，子地址空间的#address-cells为2，父地址空间的#address-cells值为1，因此0 0  0x10100000   0x10000的前2个cell为external-bus后片选0上偏移0，第3个cell表示external-bus后片选0上偏移0的地址空间被映射到CPU的0x10100000位置，第4个cell表示映射的大小为0x10000。ranges的后面2个项目的含义可以类推。  
Device Tree中还可以中断连接信息，对于中断控制器而言，它提供如下属性：  
interrupt-controller – 这个属性为空，中断控制器应该加上此属性表明自己的身份；  
#interrupt-cells – 与#address-cells 和 #size-cells相似，它表明连接此中断控制器的设备的interrupts属性的cell大小。  
在整个Device Tree中，与中断相关的属性还包括：  
interrupt-parent – 设备结点透过它来指定它所依附的中断控制器的phandle，当结点没有指定interrupt-parent 时，则从父级结点继承。对于本例而言，root结点指定了interrupt-parent = <&intc>;其对应于intc: interrupt-controller@10140000，而root结点的子结点并未指定interrupt-parent，因此它们都继承了intc，即位于0x10140000的中断控制器。  
interrupts – 用到了中断的设备结点透过它指定中断号、触发方法等，具体这个属性含有多少个cell，由它依附的中断控制器结点的#interrupt-cells属性决定。而具体每个cell又是什么含义，一般由驱动的实现决定，而且也会在Device Tree的binding文档中说明。譬如，对于ARM GIC中断控制器而言，#interrupt-cells为3，它3个cell的具体含义Documentation/devicetree/bindings/arm/gic.txt就有如下文字说明：

**[plain]** [view plain](http://blog.csdn.net/21cnbao/article/details/8457546" \o "view plain)[copy](http://blog.csdn.net/21cnbao/article/details/8457546" \o "copy)[print](http://blog.csdn.net/21cnbao/article/details/8457546" \o "print)[?](http://blog.csdn.net/21cnbao/article/details/8457546" \o "?)

1. 01   The 1st cell is the interrupt type; 0 for SPI interrupts, 1 for PPI
2. 02   interrupts.
3. 03
4. 04   The 2nd cell contains the interrupt number for the interrupt type.
5. 05   SPI interrupts are in the range [0-987].  PPI interrupts are in the
6. 06   range [0-15].
7. 07
8. 08   The 3rd cell is the flags, encoded as follows:
9. 09         bits[3:0] trigger type and level flags.
10. 10                 1 = low-to-high edge triggered
11. 11                 2 = high-to-low edge triggered
12. 12                 4 = active high level-sensitive
13. 13                 8 = active low level-sensitive
14. 14         bits[15:8] PPI interrupt cpu mask.  Each bit corresponds to each of
15. 15         the 8 possible cpus attached to the GIC.  A bit set to '1' indicated
16. 16         the interrupt is wired to that CPU.  Only valid for PPI interrupts.

01 The 1st cell is the interrupt type; 0 for SPI interrupts, 1 for PPI

02 interrupts.

03

04 The 2nd cell contains the interrupt number for the interrupt type.

05 SPI interrupts are in the range [0-987]. PPI interrupts are in the

06 range [0-15].

07

08 The 3rd cell is the flags, encoded as follows:

09 bits[3:0] trigger type and level flags.

10 1 = low-to-high edge triggered

11 2 = high-to-low edge triggered

12 4 = active high level-sensitive

13 8 = active low level-sensitive

14 bits[15:8] PPI interrupt cpu mask. Each bit corresponds to each of

15 the 8 possible cpus attached to the GIC. A bit set to '1' indicated

16 the interrupt is wired to that CPU. Only valid for PPI interrupts.

另外，值得注意的是，一个设备还可能用到多个中断号。对于ARM GIC而言，若某设备使用了SPI的168、169号2个中断，而言都是高电平触发，则该设备结点的interrupts属性可定义为：interrupts = <0 168 4>, <0 169 4>;  
除了中断以外，在ARM Linux中clock、GPIO、pinmux都可以透过.dts中的结点和属性进行描述。

**DTC (device tree compiler)**

将.dts编译为.dtb的工具。DTC的源代码位于内核的scripts/dtc目录，在Linux内核使能了Device Tree的情况下，编译内核的时候主机工具dtc会被编译出来，对应scripts/dtc/Makefile中的“hostprogs-y := dtc”这一hostprogs编译target。  
在Linux内核的arch/arm/boot/dts/Makefile中，描述了当某种SoC被选中后，哪些.dtb文件会被编译出来，如与VEXPRESS对应的.dtb包括：

**[plain]** [view plain](http://blog.csdn.net/21cnbao/article/details/8457546" \o "view plain)[copy](http://blog.csdn.net/21cnbao/article/details/8457546" \o "copy)[print](http://blog.csdn.net/21cnbao/article/details/8457546" \o "print)[?](http://blog.csdn.net/21cnbao/article/details/8457546" \o "?)

