设备树

一、基本介绍

1.1 什么是设备树

在设备树出现之前，Linux 代码中有大量关于芯片平台以及板级相关的硬件描述代码，每个主板对应的硬件参数是不一样的，就会有很多冗余的垃圾代码。在 Linux 内核中有很多的 dts 和 dtsi 文件，这些就是设备树的源文件，描述的是板级的硬件信息，包括 CPU，memory，clock 以及各种外设的参数，对于不同的主板需要对应不同的设备树文件。有些可以复用的代码，一般写在 dtsi 文件中，然后像 C 语言的头文件一样通过 #include 包含到 dts 文件中。这些 dts 和 dtsi 文件经过 dtc 编译之后生成 dtb 文件，与内核镜像就完全分开，在 bootloader 跳转到 kernel 的时候，会通过寄存器 x0(arm64 架构，arm32 架构使用的是寄存器 r2)传递 dtb 镜像的物理地址，之后在 kernel 初始化阶段对设备树中的参数进行解析。然而对于老旧而不支持 dtb 的 bootloader，可以打开 kernel 的配置选项 CONFIG_ARM_APPENDED_DTB，并且使用 cat 命令把 dtb 合并到 kernel 镜像后面，这样 kernel 会自己从镜像末尾找到 dtb。

1.2 设备树源文件

设备树源文件中包含根节点，以”/”来标识，根节点下面会有很多子节点。对于多个根节点，在编译的时候会进行合并，最终只有一个根节点。除了根节点以外的节点都有父节点，还可能会有一系列子节点。每个节点都有一系列的属性。接下来介绍一些常用的属性。

```
27 / {
28   model = "Qualcomm Technologies, Inc. Lahaina";
29   compatible = "qcom,lahaina";
30   qcom,msm-id = <415 0x10000>, <456 0x10000>, <501 0x10000>;
31   interrupt-parent = <&intc>;
32
33   #address-cells = <2>;
34   #size-cells = <2>;
35
36   chosen {
37           bootargs = "log_buf_len=256K earlycon=msm_geni_serial,0x98c000
38   };
39
40   memory { device_type = "memory"; reg = <0 0 0 0>; };
41   reserved_memory: reserved-memory { };
42
43   mem-offline {
44           compatible = "qcom,mem-offline";
45           offline-sizes = <0x1 0x40000000 0x0 0x40000000>,
46                           <0x1 0xc0000000 0x0 0x80000000>,
47                           <0x2 0xc0000000 0x1 0x40000000>;
48           granule = <512>;
49           mboxes = <&qmp_aop 0>;
```

dts 文件示例

1.2.1 compatible

每个设备节点都有 compatible 属性，是表明设备的制造商和硬件类型的字符串，在驱

动绑定设备的时候就是通过匹配 compatible 属性来完成的。

1.2.2 #address-cells、size-cells 和 reg

在设备需要用到寄存器等物理地址的时候，可以通过 reg 属性来描述寄存器起始地址和需要的地址范围的大小。在设备树中的地址参数是以 cell 为单位的，一个 cell 是 4 字节，#address-cells 属性定义的是 reg 属性中的起始地址所占用的字长，#size-cell 定义的是地址范围参数所占的字长。另外，从当前节点往父节点遍历，第一次找到#address-cells 和 #size-cells 属性就是当前节点需要使用的参数，例如下图定义的地址和长度都是占 1 个 cell，apps_rsc 节点定义了三个寄存器 drv-0，drv-1 和 drv-2，地址范围分别为 0x18200000~0x1820FFFF，0x18210000~0x1821FFFF，0x18220000~0x1822FFFF。

```
571 &soc {
572   #address-cells = <1>;
573   #size-cells = <1>;
574   ranges = <0 0 0 0xffffffff>;
575   compatible = "simple-bus";
576
577   psci {
578           compatible = "arm,psci-1.0";
579           method = "smc";
580   };
581
582   apps_rsc: rsc@18200000 {
583           label = "apps_rsc";
584           compatible = "qcom,rpmh-rsc";
585           reg = <0x18200000 0x10000>,
586                 <0x18210000 0x10000>,
587                 <0x18220000 0x10000>;
588           reg-names = "drv-0", "drv-1", "drv-2";
```

