全志科技
Allwinner Technology

Tina3.0

Device Tree 介绍文档 V1.0

文档履历

版本号	日期	制/修订人	制/修订记录
V1.0	2018/04/27		初始版本，适配 Tina3.0

目　　录

1. Device tree 介绍

ARM Linux 中，arch/arm/mach-xxx 中充斥着大量描述板级细节的代码，而这些板级细节对于内核来讲，就是垃圾，如板上的 platform 设备、resource、i2c_board_info、spi_board_info 以及各种硬件的 platform_data。

内核社区为了改变这个局面，引用了 PowerPC 等其他体系结构下已经使用的 Flattened Device Tree(FDT)。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 是一种描述硬件的数据结构，它表现为一颗由电路板上 cpu、总线、设备组成的树，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：

CPU 的数量和类别
内存基地址和大小
总线
外设
中断控制器
GPIO 控制器
Clock 控制器

Bootloader 会将这棵树传递给内核，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，也会通过 dtb 传递给了内核，内核会将这些资源绑定给展开的相应的设备。

Device tree 牵扯的东西还是比较多的，对 device tree 的理解，可以分为 5 个步骤：

1. 用于描述硬件设备信息的文本格式，如 dts\dtsi。
2. 认识 DTC 工具。
3. Bootloader 怎么把二进制文件写入到指定的内存位置。
4. 内核时如何展开文件，获取硬件设备信息。
5. 设备驱动如何使用。

2. Device tree source file

.dts 文件是一种 ASCII 文本格式的 Device Tree 描述，在 ARM Linux 中，一个.dts 文件对应一个 ARM 的 machine。

ARMv7 架构下，dts 文件放置在内核的 ***arch/arm/boot/dts/***目录。
ARMv8 架构下，dts 文件放置在内核的 ***arch/arm64/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。