1. dtb-$(CONFIG\_ARCH\_VEXPRESS) += vexpress-v2p-ca5s.dtb \
2. vexpress-v2p-ca9.dtb \
3. vexpress-v2p-ca15-tc1.dtb \
4. vexpress-v2p-ca15\_a7.dtb \
5. xenvm-4.2.dtb

dtb-$(CONFIG\_ARCH\_VEXPRESS) += vexpress-v2p-ca5s.dtb \

vexpress-v2p-ca9.dtb \

vexpress-v2p-ca15-tc1.dtb \

vexpress-v2p-ca15\_a7.dtb \

xenvm-4.2.dtb

在Linux下，我们可以单独编译Device Tree文件。当我们在Linux内核下运行make dtbs时，若我们之前选择了ARCH\_VEXPRESS，上述.dtb都会由对应的.dts编译出来。因为arch/arm/Makefile中含有一个dtbs编译target项目。

**Device Tree Blob (.dtb)**

.dtb是.dts被DTC编译后的二进制格式的Device Tree描述，可由Linux内核解析。通常在我们为电路板制作NAND、SD启动image时，会为.dtb文件单独留下一个很小的区域以存放之，之后bootloader在引导kernel的过程中，会先读取该.dtb到内存。

**Binding**

对于Device Tree中的结点和属性具体是如何来描述设备的硬件细节的，一般需要文档来进行讲解，文档的后缀名一般为.txt。这些文档位于内核的Documentation/devicetree/bindings目录，其下又分为很多子目录。

**Bootloader**

Uboot mainline 从 v1.1.3开始支持Device Tree，其对ARM的支持则是和ARM内核支持Device Tree同期完成。  
为了使能Device Tree，需要编译Uboot的时候在config文件中加入  
#define CONFIG\_OF\_LIBFDT   
在Uboot中，可以从NAND、SD或者TFTP等任意介质将.dtb读入内存，假设.dtb放入的内存地址为0x71000000，之后可在Uboot运行命令fdt addr命令设置.dtb的地址，如：  
U-Boot> fdt addr 0x71000000  
fdt的其他命令就变地可以使用，如fdt resize、fdt print等。  
对于ARM来讲，可以透过bootz kernel\_addr initrd\_address dtb\_address的命令来启动内核，即dtb\_address作为bootz或者bootm的最后一次参数，第一个参数为内核映像的地址，第二个参数为initrd的地址，若不存在initrd，可以用 -代替。

**3.    Device Tree引发的BSP和驱动变更**

有了Device Tree后，大量的板级信息都不再需要，譬如过去经常在arch/arm/plat-xxx和arch/arm/mach-xxx实施的如下事情：  
1.    注册platform\_device，绑定resource，即内存、IRQ等板级信息。

透过Device Tree后，形如

**[cpp]** [view plain](http://blog.csdn.net/21cnbao/article/details/8457546" \o "view plain)[copy](http://blog.csdn.net/21cnbao/article/details/8457546" \o "copy)[print](http://blog.csdn.net/21cnbao/article/details/8457546" \o "print)[?](http://blog.csdn.net/21cnbao/article/details/8457546" \o "?)

1. 90 static struct resource xxx\_resources[] = {
2. 91         [0] = {
3. 92                 .start  = …,
4. 93                 .end    = …,
5. 94                 .flags  = IORESOURCE\_MEM,
6. 95         },
7. 96         [1] = {
8. 97                 .start  = …,
9. 98                 .end    = …,
10. 99                 .flags  = IORESOURCE\_IRQ,
11. 100         },
12. 101 };
13. 102
14. 103 static struct platform\_device xxx\_device = {
15. 104         .name           = "xxx",
16. 105         .id             = -1,
17. 106         .dev            = {
18. 107                                 .platform\_data          = &xxx\_data,
19. 108         },
20. 109         .resource       = xxx\_resources,
21. 110         .num\_resources  = ARRAY\_SIZE(xxx\_resources),
22. 111 };

90 static struct resource xxx\_resources[] = {

91 [0] = {

92 .start = …,

93 .end = …,

94 .flags = IORESOURCE\_MEM,

95 },

96 [1] = {

97 .start = …,

98 .end = …,

99 .flags = IORESOURCE\_IRQ,

100 },

101 };

102

103 static struct platform\_device xxx\_device = {

104 .name = "xxx",

105 .id = -1,

106 .dev = {

107 .platform\_data = &xxx\_data,

108 },

109 .resource = xxx\_resources,

110 .num\_resources = ARRAY\_SIZE(xxx\_resources),

111 };

之类的platform\_device代码都不再需要，其中platform\_device会由kernel自动展开。而这些resource实际来源于.dts中设备结点的reg、interrupts属性。典型地，大多数总线都与“simple\_bus”兼容，而在SoC对应的machine的.init\_machine成员函数中，调用of\_platform\_bus\_probe(NULL, xxx\_of\_bus\_ids, NULL);即可自动展开所有的platform\_device。譬如，假设我们有个XXX SoC，则可在arch/arm/mach-xxx/的板文件中透过如下方式展开.dts中的设备结点对应的platform\_device：

**[cpp]** [view plain](http://blog.csdn.net/21cnbao/article/details/8457546" \o "view plain)[copy](http://blog.csdn.net/21cnbao/article/details/8457546" \o "copy)[print](http://blog.csdn.net/21cnbao/article/details/8457546" \o "print)[?](http://blog.csdn.net/21cnbao/article/details/8457546" \o "?)