1.2.3 interrupts

中断相关的参数在设备树中用 interrupts 属性进行描述，该属性会有三个参数，这里分别以 irq_param[0]，irq_param[1]和 irq_param[2]表示。irq_param[0]是中断类型，包括 SPI，PPI 和 LPI，分别定义为 0，1，0xa110c8ed。这三类中断的区别以后在介绍中断的时候可以详细说说。irq_param[1]与硬件中断号 hwirq 相关，之所以说相关，是因为具体的硬件中断号还与中断类型有关。对于 SPI，也就是 irq_param[0]为 0 时，hwirq= irq_param[1] + 32；对于 PPI，也就是 irq_param[0]为 1 时，hwirq= irq_param[1]+16；对于 LPI，也就是 irq_param[0]为 0xa110c8ed 时，hwirq= irq_param[1]。irq_param[2]是中断信号触发类型，比如高电平，低电平触发或者电平的上升沿触发，下降沿触发等等。下面的例子中的中断参数可以解析出来是 SPI 中断，中断号为 32，高电平触发。

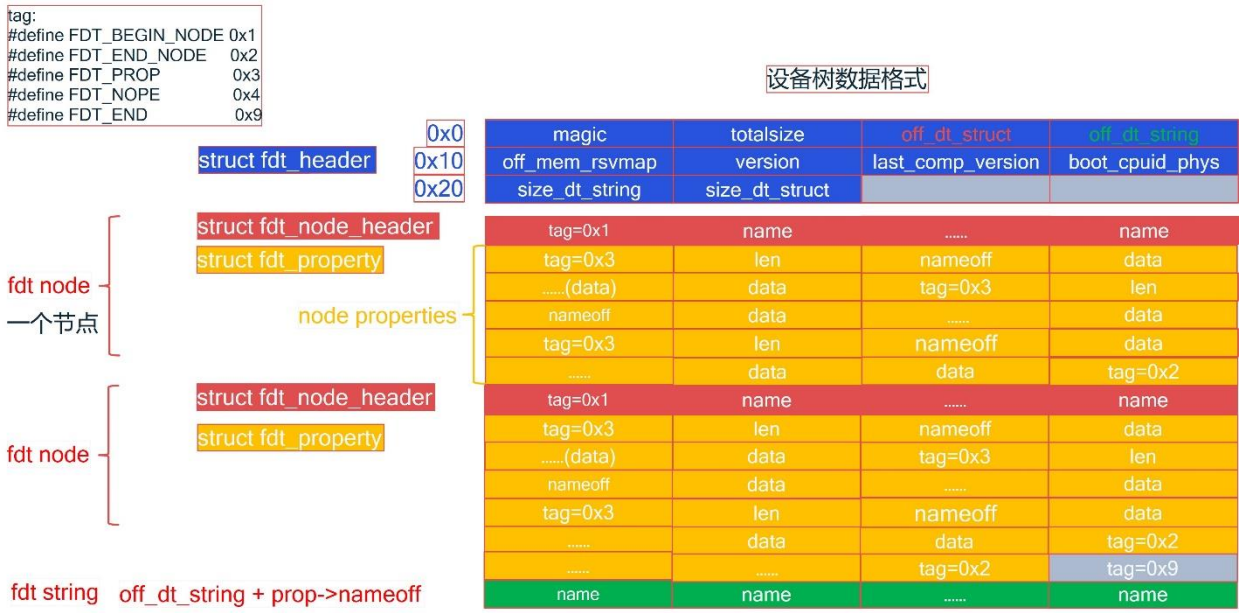
```
4174   wdog: qcom,wdt@17c10000{
4175           compatible = "qcom,msm-watchdog";
4176           reg = <0x17c10000 0x1000>;
4177           reg-names = "wdt-base";
4178           interrupts = <0 0 IRQ_TYPE_LEVEL_HIGH>;
4179   };
```