设备树是一个包含节点和属性的简单树状结构。属性就是键－值对，而节点可以同时包含属性和子节点。例如，以下就是一个 .dts 格式的简单树：

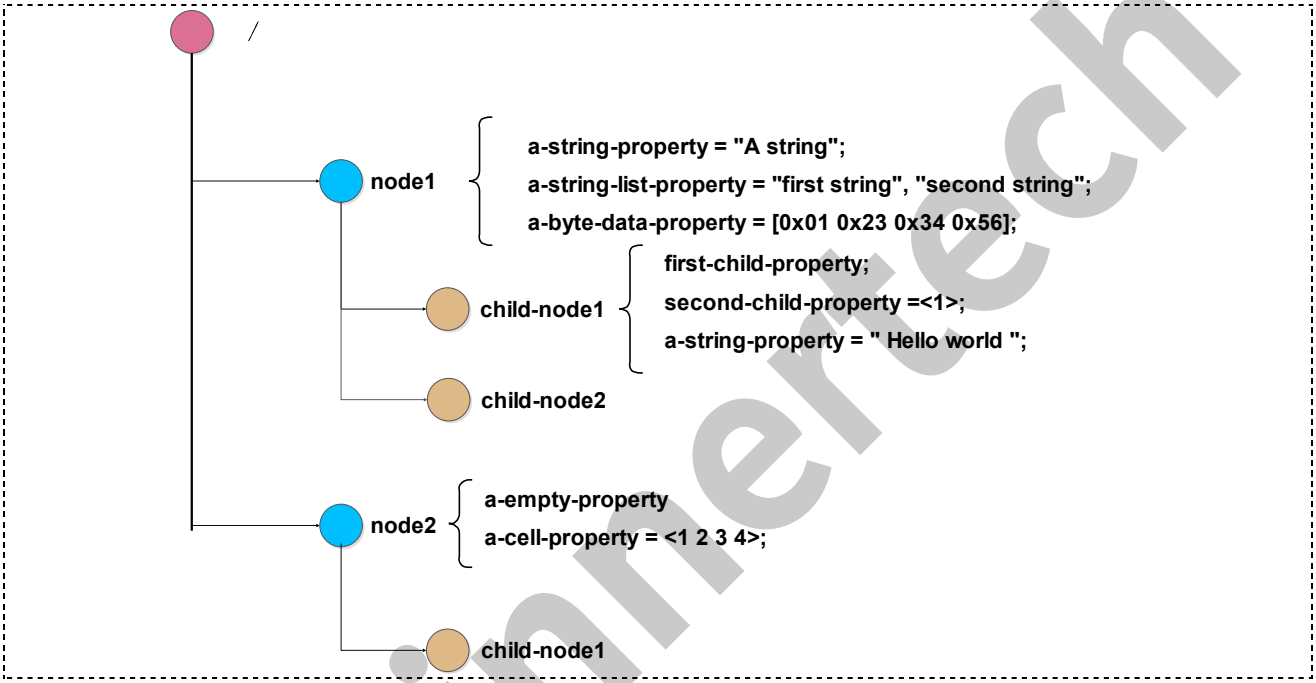

图 2-1 dts 简单树示例

这棵树显然是没什么用的，因为它并没有描述任何东西，但它确实体现了节点的一些属性：

1. 一个单独的根节点：“/”
2. 两个子节点：“node1”和“node2”
3. 两个 node1 的子节点：“child-node1”和“child-node2”
4. 一堆分散在树里的属性。

属性是简单的键－值对，它的值可以为空或者包含一个任意字节流。虽然数据类型并没有编码进数据结构，但在设备树源文件中仍有几个基本的数据表示形式。

1. 文本字符串（无结束符）可以用双引号表示：a-string-property=”hello world”
2. 二进制数据用方括号限定：
3. 不同表示形式的数据可以使用逗号连在一起：
4. 逗号也可用于创建字符串列表：a-string-list-property=”first string”,”second string”

2.1. Device tree 结构约定

2.1.1. 节点名称(node names)

规范：device tree 中每个节点的命名必须遵从一下规范：

node-name@unit-address (详见①②③④)

详注：

①node-name：节点的名称，小于 31 字符长度的字符串，可以包括 图 2-2 字符。节点名称的首字符必须是英文字母，可大写或者小写。通常，节点的命名应该根据它所体现的是什么样的设备。

Character	Description
0-9	digit
a-z	lowercase letter
A-Z	uppercase letter
,	comma
.	period
_	underscore
+	plus sign
-	dash

图 2-2 节点名称支持字符

②@unit-address：如果该节点描述的设备有一个地址，则应该加上设备地址（unit-address）。通常，设备地址就是用来访问该设备的主地址，并且该地址也在节点的 reg 属性中列出。

③同级节点命名必须是唯一的，但只要地址不同，多个节点也可以使用一样的通用名称（例如 serial@101f1000 和 serial@101f2000）。

④根节点没有 node-name 或者 unit-address，它通过“/”来识别。

实例：

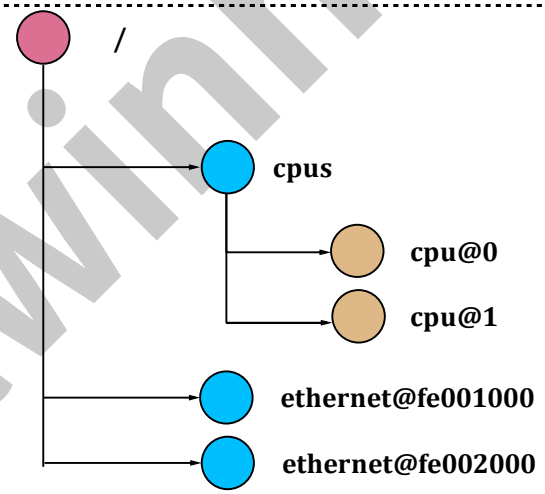

图 2-3 节点名称规范示例

在实例中，一个根节点/下有 3 个子节点；节点名称为 cpu 的节点，通过地址 0 和 1 来区别；节点名称为 ethernet 的节点，通过地址 fe001000 和 fe002000 来区别。

2.1.2. 路径名称(path names)

在 device tree 中唯一识别节点的另一个方法，通过给节点指定从根节点到该节点的完整路径。
device tree 中约定了完整路径表达方式：

```
/node-name-1/node-name-2/.../node-name-N
```

实例：
如 图 2-3 节点名称规范示例 ，

指定根节点路径： /
指定 cpu#1 的完整路径： /cpus/cpu@1
指定 ethernet#fe002000： /cpus/ethernet@fe002000

注：如果完整的路径可以明确表示我们所需的节点，那么 unit-address 可以省略。

2.1.3. 属性(properties)

Device tree 中，节点可以用属性来描述该节点的特征，属性由两个部分组成：名称和值。

2.1.3.1. 属性名称(property names)

由长度小于 31 的字符串组成。属性名称支持的字符如下图：

Character	Description
0-9	digit
a-z	lowercase letter
A-Z	uppercase letter
,	comma
.	period
_	underscore
+	plus sign
-	dash

