


个人资料



21cnbao

访问： 334636次

积分： 3805分

排名： 第2560名

原创： 52篇 转载： 1篇

译文： 3篇 评论： 270条

文章搜索

文章分类

Android系统架构 (31)

Linux Driver开发 (15)

Linux Kernel开发 (14)

Linux Userspace开发 (3)

技术人生 (2)

文章存档

2014年10月 (1)

2014年07月 (2)

2014年06月 (1)

2014年03月 (1)

2014年02月 (1)

展开

ARM Linux 3.x的设备树 (Device Tree)

分类： Linux Driver开发 Linux Kernel开发 2013-01-01 17:32 47336人阅读 评论(31) 收藏 举报

目录(?) [+]

宋宝华 Barry Song <21cnbao@gmail.com>

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 f*cking 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，而這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架构的板级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。

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

阅读排行	
ARM Linux 3.x的设备树	(47269)
Linux芯片级移植与底层	(14895)
Linux gdb调试器用法全	(13265)
Service与Android系统实	(12508)
Android应用程序开发以	(12493)
分享《Linux设备驱动开	(10821)
Android架构纵横谈之—	(10448)
Android应用程序开发以	(8868)
Android架构纵横谈之—	(8022)
Android应用程序开发以	(7669)

评论排行	
《Linux设备驱动开发详	(42)
ARM Linux 3.x的设备树	(31)
《Linux设备驱动开发详	(19)
分享《Linux设备驱动开	(17)
Android架构纵横谈之—	(16)
Android应用程序开发以	(14)
炼狱与逐光——我的十杰	(14)
Android架构纵横谈之—	(13)
宋宝华ABC	(13)
Android架构纵横谈之—	(10)

推荐文章	
------	--

最新评论	
《Linux设备驱动开发详解(第3版特警屠龙: @21cnbao:年前? 是12月份吗?	
mbed OS - ARM关于物联网(IoT) jixia松: mbed前几周还来我公司给我们培训	
mbed OS - ARM关于物联网(IoT) bona020: 看好mbed! 准备往这方面发力有兴趣的朋友 +群 275951878	
mbed OS - ARM关于物联网(IoT) Frank0525: 宋老师, 有地方可以下载到源码和相关开发工具吗? 谢谢!	
《Linux设备驱动开发详解(第3版)ximinchao: 刚听完宋老师的课, 受益匪浅, 回来就买了一本第二版的。第三版出来还要继续买一本。赞!	
《Linux设备驱动开发详解(第3版)cjok376240497: 新增的这些内容都很赞, 期待。。	
Android架构纵横谈之——软件自sinat_18374237: 死得其所, 快哉, 快哉!	
Android架构纵横谈之——软件自sinat_18374237: 被故事感动啦~~、~~	
《Linux设备驱动开发详解(第3版)yilin1002: 第三版什么时候可以出版啊? 等不及了。。	

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]
01. / {
02.     node1 {
03.         a-string-property = "A string";
04.         a-string-list-property = "first string", "second string";
05.         a-byte-data-property = [0x01 0x23 0x34 0x56];
06.         child-node1 {
07.             first-child-property;
08.             second-child-property = <1>;
09.             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. };
```

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 SMC9111 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]
01. / {
02.     compatible = "acme,coyotes-revenge";
03.     #address-cells = <1>;
04.     #size-cells = <1>;
05.     interrupt-parent = <&intc>;
06.
07.     cpus {
08.         #address-cells = <1>;
09.         #size-cells = <0>;
10.         cpu@0 {
11.             compatible = "arm,cortex-a9";
12.             reg = <0>;
13.         };
```

Service与Android系统设计 (3)
hgzlpmg: 支持楼主啊啊啊

```

14.         cpu@1 {
15.             compatible = "arm,cortex-a9";
16.             reg = <1>;
17.         };
18.     };
19.
20.     serial@101f0000 {
21.         compatible = "arm,pl011";
22.         reg = <0x101f0000 0x1000 >;
23.         interrupts = < 1 0 >;
24.     };
25.
26.     serial@101f2000 {
27.         compatible = "arm,pl011";
28.         reg = <0x101f2000 0x1000 >;
29.         interrupts = < 2 0 >;
30.     };
31.
32.     gpio@101f3000 {
33.         compatible = "arm,pl061";
34.         reg = <0x101f3000 0x1000
35.             0x101f4000 0x0010>;
36.         interrupts = < 3 0 >;
37.     };
38.
39.     intc: interrupt-controller@10140000 {
40.         compatible = "arm,pl190";
41.         reg = <0x10140000 0x1000 >;
42.         interrupt-controller;
43.         #interrupt-cells = <2>;
44.     };
45.
46.     spi@10115000 {
47.         compatible = "arm,pl022";
48.         reg = <0x10115000 0x1000 >;
49.         interrupts = < 4 0 >;
50.     };
51.
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
58.
59.         ethernet@0,0 {
60.             compatible = "smc,smc91c111";
61.             reg = <0 0 0x1000>;
62.             interrupts = < 5 2 >;
63.         };
64.
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.         };
77.
78.         flash@2,0 {
79.             compatible = "samsung,k8f1315ebm", "cfi-flash";
80.             reg = <2 0 0x4000000>;
81.         };
82.     };
83. };

