Device Tree（二）：基本概念

作者：linuxer 发布于：2014-5-30 16:47 分类：统一设备模型

一、前言

一些背景知识（例如：为何要引入Device Tree，这个机制是用来解决什么问题的）请参考引入Device Tree的原因，本文主要是介绍Device Tree的基础概念。

简单的说，如果要使用Device Tree，首先用户要了解自己的硬件配置和系统运行参数，并把这些信息组织成Device Tree source file。通过DTC（Device Tree Compiler），可以将这些适合人类阅读的Device Tree source file变成适合机器处理的Device Tree binary file（有一个更好听的名字，DTB，device tree blob）。在系统启动的时候，boot program（例如：firmware、bootloader）可以将保存在flash中的DTB copy到内存（当然也可以通过其他方式，例如可以通过bootloader的交互式命令加载DTB，或者firmware可以探测到device的信息，组织成DTB保存在内存中），并把DTB的起始地址传递给client program（例如OS kernel，bootloader或者其他特殊功能的程序）。对于计算机系统（computer system），一般是firmware->bootloader->OS，对于嵌入式系统，一般是bootloader->OS。

本文主要描述下面两个主题：

1、Device Tree source file语法介绍

2、Device Tree binaryfile格式介绍

二、Device Tree的结构

在描述Device Tree的结构之前，我们先问一个基础问题：是否Device Tree要描述系统中的所有硬件信息？答案是否定的。基本上，那些可以动态探测到的设备是不需要描述的，例如USB device。不过对于SOC上的usb host controller，它是无法动态识别的，需要在device tree中描述。同样的道理，在computer system中，PCI device可以被动态探测到，不需要在device tree中描述，但是PCI bridge如果不能被探测，那么就需要描述之。

为了了解Device Tree的结构，我们首先给出一个Device Tree的示例：

```
/ o device-tree
            |- name = "device-tree"
            |- model = "MyBoardName"
            |- compatible = "MyBoardFamilyName"
            |- #address-cells = <2>
            |- #size-cells = <2>
            |- linux,phandle = <0>
            |
            o cpus
            | | - name = "cpus"
            | | - linux,phandle = <1>
            | | - #address-cells = <1>
            | | - #size-cells = <0>
            | |
            | o PowerPC,970@0
            |      |- name = "PowerPC,970"
            |      |- device_type = "cpu"
            |      |- reg = <0>
            |      |- clock-frequency = <0x5f5e1000>
            |      |- 64-bit
            |      |- linux,phandle = <2>
            |
          o memory@0
            | |- name = "memory"
            | |- device_type = "memory"
            | |- reg = <0x00000000 0x00000000 0x00000000 0x20000000>
            | |- linux,phandle = <3>
            |
          o chosen
              |- name = "chosen"
              |- bootargs = "root=/dev/sda2"
              |- linux,phandle = <4>
```

从上图中可以看出，device tree的基本单元是node。这些node被组织成树状结构，除了root node，每个node都只有一个parent。一个device tree文件中只能有一个root node。每个node中包含了若干的property/value来描述该node的一些特性。每个node用节点名字（node name）标识，节点名字的格式是node-name@unit-address。如果该node没有reg属性（后面会描述这个property），那么该节点名字中必须不能包括@和unit-address。unit-address的具体格式是和设备挂在那个bus上相关。例如对于cpu，其unit-address就是从0开始编址，以此加一。而具体的设备，例如以太网控制器，其unit-address就是寄存器地址。root node的node name是确定的，必须是“/”。

在一个树状结构的device tree中，如何引用一个node呢？要想唯一指定一个node必须使用full path，例如/node-name-1/node-name-2/node-name-N。在上面的例子中，cpu node我们可以通过/cpus/PowerPC,970@0访问。

属性（property）值标识了设备的特性，它的值（value）是多种多样的：

1、可能是空，也就是没有值的定义。例如上图中的64-bit ，这个属性没有赋值。

2、可能是一个u32、u64的数值（值得一提的是cell这个术语，在Device Tree表示32bit的信息单位）。例如#address-cells = <1> 。当然，可能是一个数组。例如<0x00000000 0x00000000 0x00000000 0x20000000>

4、可能是一个字符串。例如device_type = "memory" ，当然也可能是一个string list。例如"PowerPC,970"

