

第 18 章

ARM Linux 设备树

本章导读

本章将介绍 Linux 设备树（Device Tree）的起源、结构和因为设备树而引起的驱动和 BSP 变更。

18.1 节阐明了 ARM Linux 为什么要采用设备树。

18.2 节详细剖析了设备树的结构、节点和属性，设备树的编译方法以及如何用设备树来描述板上的设备、设备的地址、设备的中断号、时钟等信息。

18.3 节讲解了采用设备树后，BSP 和驱动的代码需要怎么改，哪些地方变了。

18.4 节补充了一些与设备树相关的 API 定义以及用法。

本章内容是步入 Linux 3.x 时代后，嵌入式 Linux 工程师必备的知识体系。

18.1 ARM 设备树起源

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，很多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。读者若有兴趣，可以统计一下常见的 s3c2410、s3c6410 等板级目录，代码量在数万行。

设备树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 中，采用设备树后，许多硬件的细节可以直接通过它传递给 Linux，而不再需要在内核中进行大量的冗余编码。

设备树由一系列被命名的节点（Node）和属性（Property）组成，而节点本身可包含子节点。所谓属性，其实就是成对出现的名称和值。在设备树中，可描述的信息包括（原先这些信息大多被硬编码在内核中）：

- 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。
- 时钟控制器和时钟使用情况。

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。

18.2 设备树的组成和结构

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

18.2.1 DTS、DTC 和 DTB 等

1. DTS

文件 `.dts` 是一种 ASCII 文本格式的设备树描述，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 中，一个 `.dts` 文件对应一个 ARM 的设备，一般放置在内核的 `arch/arm/boot/dts/` 目录中。值得注意的是，在 `arch/powerpc/boot/dts`、`arch/powerpc/boot/dts`、`arch/c6x/boot/dts`、`arch/openrisc/boot/dts` 等目录中，也存在大量的 `.dts` 文件，这证明 DTS 绝对不是 ARM 的专利。

由于一个 SoC 可能对应多个设备（一个 SoC 可以对应多个产品和电路板），这些 `.dts` 文件势必须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设备共同的部分一般提炼为 `.dtsi`，类似于 C 语言的头文件。其他的设备对应的 `.dts` 就包括这个 `.dtsi`。譬如，对于 VEXPRESS 而言，`vexpress-v2m.dtsi` 就被 `vexpress-v2p-ca9.dts` 所引用，`vexpress-v2p-ca9.dts` 有如下一行代码：

```
/include/ "vexpress-v2m.dtsi"
```

当然，和 C 语言的头文件类似，`.dtsi` 也可以包括其他的 `.dtsi`，譬如几乎所有的 ARM SoC 的 `.dtsi` 都引用了 `skeleton.dtsi`。

文件 `.dts`（或者其包括的 `.dtsi`）的基本元素即为前文所述的节点和属性，代码清单 18.1 给出了一个设备树结构的模版。

代码清单 18.1 设备树结构模版

```

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 };

```

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

1 个 root 节点 "/"；root 节点下面含一系列子节点，本例中为 node1 和 node2；节点 node1 下又含有一系列子节点，本例中为 child-node1 和 child-node2；各节点都有一系列属性。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设备为例来看如何写一个 .dts 文件。如图 18.1 所示，假设此设备的配置如下：

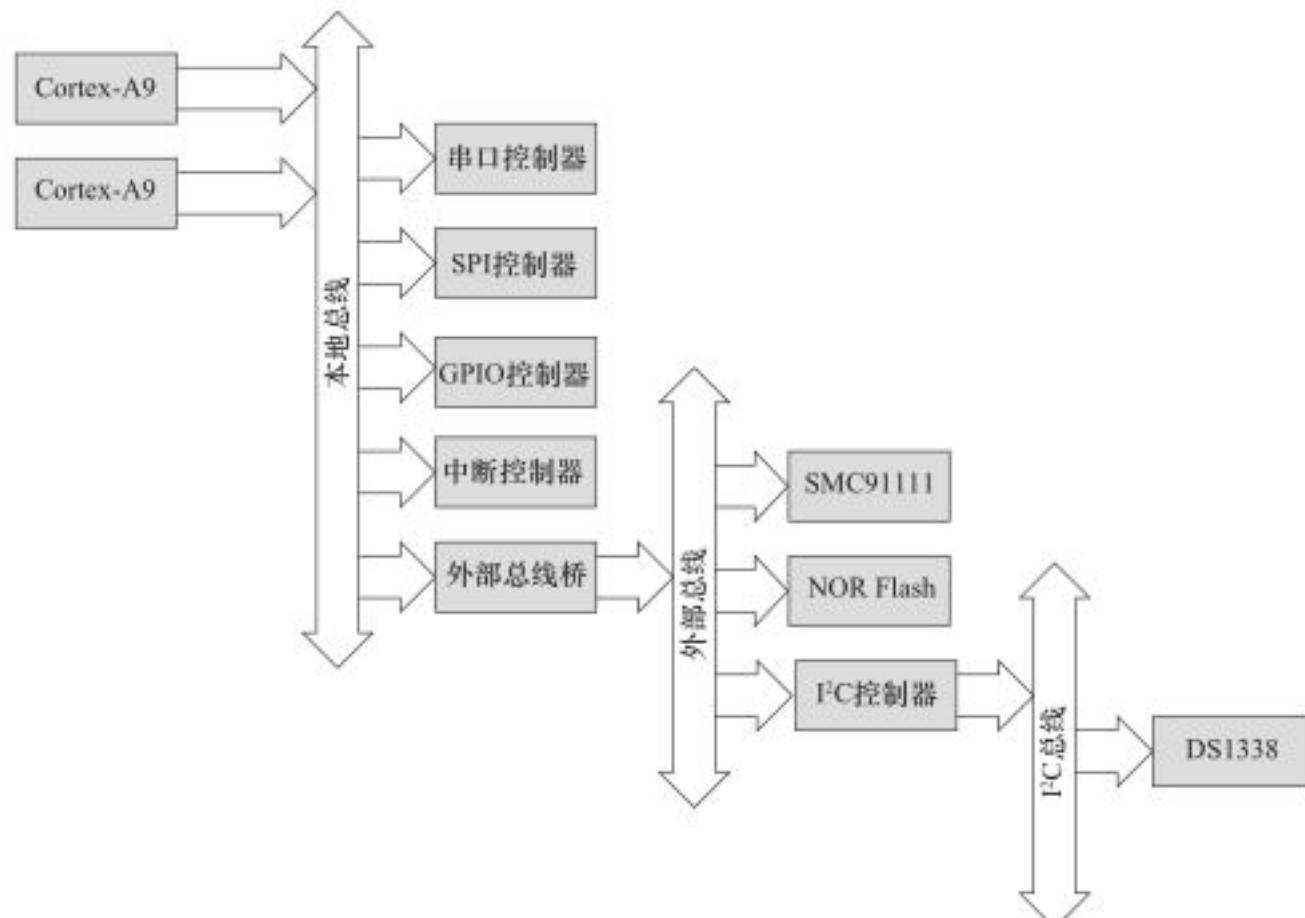


图 18.1 设备树参考硬件结构图

1个双核 ARM Cortex-A9 32位处理器；ARM本地总线上的内存映射区域分布有两个串口（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）和一个外部总线桥；外部总线桥上又连接了SMC SMC9111以太网（位于0x10100000）、I²C控制器（位于0x10160000）、64MB NOR Flash（位于0x30000000）；外部总线桥上连接的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）。