1. 18 static struct of\_device\_id xxx\_of\_bus\_ids[] \_\_initdata = {
2. 19         { .compatible = "simple-bus", },
3. 20         {},
4. 21 };
5. 22
6. 23 void \_\_init xxx\_mach\_init(void)
7. 24 {
8. 25         of\_platform\_bus\_probe(NULL, xxx\_of\_bus\_ids, NULL);
9. 26 }
10. 32
11. 33 #ifdef CONFIG\_ARCH\_XXX
12. 38
13. 39 DT\_MACHINE\_START(XXX\_DT, "Generic XXX (Flattened Device Tree)")
14. 41         …
15. 45         .init\_machine   = xxx\_mach\_init,
16. 46         …
17. 49 MACHINE\_END
18. 50 #endif

18 static struct of\_device\_id xxx\_of\_bus\_ids[] \_\_initdata = {

19 { .compatible = "simple-bus", },

20 {},

21 };

22

23 void \_\_init xxx\_mach\_init(void)

24 {

25 of\_platform\_bus\_probe(NULL, xxx\_of\_bus\_ids, NULL);

26 }

32

33 #ifdef CONFIG\_ARCH\_XXX

38

39 DT\_MACHINE\_START(XXX\_DT, "Generic XXX (Flattened Device Tree)")

41 …

45 .init\_machine = xxx\_mach\_init,

46 …

49 MACHINE\_END

50 #endif

2.    注册i2c\_board\_info，指定IRQ等板级信息。

形如

**[cpp]** [view plain](http://blog.csdn.net/21cnbao/article/details/8457546" \o "view plain)[copy](http://blog.csdn.net/21cnbao/article/details/8457546" \o "copy)[print](http://blog.csdn.net/21cnbao/article/details/8457546" \o "print)[?](http://blog.csdn.net/21cnbao/article/details/8457546" \o "?)

1. 145 static struct i2c\_board\_info \_\_initdata afeb9260\_i2c\_devices[] = {
2. 146         {
3. 147                 I2C\_BOARD\_INFO("tlv320aic23", 0x1a),
4. 148         }, {
5. 149                 I2C\_BOARD\_INFO("fm3130", 0x68),
6. 150         }, {
7. 151                 I2C\_BOARD\_INFO("24c64", 0x50),
8. 152         },
9. 153 };

145 static struct i2c\_board\_info \_\_initdata afeb9260\_i2c\_devices[] = {

146 {

147 I2C\_BOARD\_INFO("tlv320aic23", 0x1a),

148 }, {

149 I2C\_BOARD\_INFO("fm3130", 0x68),

150 }, {

151 I2C\_BOARD\_INFO("24c64", 0x50),

152 },

153 };

之类的i2c\_board\_info代码，目前不再需要出现，现在只需要把tlv320aic23、fm3130、24c64这些设备结点填充作为相应的I2C controller结点的子结点即可，类似于前面的

**[cpp]** [view plain](http://blog.csdn.net/21cnbao/article/details/8457546" \o "view plain)[copy](http://blog.csdn.net/21cnbao/article/details/8457546" \o "copy)[print](http://blog.csdn.net/21cnbao/article/details/8457546" \o "print)[?](http://blog.csdn.net/21cnbao/article/details/8457546" \o "?)

1. i2c@1,0 {
2. compatible = "acme,a1234-i2c-bus";
3. …
4. rtc@58 {
5. compatible = "maxim,ds1338";
6. reg = <58>;
7. interrupts = < 7 3 >;
8. };
9. };

i2c@1,0 {

compatible = "acme,a1234-i2c-bus";

…

rtc@58 {

compatible = "maxim,ds1338";

reg = <58>;

interrupts = < 7 3 >;

};

};

Device Tree中的I2C client会透过I2C host驱动的probe()函数中调用of\_i2c\_register\_devices(&i2c\_dev->adapter);被自动展开。

3.    注册spi\_board\_info，指定IRQ等板级信息。

形如

**[cpp]** [view plain](http://blog.csdn.net/21cnbao/article/details/8457546" \o "view plain)[copy](http://blog.csdn.net/21cnbao/article/details/8457546" \o "copy)[print](http://blog.csdn.net/21cnbao/article/details/8457546" \o "print)[?](http://blog.csdn.net/21cnbao/article/details/8457546" \o "?)

1. 79 static struct spi\_board\_info afeb9260\_spi\_devices[] = {
2. 80         {       /\* DataFlash chip \*/
3. 81                 .modalias       = "mtd\_dataflash",
4. 82                 .chip\_select    = 1,
5. 83                 .max\_speed\_hz   = 15 \* 1000 \* 1000,
6. 84                 .bus\_num        = 0,
7. 85         },
8. 86 };

79 static struct spi\_board\_info afeb9260\_spi\_devices[] = {

80 { /\* DataFlash chip \*/

81 .modalias = "mtd\_dataflash",

82 .chip\_select = 1,

83 .max\_speed\_hz = 15 \* 1000 \* 1000,

84 .bus\_num = 0,

85 },

86 };

之类的spi\_board\_info代码，目前不再需要出现，与I2C类似，现在只需要把mtd\_dataflash之类的结点，作为SPI控制器的子结点即可，SPI host驱动的probe函数透过spi\_register\_master()注册master的时候，会自动展开依附于它的slave。

4.    多个针对不同电路板的machine，以及相关的callback。

过去，ARM Linux针对不同的电路板会建立由MACHINE\_START和MACHINE\_END包围起来的针对这个machine的一系列callback，譬如：

**[cpp]** [view plain](http://blog.csdn.net/21cnbao/article/details/8457546" \o "view plain)[copy](http://blog.csdn.net/21cnbao/article/details/8457546" \o "copy)[print](http://blog.csdn.net/21cnbao/article/details/8457546" \o "print)[?](http://blog.csdn.net/21cnbao/article/details/8457546" \o "?)