三、Device Tree source file语法介绍

了解了基本的device tree的结构后，我们总要把这些结构体现在device tree source code上来。在linux kernel中，扩展名是dts的文件就是描述硬件信息的device tree source file，在dts文件中，一个node被定义成：

```
[label:] node-name[@unit-address] {
     [properties definitions]
     [child nodes]
}
```

“[]”表示option，因此可以定义一个只有node name的空节点。label方便在dts文件中引用，具体后面会描述。child node的格式和node是完全一样的，因此，一个dts文件中就是若干嵌套组成的node，property以及child note、child note property描述。

考虑到空泛的谈比较枯燥，我们用实例来讲解Device Tree Source file 的数据格式。假设蜗窝科技制作了一个S3C2416的开发板，我们把该development board命名为snail，那么需要撰写一个s3c2416-snail.dts的文件。如果把所有的开发板的硬件信息（SOC以及外设）都描述在一个文件中是不合理的，因此有可能其他公司也使用S3C2416搭建自己的开发板并命令pig、cow什么的，如果大家都用自己的dts文件描述硬件，那么其中大部分是重复的，因此我们把和S3C2416相关的硬件描述保存成一个单独的dts文件可以供使用S3C2416的target board来引用并将文件的扩展名变成dtsi（i表示include）。同理，三星公司的S3C24xx系列是一个SOC family，这些SOCs（2410、2416、2450等）也有相同的内容，因此同样的道理，我们可以将公共部分抽取出来，变成s3c24xx.dtsi，方便大家include。同样的道理，各家ARM vendor也会共用一些硬件定义信息，这个文件就是skeleton.dtsi。我们自下而上（类似C＋＋中的从基类到顶层的派生类）逐个进行分析。

1、skeleton.dtsi。位于linux-3.14\arch\arm\boot\dts目录下，具体该文件的内容如下：

```
/ {
        #address-cells = <1>;
        #size-cells = <1>;
        chosen { };
        aliases { };
        memory { device_type = "memory"; reg = <0 0>; };
};
```

device tree顾名思义是一个树状的结构，既然是树，必然有根。“/”是根节点的node name。“{”和“}”之间的内容是该节点的具体的定义，其内容包括各种属性的定义以及child node的定义。chosen、aliases和memory都是sub node，sub node的结构和root node是完全一样的，因此，sub node也有自己的属性和它自己的sub node，最终形成了一个树状的device tree。属性的定义采用property ＝ value的形式。例如#address-cells和#size-cells就是property，而<1>就是value。value有三种情况：

1）属性值是text string或者string list，用双引号表示。例如device_type = "memory"

2）属性值是32bit unsigned integers，用尖括号表示。例如#size-cells = <1>

3）属性值是binary data，用方括号表示。例如binary-property = [0x01 0x23 0x45 0x67]

如果一个device node中包含了有寻址需求（要定义reg property）的sub node（后文也许会用child node，和sub node是一样的意思），那么就必须要定义这两个属性。“#”是number的意思，#address-cells这个属性是用来描述sub node中的reg属性的地址域特性的，也就是说需要用多少个u32的cell来描述该地址域。同理可以推断#size-cells的含义，下面对reg的描述中会给出更详细的信息。

chosen node主要用来描述由系统firmware指定的runtime parameter。如果存在chosen这个node，其parent node必须是名字是“/”的根节点。原来通过tag list传递的一些linux kernel的运行时参数可以通过Device Tree传递。例如command line可以通过bootargs这个property这个属性传递；initrd的开始地址也可以通过linux,initrd-start这个property这个属性传递。在本例中，chosen节点是空的，在实际中，建议增加一个bootargs的属性，例如：

> "root=/dev/nfs nfsroot=1.1.1.1:/nfsboot ip=1.1.1.2:1.1.1.1:1.1.1.1:255.255.255.0::usbd0:off console=ttyS0,115200 mem=64M@0x30000000"