对于图18.1所示硬件结构图，如果用“.dts”描述，则其对应的“.dts”文件如代码清单18.2所示。

代码清单18.2 参考硬件的设备树文件

```

1  / {
2      compatible = "acme,coyotes-revenge";
3      #address-cells = <1>;
4      #size-cells = <1>;
5      interrupt-parent = <&intc>;
6
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     };
19
20     serial@101f0000 {
21         compatible = "arm,p1011";
22         reg = <0x101f0000 0x1000 >;
23         interrupts = < 1 0 >;
24     };
25
26     serial@101f2000 {
27         compatible = "arm,p1011";
28         reg = <0x101f2000 0x1000 >;
29         interrupts = < 2 0 >;
30     };
31
32     gpio@101f3000 {
33         compatible = "arm,p1061";
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,k8fl315ebm", "cfi-flash";
80             reg = <2 0 0x4000000>;
81         };
82     };
83 }

```

在上述.dts文件中，可以看出external-bus是根节点的子节点，而I²C又是external-bus的子节点，RTC又进一步是I²C的子节点。每一级节点都有一些属性信息，本章后续部分会进行详细解释。

2. DTC(Device Tree Compiler)

DTC 是将 .dts 编译为 .dtb 的工具。DTC 的源代码位于内核的 scripts/dtc 目录中，在 Linux 内核使能了设备树的情况下，编译内核的时候主机工具 DTC 会被编译出来，对应于 scripts/dtc/Makefile 中 “hostprogs-y := dtc” 这一 hostprogs 的编译目标。

当然，DTC 也可以在 Ubuntu 中单独安装，命令如下：

```
sudo apt-get install device-tree-compiler
```

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：

```
dtb-$(CONFIG_ARCH_VEXPRESS) += vexpress-v2p-ca5s.dtb \
vexpress-v2p-ca9.dtb \
vexpress-v2p-ca15-tcl.dtb \
vexpress-v2p-ca15_a7.dtb \
xenvm-4.2.dtb
```

在 Linux 下，我们可以单独编译设备树文件。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 编译目标项目。

DTC 除了可以编译 .dts 文件以外，其实也可以“反汇编” .dtb 文件为 .dts 文件，其指令格式为：

```
./scripts/dtc/dtc -I dtb -O dts -o xxx.dts arch/arm/boot/dts/xxx.dtb
```

3. DTB(Device Tree Blob)

文件 .dtb 是 .dts 被 DTC 编译后的二进制格式的设备树描述，可由 Linux 内核解析，当然 U-Boot 这样的 bootloader 也是可以识别 .dtb 的。

通常在我们为电路板制作 NAND、SD 启动映像时，会为 .dtb 文件单独留下一个很小的区域以存放之，之后 bootloader 在引导内核的过程中，会先读取该 .dtb 到内存。

Linux 内核也支持一种变通的模式，可以不把 .dtb 文件单独存放，而是直接和 zImage 绑定在一起做成一个映像文件，类似 cat zImage xxx.dtb > zImage_with_dtb 的效果。当然内核编译时候要使能 CONFIG_ARM_APPENDED_DTB 这个选项，以支持“Use appended device tree blob to zImage”（见 Linux 内核中的菜单）。

4. 绑定 (Binding)

对于设备树中的节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。在这个 .txt 文件中，需要描述对应节点的兼容性、必需的属性和可选的属性。

这些文档位于内核的 Documentation/devicetree/bindings 目录下，其下又分为很多子目录。譬如，Documentation/devicetree/bindings/i2c/i2c-xiic.txt 描述了 Xilinx 的 I²C 控制器，其内容如下：

```

Xilinx IIC controller:

Required properties:
- compatible : Must be "xlnx,xps-iic-2.00.a"
- reg : IIC register location and length
- interrupts : IIC controller interrupt
- #address-cells = <1>
- #size-cells = <0>

Optional properties:
- Child nodes conforming to i2c bus binding

Example:

    axi_iic_0: i2c@40800000 {
        compatible = "xlnx,xps-iic-2.00.a";
        interrupts = < 1 2 >;
        reg = < 0x40800000 0x10000 >

        #size-cells = <0>;
        #address-cells = <1>;
    };

```

基本可以看出，设备树绑定文档的主要内容包括：

- 关于该模块最基本的描述。
- 必需属性（Required Properties）的描述。
- 可选属性（Optional Properties）的描述。
- 一个实例。

Linux 内核下的 scripts/checkpatch.pl 会运行一个检查，如果有人在设备树中新添加了 compatible 字符串，而没有添加相应的文档进行解释，checkpatch 程序会报出警告：UNDOCUMENTED_DT_STRINGDT compatible string xxx appears un-documented，因此程序员要养成及时写 DT Binding 文档的习惯。

5. Bootloader

Uboot 设备从 v1.1.3 开始支持设备树，其对 ARM 的支持则是和 ARM 内核支持设备树同期完成。

为了使能设备树，需要在编译 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 等任意介质中将 .dtb 读入内存，假设 .dtb 放入的内存地址为 0x71000000，之后可在 Uboot 中运行 fdt addr 命令设置 .dtb 的地址，如：

```
UBoot> fdt addr 0x71000000
```

fdt 的其他命令就变得可以使用，如 fdt resize、fdt print 等。

对于 ARM 来讲，可以通过 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“-”符号代替。

18.2.2 根节点兼容性

上述 .dts 文件中，第 2 行根节点 “/” 的兼容属性 compatible = "acme,coyotes-revenge"；定义了整个系统（设备级别）的名称，它的组织形式为：<manufacturer>,<model>。

Linux 内核通过根节点 “/” 的兼容属性即可判断它启动的是什么设备。在真实项目中，这个顶层设备的兼容属性一般包括两个或者两个以上的兼容性字符串，首个兼容性字符串是板子级别的名字，后面一个兼容性是芯片级别（或者芯片系列级别）的名字。

譬如板子 arch/arm/boot/dts/vexpress-v2p-ca9.dts 兼容于 arm,vexpress,v2p-ca9 和 “arm,vexpress”：

```
compatible = "arm,vexpress,v2p-ca9", "arm,vexpress";
```

板子 arch/arm/boot/dts/vexpress-v2p-ca5s.dts 的兼容性则为：

```
compatible = "arm,vexpress,v2p-ca5s", "arm,vexpress";
```

板子 arch/arm/boot/dts/vexpress-v2p-ca15_a7.dts 的兼容性为：

```
compatible = "arm,vexpress,v2p-ca15_a7", "arm,vexpress";
```

可以看出，上述各个电路板的共性是兼容于 arm,vexpress，而特性是分别兼容于 arm,vexpress,v2p-ca9、arm,vexpress,v2p-ca5s 和 arm,vexpress,v2p-ca15_a7。

进一步地看，arch/arm/boot/dts/exynos4210-origen.dts 的兼容性字段如下：

```
compatible = "insignal,origen", "samsung,exynos4210", "samsung,exynos4";
```

第一个字符串是板子名字（很特定），第 2 个字符串是芯片名字（比较特定），第 3 个字段是芯片系列的名字（比较通用）。

作为类比，arch/arm/boot/dts/exynos4210-universal_c210.dts 的兼容性字段则如下：

```
compatible = "samsung,universal_c210", "samsung,exynos4210", "samsung,exynos4";
```

由此可见，它与 exynos4210-origen.dts 的区别只在于第 1 个字符串（特定的板子名字）不一样，后面芯片名和芯片系列的名字都一样。

在 Linux 2.6 内核中，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设备的一系列回调函数，如代码清单 18.3 所示。

代码清单 18.3 ARM Linux 2.6 时代的设备

```
1 MACHINE_START(VEXPRESS, "ARM-Versatile Express")
2     .atag_offset    = 0x100,
3     .smp           = smp_ops(vexpress_smp_ops),
```

```

4     .map_io      = v2m_map_io,
5     .init_early   = v2m_init_early,
6     .init_irq     = v2m_init_irq,
7     .timer        = &v2m_timer,
8     .handle_irq   = gic_handle_irq,
9     .init_machine = v2m_init,
10    .restart      = vexpress_restart,
11 MACHINE_END

```

这些不同的设备会有不同的 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，然后执行相应设备的一系列初始化函数。