```

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]
01. flash@0,00000000 {
02.     compatible = "arm,vexpress-flash", "cfi-flash";
03.     reg = <0 0x00000000 0x04000000>,
04.     <1 0x00000000 0x04000000>;
05.     bank-width = <4>;
06. };
```

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]
01. ranges = <0 0 0x10100000 0x10000 // Chipselect 1, Ethernet
```

```

02.      1 0 0x10160000 0x10000 // Chipselect 2, i2c controller
03.      2 0 0x30000000 0x100000>; // 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]
01. 01 The 1st cell is the interrupt type; 0 for SPI interrupts, 1 for PPI
02. 02 interrupts.
03. 03
04. 04 The 2nd cell contains the interrupt number for the interrupt type.
05. 05 SPI interrupts are in the range [0-987]. PPI interrupts are in the
06. 06 range [0-15].
07. 07
08. 08 The 3rd cell is the flags, encoded as follows:
09. 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.

```

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]
01. dtb-$(CONFIG_ARCH_VEXPRESS) += vexpress-v2p-ca5s.dtb \
02.     vexpress-v2p-ca9.dtb \
03.     vexpress-v2p-ca15-tc1.dtb \
04.     vexpress-v2p-ca15_a7.dtb \
05.     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]
01. 90 static struct resource xxx_resources[] = {
02.     91     [0] = {
03.         92         .start = ...,
04.         93         .end   = ...,
05.         94         .flags  = IORESOURCE_MEM,
06.         95     },
07.     96     [1] = {
08.         97         .start = ...,
09.         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 };
```

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]
01. 18 static struct of_device_id xxx_of_bus_ids[] __initdata = {
02. 19     { .compatible = "simple-bus", },
03. 20     {}},
04. 21 };
05. 22
06. 23 void __init xxx_mach_init(void)
07. 24 {
08. 25     of_platform_bus_probe(NULL, xxx_of_bus_ids, NULL);
09. 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
```

2. 注册i2c_board_info, 指定IRQ等板级信息。

形如

```
[cpp]
01. 145 static struct i2c_board_info __initdata afeb9260_i2c_devices[] = {
02. 146     {
03. 147         I2C_BOARD_INFO("tlv320aic23", 0x1a),
04. 148     }, {
05. 149         I2C_BOARD_INFO("fm3130", 0x68),
06. 150     }, {
07. 151         I2C_BOARD_INFO("24c64", 0x50),
08. 152     },
09. 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]
01. i2c@1,0 {
02.     compatible = "acme,a1234-i2c-bus";
03.     ...
04.     rtc@58 {
05.         compatible = "maxim,ds1338";
06.         reg = <58>;
07.         interrupts = < 7 3 >;
08.     };
09. };
```

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]
01. 79 static struct spi_board_info afeb9260_spi_devices[] = {
02. 80     { /* DataFlash chip */
03. 81         .modalias = "mtd_dataflash",
04. 82         .chip_select = 1,
05. 83         .max_speed_hz = 15 * 1000 * 1000,
06. 84         .bus_num = 0,
07. 85     },
08. 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]
01. 373 MACHINE_START(VEXPRESS, "ARM-Versatile Express")
02. 374     .atag_offset    = 0x100,
03. 375     .smp            = smp_ops(vexpress_smp_ops),
04. 376     .map_io         = v2m_map_io,
05. 377     .init_early      = v2m_init_early,
06. 378     .init_irq        = v2m_init_irq,
07. 379     .timer          = &v2m_timer,
08. 380     .handle_irq      = gic_handle_irq,
09. 381     .init_machine    = v2m_init,
10. 382     .restart         = vexpress_restart,
11. 383 MACHINE_END
```

这些不同的machine会有不同的MACHINE ID, Uboot在启动Linux内核时会把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]
01. 489 static const char * const v2m_dt_match[] __initconst = {
02. 490     "arm,vexpress",
03. 491     "xen,xenvm",
04. 492     NULL,
05. 493 };
06. 495 DT_MACHINE_START(VEXPRESS_DT, "ARM-Versatile Express")
07. 496     .dt_compat      = v2m_dt_match,
08. 497     .smp            = smp_ops(vexpress_smp_ops),
09. 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
```