通过该command line可以控制内核从usbnet启动，当然，具体项目要相应修改command line以便适应不同的需求。我们知道，device tree用于HW platform识别，runtime parameter传递以及硬件设备描述。chosen节点并没有描述任何硬件设备节点的信息，它只是传递了runtime parameter。

aliases 节点定义了一些别名。为何要定义这个node呢？因为Device tree是树状结构，当要引用一个node的时候要指明相对于root node的full path，例如/node-name-1/node-name-2/node-name-N。如果多次引用，每次都要写这么复杂的字符串多少是有些麻烦，因此可以在aliases 节点定义一些设备节点full path的缩写。skeleton.dtsi中没有定义aliases，下面的section中会进一步用具体的例子描述之。

memory device node是所有设备树文件的必备节点，它定义了系统物理内存的layout。device_type属性定义了该node的设备类型，例如cpu、serial等。对于memory node，其device_type必须等于memory。reg属性定义了访问该device node的地址信息，该属性的值被解析成任意长度的（address，size）数组，具体用多长的数据来表示address和size是在其parent node中定义（#address-cells和#size-cells）。对于device node，reg描述了memory-mapped IO register的offset和length。对于memory node，定义了该memory的起始地址和长度。

本例中的物理内存的布局并没有通过memory node传递，其实我们可以使用command line传递，我们command line中的参数“mem=64M@0x30000000”已经给出了具体的信息。我们用另外一个例子来加深对本节描述的各个属性以及memory node的理解。假设我们的系统是64bit的，physical memory分成两段，定义如下：

RAM: starting address 0x0, length 0x80000000 (2GB)
RAM: starting address 0x100000000, length 0x100000000 (4GB)

对于这样的系统，我们可以将root node中的#address-cells和#size-cells这两个属性值设定为2，可以用下面两种方法来描述物理内存：

> 方法1：
>
> ```
> memory@0 {
>         device_type = "memory";
>         reg = <0x000000000 0x00000000 0x00000000 0x80000000
>                         0x000000001 0x00000000 0x00000001 0x00000000>;
> };
> ```
>
> 方法2：
>
> ```
> memory@0 {
>         device_type = "memory";
>         reg = <0x000000000 0x00000000 0x00000000 0x80000000>;
> };
>
> memory@100000000 {
>         device_type = "memory";
>         reg = <0x000000001 0x00000000 0x00000001 0x00000000>;
> };
> ```

2、s3c24xx.dtsi。位于linux-3.14\arch\arm\boot\dts目录下，具体该文件的内容如下（有些内容省略了，领会精神即可，不需要描述每一个硬件定义的细节）：

```
#include "skeleton.dtsi"

/ {
        compatible = "samsung,s3c24xx"; ———————————————————— (A)
        interrupt-parent = <&intc>; ———————————————————————— (B)

        aliases {
                pinctrl0 = &pinctrl_0; ———————————————————————— (C)
        };

        intc:interrupt-controller@4a000000 { ———————————————— (D)
                compatible = "samsung,s3c2410-irq";
                reg = <0x4a000000 0x100>;
                interrupt-controller;
                #interrupt-cells = <4>;
        };

        serial@50000000 { ———————————————————————————— (E)
                compatible = "samsung,s3c2410-uart";
                reg = <0x50000000 0x4000>;
```

```
                interrupts = <1 0 4 28>, <1 1 4 28>;
                status = "disabled";
        };

        pinctrl_0: pinctrl@56000000 {————————————————————— (F)
                reg = <0x56000000 0x1000>;

                wakeup-interrupt-controller {
                        compatible = "samsung,s3c2410-wakeup-eint";
                        interrupts = <0 0 0 3>,
                                     <0 0 1 3>,
                                     <0 0 2 3>,
                                     <0 0 3 3>,
                                     <0 0 4 4>,
                                     <0 0 5 4>;
                };
        };

……
};
```

