设备树的用法（Device Tree Usage）

2014年09月05日／综合／共 15865字／字号 小 中 大／评论关闭

设备树手册（Device Tree Usage）原文地址：
http://www.devicetree.org/Device_Tree_Usage
本文概述了如何为一个全新的计算机编写设备树。意在提供一个device tree概念的概述以及如何使用device tree描述一台计算机。
有关device tree数据格式的更完整技术说明，读者可以参考ePAPR规范
（http://www.power.org/resources/downloads/Power_ePAPR_APPROVED_v1.0.pdf）

HTTP/1.1 400 Bad Request

基本数据格式

---

device tree是一个简单的节点和属性树，属性是键值对，节点可以包含属性和子节点。下面是一个.dts格式的简单设备树。

```
/ {
    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 {
        };
    };
};
```

该树并未描述任何东西，也不具备任何实际意义，但它却揭示了节点和属性的结构。即：
一个的根节点:'/'，两个子节点：node1和node2；node1的子节点：child-node1和child-node2，一些属性分散在树之间。
属性是一些简单的键值对（key-value pairs）：value可以为空也可以包含任意的字节流。而数据类型并没有编码成数据结构，有一些基本数据表示可以在device tree源文件中表示。

- 文本字符串（null 终止）用双引号来表示：string-property = "a string"
- “Cells”是由尖括号分隔的32位无符号整数：cell-property = <0xbeef 123 0xabcd1234>
- 二进制数据是用方括号分隔：binary-property = [0x01 0x23 0x45 0x67];
- 不同格式的数据可以用逗号连接在一起：mixed-property = "a string", [0x01 0x23 0x45 0x67], <0x12345678>;

逗号也可以用来创建字符串列表：string-list = "red fish", "blue fish";

基本概念

为了帮助理解device tree的用法，我们从一个简单的计算机开始， 手把手创建一个device tree来描述它。

例子

假设有这样一台计算机（基于ARM Versatile），由“Acme”制造并命名为"Coyote's Revenge"：

- 一个32位的ARM CPU

- 连接到内存映射串行端口的处理器本地总线(processor local bus),spi bus controller, i2c controller, interrupt controller, 和external bus bridge
- 256MB SDRAM，起始地址为0
- 两个串口起始地址为0x101F1000，0x101F2000
- GPIO controller，起始地址为0x101F3000
- SPI controller起始地址为0x10170000，并挂载以下设备：
- MMC插槽（SS管脚连接到GPIO #1）
- External bus bridge，挂载以下设备：
- SMC SMC91111以太网设备连接到external bus，基地址0x10100000
- i2c controller起始地址为0x10160000，并挂载以下设备：
- Maxim DS1338 real time clock.响应从机地址1101000 (0x58)
- 64MB NOR flash，基地址0x30000000

初始结构

第一步，先构建一个计算机的基本架构，即一个有效设备树的最小架构。在这一步，要唯一地标志这台计算机。

```
/ {
    compatible = "acme,coyotes-revenge";
};
```

compatible属性以"<manufacturer>,<model>"的格式指定了系统名称。指定了具体设备和制造商名称来避免命名空间的冲突是很重要的，因为一个操作系统可以使用compatible值来决定如何运行这台计算机，在该属性中填入正确的数据很重要。理论上，compatible是一个OS唯一地识别一台计算机所需要的所有数据。如果计算机的所有细节都被硬编码，那么OS可以在顶层compatible属性中专门查找"acme,coyotes-revenge"。

CPU

接下来就要描述各个CPU了。先添加一个“cpus”容器节点，再将每个CPU作为子节点添加。在本例中，系统是基于ARM的双核Cortex A9系统。

```
/ {
    compatible = "acme,coyotes-revenge";

    cpus {
```

```
        cpu@0 {
            compatible = "arm,cortex-a9";
        };
        cpu@1 {
            compatible = "arm,cortex-a9";
        };
    };
};
```

各个CPU节点的compatible属性是一个字符串，与顶层compatible属性类似，该字符串以“<manufacturer>,<model>”的格式指定了CPU的确切型号。
随后更多的属性被添加到cpu节点，但首先我们需要先了解一些基本概念。