ARM Linux 3.x 在引入设备树之后，MACHINE_START 变更为 DT_MACHINE_START，其中含有一个 .dt_compatible 成员，用于表明相关的设备与 .dts 中根节点的兼容属性兼容关系。如果 Bootloader 传递给内核的设备树中根节点的兼容属性出现在某设备的 .dt_compatible 表中，相关的设备就与对应的兼容匹配，从而引发这一设备的一系列初始化函数被执行。一个典型的 DT_MACHINE 如代码清单 18.4 所示。

代码清单 18.4 ARM Linux 3.x 时代的设备

```

1 static const char * const v2m_dt_match[] __initconst = {
2     "arm,vexpress",
3     "xen,xenvm",
4     NULL,
5 };
6 DT_MACHINE_START(VEXPRESS_DT, "ARM-Versatile Express")
7     .dt_compatible     = v2m_dt_match,
8     .smp               = smp_ops(vexpress_smp_ops),
9     .map_io            = v2m_dt_map_io,
10    .init_early         = v2m_dt_init_early,
11    .init_irq           = v2m_dt_init_irq,
12    .timer              = &v2m_dt_timer,
13    .init_machine       = v2m_dt_init,
14    .handle_irq          = gic_handle_irq,
15    .restart             = vexpress_restart,
16 MACHINE_END

```

Linux 倡导针对多个 SoC、多个电路板的通用 DT 设备，即一个 DT 设备的 .dt_compatible 包含多个电路板 .dts 文件的根节点兼容属性字符串。之后，如果这多个电路板的初始化序列不一样，可以通过 int of_machine_is_compatible(const char *compat) API 判断具体的电路板是什么。在 Linux 内核中，常常使用如下 API 来判断根节点的兼容性：

```
int of_machine_is_compatible(const char *compat);
```

此 API 判断目前运行的板子或者 SoC 的兼容性，它匹配的是设备树根节点下的兼容

属性。例如 drivers/cpufreq/exynos-cpufreq.c 中就有判断运行的 CPU 类型是 exynos4210、exynos4212、exynos4412 还是 exynos5250 的代码，进而分别处理，如代码清单 18.5 所示。

代码清单 18.5 of_machine_is_compatible() 的案例

```

1 static int exynos_cpufreq_probe(struct platform_device *pdev)
2 {
3     int ret = -EINVAL;
4
5     exynos_info = kzalloc(sizeof(*exynos_info), GFP_KERNEL);
6     if (!exynos_info)
7         return -ENOMEM;
8
9     exynos_info->dev = &pdev->dev;
10
11    if (of_machine_is_compatible("samsung,exynos4210")) {
12        exynos_info->type = EXYNOS_SOC_4210;
13        ret = exynos4210_cpufreq_init(exynos_info);
14    } else if (of_machine_is_compatible("samsung,exynos4212")) {
15        exynos_info->type = EXYNOS_SOC_4212;
16        ret = exynos4x12_cpufreq_init(exynos_info);
17    } else if (of_machine_is_compatible("samsung,exynos4412")) {
18        exynos_info->type = EXYNOS_SOC_4412;
19        ret = exynos4x12_cpufreq_init(exynos_info);
20    } else if (of_machine_is_compatible("samsung,exynos5250")) {
21        exynos_info->type = EXYNOS_SOC_5250;
22        ret = exynos5250_cpufreq_init(exynos_info);
23    } else {
24        pr_err("%s: Unknown SoC type\n", __func__);
25        return -ENODEV;
26    }
27    ...
28 }

```

如果一个兼容包含多个字符串，譬如对于前面介绍的根节点兼容 compatible = "samsung,universal_c210", "samsung,exynos4210", "samsung,exynos4" 的情况，如下 3 个表达式都是成立的。

```

of_machine_is_compatible("samsung,universal_c210")
of_machine_is_compatible("samsung,exynos4210")
of_machine_is_compatible("samsung,exynos4")

```

18.2.3 设备节点兼容性

在 .dts 文件的每个设备节点中，都有一个兼容属性，兼容属性用于驱动和设备的绑定。兼容属性是一个字符串的列表，列表中的第一个字符串表征了节点代表的确切设备，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 vexpress-v2m.dtsi 中的 Flash 节点如下：

```

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

```

兼容属性的第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 串口设备的兼容属性为 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 NS16550 UART 保持了寄存器兼容。因此，设备节点的兼容性和根节点的兼容性是类似的，都是“从具体到抽象”。

使用设备树后，驱动需要与 .dts 中描述的设备节点进行匹配，从而使驱动的 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 控制器节点的 OF 匹配表，具体代码清单 18.6 所示。

代码清单 18.6 platform 设备驱动中的 of_match_table

```

1 static const struct of_device_id a1234_i2c_of_match[] = {
2     { .compatible = "acme,a1234-i2c-bus", },
3     {}
4 };
5 MODULE_DEVICE_TABLE(of, a1234_i2c_of_match);
6
7 static struct platform_driver i2c_a1234_driver = {
8     .driver = {
9         .name = "a1234-i2c-bus",
10        .owner = THIS_MODULE,
11        .of_match_table = a1234_i2c_of_match,
12    },
13        .probe = i2c_a1234_probe,
14        .remove = i2c_a1234_remove,
15    };
16 module_platform_driver(i2c_a1234_driver);

```

对于 I²C 和 SPI 从设备而言，同样也可以通过 of_match_table 添加匹配的 .dts 中的相关节点的兼容属性，如 sound/soc/codecs/wm8753.c 中的针对 WolfsonWM8753 的 of_match_table，具体如代码清单 18.7 所示。

代码清单 18.7 I²C、SPI 设备驱动中的 of_match_table

```

1 static const struct of_device_id wm8753_of_match[] = {
2     { .compatible = "wlf,wm8753", },
3     {}

```

```

4  };
5 MODULE_DEVICE_TABLE(of, wm8753_of_match);
6 static struct spi_driver wm8753_spi_driver = {
7     .driver = {
8         .name    = "wm8753",
9         .owner   = THIS_MODULE,
10        .of_match_table = wm8753_of_match,
11    },
12    .probe      = wm8753_spi_probe,
13    .remove     = wm8753_spi_remove,
14 };
15 static struct i2c_driver wm8753_i2c_driver = {
16     .driver = {
17         .name = "wm8753",
18         .owner = THIS_MODULE,
19         .of_match_table = wm8753_of_match,
20     },
21     .probe =   wm8753_i2c_probe,
22     .remove =  wm8753_i2c_remove,
23     .id_table = wm8753_i2c_id,
24 };

```

上述代码中的第2行显示WM8753的供应商是“wlf”，它其实是对应于Wolfson Microelectronics的前缀。详细的前缀可见于内核文档：Documentation/devicetree/bindings/vendor-prefixes.txt

对于I²C、SPI还有一点需要提醒的是，I²C和SPI外设驱动和设备树中设备节点的兼容属性还有一种弱式匹配方法，就是“别名”匹配。兼容属性的组织形式为<manufacturer>,<model>，别名其实就是去掉兼容属性中 manufacturer 前缀后的 <model> 部分。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，代码清单 18.8 显示了 SPI 的别名匹配。

代码清单 18.8 SPI 的别名匹配

```

1 static int spi_match_device(struct device *dev, struct device_driver *drv)
2 {
3     const struct spi_device *spi = to_spi_device(dev);
4     const struct spi_driver *sdrv = to_spi_driver(drv);
5
6     /* Attempt an OF style match */
7     if (of_driver_match_device(dev, drv))
8         return 1;
9
10    /* Then try ACPI */
11    if (acpi_driver_match_device(dev, drv))
12        return 1;
13

```



```

        ) else if (of_device_is_compatible(immr_node, "fsl,mpc8315-immr")) {
            clrsetbits_be32(immap + MPC83XX_SICRL_OFFSET,
                            MPC8315_SICRL_USB_MASK,
                            MPC8315_SICRL_USB_ULPI);
            clrsetbits_be32(immap + MPC83XX_SICRH_OFFSET,
                            MPC8315_SICRH_USB_MASK,
                            MPC8315_SICRH_USB_ULPI);
        } else {
            clrsetbits_be32(immap + MPC83XX_SICRL_OFFSET,
                            MPC831X_SICRL_USB_MASK,
                            MPC831X_SICRL_USB_ULPI);
            clrsetbits_be32(immap + MPC83XX_SICRH_OFFSET,
                            MPC831X_SICRH_USB_MASK,
                            MPC831X_SICRH_USB_ULPI);
        }
    }
}