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]
01. 158 static char const *exynos5_dt_compat[] __initdata = {
02. 159     "samsung,exynos5250",
03. 160     "samsung,exynos5440",
04. 161     NULL
05. 162 };
06. 163
07. 177 DT_MACHINE_START(EXYNOS5_DT, "SAMSUNG EXYNOS5 (Flattened Device Tree)")
08. 178     /* Maintainer: Kukjin Kim <kgene.kim@samsung.com> */
```



```

09. 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

```

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]
01. 126 static void __init exynos5_dt_machine_init(void)
02. 127 {
03. 128     ...
04. 149
05. 150     if (of_machine_is_compatible("samsung,exynos5250"))
06. 151         of_platform_populate(NULL, of_default_bus_match_table,
07. 152                               exynos5250_auxdata_lookup, NULL);
08. 153     else if (of_machine_is_compatible("samsung,exynos5440"))
09. 154         of_platform_populate(NULL, of_default_bus_match_table,
10. 155                               exynos5440_auxdata_lookup, NULL);
11. 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]
01. 436 static const struct of_device_id a1234_i2c_of_match[] = {
02. 437     { .compatible = "acme,a1234-i2c-bus ", },
03. 438     {}},
04. 439 };
05. 440 MODULE_DEVICE_TABLE(of, a1234_i2c_of_match);
06. 441
07. 442 static struct platform_driver i2c_a1234_driver = {
08. 443     .driver = {
09. 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);

```

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]
01. 1533 static const struct of_device_id wm8753_of_match[] = {
02. 1534     { .compatible = "wlf,wm8753", },
03. 1535     {} }
04. 1536 };
05. 1537 MODULE_DEVICE_TABLE(of, wm8753_of_match);
06. 1587 static struct spi_driver wm8753_spi_driver = {
07. 1588     .driver = {
08. 1589         .name = "wm8753",
09. 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 };

```

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]
01. 90 static int spi_match_device(struct device *dev, struct device_driver *drv)
02. 91 {
03. 92     const struct spi_device *spi = to_spi_device(dev);
04. 93     const struct spi_driver *sdrv = to_spi_driver(drv);
05. 94
06. 95     /* Attempt an OF style match */
07. 96     if (of_driver_match_device(dev, drv))
08. 97         return 1;
09. 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 }

```

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]
01. 1682 if (of_device_is_compatible(np, "sirf,marco-pinctrl"))
02. 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]
01. 534      of_property_read_u32_array(np, "arm,data-latency",
02. 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]
01. 137      L2: cache-controller@1e00a000 {
02. 138          compatible = "arm,pl310-cache";
03. 139          reg = <0x1e00a000 0x1000>;
04. 140          interrupts = <0 43 4>;
05. 141          cache-level = <2>;
06. 142          arm,data-latency = <1 1 1>;
07. 143          arm,tag-latency = <1 1 1>;
08. 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]
01. 513 static inline int of_property_read_u8(const struct device_node *np,
02. 514      const char *propname,
03. 515      u8 *out_value)
04. 516 {
05. 517      return of_property_read_u8_array(np, propname, out_value, 1);
06. 518 }
07. 519
08. 520 static inline int of_property_read_u16(const struct device_node *np,
09. 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 }
```

```
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/clock/clock.c中的of_clk_get_parent_name()透过of_property_read_string_index()遍历clk_spec结点的所有"clock-output-names"字符串数组属性。

```
[cpp]
01. 1759 const char *of_clk_get_parent_name(struct device_node *np, int index)
02. 1760 {
03. 1761     struct of_phandle_args clk_spec;
04. 1762     const char *clk_name;
05. 1763     int rc;
06. 1764
07. 1765     if (index < 0)
08. 1766         return NULL;
09. 1767
10. 1768     rc = of_parse_phandle_with_args(np, "clocks", "#clock-cells", index,
11. 1769                                     &clk_spec);
12. 1770     if (rc)
13. 1771         return NULL;
14. 1772
15. 1773     if (of_property_read_string_index(clk_spec.np, "clock-output-names",
16. 1774                                     clk_spec.args_count ? clk_spec.args[0] : 0,
17. 1775                                     &clk_name) < 0)
18. 1776         clk_name = clk_spec.np->name;
19. 1777
20. 1778     of_node_put(clk_spec.np);
21. 1779     return clk_name;
22. 1780 }
23. 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内核的时候会先将.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。