节点命名

花些时间谈谈命名习惯是值得的。每个节点都必须有一个<name>[@<unit-address>]格式的名称。<name>是一个简单的ascii字符串，最长为31个字符，总的来说，节点命名是根据它代表什么设备。比如说，一个代表3com以太网适配器的节点应该命名为ethernet，而不是3com509。
如果节点描述的设备有地址的话，就应该加上unit-address，unit-address通常是用来访问设备的主地址，并在节点的reg属性中被列出。后面我们将谈到reg属性。
同级节点的命名必须是唯一，但多个节点的通用名称可以相同，只要地址不同就行。（即serial@101f1000 & serial@101f2000）。
关于节点命名的全部细节请参考ePAPR规范2.2.1节。

设备

系统中的每个设备由device tree的一个节点来表示，接下来将为设备树添加设备节点。在我们讲到如何寻址和如何处理中断之前，暂时将新节点置空。

```
/ {
    compatible = "acme,coyotes-revenge";


    cpus {
        cpu@0 {
            compatible = "arm,cortex-a9";
        };
        cpu@1 {
            compatible = "arm,cortex-a9";
        };
    };
```

```
    serial@101F0000 {
        compatible = "arm,pl011";
    };


    serial@101F2000 {
        compatible = "arm,pl011";
    };


    gpio@101F3000 {
        compatible = "arm,pl061";
    };


    interrupt-controller@10140000 {
        compatible = "arm,pl190";
    };


    spi@10115000 {
        compatible = "arm,pl022";
    };


    external-bus {
        ethernet@0,0 {
            compatible = "smc,smc91c111";
        };


        i2c@1,0 {
            compatible = "acme,a1234-i2c-bus";
            rtc@58 {
                compatible = "maxim,ds1338";
            };
        };


        flash@2,0 {
            compatible = "samsung,k8f1315ebm", "cfi-flash";
        };
    };
};
```

在上面的设备树中，系统中的设备节点已经添加进来，树的层次结构反映了设备如何连到系统中。外部总线上的设备就是外部总线节点的子节点，i2c设备是i2c总线控制节点的子节点。总的来说，层次结构表现的是从CPU视角来看的系统视图。在这里这棵树是依然是无效的。它缺少关于设备之间的连接信息。稍后将添加这些数据。
设备树中应当注意：每个设备节点有一个compatible属性。flash节点的compatible属性有两个字符串。请阅读下一节以了解更多内容。 之前提到的，节点命名应当反映设备的类型，而不是特定型号。请参考ePAPR规范2.2.2节的通用节点命名，应优先使用这些命名。

compatible 属性

树中的每一个代表了一个设备的节点都要有一个compatible属性。compatible是OS用来决定绑定到设备的设备驱动的关键。
compatible是字符串的列表。列表中的第一个字符串指定了"<manufacturer>,<model>"格式的节点代表的确切设备，第二个字符串代表了与该设备兼容的其他设备。例如，Freescale MPC8349 SoC有一个串口设备实现了National Semiconductor ns16550寄存器接口。因此MPC8349串口设备的compatible属性为：compatible = "fsl,mpc8349-uart", "ns16550"。在这里，fsl,mpc8349-uart指定了确切的设备，ns16550表明它与National
Semiconductor 16550 UART是寄存器级兼容的。
注：由于历史原因，ns16550没有制造商前缀，所有新的compatible值都应使用制造商的前缀。这种做法使得现有的设备驱动程序可以绑定到一个新设备上，同时仍能唯一准确的识别硬件。

警告：不要使用通配符compatible值，如"fsl,mpc83xx-uart"等类似表达，芯片厂商总会改变并打破你的通配符假设，到时候再想修改就为时已晚了。相反，你应当选择一个特定的芯片实现，并与所有后续芯片保持兼容。

编址

可编址的设备使用下列属性来将地址信息编码进设备树：

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

每个可寻址的设备有一个reg属性，即以下面形式表示的元组列表：reg = <address1 length1 [address2 length2] [address3 length3] ... >
每个元组表示该设备的地址范围。每个地址值由一个或多个32位整数列表组成，被称做cells。同样地，长度值可以是cells列表，也可以为空。
既然address和length字段是大小可变的变量，父节点的#address-cells和#size-cells属性用来说明各个子节点有多少个cells。换句话说，正确解释一个子节点的reg属性需要父节点的#address-cells和#size-cells值。让我们从CPU开始，添加编址属性到示例设备树。