图 2-4 属性名称支持字符

非标准的属性名称，需要指定一个唯一的前缀，用来识别是那个公司或者机构定义了该属性。例如：

```
fsl,channel-fifo-len 29
ibm,ppc-interrupt-server#s 30
linux,network-index
```

2.1.3.2. 属性值(property values)

属性值是一个包含属性相关信息的数组，数组可能有 0 个或者多个字节。当属性是为了传递真伪信息时，属性值可能为空值，这个时候，属性值的存在或者不存在，就已经足够描述属性的相关信息了。

value	description
<empty>	属性表达真伪信息，判断值有没有存在就可以识别
<u32>	大端格式的 32 位整数.例如：值 0x11223344 address 11 address+1 22 address+2 33 address+3 44
<u64>	大端格式的 64 位整数，有两个<u32>组成。第一<u32>表示高位，第二<u32>表示地位。例如，0x1122334455667788 由两个单元组成<0x11223344,0x55667788> address 11 address+1 22 address+2 33 address+3 44 address+4 55 address+5 66 address+6 77 address+7 88
<string>	字符串可打印，并且有终结符。例如：“hello” address 68 address+1 65 address+2 6c address+3 6c address+4 6f address+5 00
<prop-encoded-array>	跟特定的属性有关
<phandle>	一个<u32>值，phandle 值提供了一种引用设备树中其他节点的方法。通过定义 phandle 属性值，任何节点都可以被其他节点引用。
<stringlist>	由一系列<string>值串连在一起，例如”hello”,”world”. address 68 address+1 65 address+2 6C address+3 6C address+4 6F address+5 00 address+6 77 address+7 6F address+8 72 address+9 6C address+10 64 address+11 00

表格 1 属性值

2.1.4. 标准属性类型

2.1.4.1. Compatible

1. 属性：compatible
2. 值类型：<stringlist>
3. 说明：

树中每个表示一个设备的节点都需要一个 compatible 属性。compatible 属性是操作系统用来决定使用哪个设备驱动来绑定到一个设备上的关键因素。compatible 是一个字符串列表，之中第一个字符串指定了这个节点所表示的确切的设备，该字符串的格式为：

"<制造商>,<型号>"

剩下的字符串的则表示其它与之相兼容的设备。

例如：compatible = "fsl,mpc8641-uart", "ns16550";

系统首先会查找跟 fsl,mpc8641-uart 相匹配的驱动，如果找不到，就找更通用的，跟 ns16550 想匹配的驱动。

2.1.4.2. Model

1. 属性：model
2. 值类型：<string>
3. 说明：

model 属性值是<string>,该值指定了设备的型号。推荐的使用形式如下：

"manufacturer，model"

其中，字符 manufacturer 表示厂商的名称，字符 model 表示设备的型号。

例如：model = "fsl,MPC8349EMITX";

2.1.4.3. Phandle

1. 属性：phandle
2. 值类型：<u32>
3. 说明：

device tree 中，定义了 phandle 属性，它是一个 u32 的值。每个节点都可以拥有一个相关的 phandle,通过它的值来唯一标识。(实际实现中常采用指针或者偏移)，phandle 常用于查询或者遍历设备树，也有用于指向设备树中的其它节点。例如，在设备树中，pic 节点如下所示：

```
pic@10000000 {
        phandle = <1>;
        interrupt-controller;
};
```

定义 pic 节点的 phandle 为 1，那么其他设备节点引用 pic 节点时，只需要在本节点中添加：

interrupt-parent = <1>;

2.1.4.4. Status

a. 属性：status
b. 值类型：<string>
c. 说明：

该属性指明设备的运行状态，见表格 2 设备运行状态。

value	description
“okay”	表明设备可运行
“disabled”	表明设备当前不可运行，但条件满足，它还是可以运行的。
“fail”	表明设备不可运行，设备产生严重错误，如果不修复，将一直不可运行
“fail-sss”	表明设备不可运行，设备产生严重错误，如果不修复，将一直不可运行.sss 部分特定设备相关，指明错误检测条件。

表格 2 设备运行状态

2.1.4.5. #address-cells 和#size-cells

1. 属性：#address-cells，#size-cells
2. 值类型：<u32>
3. 说明：

#address-cells 和#size-cells 属性常备用在拥有孩子节点的父节点上，用来描述孩子节点时如何编址的。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。例如：

```
/{
        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>;
                };
        };
        [....]
};
```

图 2-5 #address-cells 和#size-cells 示例

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 1 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 #addresscells= <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。