```

它根据具体的设备是 fsl,mpc8315-immr 和 fsl,mpc8308-immr、中的哪一种来进行不同的处理。

当一个驱动可以兼容多种设备的时候，除了 of_device_is_compatible() 这种判断方法以外，还可以采用在驱动的 of_device_id 表中填充 .data 成员的形式。譬如，arch/arm/mm/cache-l2x0.c 支持 “arm,l210-cache” “arm,pl310-cache” “arm,l220-cache” 等多种设备，其 of_device_id 表如代码清单 18.9 所示。

代码清单 18.9 支持多个兼容性以及 .data 成员的 of_device_id 表

```

1 #define L2C_ID(name, fns) { .compatible = name, .data = (void *)&fns }
2 static const struct of_device_id l2x0_ids[] __initconst = {
3     L2C_ID("arm,l210-cache", of_l2c210_data),
4     L2C_ID("arm,l220-cache", of_l2c220_data),
5     L2C_ID("arm,pl310-cache", of_l2c310_data),
6     L2C_ID("brcm,bcm11351-a2-pl310-cache", of_bcm_l2x0_data),
7     L2C_ID("marvell,aurora-outer-cache", of_aurora_with_outer_data),
8     L2C_ID("marvell,aurora-system-cache", of_aurora_no_outer_data),
9     L2C_ID("marvell,tauros3-cache", of_tauros3_data),
10    /* Deprecated IDs */
11    L2C_ID("bcm,bcm11351-a2-pl310-cache", of_bcm_l2x0_data),
12 };
13 };

```

在驱动中，通过如代码清单 18.10 的方法拿到了对应于 L2 缓存类型的 .data 成员，其中主要用到了 of_match_node() 这个 API。

代码清单 18.10 通过 of_match_node() 找到 .data

```

1 int __init l2x0_of_init(u32 aux_val, u32 aux_mask)
2 {
3     const struct l2c_init_data *data;

```

```

4         struct device_node *np;
5
6         np = of_find_matching_node(NULL, l2x0_ids);
7         if (!np)
8             return -ENODEV;
9         ...
10        data = of_match_node(l2x0_ids, np)->data;
11    }

```

如果电路板的.dts文件中L2缓存是arm,pl310-cache，那么上述代码第10行找到的数据就是of_l2c310_data，它是l2c_init_data结构体的一个实例。l2c_init_data是一个由L2缓存驱动自定义的数据结构，在其定义中既可以保护数据成员，又可以包含函数指针，如代码清单18.11所示。

代码清单18.11 与兼容对应的特定data实例

```

1 struct l2c_init_data {
2     const char *type;
3     unsigned way_size_0;
4     unsigned num_lock;
5     void (*of_parse)(const struct device_node *, u32 *, u32 *);
6     void (*enable)(void __iomem *, u32, unsigned);
7     void (*fixup)(void __iomem *, u32, struct outer_cache_fns *);
8     void (*save)(void __iomem *);
9     struct outer_cache_fns outer_cache;
10    };

```

通过这种方法，驱动可以把与某个设备兼容的私有数据寻找出来，如此体现了一种面向对象的设计思想，避免了大量的if, else或者switch, case语句。

18.2.4 设备节点及label的命名

代码清单18.2的.dts文件中，根节点“/”的cpus子节点下面又包含两个cpu子节点，描述了此设备上的两个CPU，并且两者的兼容属性为：“arm,cortex-a9”。

注意cpus和cpus的两个cpu子节点的命名，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，用于描述节点对应的设备类型，如3com Ethernet适配器对应的节点name宜为ethernet，而不是3com509。如果一个节点描述的设备有地址，则应该给出@unit-address。多个相同类型的设备节点的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这样的同名节点。设备的unit-address地址也经常在其对应节点的reg属性中给出。

对于挂在内存空间的设备而言，@字符后跟的一般就是该设备在内存空间的地址，譬如arch/arm/boot/dts/exynos4210.dtsi中存在的：

```

sysram@02020000 {
    compatible = "mmio-sram";
    reg = <0x02020000 0x20000>;
    ...
}

```

上述节点的 reg 属性的开始位置与 @ 后面的地址一样。

对于挂在 I²C 总线上的外设而言，@ 后面一般跟的是从设备的 I²C 地址，譬如 arch/arm/boot/dts/exynos4210-trats.dts 中的 mmsl14-touchscreen：

```

i2c@13890000 {
    ...
    mmsl14-touchscreen@48 {
        compatible = "melfas,mmsl14";
        reg = <0x48>;
        ...
    };
}

```

上述节点的 reg 属性标示的 I²C 从地址与 @ 后面的地址一样。

具体的节点命名规范可见 ePAPR (embedded Power Architecture Platform Reference) 标准，在 <https://www.power.org> 中可下载该标准。

我们还可以给一个设备节点添加 label，之后可以通过 &label 的形式访问这个 label，这种引用是通过 phandle (pointer handle) 进行的。

例如，在 arch/arm/boot/dts/omap5.dtsi 中，第 3 组 GPIO 有 gpio3 这个 label，如代码清单 18.12 所示。

代码清单 18.12 在设备树中定义 label

```

1 gpio3: gpio@48057000 {
2     compatible = "ti,omap4-gpio";
3     reg = <0x48057000 0x200>;
4     interrupts = <GIC_SPI 31 IRQ_TYPE_LEVEL_HIGH>;
5     ti,hwmods = "gpio3";
6     gpio-controller;
7     #gpio-cells = <2>;
8     interrupt-controller;
9     #interrupt-cells = <2>;
10 };

```

而 hsub2_phy 这个 USB 的 PHY 复位 GPIO 用的是这组 GPIO 中的一个，所以它通过 phandle 引用了 “gpio3”，如代码清单 18.13 所示。

代码清单 18.13 通过 phandle 引用其他节点

```

1 /* HS USB Host PHY on PORT 2 */
2 hsub2_phy: hsub2_phy [

```

```

3         compatible = "usb-nop-xceiv";
4         reset-gpios = <&gpio3 12 GPIO_ACTIVE_LOW>; /* gpio3_76 HUB_RESET */
5 };

```

代码清单 18.12 第 1 行的 gpio3 是 gpio@48057000 节点的 label，而代码清单 18.13 的 hsusb2_phy 则通过 &gpio3 引用了这个节点，表明自己要使用这一组 GPIO 中的第 12 个 GPIO。很显然，这种 phandle 引用其实表明硬件之间的一种关联性。

再举一例，在 arch/arm/boot/dts/omap5.dtsi 中，我们可以看到类似如下的 label，从这些实例可以看出，label 习惯以 <设备类型><index> 进行命名：

```

i2c1: i2c@48070000 {
}
i2c2: i2c@48072000 {
}
i2c3: i2c@48060000 {
}
...