CPU编址

Each CPU is assigned a single unique ID, and there is no size associated with CPU ids.
谈到编址，最简单的例子就是CPU节点。每个CPU被分配了一个唯一的ID，并且不存在与CPU ids的相关大小信息。

```
    cpus {
        #address-cells = <1>;
        #size-cells = <0>;
        cpu@0 {
            compatible = "arm,cortex-a9";
            reg = <0>;
        };
        cpu@1 {
            compatible = "arm,cortex-a9";
            reg = <1>;
        };
    };
```

- 在cpus节点中，#address-cells为1，#size-cells为0，这意味着子寄存器值是一个uint32，是一个不包含大小字段的地址。在本例中，两个CPU被分配为地址0和1。cpu节点的#size-cells为0，是因为只为每个CPU分配了地址值。你一定注意到了，reg值与节点名中的值是匹配。按照惯例，如果一个节点有reg属性，则节点名称必须包含unit-address属性，unit-address属性值是reg属性中的第一个地址值。
- 注：ePAPR中对cell的定义是”一个包含32bit信息的单元“。

内存映射设备

与CPU节点中的单一地址值不同，内存映射设备会被分配一个它能响应的地址范围。#size-cells用来说明每个子节点种reg元组的长度大小。在下面的示例中，每个地址值是1 cell (32位)，并且每个的长度值也为1 cell，这在32位系统中是非常典型的。64位计算机可以在设备树中使用2作为#address-cells和#size-cells的值来实现64位寻址。

```
/ {
    #address-cells = <1>;
    #size-cells = <1>;


    ...


    serial@101f0000 {
        compatible = "arm,pl011";
        reg = <0x101f0000 0x1000 >;
```

```
    };

    serial@101f2000 {
        compatible = "arm,pl011";
        reg = <0x101f2000 0x1000 >;
    };

    gpio@101f3000 {
        compatible = "arm,pl061";
        reg = <0x101f3000 0x1000
               0x101f4000 0x0010>;
    };

    interrupt-controller@10140000 {
        compatible = "arm,pl190";
        reg = <0x10140000 0x1000 >;
    };

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

    ...

};
```

每个设备都被分配了一个基地址及该区域大小。本例中的GPIO设备地址被分成两个地址范围:0x101f3000~0x101f3fff和0x101f4000~0x101f400f。
有些挂载于总线上的设备有不同的编址方案。例如，设备也可以通过独立片选线连接到外部总线。因为父节点定义了它的子节点的地址范围，可根据需要选择地址映射来最佳地描述该系统。下面的代码显示了连接到外部总线并将芯片片选编码进地址的设备地址分配。

```
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";
            reg = <1 0 0x1000>;
            rtc@58 {
                compatible = "maxim,ds1338";
            };
        };


        flash@2,0 {
            compatible = "samsung,k8f1315ebm", "cfi-flash";
            reg = <2 0 0x4000000>;
        };
    };
```

外部总线用了2个cells来表示地址值;一个是片选号，一个是基于片选的偏移量。长度字段还是一个cell，这是因为只有地址的偏移部分需要一个范围。所以，在本例中，每个reg条目包含3个cell；片选号码，偏移，长度。由于地址范围包含节点及其子节点，父节点可以自由定义任何对总线而言有意义的编址方案。直接父节点和子节点之外的其他节点，通常不需要关心本地节点地址域。