2.1.4.6. Reg

1. 属性：reg
2. 值类型：<address1 length1 [address2 length2] [address3 length3] ... >
3. 说明：

reg 属性描述了设备拥有资源的地址信息，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。

2.1.4.7. Virtual-reg

1. 属性：virtual-reg
2. 值类型：<u32>
3. 说明：virutal-reg 属性指定一个有效的地址映射到物理地址。

2.1.4.8. Ranges

1. 属性：ranges
2. 值类型：<empty>或者<prop-encoded-array>
3. 说明：

前边 reg 属性说明中，我们已经知道如何给设备分配地址，但目前来说这些地址还只是设备节点的本地地址，我们还没有描述如何将这些地址映射成 CPU 可使用的地址。根节点始终描述的是 CPU 视角的地址空间。根节点的子节点已经使用的是 CPU 的地址域，所以它们不需要任何直接映射。例如，serial@101f0000 设备就是直接分配的 0x101f0000 地址。那些非根节点直接子节点的节点就没有使用 CPU 地址域。为了得到一个内存映射地址，设备树必须指定从一个域到另一个域地址转换的方法，而 ranges 属性就为此而生。还以图 2-5 的设备数来分析：

```
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 是一个地址转换列表。ranges 表中的每一项都是一个包含子地址、父地址和在子地址空间中区域大小的元组。每个字段的值都取决于子节点的 #address-cells 、父节点的 #address-cells 和子节点的 #size-cells。以本例中的外部总线来说，子地址是 #address-cells 是 2、父地址#address-cells 是 1 、区域大小#size-cells 是 1。那么三个 ranges 被翻译为：

从片选 0 开始的偏移量 0 被映射为地址范围：0x10100000..0x1010ffff
从片选 0 开始的偏移量 1 被映射为地址范围：0x10160000..0x1016ffff
从片选 0 开始的偏移量 2 被映射为地址范围：0x30000000..0x10000000

另外，如果父地址空间和子地址空间是相同的，那么该节点可以添加一个空的 range 属性。一个空的 range 属性意味着子地址将被 1:1 映射到父地址空间。

2.2. 常用节点类型

所有 device tree 都必须拥有一个根节点，还必须在根节点下边有以下的节点：

1. Cpu 节点
2. Memory 节点

2.2.1. 根节点(root node)

设备树都必须有一个根节点，树中其它的节点都是根节点的后代，根节点的完整路径是/。

根节点具有如下属性：

属性名称	需要使用	属性值类型	定义
#address-cells	需要	<u32>	表示子节点寄存器属性中的地址
#size-cells	需要	<u32>	表示子节点寄存器属性中的大小
model	需要	<string>	指定一个字符串用来识别不同板子
compatible	需要	<stringlist>	指定平台的兼容列表
Epapr-version	需要	<string>	这个属性必须包含下边字符串 “ePAPR-<ePAPR version>” 其中,<ePAPR version>是平台遵从的 PAPR 规范版本号， 例如：Epapr-version =”ePAPR-1.1”

2.2.2. 别名节点(aliases node)

Device tree 中采用别名节点来定义设备节点全路径的别名，别名节点必须是根节点的孩子，而且还必须采用 aliases 的节点名称。

/aliases 节点中每个属性定义了一个别名，属性的名字指定了别名，属性值指定了 device tree 中设备节点的完整路径。例如：

```
serial0 = “/simple-bus@fe000000/serial@llc500”
```

指定该路径下 serial@llc500 设备节点全路径的别名为 serial0。当用户想知道的只是“那个设备是 serial”时，这样的全路径就会变得很冗长，采用 aliases 节点指定一个设备节点全路径的别名，好处就在这个时候体现出来了。

2.2.3. 内存节点(memory node)

ePAPR 规范中指定了内存节点是 device tree 中必须的节点。内存节点描绘了系统物理内存的信息，如果系统中有多个内存范围，那么 device tree 中可能会创建多个内存节点，或者在一个单独的内存节点中通过 reg 属性指定内存的范围。节点的名称必须是 memory。

内存节点属性如下：

属性名称	是否使用	值类型	定义
Device_type	需要	<string>	属性值必须为”memory”
reg	需要	<prop-encoded-array>	包含任意数量的用来指示地址和地址空间大小的对。
Initial-mapped-area	可选择	<prop-encoded-array>	指定初始映射区的内存地址和地址空间的大小

假设一个 64 位系统具有以下的物理内存块：

1. RAM：起始地址 0x0,长度 0x80000000(2GB)
2. RAM：起始地址 0x100000000,长度 0x100000000(4GB)

内存节点的定义可以采用以下方式，假设#address-cells =2，#size-cells =2。

方式 1：

