**Device Tree（一）：背景介绍**

一、前言

作为一个多年耕耘在linux 2.6.23内核的开发者，各个不同项目中各种不同周边外设驱动的开发以及各种琐碎的、扯皮的俗务占据了大部分的时间。当有机会下载3.14的内核并准备学习的时候，突然发现linux kernel对于我似乎变得非常的陌生了，各种新的机制，各种framework、各种新的概念让我感到阅读内核代码变得举步维艰。 还好，剖析内核的热情还在，剩下的就交给时间的。首先进入视线的是Device Tree机制，这是和porting内核非常相关的机制，如果想让将我们的硬件平台迁移到高版本的内核上，Device Tree是一个必须要扫清的障碍。

我想从下面三个方面来了解Device Tree：

1、为何要引入Device Tree，这个机制是用来解决什么问题的？（这是本文的主题）

2、Device Tree的基础概念（请参考[DT基础概念](http://www.wowotech.net/linux_kenrel/dt_basic_concept.html" \t "_blank)）

3、ARM linux中和Device Tree相关的代码分析（请参考[DT代码分析](http://www.wowotech.net/linux_kenrel/dt-code-analysis.html" \t "_blank)）

阅读linux内核代码就像欣赏冰山，有看得到的美景（各种内核机制及其代码），也有埋在水面之下看不到的基础（机制背后的源由和目的）。沉醉于各种内核机制的代码固然有无限乐趣，但更重要的是注入更多的思考，思考其背后的机理，真正理解软件抽象。这样才能举一反三，并应用在具体的工作和生活中。

本文主要从下面几个方面阐述为何ARM linux会引入Device Tree：

1、没有Device Tree的ARM linux是如何运转的？

2、混乱的ARM architecture代码和存在的问题

3、新内核的解决之道

二、没有Device Tree的ARM linux是如何运转的？

我曾经porting内核到两个ARM-based的平台上。一个是小的芯片公司的应用处理器，公司自己购买了CPU core，该CPU core使用ARM兼容的指令集（但不是ARM）加上各种公司自行设计的多媒体外设整合成公司的产品进行销售。而我的任务就是porting 2.4.18内核到该平台上。在黑白屏幕的手机时代，那颗AP（application process）支持了彩屏、camera、JPEG硬件加速、2D/3D加速、MMC/SD卡、各种音频加速（内置DSP）等等特性，功能强大到无法直视。另外一次移植经历是让2.6.23内核跑在一个大公司的冷门BP（baseband processor）上。具体porting的方法是很简单的：

1、自己撰写一个bootloader并传递适当的参数给kernel。除了传统的command line以及tag list之类的，最重要的是申请一个machine type，当拿到属于自己项目的machine type ID的时候，当时心情雀跃，似乎自己已经是开源社区的一份子了（其实当时是有意愿，或者说有目标是想将大家的代码并入到linux kernel main line的）。

2、在内核的arch/arm目录下建立mach-xxx目录，这个目录下，放入该SOC的相关代码，例如中断controller的代码，时间相关的代码，内存映射，睡眠相关的代码等等。此外，最重要的是建立一个board specific文件，定义一个machine的宏：

MACHINE\_START(project name, "xxx公司的xxx硬件平台")   
    .phys\_io    = 0x40000000,   
    .boot\_params    = 0xa0000100,     
    .io\_pg\_offst    = (io\_p2v(0x40000000) >> 18) & 0xfffc,   
    .map\_io        = xxx\_map\_io,   
    .init\_irq    = xxx\_init\_irq,   
    .timer        = &xxx\_timer,   
    .init\_machine    = xxx\_init,   
MACHINE\_END

在xxx\_init函数中，一般会加入很多的platform device。因此，伴随这个board specific文件中是大量的静态table，描述了各种硬件设备信息。

3、调通了system level的driver（timer，中断处理，clock等）以及串口terminal之后，linux kernel基本是可以起来了，后续各种driver不断的添加，直到系统软件支持所有的硬件。

综上所述，在linux kernel中支持一个SOC平台其实是非常简单的，让linux kernel在一个特定的平台上“跑”起来也是非常简单的，问题的重点是如何优雅的”跑”。

三、混乱的ARM architecture代码和存在的问题

每次正式的linux kernel release之后都会有两周的merge window，在这个窗口期间，kernel各个部分的维护者都会提交各自的patch，将自己测试稳定的代码请求并入kernel main line。每到这个时候，Linus就会比较繁忙，他需要从各个内核维护者的分支上取得最新代码并merge到自己的kernel source tree中。Tony Lindgren，内核OMAP development tree的维护者，发送了一个邮件给Linus，请求提交OMAP平台代码修改，并给出了一些细节描述：