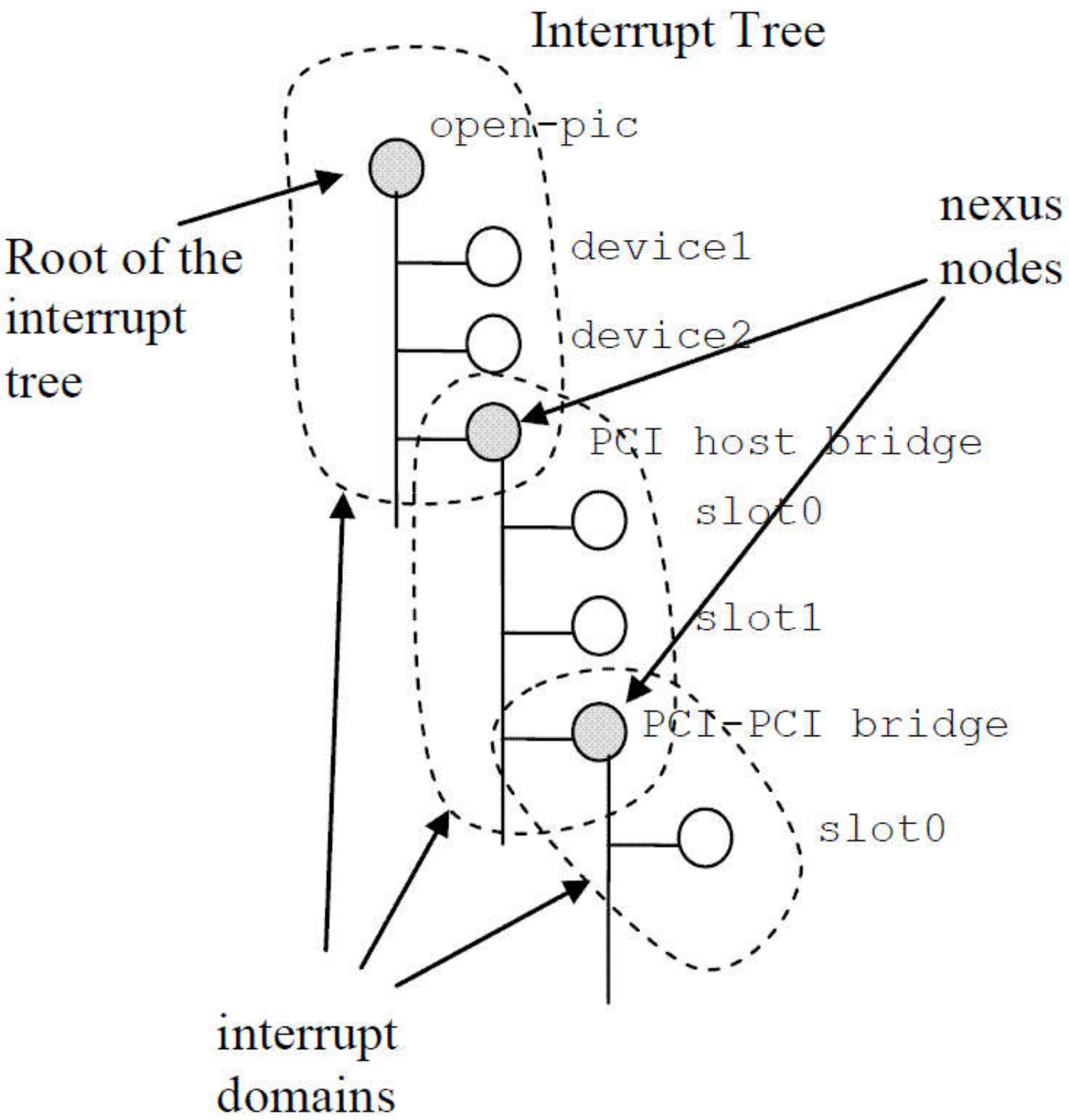
这个文件描述了三星公司的S3C24xx系列SOC family共同的硬件block信息。首先提出的问题就是：为何定义了两个根节点？按理说Device Tree只能有一个根节点，所有其他的节点都是派生于根节点的。我的猜测是这样的：Device Tree Compiler会对DTS的node进行合并，最终生成的DTB只有一个root node。OK，我们下面开始逐一分析：

（A）在描述compatible属性之前要先描述model属性。model属性指明了该设备属于哪个设备生产商的哪一个model。一般而言，我们会给model赋值“manufacturer,model”。例如model = "samsung,s3c24xx"。samsung是生产商，s3c24xx是model类型，指明了具体的是哪一个系列的SOC。OK，现在我们回到compatible属性，该属性的值是string list，定义了一系列的modle（每个string是一个model）。这些字符串列表被操作系统用来选择用哪一个driver来驱动该设备。假设定义该属性：compatible = “aaaaaa”，“bbbbb"。那么操作操作系统可能首先使用aaaaaa来匹配适合的driver，如果没有匹配到，那么使用字符串bbbbb来继续寻找适合的driver，对于本例，compatible = "samsung,s3c24xx"，这里只定义了一个modle而不是一个list。对于root node，compatible属性是用来匹配machine type的（在device tree代码分析文章中会给出更细致的描述）。对于普通的HW block的节点，例如interrupt-controller，compatible属性是用来匹配适合的driver的。