```
memory@0 {
        device_type = "memory";
        reg = <    0x000000000 0x00000000 0x00000000 0x80000000
                   0x000000001 0x00000000 0x00000001 0x00000000>;
};
```

方式 2：

```
memory@0 {
        device_type = "memory";
        reg = <    0x000000000 0x00000000 0x00000000 0x80000000>;
};
memory@100000000 {
        device_type = "memory";
        reg = <    0x000000001 0x00000000 0x00000001 0x00000000>;
};
```

2.2.4. chosen 节点

chosen 节点并不代表一个真正的设备，只是作为一个为固件和操作系统之间传递数据的地方，比如引导参数。chosen 节点里的数据也不代表硬件。通常，chosen 节点在 .dts 源文件中为空，并在启动时填充。它必须是根节点的孩子。

节点属性如下：

属性名	是否使用	值类型	定义
bootargs	可选择	<string>	为用户指定 boot 参数
Stdout-path	可选择	<string>	指定 boot 控制台输出路径
Stdin-path	可选择	<string>	指定 boot 控制台输入路径

例子：

```
chosen {
        bootargs = "root=/dev/nfs rw nfsroot=192.168.1.1 console=ttyS0,115200";
};
```

2.2.5. cpus 节点

ePAPR 规范指定 cpus 节点是 device tree 中必须的节点，它并不代表系统中真实设备，可以理解 cpus 节点仅作为存放子节点 cpu 的一个容器。

节点属性如下：

属性值	是否使用	值类型
#address-cells	必须	<u32>
#size-cells	必须	<u32>

2.2.6. cpu 节点

Device tree 中每一个 cpu 节点描述一个具体的硬件执行单元。每个 cpu 节点的 compatible 属性是一个“<制造商>,<型号>”形式的字符串，并指定了确切的 cpu，就像顶层的 compatible 属性一样。如果系统的 cpu 拓扑结构很复杂，还必须在 binding 文档中详细说明。

cpu 节点所拥有的属性：

属性名	是否使用	属性值	定义
Device_type	必须	<string>	属性值必须是“cpu”的字符串
reg	必须	<prop-encoded-array>	定义 cpu/thread id。
Clock-frequency	必须	<prop-encodec-array>	指定 cpu 的时钟频率
Timebase-frequency	必须	<prop-encoded-array>	指定当前 timebase 的是时钟频率信息。
status		<u32>	描述 cpu 的状态 okay/disabled
Enable-method		<stringlist>	指定了 cpu 从 disabled 状态到 enabled 的方式。
Mmu-type	可选	<string>	指定 cpu mmu 的类型

cpu 节点实例：

```
cpus {
        #address-cells = <1>;
        #size-cells = <0>;
        cpu@0 {
                device_type = "cpu";
                compatible = "arm,cortex-a8";
                reg = <0x0>;
        };
};
```

2.2.7. soc 节点

这个节点用来表示一个系统级芯片（soc），如果处理器就是一个系统级芯片，那么这个节点就必须包含，soc 节点的顶层包含 soc 上所有设备可见的信息。节点名字必须包含 soc 的地址并且以"soc"字符开头。

实例：

```
soc@01c20000 {
        compatible = "simple-bus";
        #address-cells = <1>;
        #size-cells = <1>;
        reg = <0x01c20000 0x300000>;
        ranges;

        intc: interrupt-controller@01c20400 {
            compatible = "allwinner,sun4i-ic";
            reg = <0x01c20400 0x400>;
            interrupt-controller;
            #interrupt-cells = <1>;
        };

        pio: pinctrl@01c20800 {
            compatible = "allwinner,sun5i-a13-pinctrl";
            reg = <0x01c20800 0x400>;
            interrupts = <28>;
            clocks = <&apb0_gates 5>;
            gpio-controller;
            interrupt-controller;
            #address-cells = <1>;
            #size-cells = <0>;
            #gpio-cells = <3>;

            uart1_pins_a: uart1@0 {
                allwinner,pins = "PE10", "PE11";
                allwinner,function = "uart1";
                allwinner,drive = <0>;
                allwinner,pull = <0>;
            };
            ......
}
```

2.3. 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，这些文档位于内核的 Documentation/devicetree/bindings/arm 路径下。

3. Device tree block file

3.1. 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，编译内核时同时会编译 dtc。通过 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，描述了当某个 SoC 被选中后，哪些.dtb 文件会被编译出来，如与 sunxi 对应的.dtb 包括