```

读者也许发现了一个奇怪的现象，就是代码清单 18.13 中居然引用了 GPIO_ACTIVE_LOW 这个类似 C 语言的宏。文件 .dts 的编译过程确实支持 C 的预处理，相应的 .dts 文件也包括了包含 GPIO_ACTIVE_LOW 这个宏定义的头文件：

```
#include <dt-bindings/gpio/gpio.h>
```

对于 ARM 而言，dt-bindings 头文件位于内核的 arch/arm/boot/dts/include/dt-bindings 目录中。观察该目录的属性，它实际上是一个符号链接：

```

baohua@baohua-VirtualBox:~/develop/linux/arch/arm/boot/dts/include$ ls -l dt-
    bindings
lrwxrwxrwx 1 baohua baohua 34 11月 28 00:16 dt-bindings -> ../../../../../../
    include/dt-bindings

```

从内核的 scripts/Makefile.lib 这个文件可以看出，文件 .dts 的编译过程确实是支持 C 预处理的。

```

cmd_dtc = $(CPP) $(dtc_cpp_flags) -x assembler-with-cpp -o $(dtc-tmp) $< ; \
    $(objtree)/scripts/dtc/dtc -O dtb -o $@ -b 0 \
        -i $(dir $<) $(DTC_FLAGS) \
        -d $(depfile).dtc.tmp $(dtc-tmp) ; \
    cat $(depfile).pre.tmp $(depfile).dtc.tmp > $(depfile)

```

它是先做了 \$(CPP) \$(dtc_cpp_flags) -x assembler-with-cpp -o \$(dtc-tmp) \$<，再做的 .dtc 编译。

18.2.5 地址编码

可寻址的设备使用如下信息在设备树中编码地址信息：

```
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` 的意义则意味着从 `address` 到 `address+length-1` 的地址范围都属于该节点。若 `#size-cells = 0`，则 `length` 字段为空。

`address` 和 `length` 字段是可变长的，父节点的 `#address-cells` 和 `#size-cells` 分别决定了子节点 `reg` 属性的 `address` 和 `length` 字段的长度。

在代码清单 18.2 中，根节点的 `#address-cells = <1>`；和 `#size-cells = <1>`；决定了 `serial`、`gpio`、`spi` 等节点的 `address` 和 `length` 字段的长度分别为 1。

`cpus` 节点的 `#address-cells = <1>`；和 `#size-cells = <0>`；决定了两个 `cpu` 子节点的 `address` 为 1，而 `length` 为空，于是形成了两个 `cpu` 的 `reg = <0>`；和 `reg = <1>`；。

`external-bus` 节点的 `#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 2，开始的第一个 `cell`（即“<”后的 0、1、2）是对应的片选，第 2 个 `cell`（即 `<0 0 0x1000>`、`<1 0 0x1000>` 和 `<2 0 0x4000000>` 中间的 0、0、0）是相对该片选的基址，第 3 个 `cell`（即“>”前的 0x1000、0x1000、0x4000000）为 `length`。

特别要留意的是 `i2c` 节点中定义的 `#address-cells = <1>`；和 `#size-cells = <0>`；，其作用到了 I²C 总线上连接的 RTC，它的 `address` 字段为 0x58，是 RTC 设备的 I²C 地址。

根节点的直接子节点描述的是 CPU 的视图，因此根子节点的 `address` 区域就直接位于 CPU 的内存区域。但是，经过总线桥后的 `address` 往往需要经过转换才能对应 CPU 的内存映射。`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 的内存区域。

```
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` 桥后 `external-bus` 上片选 0 偏移 0，第 3 个 `cell` 表示 `external-bus` 上片选 0 偏移 0 的地址空间被映射到 CPU 的本地总线的 `0x10100000` 位置，第 4 个 `cell` 表示映射的大小为 `0x10000`。`ranges` 后面两个项目的含义可以类推。

18.2.6 中断连接

设备树中还可以包含中断连接信息，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中断属性的 cell 大小。

在整个设备树中，与中断相关的属性还包括：

`interrupt-parent`- 设备节点通过它来指定它所依附的中断控制器的 phandle，当节点没有指定 `interrupt-parent` 时，则从父级节点继承。对于本例（代码清单 18.2）而言，根节点指定了 `interrupt-parent = <&intc>;`，其对应于 `intc: interrupt-controller@10140000`，而根节点的子节点并未指定 `interrupt-parent`，因此它们都继承了 `intc`，即位于 `0x10140000` 的中断控制器中。

`interrupts`- 用到了中断的设备节点，通过它指定中断号、触发方法等，这个属性具体含有多少个 cell，由它依附的中断控制器节点的 `#interrupt-cells` 属性决定。而每个 cell 具体又是什么含义，一般由驱动的实现决定，而且也会在设备树的绑定文档中说明。譬如，对于 ARM GIC 中断控制器而言，`#interrupt-cells` 为 3，3 个 cell 的具体含义在 `Documentation/devicetree/bindings/arm/gic.txt` 中就有如下文字说明：

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

The 2nd cell contains the interrupt number for the interrupt type.
SPI interrupts are in the range [0-987]. PPI interrupts are in the range [0-15].

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

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

1 = low-to-high edge triggered

2 = high-to-low edge triggered

4 = active high level-sensitive

8 = active low level-sensitive

`bits[15:8]` PPI interrupt cpu mask. Each bit corresponds to each of the 8 possible cpus attached to the GIC. A bit set to '1' indicated 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 号、169 号两个中断，而且都是高电平触发，则该设备节点的中断属性可定义为 `interrupts = <0 168 4>,<0 169 4>;`。

对于平台设备而言，简单的通过如下 API 就可以指定想取哪一个中断，其中的参数 `num` 就是中断的 `index`。

```
int platform_get_irq(struct platform_device *dev, unsigned int num);
```

当然在.dts文件中可以对中断进行命名，而后在驱动中通过platform_get_irq_byname()来获取对应的中断号。譬如代码清单18.14演示了在drivers/dma/fsl-edma.c中通过platform_get_irq_byname()获取IRQ，以及arch/arm/boot/dts/vf610.dtsi与fsl-edma驱动对应节点的中断描述。

代码清单18.14 设备树中的中断名称以及驱动获取中断

```

1 static int
2 fsl_edma_irq_init(struct platform_device *pdev, struct fsl_edma_engine *fsl_edma)
3 {
4     fsl_edma->txirq = platform_get_irq_byname(pdev, "edma-tx");
5     fsl_edma->errirq = platform_get_irq_byname(pdev, "edma-err");
6 }
7
8 edma0: dma-controller@40018000 {
9     #dma-cells = <2>;
10    compatible = "fsl,vf610-edma";
11    reg = <0x40018000 0x2000>;
12        <0x40024000 0x1000>;
13        <0x40025000 0x1000>;
14    interrupts = <0 8 IRQ_TYPE_LEVEL_HIGH>;
15        <0 9 IRQ_TYPE_LEVEL_HIGH>;
16    interrupt-names = "edma-tx", "edma-err";
17    dma-channels = <32>;
18    clock-names = "dmamux0", "dmamux1";
19    clocks = <&clks VF610_CLK_DMAMUX0>;
20        <&clks VF610_CLK_DMAMUX1>;
21 };

```

第4行、第5行的platform_get_irq_byname()的第2个参数与.dts中的interrupt-names是一致的。

18.2.7 GPIO、时钟、pinmux连接

除了中断以外，在ARM Linux中时钟、GPIO、pinmux都可以通过.dts中的节点和属性进行描述。

1. GPIO

譬如，对于GPIO控制器而言，其对应的设备节点需声明gpio-controller属性，并设置#gpio-cells的大小。譬如，对于兼容性为fsl,imx28-pinctrl的pinctrl驱动而言，其GPIO控制器的设备节点类似于：

```

pinctrl@80018000 {
    compatible = "fsl,imx28-pinctrl", "simple-bus";
    reg = <0x80018000 2000>;
    gpio0: gpio@0 {

```

```

compatible = "fsl,imx28-gpio";
interrupts = <127>;
gpio-controller;
    #gpio-cells = <2>;
interrupt-controller;
    #interrupt-cells = <2>;
};

gpiol: gpio@1 {
compatible = "fsl,imx28-gpio";
interrupts = <126>;
gpio-controller;
    #gpio-cells = <2>;
interrupt-controller;
    #interrupt-cells = <2>;
};
...
};

```

其中，#gpio-cells 为 2，第 1 个 cell 为 GPIO 号，第 2 个为 GPIO 的极性。为 0 的时候是高电平有效，为 1 的时候则是低电平有效。

