# 语法

参考：<http://www.wowotech.net/device_model/dt_basic_concept.html>

其中，需要特别提醒的有如下几点：

## 通用语法模型

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

关于属性定义：

* 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：
  + 属性值是text string或者string list，用**双引号**表示。例如device\_type = "memory"。
  + 属性值是32bit unsigned integers，用**尖括号**表示。例如#size-cells = <1>
  + 属性值是binary data，用**方括号**表示。例如binary-property = [0x01 0x23 0x45 0x67]。

## 关于memory node

|  |
| --- |
| / {  #address-cells = <2>;  #size-cells = <2>;  chosen { };  aliases { };  memory@0 {  device\_type = "memory";  reg = <0x000000000 0x00000000 0x00000000 0x80000000  0x000000001 0x00000000 0x00000001 0x00000000>;  }; |

主要关注上面的memory node，它描述的是64位系统下的2段物理内存：

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

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

说明如下：

* memory node是设备树必备节点。
* device\_type的名字固定的，必须是memory。
* reg定义了该device node（即memory node）地址信息。
* 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。
* #address-cells = <2>;#size-cells = <2>; 这两句话中的<>号中的数字，表示的是多少个32bit的数，如果为2，表示2个32bit数。由于64位系统中，address和size都是64位的数，因此这里需要用到2个32bit的数来表示一个64bit的数。因此，<>里面的数值填2。
* 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。

## 关于chosen node

|  |
| --- |
| / {  #address-cells = <2>;  #size-cells = <2>;  chosen { };  aliases { };  memory@0 {  device\_type = "memory";  reg = <0x000000000 0x00000000 0x00000000 0x80000000  0x000000001 0x00000000 0x00000001 0x00000000>;  }; |

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：

|  |
| --- |
| bootargs = "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。

## 关于aliases node

|  |
| --- |
| / {  #address-cells = <2>;  #size-cells = <2>;  chosen { };  aliases { };  memory@0 {  device\_type = "memory";  reg = <0x000000000 0x00000000 0x00000000 0x80000000  0x000000001 0x00000000 0x00000001 0x00000000>;  }; |

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。比如

|  |
| --- |
| aliases {  pinctrl0 = &pinctrl\_0;  }; |

## 关于多个根节点

每一个文件似乎都定义了一个根节点，那么为何定义了多个个根节点？

答：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。

## 关于多个相同的compatible属性

处于同一级(比如根节点)下的不同文件有可能都会定义compatible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。类似于C语言的#define 宏定义。比如s3c24xx.dtsi有如下定义：

|  |
| --- |
| #include "skeleton.dtsi"  / {  compatible = "samsung,s3c24xx";  interrupt-parent = <&intc>;  aliases {  pinctrl0 = &pinctrl\_0;  };  …  } |

而在s3c2416.dtsi有如下定义：

|  |
| --- |
| #include "s3c24xx.dtsi"  #include "s3c2416-pinctrl.dtsi"  / {  model = "Samsung S3C2416 SoC";  compatible = "samsung,s3c2416";  ...  } |

最终，compatible的属性为：samsung,s3c2416

## 关于多个相同的node

多个不同文件，可能出现相同的node的定义。DTC的处理方式是会对这两个node进行合并。比如s3c24xx.dtsi有如下定义：

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

而在s3c2416.dtsi有如下定义：

|  |
| --- |
| #include "s3c24xx.dtsi"  #include "s3c2416-pinctrl.dtsi"  / {  interrupt-controller@4a000000 {  compatible = "samsung,s3c2416-irq";  };  ...  } |

合并后为：

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

## 关于label

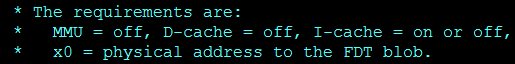
|  |
| --- |
| #include "skeleton.dtsi"  / {  compatible = "samsung,s3c24xx";  interrupt-parent = <&intc>;  aliases {  pinctrl0 = &pinctrl\_0;  };  intc:interrupt-controller@4a000000 {  compatible = "samsung,s3c2410-irq";  reg = <0x4a000000 0x100>;  interrupt-controller;  #interrupt-cells = <4>;  };  serial@50000000 {  compatible = "samsung,s3c2410-uart";  reg = <0x50000000 0x4000>;  interrupts = <1 0 4 28>, <1 1 4 28>;  status = "disabled";  };  pinctrl\_0: pinctrl@56000000 {  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>;  };  };  ……  }; |