```
dtb-$(CONFIG_ARCH_SUNXI) += \
	sun4i-a10-cubieboard.dtb \
	sun4i-a10-mini-xplus.dtb \
	sun4i-a10-hackberry.dtb \
	sun5i-a10s-olinuxino-micro.dtb \
	sun5i-a13-olinuxino.dtb
```

3.2. 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。

3.3. DTB 的内存布局

Device tree block 内存布局大致如下(地址从上往下递增)。我们可以看到，dtb 文件结构主要由 4 个部分组成，一个小的文件头、一个 memory reserve map、一个 device tree structure 、一个 device-tree strings。这几个部分构成一个整体，一起加载到内存中。

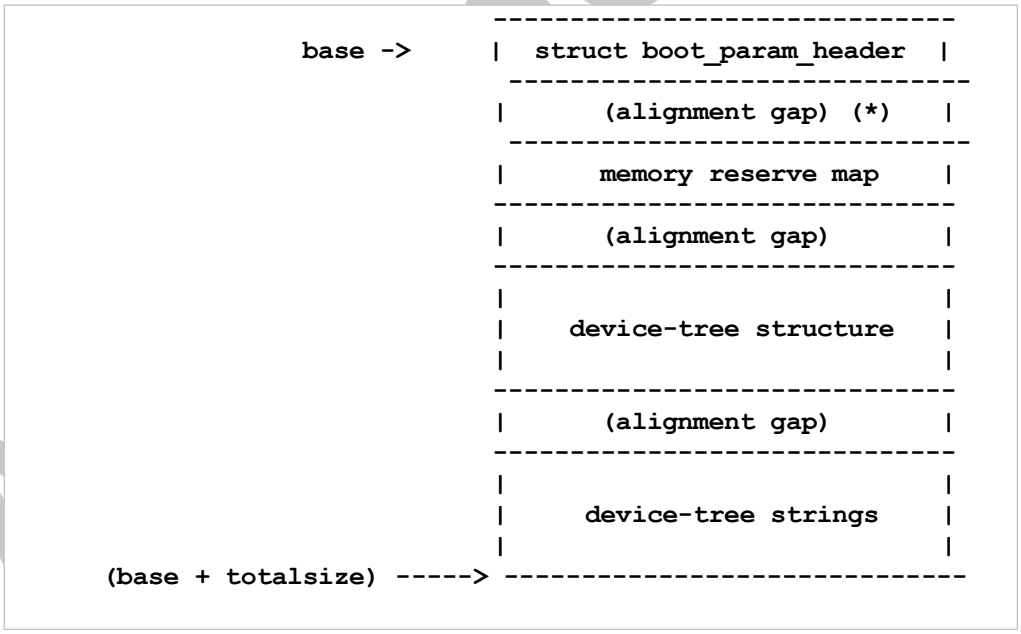

图 3-1 dtb 内存布局

3.3.1. 文件头-boot_param_header

内核的物理指针指向的内存区域在 structure boot_param_header 这个结构体中大概描述到了：

include/linux/of_fdt.h

```
/* Definitions used by the flattened device tree */
#define OF_DT_HEADER            0xd00dfeed  /* marker */
#define OF_DT_BEGIN_NODE        0x1         /* Start of node, full name */
#define OF_DT_END_NODE          0x2         /* End node */
#define OF_DT_PROP              0x3         /* Property: name off, size,* content */
#define OF_DT_NOP               0x4         /* nop */
#define OF_DT_END               0x9
#define OF_DT_VERSION           0x10