1、简单介绍本次改动

2、关于如何解决merge conficts。有些git mergetool就可以处理，不能处理的，给出了详细介绍和解决方案

一切都很平常，也给出了足够的信息，然而，正是这个pull request引发了一场针对ARM linux的内核代码的争论。我相信Linus一定是对ARM相关的代码早就不爽了，ARM的merge工作量较大倒在其次，主要是他认为ARM很多的代码都是垃圾，代码里面有若干愚蠢的table，而多个人在维护这个table，从而导致了冲突。因此，在处理完OMAP的pull request之后（Linus并非针对OMAP平台，只是Tony Lindgren撞在枪口上了），他发出了怒吼：

Gaah. Guys, this whole ARM thing is a f\*cking pain in the ass.

负责ARM linux开发的Russell King脸上挂不住，进行了反驳：事情没有那么严重，这次的merge conficts就是OMAP和IMX/MXC之间一点协调的问题，不能抹杀整个ARM linux团队的努力。其他的各个ARM平台维护者也加入讨论：ARM平台如何复杂，如何庞大，对于arm linux code我们已经有一些思考，正在进行中……一时间，讨论的气氛有些尖锐，但总体是坦诚和友好的。

对于一件事情，不同层次的人有不同层次的思考。这次争论涉及的人包括：

1、内核维护者（CPU体系结构无关的代码）

2、维护ARM系统结构代码的人

3、维护ARM sub architecture的人（来自各个ARM SOC vendor）

维护ARM sub architecture的人并没有强烈的使命感，作为公司的一员，他们最大的目标是以最快的速度支持自己公司的SOC，尽快的占领市场。这些人的软件功力未必强，对linux kernel的理解未必深入（有些人可能很强，但是人在江湖身不由己）。在这样的情况下，很多SOC specific的代码都是通过copy and paste，然后稍加修改代码就提交了。此外，各个ARM vendor的SOC family是一长串的CPU list，每个CPU多多少少有些不同，这时候＃ifdef就充斥了各个源代码中，让ARM mach-和plat-目录下的代码有些不忍直视。

作为维护ARM体系结构的人，其能力不容置疑。以Russell King为首的team很好的维护了ARM体系结构的代码。基本上，除了mach-和plat-目录，其他的目录中的代码和目录组织是很好的。作为ARM linux的维护者，维护一个不断有新的SOC加入的CPU architecture code的确是一个挑战。在Intel X86的架构一统天下的时候，任何想正面攻击Intel的对手都败下阵来。想要击倒巨人（或者说想要和巨人并存）必须另辟蹊径。ARM的策略有两个，一个是focus在嵌入式应用上，也就意味着要求低功耗，同时也避免了和Intel的正面对抗。另外一个就是博采众家之长，采用license IP的方式，让更多的厂商加入ARM建立的生态系统。毫无疑问，ARM公司是成功的，但是这种模式也给ARM linux的维护者带来了噩梦。越来越多的芯片厂商加入ARM阵营，越来越多的ARM platform相关的代码被加入到内核，不同厂商的周边HW block设计又各不相同……

内核维护者是真正对操作系统内核软件有深入理解的人，他们往往能站在更高的层次上去观察问题，发现问题。Linus注意到每次merge window中，ARM的代码变化大约占整个ARCH目录的60％，他认为这是一个很明显的符号，意味着ARM linux的代码可能存在问题。其实，60％这个比率的确很夸张，因为unicore32是在2.6.39 merge window中第一次全新提交，它的代码是全新的，但是其代码变化大约占整个ARCH目录的9.6％（需要提及的是unicore32是一个中国芯）。有些维护ARM linux的人认为这是CPU市场占用率的体现，不是问题，直到内核维护者贴出实际的代码并指出问题所在。内核维护者当然想linux kernel支持更多的硬件平台，但是他们更愿意为linux kernel制定更长远的规划。例如：对于各种繁杂的ARM平台，用一个kernel image来支持。

经过争论，确定的问题如下：

1、ARM linux缺少platform（各个ARM sub architecture，或者说各个SOC）之间的协调，导致arm linux的代码有重复。值得一提的是在本次争论之前，ARM维护者已经进行了不少相关的工作（例如PM和clock tree）来抽象相同的功能模块。