1. 373 MACHINE\_START(VEXPRESS, "ARM-Versatile Express")
2. 374         .atag\_offset    = 0x100,
3. 375         .smp            = smp\_ops(vexpress\_smp\_ops),
4. 376         .map\_io         = v2m\_map\_io,
5. 377         .init\_early     = v2m\_init\_early,
6. 378         .init\_irq       = v2m\_init\_irq,
7. 379         .timer          = &v2m\_timer,
8. 380         .handle\_irq     = gic\_handle\_irq,
9. 381         .init\_machine   = v2m\_init,
10. 382         .restart        = vexpress\_restart,
11. 383 MACHINE\_END

373 MACHINE\_START(VEXPRESS, "ARM-Versatile Express")

374 .atag\_offset = 0x100,

375 .smp = smp\_ops(vexpress\_smp\_ops),

376 .map\_io = v2m\_map\_io,

377 .init\_early = v2m\_init\_early,

378 .init\_irq = v2m\_init\_irq,

379 .timer = &v2m\_timer,

380 .handle\_irq = gic\_handle\_irq,

381 .init\_machine = v2m\_init,

382 .restart = vexpress\_restart,

383 MACHINE\_END

这些不同的machine会有不同的MACHINE ID，Uboot在启动Linux内核时会将MACHINE ID存放在r1寄存器，Linux启动时会匹配Bootloader传递的MACHINE ID和MACHINE\_START声明的MACHINE ID，然后执行相应machine的一系列初始化函数。

引入Device Tree之后，MACHINE\_START变更为DT\_MACHINE\_START，其中含有一个.dt\_compat成员，用于表明相关的machine与.dts中root结点的compatible属性兼容关系。如果Bootloader传递给内核的Device Tree中root结点的compatible属性出现在某machine的.dt\_compat表中，相关的machine就与对应的Device Tree匹配，从而引发这一machine的一系列初始化函数被执行。

**[cpp]** [view plain](http://blog.csdn.net/21cnbao/article/details/8457546" \o "view plain)[copy](http://blog.csdn.net/21cnbao/article/details/8457546" \o "copy)[print](http://blog.csdn.net/21cnbao/article/details/8457546" \o "print)[?](http://blog.csdn.net/21cnbao/article/details/8457546" \o "?)

1. 489 static const **char** \* const v2m\_dt\_match[] \_\_initconst = {
2. 490         "arm,vexpress",
3. 491         "xen,xenvm",
4. 492         NULL,
5. 493 };
6. 495 DT\_MACHINE\_START(VEXPRESS\_DT, "ARM-Versatile Express")
7. 496         .dt\_compat      = v2m\_dt\_match,
8. 497         .smp            = smp\_ops(vexpress\_smp\_ops),
9. 498         .map\_io         = v2m\_dt\_map\_io,
10. 499         .init\_early     = v2m\_dt\_init\_early,
11. 500         .init\_irq       = v2m\_dt\_init\_irq,
12. 501         .timer          = &v2m\_dt\_timer,
13. 502         .init\_machine   = v2m\_dt\_init,
14. 503         .handle\_irq     = gic\_handle\_irq,
15. 504         .restart        = vexpress\_restart,
16. 505 MACHINE\_END

489 static const char \* const v2m\_dt\_match[] \_\_initconst = {

490 "arm,vexpress",

491 "xen,xenvm",

492 NULL,

493 };

495 DT\_MACHINE\_START(VEXPRESS\_DT, "ARM-Versatile Express")

496 .dt\_compat = v2m\_dt\_match,

497 .smp = smp\_ops(vexpress\_smp\_ops),

498 .map\_io = v2m\_dt\_map\_io,

499 .init\_early = v2m\_dt\_init\_early,

500 .init\_irq = v2m\_dt\_init\_irq,

501 .timer = &v2m\_dt\_timer,

502 .init\_machine = v2m\_dt\_init,

503 .handle\_irq = gic\_handle\_irq,

504 .restart = vexpress\_restart,

505 MACHINE\_END

Linux倡导针对多个SoC、多个电路板的通用DT machine，即一个DT machine的.dt\_compat表含多个电路板.dts文件的root结点compatible属性字符串。之后，如果的电路板的初始化序列不一样，可以透过int of\_machine\_is\_compatible(const char \*compat) API判断具体的电路板是什么。

    譬如arch/arm/mach-exynos/mach-exynos5-dt.c的EXYNOS5\_DT machine同时兼容"samsung,exynos5250"和"samsung,exynos5440"：

**[cpp]** [view plain](http://blog.csdn.net/21cnbao/article/details/8457546" \o "view plain)[copy](http://blog.csdn.net/21cnbao/article/details/8457546" \o "copy)[print](http://blog.csdn.net/21cnbao/article/details/8457546" \o "print)[?](http://blog.csdn.net/21cnbao/article/details/8457546" \o "?)