使用 GPIO 的设备则通过定义命名 xxx-gpios 属性来引用 GPIO 控制器的设备节点，如：

```

sdhci@c8000400 {
status = "okay";
cd-gpios = <&gpio01 0>;
wp-gpios = <&gpio02 0>;
power-gpios = <&gpio03 0>;
bus-width = <4>;
};

```

而具体的设备驱动则通过类似如下的方法来获取 GPIO：

```

cd_gpio = of_get_named_gpio(np, "cd-gpios", 0);
wp_gpio = of_get_named_gpio(np, "wp-gpios", 0);
power_gpio = of_get_named_gpio(np, "power-gpios", 0);

```

of_get_named_gpio() 这个 API 的原型如下：

```

static inline int of_get_named_gpio(struct device_node *np,
const char *propname, int index);

```

在 .dts 和设备驱动不关心 GPIO 名字的情况下，也可以直接通过 of_get_gpio() 获取 GPIO，此函数原型为：

```

static inline int of_get_gpio(struct device_node *np, int index);

```

如对于 compatible = "gpio-control-nand" 的基于 GPIO 的 NAND 控制器而言，在 .dts 中会定义多个 gpio 属性：

```

gpio-nand@1,0 {
    compatible = "gpio-control-nand";
    reg = <1 0x0000 0x2>;
        #address-cells = <1>;
        #size-cells = <1>;
    gpios = <&banka 1 0      /* rdy */>
            <&banka 2 0      /* nce */>
            <&banka 3 0      /* ale */>
            <&banka 4 0      /* cle */>
            0              /* nwp */>;
}

partition@0 {
    ...
};

};


```

在相应的驱动代码 drivers/mtd/nand/gpio.c 中是这样获取这些 GPIO 的：

```

plat->gpio_rdy = of_get_gpio(dev->of_node, 0);
plat->gpio_nce = of_get_gpio(dev->of_node, 1);
plat->gpio_ale = of_get_gpio(dev->of_node, 2);
plat->gpio_cle = of_get_gpio(dev->of_node, 3);
plat->gpio_nwp = of_get_gpio(dev->of_node, 4);


```

2. 时钟

时钟和 GPIO 也是类似的，时钟控制器的节点被使用时钟的模块引用：

```

clocks = <&clks 138>, <&clks 140>, <&clks 141>;
clock-names = "uart", "general", "noc";


```

而驱动中则使用上述的 clock-names 属性作为 clk_get() 或 devm_clk_get() 的第二个参数来申请时钟，譬如获取第 2 个时钟：

```

devm_clk_get(&pdev->dev, "general");


```

<&clks 138> 里的 138 这个 index 是与相应时钟驱动中 clk 的表的顺序对应的，很多开发者也认为这种数字出现在设备树中不太好，因此他们把 clk 的 index 作为宏定义到了 arch/arm/boot/dts/include/dt-bindings/clock 中。譬如 include/dt-bindings/clock/imx6qdl-clock.h 中存在这样的宏：

#define IMX6QDL_CLK_STEP	16
#define IMX6QDL_CLK_PLL1_SW	17
...	
#define IMX6QDL_CLK_ARM	104
...	

而 arch/arm/boot/dts/imx6q.dtsi 则是这样引用它们的：

```

clocks = <&clks IMX6QDL_CLK_ARM>,


```

```
<&clks IMX6QDL_CLK_PLL2_PFD2_396M>,
<&clks IMX6QDL_CLK_STEP>,
<&clks IMX6QDL_CLK_PLL1_SW>,
<&clks IMX6QDL_CLK_PLL1_SYS>;
```

3. pinmux

在设备树中，某个设备节点使用的 pinmux 的引脚群是通过 phandle 来指定的。譬如在 arch/arm/boot/dts/atlas6.dtsi 的 pinctrl 节点中包含所有引脚群的描述，如代码清单 18.15 所示。

代码清单 18.15 设备树中 pinctrl 控制器的引脚群

```
1 gpio: pinctrl@b0120000 {
2     #gpio-cells = <2>;
3     #interrupt-cells = <2>;
4     compatible = "sirf,atlas6-pinctrl";
5     ...
6
7     lcd_16pins_a: lcd0@0 {
8         lcd {
9             sirf,pins = "lcd_16bitsgrp";
10            sirf,function = "lcd_16bits";
11        };
12    };
13    ...
14    spi0_pins_a: spi0@0 {
15        spi {
16            sirf,pins = "spi0grp";
17            sirf,function = "spi0";
18        };
19    };
20    spil_pins_a: spi1@0 {
21        spi {
22            sirf,pins = "spilgrp";
23            sirf,function = "spil";
24        };
25    };
26    ...
27};
```

而 SPI0 这个硬件实际上需要用到 spi0_pins_a 对应的 spi0grp 这一组引脚，因此在 atlas6-evb.dts 中通过 pinctrl-0 引用了它，如代码清单 18.16 所示。

代码清单 18.16 给设备节点指定引脚群

```
1 spi@b00d0000 {
2     status = "okay";
3     pinctrl-names = "default";
4     pinctrl-0 = <&spi0_pins_a>;
5     ...
6};
```