2、ARM linux中大量的board specific的源代码应该踢出kernel，否则这些垃圾代码和table会影响linux kernel的长期目标。

3、各个sub architecture的维护者直接提交给Linux并入主线的机制缺乏层次。

四、新内核的解决之道

针对ARM linux的现状，最需要解决的是人员问题，也就是如何整合ARM sub architecture（各个ARM Vendor）的资源。因此，内核社区成立了一个ARM sub architecture的team，该team主要负责协调各个ARM厂商的代码（not ARM core part），Russell King继续负责ARM core part的代码。此外，建立一个ARM platform consolidation tree。ARM sub architecture team负责review各个sub architecture维护者提交的代码，并在ARM platform consolidation tree上维护。在下一个merge window到来的时候，将patch发送给Linus。

针对重复的代码问题，如果不同的SOC使用了相同的IP block（例如I2C controller），那么这个driver的code要从各个arch/arm/mach-xxx中独立出来，变成一个通用的模块供各个SOC specific的模块使用。移动到哪个目录呢？对于I2C或者USB OTG而言，这些HW block的驱动当然应该移动到kernel/drivers目录。因为，对于这些外设，可能是in-chip，也可能是off-chip的，但是对于软件而言，它们是没有差别的（或者说好的软件抽象应该掩盖底层硬件的不同）。对于那些system level的code呢？例如clock control、interrupt control。其实这些也不是ARM-specific，应该属于linux kernel的核心代码，应该放到linux/kernel目录下，属于core-Linux-kernel frameworks。当然对于ARM平台，也需要保存一些和framework交互的code，这些code叫做ARM SoC core architecture code。OK，总结一下：

1、ARM的核心代码仍然保存在arch/arm目录下

2、ARM SoC core architecture code保存在arch/arm目录下

3、ARM SOC的周边外设模块的驱动保存在drivers目录下

4、ARM SOC的特定代码在arch/arm/mach-xxx目录下

5、ARM SOC board specific的代码被移除，由Device Tree机制来负责传递硬件拓扑和硬件资源信息。

OK，终于来到了Device Tree了。本质上，Device Tree改变了原来用hardcode方式将HW 配置信息嵌入到内核代码的方法，改用bootloader传递一个DB的形式。对于基于ARM CPU的嵌入式系统，我们习惯于针对每一个platform进行内核的编译。但是随着ARM在消费类电子上的广泛应用（甚至桌面系统、服务器系统），我们期望ARM能够象X86那样用一个kernel image来支持多个platform。在这种情况下，如果我们认为kernel是一个black box，那么其输入参数应该包括：

1、识别platform的信息

2、runtime的配置参数

3、设备的拓扑结构以及特性

对于嵌入式系统，在系统启动阶段，bootloader会加载内核并将控制权转交给内核，此外，还需要把上述的三个参数信息传递给kernel，以便kernel可以有较大的灵活性。在linux kernel中，Device Tree的设计目标就是如此。

**Device Tree（二）：基本概念**

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](mailto: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。

以全志A64为例

在linux-3.10/arch/arm64/boot/dts/目录中，如sun50iw1p1.dtsi。

 \*note:<name>[@<unit-address>]是节点的格式，其中unit-address是单位偏移地址，本人验证去掉[]内的内容依然可以运行，也没在source code找到必须存在的理由。

“/"代表根节点；

“model”是板的ID；

"compatible"是平台兼容，一般格式是"manufacturer,model"。内核或者uboot依靠这个属性找到相对应driver，若"compatible"出现多个属性，按序匹配driver；

“#address-cells”是address的单位(32bit)；

“#size-cells”是length的单位(32bit)；

"reg"是寄存器，格式是"<address,length>"，作为平台内存资源；

"aliase" 是别名，必须节点全称，可以通过地址引用获取；

”chosen“是板级启动参数；

"cpus"是SOC的CPU信息，可以改变运行频率或者开关CPU；

"memory"是板级内存的信息。

"interrupts"是中断控制器，根据SOC自定义格式，这里是<输入类型 中断号 触发方式>，作为平台中断资源；

“interrupt-controller”指示这个节点是中断控制节点；

"[label:]"如gic: interrupt-controller@1c81000，这个标签可以作为地址赋值到其他节点的属性；

“device\_type":设备类型，寻找节点可以依据这个属性；

"status"是开关节点设备的状态，取值"okay"或者"ok"表示使能，"disabled"表示失能。

Device Tree由节点和属性构成。属性为key-value对，节点包括了各种属性，也可以包含子节点。下边列举一个简单的dts文件：

/ { 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 { }; }; };