1. 158 static **char** const \*exynos5\_dt\_compat[] \_\_initdata = {
2. 159         "samsung,exynos5250",
3. 160         "samsung,exynos5440",
4. 161         NULL
5. 162 };
6. 163
7. 177 DT\_MACHINE\_START(EXYNOS5\_DT, "SAMSUNG EXYNOS5 (Flattened Device Tree)")
8. 178         /\* Maintainer: Kukjin Kim <kgene.kim@samsung.com> \*/
9. 179         .init\_irq       = exynos5\_init\_irq,
10. 180         .smp            = smp\_ops(exynos\_smp\_ops),
11. 181         .map\_io         = exynos5\_dt\_map\_io,
12. 182         .handle\_irq     = gic\_handle\_irq,
13. 183         .init\_machine   = exynos5\_dt\_machine\_init,
14. 184         .init\_late      = exynos\_init\_late,
15. 185         .timer          = &exynos4\_timer,
16. 186         .dt\_compat      = exynos5\_dt\_compat,
17. 187         .restart        = exynos5\_restart,
18. 188         .reserve        = exynos5\_reserve,
19. 189 MACHINE\_END

158 static char const \*exynos5\_dt\_compat[] \_\_initdata = {

159 "samsung,exynos5250",

160 "samsung,exynos5440",

161 NULL

162 };

163

177 DT\_MACHINE\_START(EXYNOS5\_DT, "SAMSUNG EXYNOS5 (Flattened Device Tree)")

178 /\* Maintainer: Kukjin Kim <kgene.kim@samsung.com> \*/

179 .init\_irq = exynos5\_init\_irq,

180 .smp = smp\_ops(exynos\_smp\_ops),

181 .map\_io = exynos5\_dt\_map\_io,

182 .handle\_irq = gic\_handle\_irq,

183 .init\_machine = exynos5\_dt\_machine\_init,

184 .init\_late = exynos\_init\_late,

185 .timer = &exynos4\_timer,

186 .dt\_compat = exynos5\_dt\_compat,

187 .restart = exynos5\_restart,

188 .reserve = exynos5\_reserve,

189 MACHINE\_END

     它的.init\_machine成员函数就针对不同的machine进行了不同的分支处理：

**[cpp]** [view plain](http://blog.csdn.net/21cnbao/article/details/8457546" \o "view plain)[copy](http://blog.csdn.net/21cnbao/article/details/8457546" \o "copy)[print](http://blog.csdn.net/21cnbao/article/details/8457546" \o "print)[?](http://blog.csdn.net/21cnbao/article/details/8457546" \o "?)

1. 126 static void \_\_init exynos5\_dt\_machine\_init(void)
2. 127 {
3. 128         …
4. 149
5. 150         if (of\_machine\_is\_compatible("samsung,exynos5250"))
6. 151                 of\_platform\_populate(NULL, of\_default\_bus\_match\_table,
7. 152                                      exynos5250\_auxdata\_lookup, NULL);
8. 153         else if (of\_machine\_is\_compatible("samsung,exynos5440"))
9. 154                 of\_platform\_populate(NULL, of\_default\_bus\_match\_table,
10. 155                                      exynos5440\_auxdata\_lookup, NULL);
11. 156 }

126 static void \_\_init exynos5\_dt\_machine\_init(void)

127 {

128 …

149

150 if (of\_machine\_is\_compatible("samsung,exynos5250"))

151 of\_platform\_populate(NULL, of\_default\_bus\_match\_table,

152 exynos5250\_auxdata\_lookup, NULL);

153 else if (of\_machine\_is\_compatible("samsung,exynos5440"))

154 of\_platform\_populate(NULL, of\_default\_bus\_match\_table,

155 exynos5440\_auxdata\_lookup, NULL);

156 }

使用Device Tree后，驱动需要与.dts中描述的设备结点进行匹配，从而引发驱动的probe()函数执行。对于platform\_driver而言，需要添加一个OF匹配表，如前文的.dts文件的"acme,a1234-i2c-bus"兼容I2C控制器结点的OF匹配表可以是：

**[cpp]** [view plain](http://blog.csdn.net/21cnbao/article/details/8457546" \o "view plain)[copy](http://blog.csdn.net/21cnbao/article/details/8457546" \o "copy)[print](http://blog.csdn.net/21cnbao/article/details/8457546" \o "print)[?](http://blog.csdn.net/21cnbao/article/details/8457546" \o "?)

1. 436 static const struct of\_device\_id a1234\_i2c\_of\_match[] = {
2. 437         { .compatible = "acme,a1234-i2c-bus ", },
3. 438         {},
4. 439 };
5. 440 MODULE\_DEVICE\_TABLE(of, a1234\_i2c\_of\_match);
6. 441
7. 442 static struct platform\_driver i2c\_a1234\_driver = {
8. 443         .driver = {
9. 444                 .name = "a1234-i2c-bus ",
10. 445                 .owner = THIS\_MODULE,
11. 449                 .of\_match\_table = a1234\_i2c\_of\_match,
12. 450         },
13. 451         .probe = i2c\_a1234\_probe,
14. 452         .remove = i2c\_a1234\_remove,
15. 453 };
16. 454 module\_platform\_driver(i2c\_a1234\_driver);

436 static const struct of\_device\_id a1234\_i2c\_of\_match[] = {

437 { .compatible = "acme,a1234-i2c-bus ", },

438 {},

439 };

440 MODULE\_DEVICE\_TABLE(of, a1234\_i2c\_of\_match);

441

442 static struct platform\_driver i2c\_a1234\_driver = {

443 .driver = {

444 .name = "a1234-i2c-bus ",

445 .owner = THIS\_MODULE,

449 .of\_match\_table = a1234\_i2c\_of\_match,

450 },

451 .probe = i2c\_a1234\_probe,

452 .remove = i2c\_a1234\_remove,

453 };

454 module\_platform\_driver(i2c\_a1234\_driver);

对于I2C和SPI从设备而言，同样也可以透过of\_match\_table添加匹配的.dts中的相关结点的compatible属性，如sound/soc/codecs/wm8753.c中的：

**[cpp]** [view plain](http://blog.csdn.net/21cnbao/article/details/8457546" \o "view plain)[copy](http://blog.csdn.net/21cnbao/article/details/8457546" \o "copy)[print](http://blog.csdn.net/21cnbao/article/details/8457546" \o "print)[?](http://blog.csdn.net/21cnbao/article/details/8457546" \o "?)