（B）具体各个HW block的interrupt source是如何物理的连接到interruptcontroller的呢？在dts文件中是用interrupt-parent这个属性来标识的。且慢，这里定义interrupt-parent属性的是root node，难道root node会产生中断到interrupt controller吗？当然不会，只不过如果一个能够产生中断的device node没有定义interrupt-parent的话，其interrupt-parent属性就是跟随parent node。因此，与其在所有的下游设备中定义interrupt-parent，不如统一在root node中定义了。

intc是一个lable，标识了一个device node（在本例中是标识了interrupt-controller@4a000000 这个device node）。实际上，interrupt-parent属性值应该是是一个u32的整数值（这个整数值在Device Tree的范围内唯一识别了一个device node，也就是phandle），不过，在dts文件中中，可以使用类似c语言的Labels and References机制。定义一个lable，唯一标识一个node或者property，后续可以使用&来引用这个lable。DTC会将lable转换成u32的整数值放入到DTB中，用户层面就不再关心具体转换的整数值了。

关于interrupt，我们值得进一步描述。在Device Tree中，有一个概念叫做interrupt tree，也就是说interrupt也是一个树状结构。我们以下图为例（该图来自Power_ePAPR_APPROVED_v1.1）：



系统中有一个interrrupt tree的根节点，device1、device2以及PCI host bridge的interrupt line都是连接到root interrupt controller的。PCI host bridge设备中有一些下游的设备，也会产生中断，但是他们的中断都是连接到PCI host bridge上的interrupt controller（术语叫做interrupt nexus），然后报告到root interrupt controller的。每个能产生中断的设备都可以产生一个或者多个interrupt，每个interrupt source（另外一个术语叫做interrupt specifier，描述了interrupt source的信息）都是限定在其所属的interrupt domain中。

在了解了上述的概念后，我们可以回头再看看interrupt-parent这个属性。其实这个属性是建立interrupt tree的关键属性。它指明了设备树中的各个device node如何路由interrupt event。另外，需要提醒的是interrupt controller也是可以级联的，上图中没有表示出来。那么在这种情况下如何定义interrupt tree的root呢？那个没有定义interrupt-parent的interrupt controller就是root。

（C）pinctrl0是一个缩写，他是/pinctrl@56000000的别名。这里同样也是使用了Labels and References机制。

（D）intc（node name是interrupt-controller@4a000000 ，我这里直接使用lable）是描述interrupt controller的device node。根据S3C24xx的datasheet，我们知道interrupt controller的寄存器地址从0x4a000000开始，长度为0x100（实际2451的interrupt的寄存器地址空间没有那么长，0x4a000074是最后一个寄存器），也就是reg属性定义的内容。interrupt-controller属性为空，只是用来标识该node是一个interrupt controller而不是interrupt nexus（interrupt nexus需要在不同的interrupt domains之间进行翻译，需要定义interrupt-map的属性，本文不涉及这部分的内容）。#interrupt-cells 和#address-cells概念是类似的，也就是说，用多少个u32来标识一个interrupt source。我们可以看到，在具体HW block的interrupt定义中都是用了4个u32来表示，例如串口的中断是这样定义的：

```
interrupts = <1 0 4 28>, <1 1 4 28>;
```

（E） 从reg属性可以serial controller寄存器地址从0x50000000 开始，长度为0x4000。对于一个能产生中断的设备，必须定义interrupts这个属性。也可以定义interrupt-parent这个属性，如果不定义，则继承其parent node的interrupt-parent属性。 对于interrupt属性值，各个interrupt controller定义是不一样的，有的用3个u32表示，有的用4个。具体上面的各个数字的解释权归相关的interrupt controller所有。对于中断属性的具体值的描述我们会在device tree的第三份文档—代码分析中描述。

（F）这个node是描述GPIO控制的。这个节点定义了一个wakeup-interrupt-controller 的子节点，用来描述有唤醒功能的中断源。

3、s3c2416.dtsi。位于linux-3.14\arch\arm\boot\dts目录下，具体该文件的内容如下（有些内容省略了，领会精神即可，不需要描述每一个硬件定义的细节）：

```
#include "s3c24xx.dtsi"
#include "s3c2416-pinctrl.dtsi"

/ {
        model = "Samsung S3C2416 SoC";
        compatible = "samsung,s3c2416"; ————————————————————A

        cpus { ————————————————————————————————————B
                #address-cells = <1>;
                #size-cells = <0>;

                cpu {
                        compatible = "arm,arm926ejs";
                };
        };

        interrupt-controller@4a000000 { ————————————————————C
                compatible = "samsung,s3c2416-irq";
        };

……

};
```