这个文件实际上没有任何意义，但却包含了基本所有要素：

* 1 唯一的根节点 “/”
* 2 一些节点：node1 node2
* 3 子节点 node1的子节点child-node1和child-node2
* 4 一群分散的属性

属性都是简单的key-value对，其中value也可以是空的或包含任意的byte流。以下是一些属性的基本数据结构：

* 1 双引号包含的字符信息

string-property = "a string";

* 2 cells单位信息是32位无符号整型数据

cell-property = <0xFF01 412 0x12341283>;

* 3 二进制数据流

binary-property = [0x01 0x02 0x03 0x04];

* 4 混合数据用逗号隔开

mixed-property = "a string", [0x01 0x02 0x03 0x04], <0xFF01 412 0x12341283>;

* 5 字符列表

string-list = "string test1", "string test2";

3. 一些基本概念

* 每个完整的dts文件必须拥有一个根节点
* dtsi文件一般为通用文件（类似C语言的头文件），可被其他文件include  
  后边的名字涵盖的范围更加广泛，如果可以匹配到，同样会以这个dts为基础进行初始化并启动。
* 父节点名应该取类型名，而不是IC名。节点名的命名规则一般是 [name]@[address]，也可以只有name而没有@之后的内容，但是要确保name不能重名。如果加了@以及地址，那么name可以相同，只要address不同即可。
* 每一个设备节点都要有一个compatible属性
* compatible的内容是用来匹配驱动的，组成方式为"[manufacturer], [model]"，加入厂商名是为了避免重名。有的时候后边还会跟一个名字，如：

compatible = "acme,coyotes-revenge", "acmd-board";

4. 工作方式

a. 地址

设备的地址特性根据一下几个属性来控制：

* reg
* #address-cells
* #size-cells

reg意为region，区域。格式为：

reg = <address1 length1 [address2 length2] [address3 length3]>;

属性的值的含义要根据实际情况分析。

父类的address-cells和size-cells决定了子类的相关属性要包含多少个cell，如果子节点有特殊需求的话，可以自己再定义，这样就可以摆脱父节点的控制。  
address-cells决定了address1/2/3包含几个cell，size-cells决定了length1/2/3包含了几个cell。本地模块例如：

spi@10115000 { compatible = "arm,pl022"; reg = <0x10115000 0x1000 >; };

位于0x10115000的SPI设备申请地址空间，起始地址为0x10115000，长度为0x1000，即属于这个SPI设备的地址范围是0x10115000~0x10116000。

实际应用中，有另外一种情况，就是通过外部芯片片选激活模块。例如，挂载在外部总线上，需要通过片选线工作的一些模块：

external-bus { #address-cells = <2> #size-cells = <1>; ethernet@0,0 { compatible = "smc,smc91c111"; reg = <0 0 0x1000>; }; i2c@1,0 { compatible = "acme,a1234-i2c-bus"; #address-cells = <1>; #size-cells = <0>; reg = <1 0 0x1000>; rtc@58 { compatible = "maxim,ds1338"; reg = <58>; }; }; flash@2,0 { compatible = "samsung,k8f1315ebm", "cfi-flash"; reg = <2 0 0x4000000>; }; };

external-bus使用两个cell来描述地址，一个是片选序号，另一个是片选序号上的偏移量。而地址空间长度依然用一个cell来描述。所以以上的子设备们都需要3个cell来描述地址空间属性——片选、偏移量、地址长度。在上个例子中，有一个例外，就是i2c控制器模块下的rtc模块。因为I2C设备只是被分配在一个地址上，不需要其他任何空间，所以只需要一个address的cell就可以描述完整，不需要size-cells。

当需要描述的设备不是本地设备时，就需要描述一个从设备地址空间到CPU地址空间的映射关系，这里就需要用到**ranges**属性。还是以上边的external-bus举例：

#address-cells = <1>; #size-cells = <1>; ... 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 };

ranges属性为一个地址转换表。表中的每一行都包含了子地址、父地址、在自地址空间内的区域大小。他们的大小（包含的cell）分别由子节点的address-cells的值、父节点的address-cells的值和子节点的size-cells来决定。以第一行为例：