1. 1533 static const struct of\_device\_id wm8753\_of\_match[] = {
2. 1534         { .compatible = "wlf,wm8753", },
3. 1535         { }
4. 1536 };
5. 1537 MODULE\_DEVICE\_TABLE(of, wm8753\_of\_match);
6. 1587 static struct spi\_driver wm8753\_spi\_driver = {
7. 1588         .driver = {
8. 1589                 .name   = "wm8753",
9. 1590                 .owner  = THIS\_MODULE,
10. 1591                 .of\_match\_table = wm8753\_of\_match,
11. 1592         },
12. 1593         .probe          = wm8753\_spi\_probe,
13. 1594         .remove         = wm8753\_spi\_remove,
14. 1595 };
15. 1640 static struct i2c\_driver wm8753\_i2c\_driver = {
16. 1641         .driver = {
17. 1642                 .name = "wm8753",
18. 1643                 .owner = THIS\_MODULE,
19. 1644                 .of\_match\_table = wm8753\_of\_match,
20. 1645         },
21. 1646         .probe =    wm8753\_i2c\_probe,
22. 1647         .remove =   wm8753\_i2c\_remove,
23. 1648         .id\_table = wm8753\_i2c\_id,
24. 1649 };

1533 static const struct of\_device\_id wm8753\_of\_match[] = {

1534 { .compatible = "wlf,wm8753", },

1535 { }

1536 };

1537 MODULE\_DEVICE\_TABLE(of, wm8753\_of\_match);

1587 static struct spi\_driver wm8753\_spi\_driver = {

1588 .driver = {

1589 .name = "wm8753",

1590 .owner = THIS\_MODULE,

1591 .of\_match\_table = wm8753\_of\_match,

1592 },

1593 .probe = wm8753\_spi\_probe,

1594 .remove = wm8753\_spi\_remove,

1595 };

1640 static struct i2c\_driver wm8753\_i2c\_driver = {

1641 .driver = {

1642 .name = "wm8753",

1643 .owner = THIS\_MODULE,

1644 .of\_match\_table = wm8753\_of\_match,

1645 },

1646 .probe = wm8753\_i2c\_probe,

1647 .remove = wm8753\_i2c\_remove,

1648 .id\_table = wm8753\_i2c\_id,

1649 };

不过这边有一点需要提醒的是，I2C和SPI外设驱动和Device Tree中设备结点的compatible 属性还有一种弱式匹配方法，就是别名匹配。compatible 属性的组织形式为<manufacturer>,<model>，别名其实就是去掉compatible 属性中逗号前的manufacturer前缀。关于这一点，可查看drivers/spi/spi.c的源代码，函数spi\_match\_device()暴露了更多的细节，如果别名出现在设备spi\_driver的id\_table里面，或者别名与spi\_driver的name字段相同，SPI设备和驱动都可以匹配上：

**[cpp]** [view plain](http://blog.csdn.net/21cnbao/article/details/8457546" \o "view plain)[copy](http://blog.csdn.net/21cnbao/article/details/8457546" \o "copy)[print](http://blog.csdn.net/21cnbao/article/details/8457546" \o "print)[?](http://blog.csdn.net/21cnbao/article/details/8457546" \o "?)

1. 90 static **int** spi\_match\_device(struct device \*dev, struct device\_driver \*drv)
2. 91 {
3. 92         const struct spi\_device \*spi = to\_spi\_device(dev);
4. 93         const struct spi\_driver \*sdrv = to\_spi\_driver(drv);
5. 94
6. 95         /\* Attempt an OF style match \*/
7. 96         if (of\_driver\_match\_device(dev, drv))
8. 97                 return 1;
9. 98
10. 99         /\* Then try ACPI \*/
11. 100         if (acpi\_driver\_match\_device(dev, drv))
12. 101                 return 1;
13. 102
14. 103         if (sdrv->id\_table)
15. 104                 return !!spi\_match\_id(sdrv->id\_table, spi);
16. 105
17. 106         return strcmp(spi->modalias, drv->name) == 0;
18. 107 }
19. 71 static const struct spi\_device\_id \*spi\_match\_id(const struct spi\_device\_id \*id,
20. 72                                                 const struct spi\_device \*sdev)
21. 73 {
22. 74         while (id->name[0]) {
23. 75                 if (!strcmp(sdev->modalias, id->name))
24. 76                         return id;
25. 77                 id++;
26. 78         }
27. 79         return NULL;
28. 80 }

90 static int spi\_match\_device(struct device \*dev, struct device\_driver \*drv)

91 {

92 const struct spi\_device \*spi = to\_spi\_device(dev);

93 const struct spi\_driver \*sdrv = to\_spi\_driver(drv);

94

95 /\* Attempt an OF style match \*/

96 if (of\_driver\_match\_device(dev, drv))

97 return 1;

98

99 /\* Then try ACPI \*/

100 if (acpi\_driver\_match\_device(dev, drv))

101 return 1;

102

103 if (sdrv->id\_table)