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。

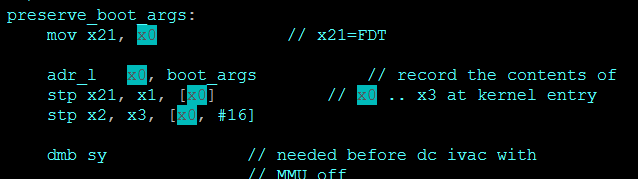
# dtb初始化流程

## 初始化流程

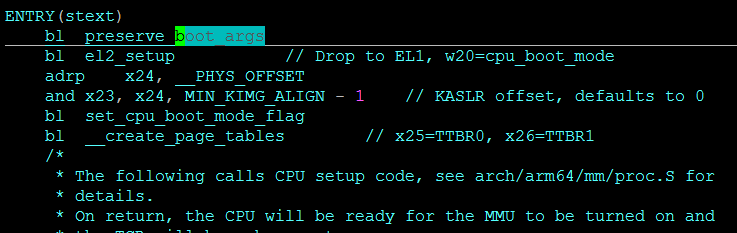
1. kernel/arch/arm64/kernel/head.S



说明x0存放的是FDT(Flattened Device Tree).

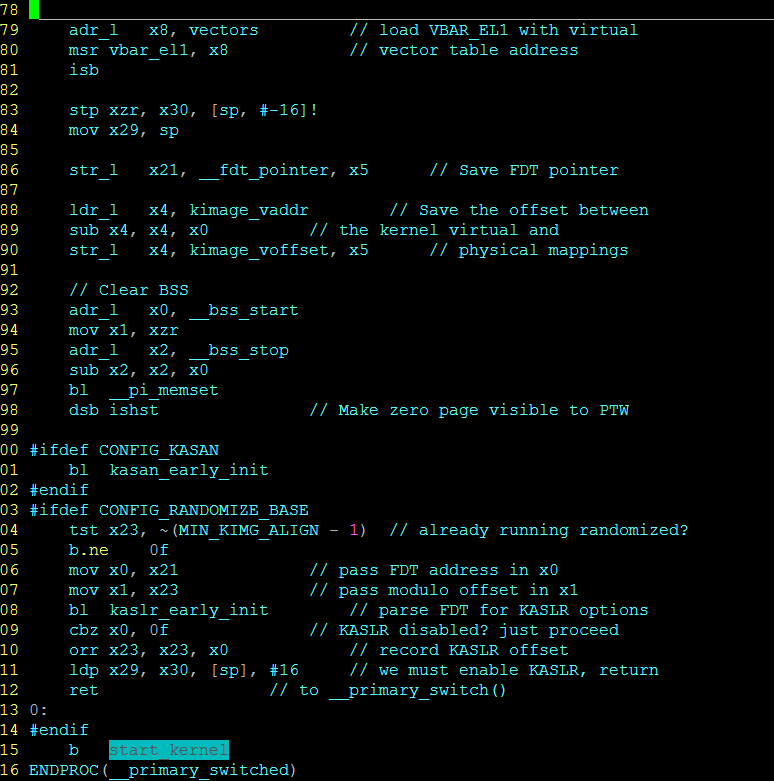


说明，x0是由bootloader传进来的。



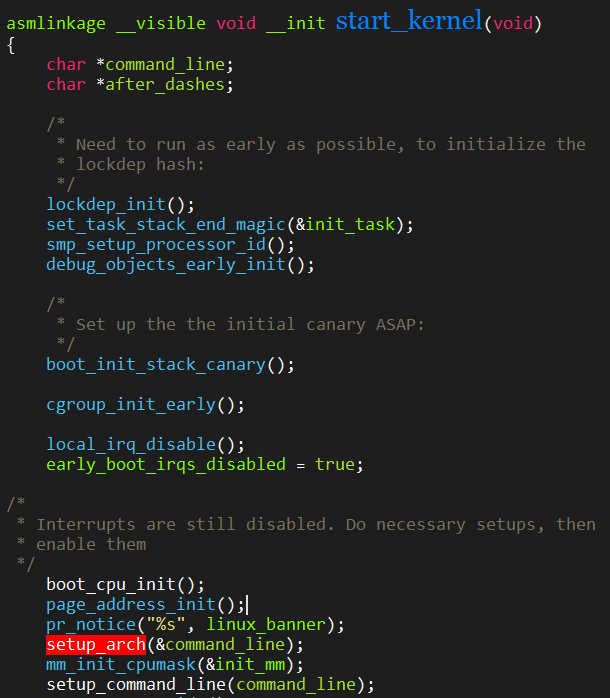
entry一进来，就先保存启动参数。

最后，呼叫start\_kernel

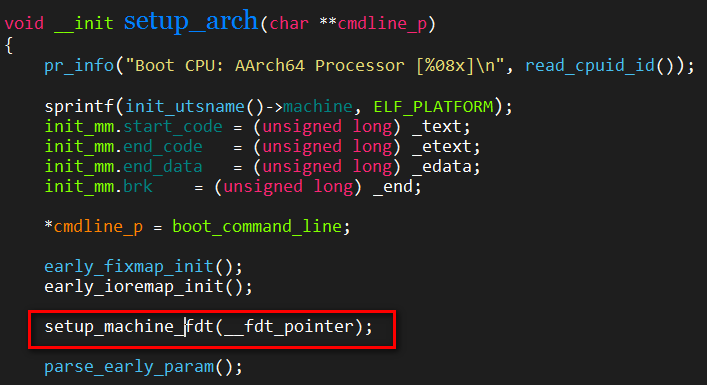


1. kernel/init/main.c

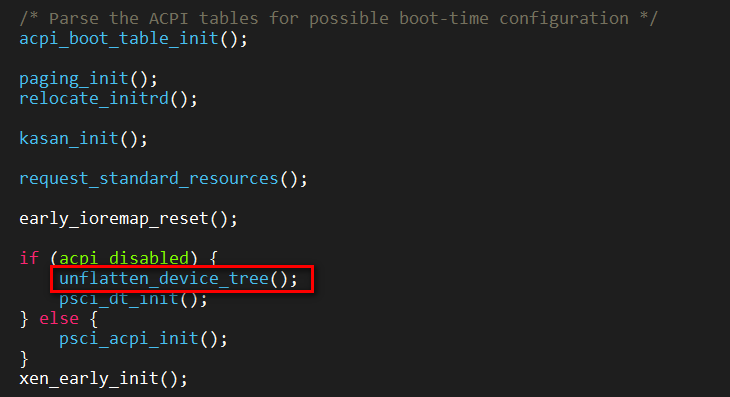
start\_kernel呼叫setup\_arch



1. kernel/arch/arm64/kernel/setup.c



上面的函数实现参数的传递及platform的识别等



上面的代码实现将dtb转换为树状的结构。

## driver probe与dtb关系

驱动被调用的顺序？驱动如何保证与他有dependency关系的模块被提前调用？内核如何展开平台设备？

关于前两个问题，解答如下：

* 在内核insmod的时候，内核会对里面的symbol做检查，如果你的模块里面用到的某个symbol在内核里面不存在，那么就会无法加载成功。这里就做好了dependency的check。
* 如果是与内核一起编译的内核模块，就不存在dependency关系，因为所有的symbol都在内核中。不会存在一个模块访问另一个模块，找不到symbol的情况。而对于动态加载的模块，则存在dependency关系，如果A模块depend on B模块，那么B模块必须先加载，否则A模块无法加载成功。

最后一个问题：

* 首先，系统在初始化的时候，解析dtb。并将dtb的结构存放在device node的结构体里面。
* 然后，会根据每一个device node，创建一个device结构体，并把它挂到platform bus上面。但是，并不是每个device node都会挂到platform bus上，比如cpus node，memory node，choose node等