到目前为止，我们可以勾勒出一个设备树的全局视图，图 18.2 显示了设备树中的节点、属性、label 以及 phandle 等信息。

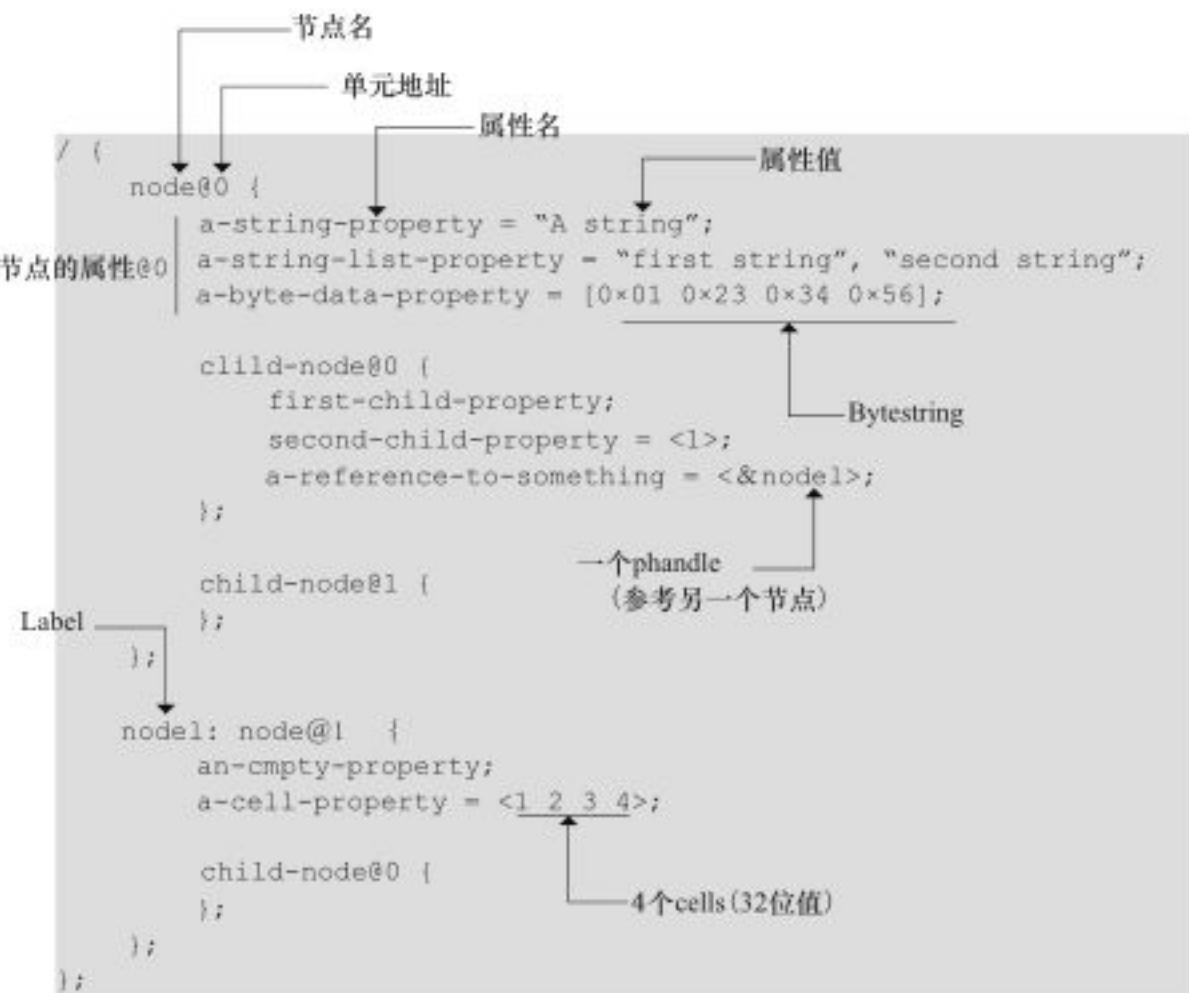


图 18.2 设备树的全景视图

18.3 由设备树引发的 BSP 和驱动变更

有了设备树后，不再需要大量的板级信息，譬如过去经常在 arch/arm/plat-xxx 和 arch/arm/mach-xxx 中实施如下事情。

1. 注册 platform_device，绑定 resource，即内存、IRQ 等板级信息

通过设备树后，形如：

```

static struct resource xxx_resources[] = {
    [0] = {
        .start  = ...,
        .end    = ...,
        .flags   = IORESOURCE_MEM,
    },
    [1] = {
        .start  = ...
    }
};

```

```

        .end      = ...,
        .flags    = IORESOURCE_IRQ,
    },
};

static struct platform_device xxx_device = {
    .name     = "xxx",
    .id       = -1,
    .dev      = {
        .platform_data          = &xxx_data,
    },
    .resource   = xxx_resources,
    .num_resources = ARRAY_SIZE(xxx_resources),
};

```

之类的 platform_device 代码都不再需要，其中 platform_device 会由内核自动展开。而这些 resource 实际来源于.dts 中设备节点的 reg、interrupts 属性。

典型的，大多数总线都与“simple_bus”兼容，而在与 SoC 对应的设备的 .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。

2. 注册 i2c_board_info，指定 IRQ 等板级信息

形如：

```

static struct i2c_board_info __initdata afeb9260_i2c_devices[] = {
{
    I2C_BOARD_INFO("tlv320aic23", 0x1a),
},
{
    I2C_BOARD_INFO("fm3130", 0x68),
},
{
    I2C_BOARD_INFO("24c64", 0x50),
}
};

```

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 这些设备节点填充作为相应的 I²C 控制器节点的子节点即可，类似于前面的代码：

```

i2c@1,0 {
compatible = "acme,al234-i2c-bus";
...
    rtc@58 {
compatible = "maxim,dsl1338";
reg = <58>;
interrupts = <7 3>;
    };
}

```

设备树中的 I²C 客户端会通过在 I²C host 驱动的 probe() 函数中调用的 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：

```
static struct spi_board_info afeb9260_spi_devices[] = {
    { /* DataFlash chip */
        .modalias      = "mtd_dataflash",
        .chip_select   = 1,
        .max_speed_hz = 15 * 1000 * 1000,
        .bus_num       = 0,
    },
};
```

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 之类的节点作为 SPI 控制器的子节点即可，SPI host 驱动的 probe() 函数通过 spi_register_master() 注册主机的时候，会自动展开依附于它的从机，spear1310-evb.dts 中的 st,m25p80 SPI 接口的 NOR Flash 节点如下：

```
spi0: spi@e0100000 {
    status = "okay";
    num-cs = <3>;
    m25p80@1 {
        compatible = "st,m25p80";
        ...
    };
}
```

4. 多个针对不同电路板的设备，以及相关的回调函数

在过去，ARM Linux 针对不同的电路板会建立由 MACHINE_START 和 MACHINE_END 包围的设备，引入设备树之后，MACHINE_START 变更为 DT_MACHINE_START，其中含有一个 .dt_compat 成员，用于表明相关的设备与 .dts 中根节点的兼容属性的兼容关系。

这样可以显著改善代码的结构并减少冗余的代码，在不支持设备树的情况下，光是一个 S3C24xx 就存在多个板文件，譬如 mach-am335900.c、mach-gta02.c、mach-smdk2410.c、mach-qt2410.c、mach-rx3715.c 等，其累计的代码量是相当大的，板级信息都用 C 语言来实现。而采用设备树后，我们可以对多个 SoC 和板子使用同一个 DT_MACHINE 和板文件，板子和板子之间的差异更多只是通过不同的 .dts 文件来体现。

5. 设备与驱动的匹配方式

使用设备树后，驱动需要与在 .dts 中描述的设备节点进行匹配，从而使驱动的 probe() 函数执行。新的驱动、设备的匹配变成了设备树节点的兼容属性和设备驱动中的 OF 匹配表的匹配。

6. 设备的平台数据属性化

在 Linux 2.6 下，驱动习惯自定义 platform_data，在 arch/arm/mach-xxx 注册 platform_

device、i2c_board_info、spi_board_info 等的时候绑定 platform_data，而后驱动通过标准 API 获取平台数据。譬如，在 arch/arm/mach-at91/board-sam9263ek.c 下用如下代码注册 gpio_keys 设备，它通过 gpio_keys_platform_data 结构体来定义 platform_data。

```

static struct gpio_keys_button ek_buttons[] = {
    { /* BP1, "leftclic" */
        .code      = BTN_LEFT,
        .gpio      = AT91_PIN_PC5,
        .active_low = 1,
        .desc      = "left_click",
        .wakeup    = 1,
    },
    { /* BP2, "rightclic" */
        ...
    }
};

static struct gpio_keys_platform_data ek_button_data = {
    .buttons      = ek_buttons,
    .nbuttons     = ARRAY_SIZE(ek_buttons),
};

static struct platform_device ek_button_device = {
    .name          = "gpio-keys",
    .id            = -1,
    .num_resources = 0,
    .dev           = {
        .platform_data = &ek_button_data,
    }
};

```

设备驱动 drivers/input/keyboard/gpio_keys.c 则通过如下简单方法取得这个信息。

```

static int gpio_keys_probe(struct platform_device *pdev)
{
    struct device *dev = &pdev->dev;
    const struct gpio_keys_platform_data *pdata = dev_get_platdata(dev);
    ...
}

```

在转移到设备树后，platform_data 便不再喜欢放在 arch/arm/mach-xxx 中了，它需要从设备树的属性中获取，比如一个电路板上有 gpio_keys，则只需要在设备树中添加类似 arch/arm/boot/dts/exynos4210-origen.dts 中的如代码清单 18.17 所示的信息则可。

代码清单 18.17 在设备树中添加 GPIO 按键信息

```

1 gpio_keys {
2     compatible = "gpio-keys";
3     #address-cells = <1>;

```

```

4      #size-cells = <0>;
5
6      up {
7          label = "Up";
8          gpios = <&gpx2 0 1>;
9          linux,code = <KEY_UP>;
10         gpio-key,wakeup;
11     };
12
13     down {
14         label = "Down";
15         gpios = <&gpx2 1 1>;
16         linux,code = <KEY_DOWN>;
17         gpio-key,wakeup;
18     };
19     ...
20 };

```

而 drivers/input/keyboard/gpio_keys.c 则通过以 of_ 开头的读属性的 API 来读取这些信息，并组织出 gpio_keys_platform_data 结构体，如代码清单 18.18 所示。