104 return !!spi\_match\_id(sdrv->id\_table, spi);

105

106 return strcmp(spi->modalias, drv->name) == 0;

107 }

71 static const struct spi\_device\_id \*spi\_match\_id(const struct spi\_device\_id \*id,

72 const struct spi\_device \*sdev)

73 {

74 while (id->name[0]) {

75 if (!strcmp(sdev->modalias, id->name))

76 return id;

77 id++;

78 }

79 return NULL;

80 }

**4.    常用OF API**

在Linux的BSP和驱动代码中，还经常会使用到Linux中一组Device Tree的API,这些API通常被冠以of\_前缀，它们的实现代码位于内核的drivers/of目录。这些常用的API包括：

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

判断设备结点的compatible 属性是否包含compat指定的字符串。当一个驱动支持2个或多个设备的时候，这些不同.dts文件中设备的compatible 属性都会进入驱动 OF匹配表。因此驱动可以透过Bootloader传递给内核的Device Tree中的真正结点的compatible 属性以确定究竟是哪一种设备，从而根据不同的设备类型进行不同的处理。如drivers/pinctrl/pinctrl-sirf.c即兼容于"sirf,prima2-pinctrl"，又兼容于"sirf,prima2-pinctrl"，在驱动中就有相应分支处理：

**[cpp]** [view plain](http://blog.csdn.net/21cnbao/article/details/8457546" \o "view plain)[copy](http://blog.csdn.net/21cnbao/article/details/8457546" \o "copy)[print](http://blog.csdn.net/21cnbao/article/details/8457546" \o "print)[?](http://blog.csdn.net/21cnbao/article/details/8457546" \o "?)

1. 1682 if (of\_device\_is\_compatible(np, "sirf,marco-pinctrl"))
2. 1683      is\_marco = 1;

1682 if (of\_device\_is\_compatible(np, "sirf,marco-pinctrl"))

1683 is\_marco = 1;

**struct device\_node \*of\_find\_compatible\_node(struct device\_node \*from,**

**const char \*type, const char \*compatible);**

根据compatible属性，获得设备结点。遍历Device Tree中所有的设备结点，看看哪个结点的类型、compatible属性与本函数的输入参数匹配，大多数情况下，from、type为NULL。

**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);**

**int of\_property\_read\_u64(const struct device\_node \*np, const char**

**\*propname, u64 \*out\_value);**

读取设备结点np的属性名为propname，类型为8、16、32、64位整型数组的属性。对于32位处理器来讲，最常用的是of\_property\_read\_u32\_array()。如在arch/arm/mm/cache-l2x0.c中，透过如下语句读取L2 cache的"arm,data-latency"属性：

**[cpp]** [view plain](http://blog.csdn.net/21cnbao/article/details/8457546" \o "view plain)[copy](http://blog.csdn.net/21cnbao/article/details/8457546" \o "copy)[print](http://blog.csdn.net/21cnbao/article/details/8457546" \o "print)[?](http://blog.csdn.net/21cnbao/article/details/8457546" \o "?)

1. 534         of\_property\_read\_u32\_array(np, "arm,data-latency",
2. 535                                    data, ARRAY\_SIZE(data));

534 of\_property\_read\_u32\_array(np, "arm,data-latency",

535 data, ARRAY\_SIZE(data));

在arch/arm/boot/dts/vexpress-v2p-ca9.dts中，含有"arm,data-latency"属性的L2 cache结点如下：

**[cpp]** [view plain](http://blog.csdn.net/21cnbao/article/details/8457546" \o "view plain)[copy](http://blog.csdn.net/21cnbao/article/details/8457546" \o "copy)[print](http://blog.csdn.net/21cnbao/article/details/8457546" \o "print)[?](http://blog.csdn.net/21cnbao/article/details/8457546" \o "?)

1. 137         L2: cache-controller@1e00a000 {
2. 138                 compatible = "arm,pl310-cache";
3. 139                 reg = <0x1e00a000 0x1000>;
4. 140                 interrupts = <0 43 4>;
5. 141                 cache-level = <2>;
6. 142                 arm,data-latency = <1 1 1>;
7. 143                 arm,tag-latency = <1 1 1>;
8. 144         }

137 L2: cache-controller@1e00a000 {

138 compatible = "arm,pl310-cache";

139 reg = <0x1e00a000 0x1000>;

140 interrupts = <0 43 4>;

141 cache-level = <2>;

142 arm,data-latency = <1 1 1>;

143 arm,tag-latency = <1 1 1>;

144 }

有些情况下，整形属性的长度可能为1，于是内核为了方便调用者，又在上述API的基础上封装出了更加简单的读单一整形属性的API，它们为int of\_property\_read\_u8()、of\_property\_read\_u16()等，实现于include/linux/of.h：

**[cpp]** [view plain](http://blog.csdn.net/21cnbao/article/details/8457546" \o "view plain)[copy](http://blog.csdn.net/21cnbao/article/details/8457546" \o "copy)[print](http://blog.csdn.net/21cnbao/article/details/8457546" \o "print)[?](http://blog.csdn.net/21cnbao/article/details/8457546" \o "?)