1.2.4 status

通过设置 status 属性的值可以启用或者停用设备。当 status 设置为“ok”或者“okay”时，或者没有 status 属性，都表示设备被启用。当设置为“disabled”时，表示被停用。

1.3 设备树镜像格式

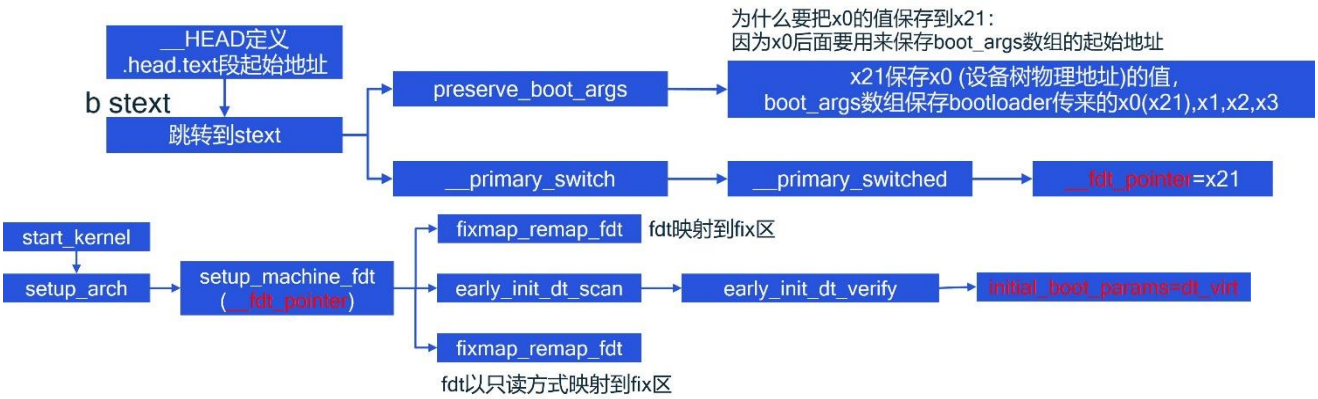
我们这里只讨论 kernel 生成的 dtb 文件的镜像格式，对于 android 生成的 dtbo 文件在 dtb 格式的镜像基础上又封装了一层 header，用于在 bootloader 中做进一步校验，不在本文的讨论范围。



二、设备树展开

2.1 获取设备树的地址

Bootloader 跳转到内核的时候会通过寄存器传递参数，arm64 平台使用的是 x0 保存设备树加载到内存中的物理地址，arm32 使用 r2 寄存器。在内核启动阶段会把从 bootloader 传递来的参数取出来，从而得到设备树的物理地址。然而在 MMU 打开之后，系统中访问地址都是以虚拟地址的方式进行，因此还需要对设备树的起始物理地址进行映射。但是问题来了，这时候 linux 仅仅创建了一个临时页表，能访问的空间是有限的，甚至内存的初始化都还没有完成，怎么进行地址映射呢？这就用到了固定映射区，也就是 linux 给 dtb 分配了一段固定区域作为虚拟地址，找到虚拟地址之后保存在 initial_boot_params 全局变量中。