代码清单 18.18 在 GPIO 按键驱动中获取 .dts 中的键描述

```

1 static struct gpio_keys_platform_data *
2 gpio_keys_get_devtree_pdata(struct device *dev)
3 {
4     struct device_node *node, *pp;
5     struct gpio_keys_platform_data *pdata;
6     struct gpio_keys_button *button;
7     int error;
8     int nbuttons;
9     int i;
10
11     node = dev->of_node;
12     if (!node)
13         return ERR_PTR(-ENODEV);
14
15     nbuttons = of_get_child_count(node);
16     if (nbuttons == 0)
17         return ERR_PTR(-ENODEV);
18
19     pdata = devm_kzalloc(dev,
20                         sizeof(*pdata) + nbuttons * sizeof(*button),
21                         GFP_KERNEL);
22     if (!pdata)
23         return ERR_PTR(-ENOMEM);
24
25     pdata->buttons = (struct gpio_keys_button *) (pdata + 1);
26     pdata->nbuttons = nbuttons;
27

```

```
28     pdata->rep = !!of_get_property(node, "autorepeat", NULL);
29
30     i = 0;
31     for_each_child_of_node(node, pp) {
32         int gpio;
33         enum of_gpio_flags flags;
34
35         if (!of_find_property(pp, "gpios", NULL)) {
36             pdata->nbuttons--;
37             dev_warn(dev, "Found button without gpios\n");
38             continue;
39         }
40
41         gpio = of_get_gpio_flags(pp, 0, &flags);
42         if (gpio < 0) {
43             error = gpio;
44             if (error != -EPROBE_DEFER)
45                 dev_err(dev,
46                         "Failed to get gpio flags, error: %d\n",
47                         error);
48             return ERR_PTR(error);
49         }
50
51         button = &pdata->buttons[i++];
52
53         button->gpio = gpio;
54         button->active_low = flags & OF_GPIO_ACTIVE_LOW;
55
56         if (of_property_read_u32(pp, "linux,code", &button->code)) {
57             dev_err(dev, "Button without keycode: 0x%x\n",
58                     button->gpio);
59             return ERR_PTR(-EINVAL);
60         }
61
62         button->desc = of_get_property(pp, "label", NULL);
63
64         if (of_property_read_u32(pp, "linux,input-type", &button->type))
65             button->type = EV_KEY;
66
67         button->wakeup = !!of_get_property(pp, "gpio-key,wakeup", NULL);
68
69         if (of_property_read_u32(pp, "debounce-interval",
70                               &button->debounce_interval))
71             button->debounce_interval = 5;
72     }
73
74     if (pdata->nbuttons == 0)
75         return ERR_PTR(-EINVAL);
76
77     return pdata;
78 }
```

上述代码通过第31行的`for_each_child_of_node()`遍历`gpio_keys`节点下的所有子节点，并通过`of_get_gpio_flags()`、`of_property_read_u32()`等API读取出来与各个子节点对应的GPIO、与每个GPIO对应的键盘键值等。

18.4 常用的OF API

除了前文介绍的`of_machine_is_compatible()`、`of_device_is_compatible()`等常用函数以外，在Linux的BSP和驱动代码中，经常会使用到一些Linux中其他设备树的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`目录下。

这些常用的API包括下面内容。

1. 寻找节点

```
struct device_node *of_find_compatible_node(struct device_node *from,
const char *type, const char *compatible);
```

根据兼容属性，获得设备节点。遍历设备树中的设备节点，看看哪个节点的类型、兼容属性与本函数的输入参数匹配，在大多数情况下，`from`、`type`为NULL，则表示遍历了所有节点。

2. 读取属性

```
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);
```

读取设备节点`np`的属性名，为`propname`，属性类型为8、16、32、64位整型数组。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，通过如下语句可读取L2 cache的"arm,data-latency"属性：

```
of_property_read_u32_array(np, "arm,data-latency",
data, ARRAY_SIZE(data));
```

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

```
L2: cache-controller@1e00a000 {
compatible = "arm,p1310-cache";
reg = <0x1e00a000 0x1000>;
interrupts = <0 43 4>;
```

```

cache-level = <2>;
arm,data-latency = <1 1 1>;
arm,tag-latency = <1 1 1>;
}

```

在有些情况下，整型属性的长度可能为 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 中，如代码清单 18.19 所示。

代码清单 18.19 设备树中整型属性的读取 API

```

1 static inline int of_property_read_u8(const struct device_node *np,
2                                     const char *propname,
3                                     u8 *out_value)
4 {
5     return of_property_read_u8_array(np, propname, out_value, 1);
6 }
7
8 static inline int of_property_read_u16(const struct device_node *np,
9                                     const char *propname,
10                                    u16 *out_value)
11 {
12     return of_property_read_u16_array(np, propname, out_value, 1);
13 }
14
15 static inline int of_property_read_u32(const struct device_node *np,
16                                     const char *propname,
17                                     u32 *out_value)
18 {
19     return of_property_read_u32_array(np, propname, out_value, 1);
20 }

```

除了整型属性外，字符串属性也比较常用，其对应的 API 包括：

```

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() 函数就通过 of_property_read_string_index() 遍历 clkspec 节点的所有 "clock-output-names" 字符串数组属性。

代码清单 18.20 在驱动中读取第 index 个字符串的例子

```

1 const char *of_clk_get_parent_name(struct device_node *np, int index)
2 {
3     struct of_phandle_args clkspec;
4     const char *clk_name;
5     int rc;

```

```

6
7     if (index < 0)
8         return NULL;
9
10    rc = of_parse_phandle_with_args(np, "clocks", "#clock-cells", index,
11                                    &clkspec);
12    if (rc)
13        return NULL;
14
15    if (of_property_read_string_index(clkspec.np, "clock-output-names",
16                                     clkspec.args_count ? clkspec.args[0] : 0,
17                                     &clk_name) < 0)
18        clk_name = clkspec.np->name;
19
20    of_node_put(clkspec.np);
21    return clk_name;
22 }
23 EXPORT_SYMBOL_GPL(of_clk_get_parent_name);

```

除整型、字符串以外的最常用属性类型就是布尔型，其对应的 API 很简单，具体如下

```
static inline bool of_property_read_bool(const struct device_node *np,
const char *propname);
```

如果设备节点 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。

3. 内存映射

```
void __iomem *of_iomap(struct device_node *node, int index);
```

上述 API 可以直接通过设备节点进行设备内存区间的 ioremap()，index 是内存段的索引。若设备节点的 reg 属性有多段，可通过 index 标示要 ioremap() 的是哪一段，在只有 1 段的情况下，index 为 0。采用设备树后，一些设备驱动通过 of_iomap() 而不再通过传统的 ioremap() 进行映射，当然，传统的 ioremap() 的用户也不少。

```
int of_address_to_resource(struct device_node *dev, int index,
struct resource *r);
```

上述 API 通过设备节点获取与它对应的内存资源的 resource 结构体。其本质是分析 reg 属性以获取内存地址、大小等信息并填充到 struct resource *r 参数指向的结构体中。

4. 解析中断

```
unsigned int irq_of_parse_and_map(struct device_node *dev, int index);
```

通过设备树获得设备的中断号，实际上是从 .dts 中的 interrupts 属性里解析出中断号。若设备使用了多个中断，index 指定中断的索引号。

5. 获取与节点对应的 platform_device

```
struct platform_device *of_find_device_by_node(struct device_node *np);
```

在可以拿到 device_node 的情况下，如果想反向获取对应的 platform_device，可使用上述 API。

当然，在已知 platform_device 的情况下，想获取 device_node 则易如反掌，例如：

```
static int sirfsoc_dma_probe(struct platform_device *op)
{
    struct device_node *dn = op->dev.of_node;
    ...
}
```

18.5 总结

充斥着 ARM 社区的大量垃圾代码导致 Linus 盛怒，因此该社区在 2011 ~ 2012 年进行了大量的修整工作。ARM Linux 开始围绕设备树展开，设备树有自己的独立语法，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。之后内核会展开设备树并创建和注册相关的设备，因此 arch/arm/mach-xxx 和 arch/arm/plat-xxx 中的大量用于注册 platform、I²C、SPI 等板级信息的代码被删除，而驱动也以新的方式与在 .dts 中定义的设备节点进行匹配。