* 0 0 两个cell，由子节点external-bus的address-cells=<2>决定；
* 0x10100000 一个cell，由父节点的address-cells=<1>决定；
* 0x10000 一个cell，由子节点external-bus的size-cells=<1>决定。  
  最终第一行说明的意思就是：片选0，偏移0（选中了网卡），被映射到CPU地址空间的0x10100000~0x10110000中，地址长度为0x10000。

b. 中断

描述中断连接需要四个属性：  
1. interrupt-controller 一个空属性用来声明这个node接收中断信号；  
2. #interrupt-cells 这是中断控制器节点的属性，用来标识这个控制器需要几个单位做中断描述符；  
3. interrupt-parent 标识此设备节点属于哪一个中断控制器，如果没有设置这个属性，会自动依附父节点的；  
4. interrupts 一个中断标识符列表，表示每一个中断输出信号。

如果有两个，第一个是中断号，第二个是中断类型，如高电平、低电平、边缘触发等触发特性。对于给定的中断控制器，应该仔细阅读相关文档来确定其中断标识该如何解析。

c. 其他

除了以上规则外，也可以自己加一些自定义的属性和子节点，但是一定要符合以下的几个规则：

1. 新的设备属性一定要以厂家名字做前缀，这样就可以避免他们会和当前的标准属性存在命名冲突问题；
2. 新加的属性具体含义以及子节点必须加以文档描述，这样设备驱动开发者就知道怎么解释这些数据了。描述文档中必须特别说明compatible的value的意义，应该有什么属性，可以有哪个（些）子节点，以及这代表了什么设备。每个独立的compatible都应该由单独的解释。
3. 新添加的这些要发送到devicetree-discuss@lists.ozlabs.org邮件列表中进行review，并且检查是否会在将来引发其他的问题。

5. 进阶例子

pci@0x10180000 { compatible = "arm,versatile-pci-hostbridge", "pci"; reg = <0x10180000 0x1000>; interrupts = <8 0>; bus-ranges = <0 0>; #address-cells = <3> #size-cells = <2>; ranges = <0x42000000 0 0x80000000 0x80000000 0 0x20000000 0x02000000 0 0xa0000000 0xa0000000 0 0x10000000 0x01000000 0 0x00000000 0xb0000000 0 0x01000000>; };

像之前描述过的本地总线一样，PCI地址空间与CPU地址空间是完全分离的，所以这里需要通过定义ranges属性进行地址转化。  
#address-cells定义PCI使用3个cell，并且PCI的地址范围通过两个单位就可以解读。所以，首先的问题就是，为什么需要用3个32位的cell来描述一个PCI地址。

**这三个cell分别代表物理地址高位、中位、低位**：

* 1 phys.high cell : npt000ss bbbbbbbb dddddfff rrrrrrrr
* 2 phys.mid cell : hhhhhhh hhhhhhhh hhhhhhhh hhhhhhh
* 3 phys.low cell : llllllll llllllll llllllll llllllll

PCI地址为64位宽度，编码在phys.mid和phys.low中。真正重要的东西在于phys.high这一位空间中：

n：代表重申请空间标志（这里没有使用）  
p：代表预读空间（缓存）标志  
t：别名地址标志（这里没有使用）  
ss：空间代码  
00： 设置空间  
01：IO空间  
10：32位存储空间  
11：64位存储空间

bbbbbbbb： PCI总线号。PCI有可能是层次性架构，所以我们可能需要区分一些子-总线  
ddddd：设备号，通常由初始化设备选择信号IDSEL连接时申请。  
fff：功能序号，有些多功能PCI设备可能用到。  
rrrrrrrr：注册号，在设置周期使用。

ranges = <0x42000000 0 0x80000000 0x80000000 0 0x20000000 0x02000000 0 0xa0000000 0xa0000000 0 0x10000000 0x01000000 0 0x00000000 0xb0000000 0 0x01000000>;

回头再看这个ranges分表代表了什么。父节点address-cells为1，子节点address-cells为3， 子节点size-cells为2。则第一行可以这样划分：

0x42000000 0 0x80000000 子节点地址| 0x80000000 父节点地址| 0 0x20000000 地址空间长度|

0x42000000为phys.high，第一位为01000010，则p为1，ss为10，即申请32位存储空间为缓存空间。phys.mid为0，phys.low为0x80000000，他们共同组成了PCI地址，即表示从PCI总线的0x80000000地址处申请出一个32位的存储空间作为缓存。后边的那个cell 0x80000000 0 0x20000000代表到CPU空间后的参数，申请的地址被映射到CPU空间的0x80000000地址处，大小共计0x20000000(512MB)。