```
[plain]
01.
```

上一篇 [炼狱与逐光——我的十杰博客获奖感言](#)
下一篇 [Linux芯片级移植与底层驱动（基于3.7.4内核）](#)

顶 27
踩 0

主题推荐 [linux](#) [arm](#) [arm架构](#) [linus torvalds](#) [数据结构](#)

猜你在找

- | | |
|--|---|
| linux驱动面试题汇总 | Linux中断处理流程 |
| Service与Android系统实现（1）— 应用程序里的 | Android如何完全调试framework层代码 |
| 见过最好的git入门教程 | TP调试 |
| 一个华为辞职创业后的几个反思 | vim显示行号、语法高亮、自动缩进的设置 |
| boot loader, Linux 内核, 文件系统之间的关系 | linux驱动学习之ioctl接口 |

查看评论

- 28楼 [andrew_sun](#) 2014-08-11 11:00发表
 期待宋老师的新书，驱动第三版，请问什么时候出版？
- 27楼 [tianfukeji](#) 2014-07-24 16:11发表
 太赞了，好清晰明了的文章！
- 26楼 [fantasyhujian](#) 2014-07-17 00:29发表
 楼主讲device tree讲的很好，正在学习过程中！
- 25楼 [hei-jude](#) 2014-07-09 13:05发表
 此前从事powerpc开发的时候对DTS一知半解，然后在MINI2440上对比才知道DTS的意义。MINI2440上添加硬件就需要修改内核代码，重新编译，而且各种遗留的代码很坑。
以前ARM上没采用DTS是因为uboot的支持不够好吗？据我所知，uboot对powerpc的支持很好
- 24楼 [confidence321](#) 2014-07-04 14:24发表
 经典，在学习中
- 23楼 [fafactx](#) 2014-05-27 09:45发表
 老大，你在网上搜开源力量第八课就是你讲的这个课件应该，好像被收费了，但是受益人好像是那个网站的站长，如果是你我就买了。。。现在在用P o w e r p c 结构处理器注册中断，用的3 . 8 . 1 1 内核。。。正在学习，多谢分享～～
- 22楼 [fafactx](#) 2014-05-20 11:29发表
 老大，有视频吗？你的视频被开源力量给收费了，建议你维护自己的权益。
Re: [21cnbao](#) 2014-05-20 16:35发表
 回复fafactx: 告诉我一下位置。
- 21楼 [ukyo111](#) 2014-04-09 11:10发表
 多谢宋老大!
最近也在看这一块.感觉引入dts也未见得方便多少.增加了一个中间解释层,driver中要读哪些参数还要调of接口,而之前是直接定义在.c里的,driver直接用就好.
以后平台的复杂度越高,要读的参数也就越多,driver中也会有越多了of类代码...
- 20楼 [yunzhi10](#) 2014-01-26 15:43发表



多谢宋老师的分享，非常受益。

"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:"

这两个兼容可能是笔误吧，其中一个应该是"sirf,marco-pinctrl"吧。

19楼 [to__to](#) 2014-01-04 16:59发表



楼主好，请问一个问题，如果我把驱动程序编译进了内核，内核是如何确定这些驱动是否被加载的。

18楼 [johhu](#) 2013-12-21 16:34发表



赞

17楼 [yhg20090519](#) 2013-12-02 16:14发表



非常好，有用！

16楼 [小驹德尔驾](#) 2013-10-23 11:52发表



大赞mark

15楼 [kangear](#) 2013-10-22 13:40发表



引用"kangear"的评论：

文章看到就会有用，最近在看CPU的Unique ID，一般的CPU，结果都是这个.dts传过去的。

老师应该再讲讲u-boot怎么传这个.dts.

14楼 [kangear](#) 2013-10-22 13:36发表



文章看到就会有用，最近在看CPU的Unique ID，一般的CPU，结果都是这个.dts传过去的。

13楼 [Legend_tboy](#) 2013-09-24 11:24发表



老师您好，我想请问在dts文件中，不是include可以包含其他的dts或者dtsi文件嘛，可是包含的dts里面又有/节点，可是一个dts中只有一个/节点啊，请问到底是按照哪个来？包含后按照C语言那样的话就是如下形式：