（A）在s3c24xx.dtsi文件中已经定义了compatible这个属性，在s3c2416.dtsi中重复定义了这个属性，一个node不可能有相同名字的属性，具体如何处理就交给DTC了。经过反编译，可以看出，DTC是丢弃掉了前一个定义。因此，到目前为止，compatible = samsung,s3c2416。在s3c24xx.dtsi文件中定义了compatible的属性值被覆盖了。

（B）对于根节点，必须有一个cpus的child node来描述系统中的CPU信息。对于CPU的编址我们用一个u32整数就可以描述了，因此，对于cpus node，#address-cells 是1，而#size-cells是0。其实CPU的node可以定义很多属性，例如TLB，cache、频率信息什么的，不过对于ARM，这里只是定义了compatible属性就OK了，arm926ejs包括了所有的processor相关的信息。

（C）s3c24xx.dtsi文件和s3c2416.dtsi中都有interrupt-controller@4a000000这个node，DTC会对这两个node进行合并，最终编译的结果如下：

```
interrupt-controller@4a000000 {
                compatible = "samsung,s3c2416-irq";
                reg = <0x4a000000 0x100>;
                interrupt-controller;
                #interrupt-cells = <0x4>;
                linux,phandle = <0x1>;
                phandle = <0x1>;
        };
```