* 当driver 初始化时(insmod)，就会根据创建好的platform device，去匹配驱动，如果匹配成功，probe函数就会被调用。

参考：<http://www.wowotech.net/linux_kenrel/dt-code-analysis.html>

# kernel提供的读取dtb的函数

|  |  |
| --- | --- |
| 函数 | 说明 |
| of\_match\_node | 判断是否与输入的match\_table匹配。并返回struct of\_device\_id \*或者NULL。 |
| of\_property\_read\_string | 读取property value为string的字符串 |
| of\_property\_count\_elems\_of\_size | 输入device node，property name，以及单个元素的size，返回共有多少个元素，适合property为<aa bb cc dd> |
| of\_property\_read\_u32\_array | 输入device node, property name，以及数组大小，返回u32数组。适合property为<aa bb cc dd> |
| of\_address\_to\_resource | 输入device node，自动读取property为”reg”的值，并转换为resource。 |
| of\_property\_read\_u32\_index | 输入device node和property name，返回32bit property value一个。 |
| syscon\_regmap\_lookup\_by\_phandle | 输入property name，以及device node，返回regmap。这个函数适用于property value是一个<&xxx>形式的引用。 |
| devm\_regulator\_get | 获取regulator电压/流校准器。 |

# reference

* <http://blog.csdn.net/zengxianyang/article/details/50732929>
* <http://www.wowotech.net/linux_kenrel/why-dt.html>
* <http://www.wowotech.net/device_model/dt_basic_concept.html>
* <http://www.wowotech.net/device_model/dt-code-analysis.html>
* <http://elinux.org/Device_Tree_Usage>