struct boot_param_header {
        __be32    magic;                    /* magic word OF_DT_HEADER */
        __be32    totalsize;                /* total size of DT block */
        __be32    off_dt_struct;            /* offset to structure */
        __be32    off_dt_strings;           /* offset to strings */
        __be32    off_mem_rsvmap;           /* offset to memory reserve map */
        __be32    version;                  /* format version */
        __be32    last_comp_version;        /* last compatible version */
        /* version 2 fields below */
        __be32    boot_cpuid_phys;          /* Physical CPU id we're booting on */
        /* version 3 fields below */
        __be32    dt_strings_size;          /* size of the DT strings block */
        /* version 17 fields below */
        __be32    dt_struct_size;           /* size of the DT structure block */
};
```

具体这个结构体怎么用，在后边会有具体描述。

3.3.2. device-tree structure

这一部分主要存储了各个结点的信息。每一个结点都都可以嵌套子结点，其中的结点以 OF_DT_BEGIN_NODE 做起始标志，接下来就是结点名。如果结点带有属性，那么就紧接就是结点的属性，其以 OF_DT_PROP 为起始标志。嵌套的子结点紧跟着父子结点之后，也是以 OF_DT_BEGIN_NODE 起始。OF_DT_END_NODE 标志着一结点的终止。

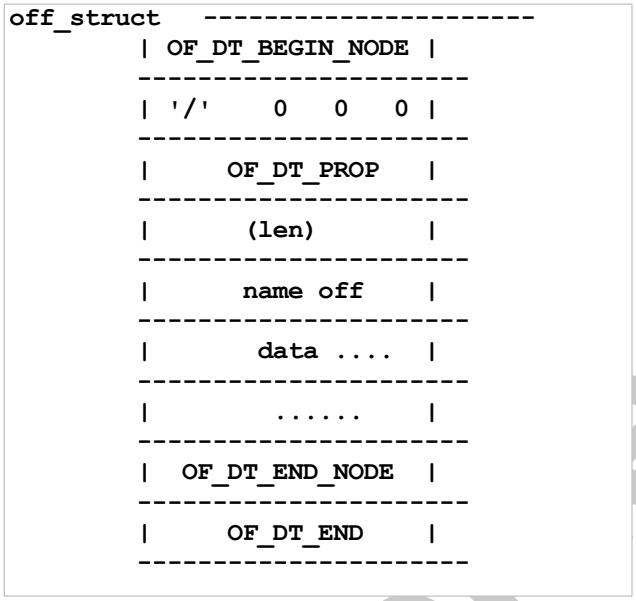

图 3-2 device-tree structure 结构

上面提到一个结点的属性，每一个属性有如下的结构：

Scripts/dtc/libfdt/fdt.h

```
struct fdt_property {
        uint32_t tag;
        uint32_t len;
        uint32_t nameoff;
        char data[0];
};
```

3.3.3. Device tree string

最后一部分就是 String，没有固定格式。其主要是把一些公共的字符串线性排布，以节约空间。

3.3.4. dtb 实例

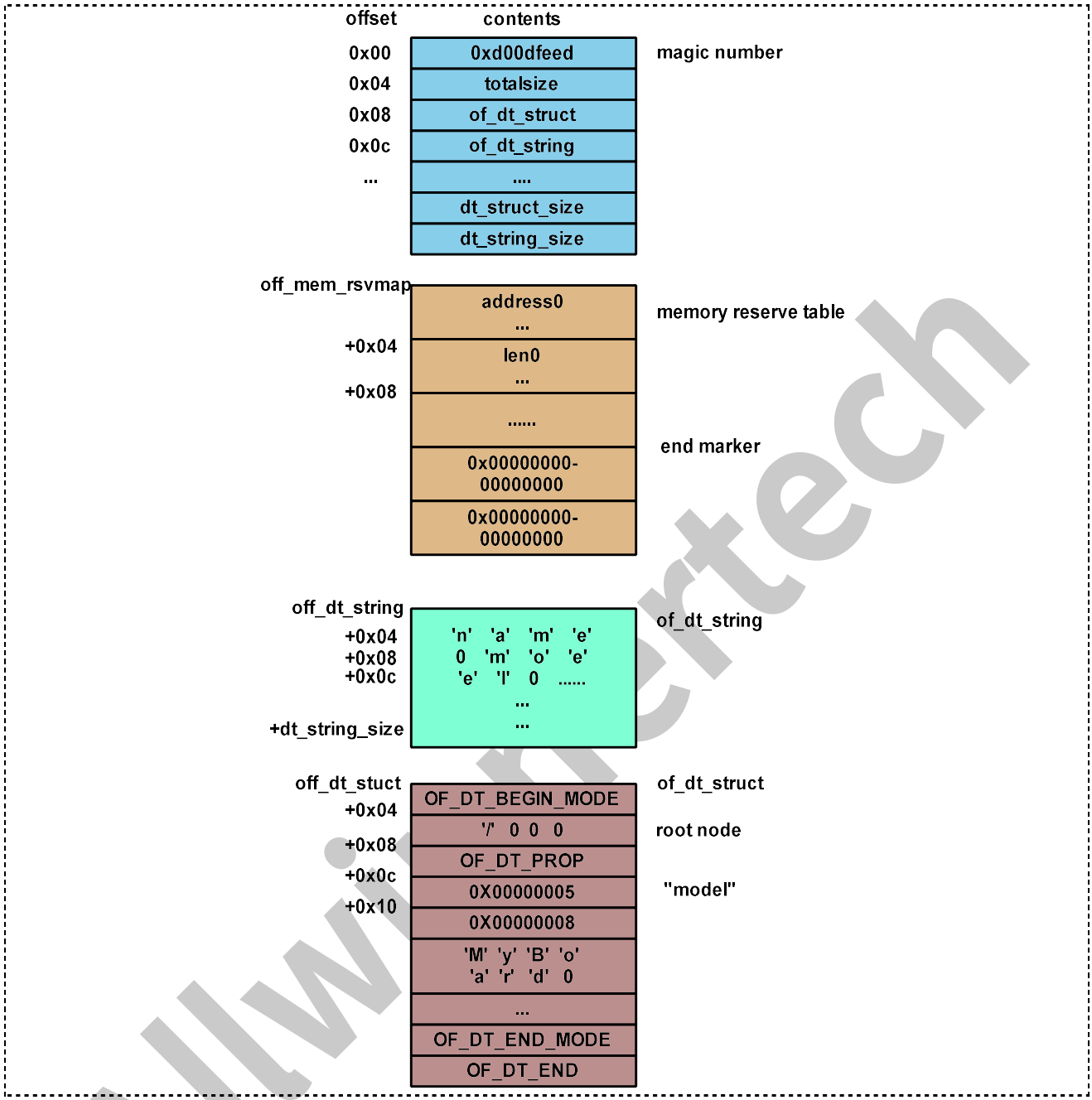

图 3-3 dtb 实例

图 3-3 dtb 实例 可以看出 dtb 结构由 4 个部分组成。

memory reserve table：给出了 kernel 不能使用的内存区域列表。

Of_device-struct：结构包含了 device tree 的属性。每个节点以 OF_DT_BEGIN_NODE 标签开始，接着紧跟着节点的名称，如果节点有属性，那么紧跟着就是节点的属性，每个属性值以 OF_DT_PROP 标签开始，紧接着是嵌套在节点中的子节点，字节点也是以 OF_DT_BEGIN_NODE 起始，以 OF_DT_END_NODE 结束，最后以标签 OF_DT_END 标示根节点结束。

对每个属性，在标签 OF_DT_PROP 之后，由一个 32 位的数指明属性名称存放在偏移 of_dt_string 结构体起始地址多少 byte 的地方。之所以采用这种做法，是因为有很多节点都有很多相同的属性名称，比如 compatible、reg 等，这些节点的名称如果一个个存放起来，显然听浪费空间的，采用一个偏移量，来指定它在 of_dt_string 的哪个地方，在 of_dt_string 中只需要保存一份属性值就可以了，有利于降低 block 占用的空间。

4. 内核常用 API

4.1. of_device_is_compatible

4.1.1. 原型

