ARM Linux 设备树（Device Tree）

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。

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：

```
/ {
   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）、I^2C 控制器（位于 0x10160000）、64MB NOR Flash（位于 0x30000000）；

External bus 桥上连接的 I^2C 控制器所对应的 I^2C 总线上又连接了 Maxim DS1338 实时钟（I^2C 地址为 0x58）。

其对应的.dts 文件为：

```
/ {
   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：

```
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 I^2C 总线上连接的 RTC，它的 address 字段为 0x58，是设备的 I^2C 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。

```
    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：

```
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：

```
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

```
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：

```
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

```
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 I^2C controller 结点的子结点即可，类似于前面的

```
i2c@1,0 {
    compatible = "acme,a1234-i2c-bus";
    …
    rtc@58 {
        compatible = "maxim,ds1338";
        reg = <58>;
        interrupts = < 7 3 >;
    };
};
```

Device Tree 中的 I^2C client 会透过 I^2C 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

```
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 I^2C 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：

```
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。

```
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"：

```
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：

```
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 I^2C 控制器结点的 OF 匹配表可以是：

```
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：

```
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，I^2C 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：

```
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：

```
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：

```
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：

```
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：

```
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。

```
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、I^2C、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。