非内存映射设备

其他设备没有映射到处理器总线上。虽然这些设备可以有地址范围，但是不能直接被CPU访问，而是由父设备的驱动代表CPU来执行间接访问。以I2C设备为例，每个设备分配一个地址，但没有与它相关的长度或范围，这与CPU地址分配很相似。

```
        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>;
            };
        };
```

ranges（地址翻译）

我们已经讨论过如何分配地址给设备，但在这些地址只是设备节点可见的，还没有描述如何将这些地址映射成CPU可使用的地址。根节点总是从CPU的角度描述地址空间。

如果根节点的子节点已经使用了CPU地址域，就不需要任何显式映射了，例如，串口serial@101f0000被直接分配到地址0x101f0000。

如果根节点的间接子节点没有使用CPU的地址域，为了获得一个内存映射地址，设备树必须指定如何翻译地址。

下面是一个添加了ranges属性的示例device tree。

```
/ {
    compatible = "acme,coyotes-revenge";
    #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


        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>;
        };
    };
};
```

ranges是地址翻译表，由3个数组成，即<子地址，父地址，区域大小>，分别对应子节点的#address-cells值，父节点的#address-cells值，子节点的#size-cells值确定。对于本例中的外部总线，子节点的地址是2个单元，父节点的地址是1个单元，子节点的区域大小是1个单元。

3个ranges被转换：

从片选0偏移0被映射到地址范围0x10100000~0x1010ffff
从片选1偏移0被映射到地址范围0x10160000~0x1016ffff
从片选2偏移0被映射到地址范围为0x30000000~0x30ffffff
另外，如果父节点和子节点的地址空间是相同的，那么一个节点可以添加一个空的ranges属性。一个空的ranges属性的存在意味着子节点地址空间的地址1：1地映射到父地址空间。你可能会问，你可能会问，为如果可以都用一一映射，为什么还需要地址转换？这是因为一些总线（如PCI ）具有完全不同的地址空间，其细节需要暴露给操作系统。其他有DMA engine的计算机需要知道总线上的真实地址。有时设备需要组合在一起，因为他们都有着相同的软件可编程的物理地址映射。是否需要一一映射取决于操作系统所需要的信息，以及硬件设计。

你也应该注意到，在i2c@1,0节点上没有ranges属性。这样做的原因是，不像外部总线，I2C总线上的设备不是内存映射到CPU地址域的。相反，CPU通过i2c@1,0设备间接访问rtc@58设备。ranges属性为空意味着设备不能被除了父设备以外的任何设备直接访问。

-
-
-

另举一例说明ranges属性

```
soc {
        compatible = "simple-bus";
        #address-cells = <1>;
        #size-cells = <1>;
        ranges = <0x0 0xe0000000 0x00100000>;

        serial {
                device_type = "serial";
                compatible = "ns16550";
                reg = <0x4600 0x100>;
```

```
            clock-frequency = <0>;
            interrupts = <0xA 0x8>;
            interrupt-parent = < &ipic >;
        }
}
```

soc的ranges属性指定了，从物理地址为0x0大小为1024KB的子节点映射到了物理地址为0xe0000000的父地址空间，有个这层映射关系，串口设备节点就可以通过load/store地址0xe0004600来访问，即0x4600的偏移+在ranges中指定的映射0xe0000000。

当然在64位操作系统中也会看到这样的映射，不要感到吃惊啦，"0xf 0x00000000"一起组成父节点地址即f00000000

```
dcsr: dcsr@f00000000 {
        ranges = <0x0 0xf 0x00000000 0x01072000>;
};
```

中断如何工作

与地址范围转换遵循树的天然结构不同，一台计算机的任何设备都可以发起和终止中断信号。不像设备编址，中断信号表现为独立于树的节点之间的链接。描述中断连接有4个属性：

- interrupt-controller - 一个空的属性定义该节点为接收中断信号的设备
- #interrupt-cells - 这是中断控制器节点的属性。它声明了中断控制器的中断说明符有多少个cell（类似#address-cells和#size-cells）。
- interrupt-parent - 设备节点的属性，包含一个指向该设备所连接中断控制器的pHandle。那些没有interrupt-parent属性节点则从它们的父节点继承该属性。
- interrupts - 设备节点属性，包含中断说明符列表，对应于该设备上的每个中断输出信号。

中断说明符是一个或多个cell的数据（由#interrupt-cells指定），指定设备连接到哪些中断输入。下面的例子中，大多数设备只有一个中断输出，但也有一个设备上有多个中断输出的情况。一个中断符的含义完全取决于绑定的中断控制器设备。每个中断控制器可以决定它需要多少cell来唯一地确定一个中断输入。