```
int of_device_is_compatible(const struct device_node *device,const char *compat);
```

4.1.2. 函数作用

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。

4.2. of_find_compatible_node

4.2.1. 原型

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

4.2.2. 函数作用

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。

4.3. of_property_read_u32_array

4.3.1. 原型

```
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);
```

4.3.2. 函数作用

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()。

4.4. of_property_read_string

4.4.1. 原型

```
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);
```

4.4.2. 函数作用

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。

4.5. bool of_property_read_bool

4.5.1. 原型

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

4.5.2. 函数作用

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。

4.6. of_iomap

4.6.1. 原型

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

4.6.2. 函数作用

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。

4.7. irq_of_parse_and_map

4.7.1. 原型

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

4.7.2. 函数作用

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。

5. Device tree 配置 demo

以 pinctrl 为例：

```
soc@01c20000 {
        compatible = "simple-bus";
        #address-cells = <1>;
        #size-cells = <1>;
        ranges;
        pio: pinctrl@01c20800 {
            compatible = "allwinner,sun50i-pinctrl";
            reg = <0x01c20800 0x400>;
            interrupts = <0 11 1>, <0 15 1>, <0 16 1>, <0 17 1>;
            clocks = <&apb1_gates 5>;
            gpio-controller;
            interrupt-controller;
            #address-cells = <1>;
            #size-cells = <0>;
            #gpio-cells = <6>;

            uart0_pins_a: uart0@0 {
                allwinner,pins = "PH20", "PH21";     //设备需要用到的 pin
                allwinner,function = "uart0";        //复用名字
                allwinner,drive = <0>;               //设置驱动力
                allwinner,pull = <0>;                //设置上下拉
                allwinner,data=<0>;                  //设置数据属性
            };
            ......
            ......
        };
};
```

6. Declaration