4、s3c2416-pinctrl.dtsi

这个文件定义了pinctrl@56000000 这个节点的若干child node，主要用来描述GPIO的bank信息。

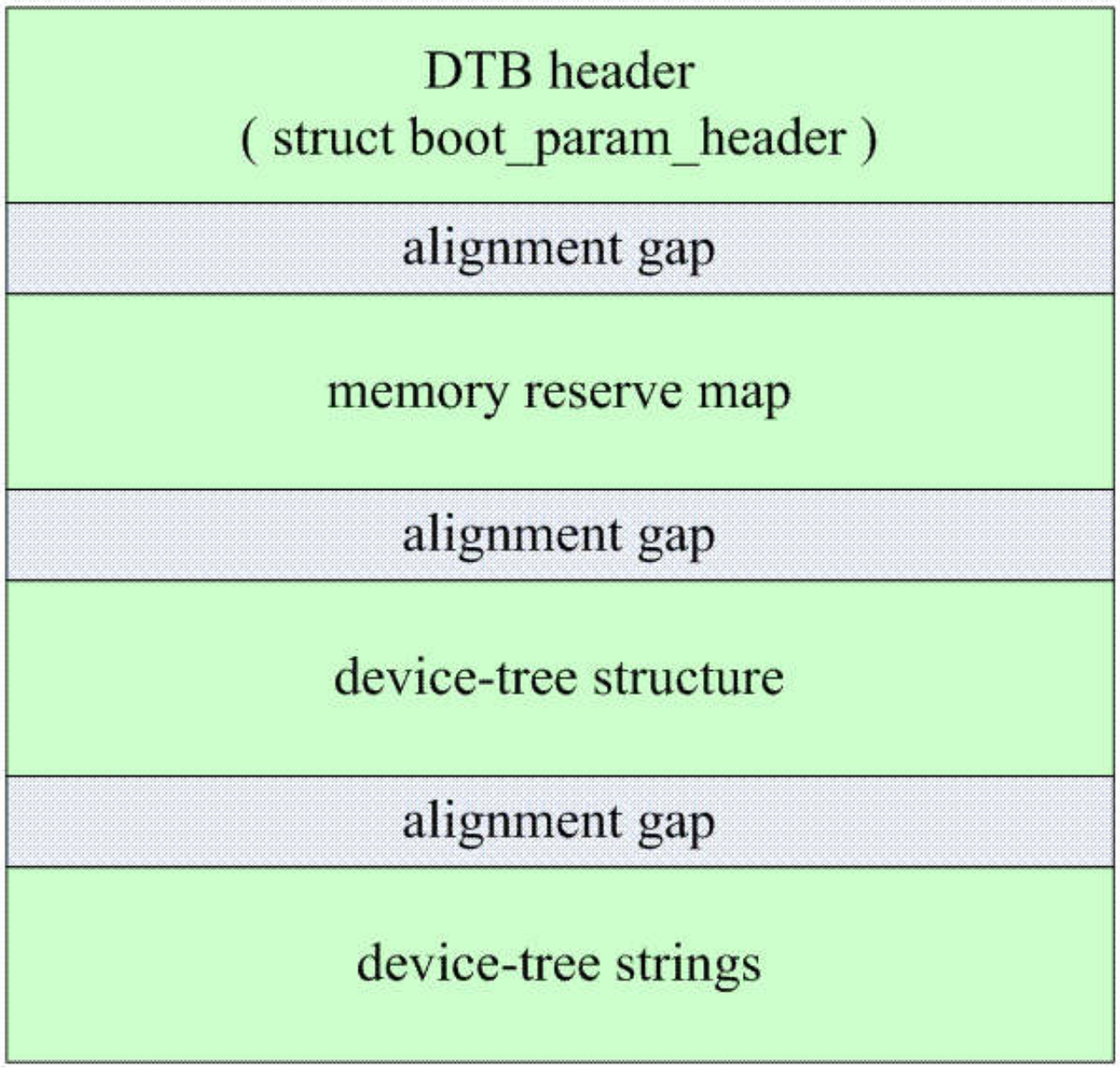
5、s3c2416-snail.dts

这个文件应该定义一些SOC之外的peripherals的定义。

四、Device Tree binary格式

1、DTB整体结构

经过Device Tree Compiler编译，Device Tree source file变成了Device Tree Blob（又称作flattened device tree）的格式。Device Tree Blob的数据组织如下图所示：



2、DTB header。

对于DTB header，其各个成员解释如下：

header field name	description
magic	用来识别DTB的。通过这个magic，kernel可以确定bootloader传递的参数block是一个DTB还是tag list。
totalsize	DTB的total size
off_dt_struct	device tree structure block的offset
off_dt_strings	device tree strings block的offset
off_mem_rsvmap	offset to memory reserve map。有些系统，我们也许会保留一些memory有特殊用途（例如DTB或者initrd image），或者在有些DSP+ARM的SOC platform上，有写memory被保留用于ARM和DSP进行信息交互。这些保留内存不会进入内存管理系统。
version	该DTB的版本。
last_comp_version	兼容版本信息
boot_cpuid_phys	我们在哪一个CPU（用ID标识）上booting
dt_strings_size	device tree strings block的size。和off_dt_strings一起确定了strings block在内存中的位置
dt_struct_size	device tree structure block的size。和和off_dt_struct一起确定了device tree structure block在内存中的位置

3、 memory reserve map的格式描述

这个区域包括了若干的reserve memory描述符。每个reserve memory描述符是由address和size组成。其中address和size都是用U64来描述。

4、device tree structure block的格式描述

device tree structure block区域是由若干的分片组成，每个分片开始位置都是保存了token，以此来描述该分片的属性和内容。共计有5种token：

（1）FDT_BEGIN_NODE (0x00000001)。该token描述了一个node的开始位置，紧挨着该token的就是node name（包括unit address）

（2）FDT_END_NODE (0x00000002)。该token描述了一个node的结束位置。

（3）FDT_PROP (0x00000003)。该token描述了一个property的开始位置，该token之后是两个u32的数据，分别是length和name offset。length表示该property value data的size。name offset表示该属性字符串在device tree strings block的偏移值。length和name offset之后就是长度为length具体的属性值数据。

（4）FDT_NOP (0x00000004)。

（5）FDT_END (0x00000009)。该token标识了一个DTB的结束位置。

一个可能的DTB的结构如下：

（1）若干个FDT_NOP（可选）

（2）FDT_BEGIN_NODE

node name

paddings

（3）若干属性定义。

（4）若干子节点定义。（被FDT_BEGIN_NODE和FDT_END_NODE包围）

（5）若干个FDT_NOP（可选）

（6）FDT_END_NODE

（7）FDT_END

5、device tree strings bloc的格式描述

device tree strings bloc定义了各个node中使用的属性的字符串表。由于很多属性会出现在多个node中，因此，所有的属性字符串组成了一个string block。这样可以压缩DTB的size。

标签: Device tree