```
{
};
/ {
};
}
这样不是有两个/节点了吗？
```

Re: [taozuiqizhong](#) 2013-12-16 21:27发表



回复wo411186312：个人理解，希望不是误导。

/是根节点，那两个大括号就是根节点上的两个分支。

如果没有include，那么根节点有一个分支，这一个分支有一个或多个分支。

如果有多个include，就是根节点有多个分支，每个分支有一个或多个分支

Re: [Legend_tboy](#) 2014-05-27 10:01发表



回复taozuiqizhong：谢谢了，我再好好研究一下

12楼 [tiandaozhang](#) 2013-09-05 10:16发表



非常好，正在学习中！

11楼 [ermuzhi](#) 2013-07-09 17:04发表



宋老大，好文章啊。哈哈。看到linux有你提交的bug呢。

10楼 [android_sniper](#) 2013-07-02 13:49发表



写得很好，我有一个想请教，如果我在同一个ARM SOC上开发多个项目，每个项目有自己的电路板，他们的区别只是部分外设不一样。请问，我应该怎样建立自己的Device Tree来实现多个项目共用一套代码呢？感谢。

9楼 [乾龙_Heron](#) 2013-05-03 14:47发表

谢谢楼主，真心实用！



8楼 [T-MEC](#) 2013-04-22 22:54发表



学习下！

7楼 [luoqiaofa](#) 2013-03-26 19:18发表



设备树是N多年前就出现的了，确切地说就是 **openfirmware**标准，POWERPC架构一直以来都支持此驱动架构，代码比ARM的要简洁很多，非得要**Linus Torvalds**生气，ARM的的内含开发者们才改进，有点可悲，很多提供内核的驱动的人们基本都是COPY代码，而不是学习别人好的代码，然后采用，改进等，这就是ARM一直以来成为这样的原因。

6楼 [wuyuwei45](#) 2013-03-14 16:55发表



写得很好

5楼 [a00185766](#) 2013-02-16 17:36发表



赞！

4楼 [xumin330774233](#) 2013-02-06 09:02发表



sbh总是经典的代名词。

3楼 [little_paul](#) 2013-02-01 14:45发表



驱动发展的一个趋势，学习ing

2楼 [UStorage](#) 2013-01-26 17:18发表



赞，先顶后看！

1楼 [jixia松](#) 2013-01-22 11:11发表



这篇文章紧跟技术潮流，来得真是时候！我之前尝试搜索“ARM Device Tree”的相关内容，即使英文文档都比较少见，只能在Linux内核邮件列表中找到。

发表评论

用户名：[benhuan99](#)

评论内容：



提交

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

- 全部主题
- [Hadoop](#) [AWS](#) [移动游戏](#) [Java](#) [Android](#) [iOS](#) [Swift](#) [智能硬件](#) [Docker](#) [OpenStack](#) [VPN](#) [Spark](#) [ERP](#) [IE10](#) [Eclipse](#) [CRM](#) [JavaScript](#) [数据库](#) [Ubuntu](#) [NFC](#) [WAP](#) [jQuery](#) [BI](#) [HTML5](#) [Spring](#) [Apache](#) [.NET](#) [API](#) [HTML](#) [SDK](#) [IIS](#) [Fedora](#) [XML](#) [LBS](#) [Unity](#) [Splashtop](#) [UML](#) [components](#) [Windows Mobile](#) [Rails](#) [QEMU](#) [KDE](#) [Cassandra](#) [CloudStack](#) [FTC](#) [coremail](#) [OPhone](#) [CouchBase](#) [云计算](#) [iOS6](#) [Rackspace](#) [Web App](#) [SpringSide](#) [Maemo](#) [Compuware](#) [大数据](#) [apttech](#) [Perl](#) [Tornado](#) [Ruby](#) [Hibernate](#) [ThinkPHP](#) [HBase](#) [Pure](#) [Solr](#) [Angular](#) [Cloud Foundry](#) [Redis](#) [Scala](#) [Django](#) [Bootstrap](#)

网站客服 杂志客服 微博客服 webmaster@csdn.net 400-600-2320

京 ICP 证 070598 号

北京创新乐知信息技术有限公司 版权所有

江苏乐知网络技术有限公司 提供商务支持

Copyright © 1999-2014, CSDN.NET, All Rights Reserved 