下面的代码将中断添加到我们的Coyote's Revenge：

```
/ {
    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>;
        };
    };
};
```

需要注意的是：

这台机器只有一个中断控制器interrupt-controller@10140000，标签'intc:'被添加到中断控制器节点，该标签用于给根节点中的interrupt-parent属性分配一个phandle。这个interrupt-parent值将成为系统默认值，所有子节点都继承它，除非它被显式覆盖。每个设备用一个interrupt属性来指定一个不同的中断输入线。#interrupt-cells是2，所以每个中断符有2个单元。本例使用通用格式，第一个cell代表中断线号码，第二个cell代表标志位，如高电平触发/低电平触发，边沿触发/水平触发。对于一个给定的中断控制器，请参阅控制器binding文档，以了解如何对说明符进行编码。

设备特有的数据

除了公共属性，任意属性和子节点都可以作为节点添加。只要遵循一些规则，操作系统所需要的任何数据可以被添加。首先，设备新的特有属性名应当使用制造商前缀，这样它们不会与现有的标准属性名称冲突。第二，属性和子节点的含义必须有文档来约束，这样一个设备驱动程序的作者才能知道如何解释这些数据。约束记录了一个特定的compatible值代表什么，它应该有什么样的属性，它可能有什么子节点，代表什么设备。每个唯一的compatible值应该有自己的约束（或者声明与其他compatible值的兼容性）。绑定新设备都记录在本wiki页。第三，发布新的约束到devicetree-discuss@lists.ozlabs.org邮件列表上进行审查。审查新的约束的确发现了很多未来会导致问题的常见错误。

特殊节点

别名节点

特定节点通常通过完整路径引用，如/external-bus/ethernet@0,0，但当用户真正想要知道的是哪个设备是eth0时，这很不具有易读性，别名节点可分配一个短的alias给一个完整的设备路径。例如：

```
    aliases {
        ethernet0 = ð0;
```

```
        serial0 = &serial0;
    };
```

分配标识符给设备时，使用别名是受操作系统欢迎的。
这里使用了一个新的语法property = &label;该语法指定通过标签引用的完整节点路径为一个字符串属性。这与phandle = <&label>;不同，它是把一个pHandle值插入到一个cell。

可选节点

可选节点并不代表真正的设备，而是作为固件和操作系统之间传递数据的地方，如启动参数。选择节点中的数据并不代表硬件。通常情况下，选择节点在DTS源文件中为空，并在开机时填充。
在我们的示例系统中，固件可以添加以下选择节点：

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

进阶主题

-
-
-

进阶例子

现在，我们已经定义了原则，让我们添加一些硬件到示例计算机中来讨论一些更复杂的用例。
先进的示例计算机增加了PCI host bridge，并将控制寄存器内存映射到0x10180000开始，BARs从0x80000000开始启动。
既然我们已经了解了device tree，我们可以从添加以下节点开始，来描述PCI host bridge。

```
        pci@10180000 {
            compatible = "arm,versatile-pci-hostbridge", "pci";
            reg = <0x10180000 0x1000>;
            interrupts = <8 0>;
        };
```

PCI Host Bridge

本节描述了Host/PCI桥节点。请注意，本节假定你已经了解了一些PCI的基本知识。这不是一个有关PCI教程，如果你需要一些更深入的信息，请阅读[1]。您也可以参考ePAPR或PCI Bus Binding to Open Firmware。在这里我们以Freescale MPC5200为例。（[1] Tom Shanley / Don Anderson: PCI System Architecture. Mindshare Inc.http://www.mindshare.com/）

PCI总线编号

每个PCI总线段被唯一地编号，总线编号在PCI节点中通过bus-ranges属性暴露，它包括两个单元。第一个单元给出分配给该节点的总线号，第二个单元给出任何从属PCI总线上的最大总线数目。
示例计算机有一个PCI总线，所以这两个单元都为0。

```
pci@0x10180000 {
    compatible = "arm,versatile-pci-hostbridge", "pci";
    reg = <0x10180000 0x1000>;
    interrupts = <8 0>;
    bus-ranges = <0 0>;
};
```