1. 513 static inline **int** of\_property\_read\_u8(const struct device\_node \*np,
2. 514                                        const **char** \*propname,
3. 515                                        u8 \*out\_value)
4. 516 {
5. 517         return of\_property\_read\_u8\_array(np, propname, out\_value, 1);
6. 518 }
7. 519
8. 520 static inline **int** of\_property\_read\_u16(const struct device\_node \*np,
9. 521                                        const **char** \*propname,
10. 522                                        u16 \*out\_value)
11. 523 {
12. 524         return of\_property\_read\_u16\_array(np, propname, out\_value, 1);
13. 525 }
14. 526
15. 527 static inline **int** of\_property\_read\_u32(const struct device\_node \*np,
16. 528                                        const **char** \*propname,
17. 529                                        u32 \*out\_value)
18. 530 {
19. 531         return of\_property\_read\_u32\_array(np, propname, out\_value, 1);
20. 532 }

513 static inline int of\_property\_read\_u8(const struct device\_node \*np,

514 const char \*propname,

515 u8 \*out\_value)

516 {

517 return of\_property\_read\_u8\_array(np, propname, out\_value, 1);

518 }

519

520 static inline int of\_property\_read\_u16(const struct device\_node \*np,

521 const char \*propname,

522 u16 \*out\_value)

523 {

524 return of\_property\_read\_u16\_array(np, propname, out\_value, 1);

525 }

526

527 static inline int of\_property\_read\_u32(const struct device\_node \*np,

528 const char \*propname,

529 u32 \*out\_value)

530 {

531 return of\_property\_read\_u32\_array(np, propname, out\_value, 1);

532 }

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

**\*propname, const char \*\*out\_string);**

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

**\*propname, int index, const char \*\*output);**

前者读取字符串属性，后者读取字符串数组属性中的第index个字符串。如drivers/clk/clk.c中的of\_clk\_get\_parent\_name()透过of\_property\_read\_string\_index()遍历clkspec结点的所有"clock-output-names"字符串数组属性。

**[cpp]** [view plain](http://blog.csdn.net/21cnbao/article/details/8457546" \o "view plain)[copy](http://blog.csdn.net/21cnbao/article/details/8457546" \o "copy)[print](http://blog.csdn.net/21cnbao/article/details/8457546" \o "print)[?](http://blog.csdn.net/21cnbao/article/details/8457546" \o "?)

1. 1759 const **char** \*of\_clk\_get\_parent\_name(struct device\_node \*np, **int** index)
2. 1760 {
3. 1761         struct of\_phandle\_args clkspec;
4. 1762         const **char** \*clk\_name;
5. 1763         **int** rc;
6. 1764
7. 1765         if (index < 0)
8. 1766                 return NULL;
9. 1767
10. 1768         rc = of\_parse\_phandle\_with\_args(np, "clocks", "#clock-cells", index,
11. 1769                                         &clkspec);
12. 1770         if (rc)
13. 1771                 return NULL;
14. 1772
15. 1773         if (of\_property\_read\_string\_index(clkspec.np, "clock-output-names",
16. 1774                                   clkspec.args\_count ? clkspec.args[0] : 0,
17. 1775                                           &clk\_name) < 0)
18. 1776                 clk\_name = clkspec.np->name;
19. 1777
20. 1778         of\_node\_put(clkspec.np);
21. 1779         return clk\_name;
22. 1780 }
23. 1781 EXPORT\_SYMBOL\_GPL(of\_clk\_get\_parent\_name);

1759 const char \*of\_clk\_get\_parent\_name(struct device\_node \*np, int index)

1760 {

1761 struct of\_phandle\_args clkspec;

1762 const char \*clk\_name;

1763 int rc;

1764

1765 if (index < 0)

1766 return NULL;

1767

1768 rc = of\_parse\_phandle\_with\_args(np, "clocks", "#clock-cells", index,

1769 &clkspec);

1770 if (rc)

1771 return NULL;

1772

1773 if (of\_property\_read\_string\_index(clkspec.np, "clock-output-names",

1774 clkspec.args\_count ? clkspec.args[0] : 0,

1775 &clk\_name) < 0)

1776 clk\_name = clkspec.np->name;

1777

1778 of\_node\_put(clkspec.np);

1779 return clk\_name;

1780 }

1781 EXPORT\_SYMBOL\_GPL(of\_clk\_get\_parent\_name);

**static inline bool of\_property\_read\_bool(const struct device\_node \*np,**

**const char \*propname);**

如果设备结点np含有propname属性，则返回true，否则返回false。一般用于检查空属性是否存在。

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

通过设备结点直接进行设备内存区间的 ioremap()，index是内存段的索引。若设备结点的reg属性有多段，可通过index标示要ioremap的是哪一段，只有1段的情况，index为0。采用Device Tree后，大量的设备驱动通过of\_iomap()进行映射，而不再通过传统的ioremap。

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

透过Device Tree或者设备的中断号，实际上是从.dts中的interrupts属性解析出中断号。若设备使用了多个中断，index指定中断的索引号。

还有一些OF API，这里不一一列举，具体可参考include/linux/of.h头文件。

**5.    总结**

ARM社区一贯充斥的大量垃圾代码导致Linus盛怒，因此社区在2011年到2012年进行了大量的工作。ARM Linux开始围绕Device Tree展开，Device Tree有自己的独立的语法，它的源文件为.dts，编译后得到.dtb，Bootloader在引导Linux内核的时候会将.dtb地址告知内核。之后内核会展开Device Tree并创建和注册相关的设备，因此arch/arm/mach-xxx和arch/arm/plat-xxx中大量的用于注册platform、I2C、SPI板级信息的代码被删除，而驱动也以新的方式和.dts中定义的设备结点进行匹配。