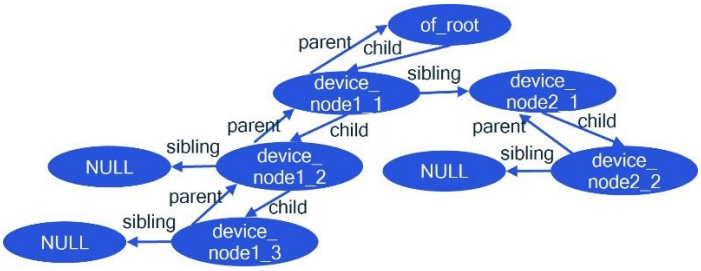


2.2 设备树展开—生成 device_node

设备树展开的过程会对设备树文件进行两次扫描，第一次扫描是用来获取需要申请的总内存大小。扫描完成之后会用 struct device_node 结构体来描述每一个节点，用 struct property 来描述每一个属性，另外还会记录每个节点的名字信息。因为设备树扫描的时候伙伴系统和 slab 都还没初始化，需要通过 memblock_alloc 来申请内存。memblock_alloc 申请内存的时候需要遍历 memblock 管理的 memory region，直到找到符合大小要求的空闲内存块，然后还会遍历 reserve region，如果把这个内存块不在 reserve region 中，才会返回地址，并把这个内存块插入到 reserve region。所以如果边扫描边申请的话，会效率极低。不如第一次扫描的时候只记录内存大小，然后一次性申请。那么第一次扫描之后会获取到需要申请的内存大小为:

size = sum_node * [sizeof(struct device_node) + strlen(fdt_node_header->name)]+ sum_property * sizeof(struct property)。第二次扫描会具体的展开每一个节点和每一个节点的属性，也就是给结构体 struct device_node 和 struct property 完成赋值，最终结果保存在 of_root 起始地址中，格式如下：

下图是以树形的结构来描述设备树各节点的关系。



设备树展开之后就在内核中通过 device_node 来描述各个节点，并且有 of_root 变量可以找到根节点。更进一步，为了融入 linux 的设备驱动框架中，会把各 device node 转化成

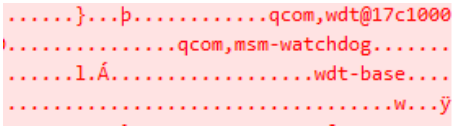
platform_device，最终把设备添加到 platform 总线中，以供后续驱动加载时候进行匹配。

三、设备树的叠加

以上分析的都是 linux 处理设备树的流程，在 bootloader 跳转到 kernel 之前，还可能会有设备树的叠加操作。也就是在 bootloader 中可以加载多个 dtb，然后对相同的节点进行修改或者是添加新的节点。这是多个 dtb 进行叠加的情况。还有一种情况是动态生成节点。在 sm8350 之后的平台，引入了 hypervisor。Hyervisor 在启动阶段会遵循设备树的格式生成一些设备节点（确切的说更像是以 dtc 的方式生成设备节点，只不过不需要 dts 文件），同时也可以对 LA 编译的设备树的节点进行修改。因此等到 linux 启动的时候看到的设备树和编译生成的设备树是会有差异的。

下面以 lahaina 平台的 watchdog 为例说明一下设备树叠加的过程。

首先，在 linux/android 代码中有 dts 文件定义了 msm-watchdog 节点，这个也是之前各平台一直使用的 watchdog 节点。编译生成的 dtb 中也可以看到该节点。



然后在 hypervisor 启动 hlos 之前会创建很多节点，包括 hh-watchdog 节点，同时还会把 msm-watchdog 节点的 status 属性设置为 disabled 表示停用该设备。之后在 abl 跳转到 kernel 之前会把 hypvisor 创建的设备树节点叠加到 android 编译生成的设备树上。这样最终 kernel 加载的设备树就会包含 hh-watchdog 节点，并且停用了 msm-watchdog 设备节点。接下来在把 device_node 转化成 platform_device 的时候就只有 hh-watchdog 而没有 msm-watchdog。后面在 msm-watchdog 驱动加载的时候会因为没有匹配到设备而不会执行 probe 回调函数，msm-watchdog 驱动相关的线程也就不会创建。而 hh-watchdog 驱动可以匹配到 hh-watchdog 设备，并执行 hh-watchdog 驱动的 probe 回调函数 hh_wdt_probe 创建 hh-watchdog 线程并对驱动进行初始化。