PCI地址转换

类似于前面描述的local bus， PCI地址空间与CPU地址空间是完全分开的，所以需要地址转换才能从PCI地址获取到CPU地址。和之前一样，通过使用range，#address-cells, 和#size-cells属性来实现。

```
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>;
};
```

正如你所看到的，子地址（PCI地址）使用3个单元，PCI ranges编码为2个单元。第一个问题可能是，为什么我们还需要3个32位的单元指定一个PCI地址。三个单元被标记为phys.hi, phys.mid和phys.low[2]。（[2] PCI Bus Bindings to Open Firmware.）

- phys.hi 单元: npt000ss bbbbbbbb dddddfff rrrrrrrr
- phys.mid 单元: hhhhhhhh hhhhhhhh hhhhhhhh hhhhhhhh
- phys.low 单元: llllllll llllllll llllllll llllllll

PCI地址是64位宽的，并被编码成phys.mid和phys.low 。然而，真正有趣是phys.high这个单比特位字段：

- n：可重定位区域标志（在这里不起作用）
- p：可预取（可缓存）区域标志
- t：别名地址标志（在这里不起作用）
- ss：空间代码
- 00：配置空间
- 01：I/O空间
- 10：32位内存空间
- 11：64位内存空间
- bbbbbbbb: PCI总线编号,PCI结构是可分层的，所以我们可能有PCI/PCI桥，来定义子总线。
- ddddd:设备编号，通常与IDSEL信号连接相关联。
- fff:功能编号。用于多功能PCI设备
- rrrrrrrr: 寄存器编号；用于配置周期。

对于PCI地址转换，重要的字段是p和ss。 phys.hi中p和ss的值决定了访问的是哪个PCI地址空间。因此，仔细观察ranges属性，我们有三个区域：

- 一个从PCI地址0x80000000处开始的512MByte大小的32位可预取内存区域，将被映射到host CPU的0x80000000地址处。
- 一个从PCI地址0xa0000000处开始的256MByte大小的32位非预取内存区域，将被映射到host CPU的0xa0000000地址处。
- 一个从PCI地址0x00000000处开始的16MByte大小的I/O区域，将被映射到host CPU的0xb0000000地址处。

为减轻工作量，phys.hi位域的存在就意味着操作系统知道该节点代表一个PCI桥，以便在地址转换时可以忽略不相关的位域。操作系统可以在PCI bus节点中查找字符串“pci”，来决定是否需要屏蔽多余的字段。

高级PCI中断映射

现在我们来到了最有趣的部分，PCI中断映射。 PCI设备可以使用引线#INTA, #INTB, #INTC 和#INTD来触发中断。如果没有多功能PCI设备，那么设备就用#INTA实现中断。但PCI插槽或设备通常会连接到中断控制器的不同输入端。所以，device tree需要一种将各种PCI中断信号映射到中断控制器输入端的方法。#interrupt-cells, interrupt-map和interrupt-map-mask属性就被用于描述中断映射。
实际上，这里所描述的中断映射不仅限于PCI总线，任何节点都可以指定复杂的中断映射，但PCI是迄今为止最常见的。

```
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>;


    #interrupt-cells = <1>;
    interrupt-map-mask = <0xf800 0 0 7>;
    interrupt-map = <0xc000 0 0 1 &intc  9 3 // 1st slot
                     0xc000 0 0 2 &intc 10 3
                     0xc000 0 0 3 &intc 11 3
                     0xc000 0 0 4 &intc 12 3

                     0xc800 0 0 1 &intc 10 3 // 2nd slot
                     0xc800 0 0 2 &intc 11 3
                     0xc800 0 0 3 &intc 12 3
                     0xc800 0 0 4 &intc  9 3>;
};
```

首先，你会发现， PCI中断号只使用了一个单元，不像系统中断控制器，使用了两个单元;一个IRQ号，一个标志位。 PCI中断只需要一个cell，因为PCI中断永远是低电平敏感的。在我们的示例板子上，我们分别有2个PCI插槽，4个中断线，所以我们要映射8个中断线到中断控制器。这是利用interrupt-map属性来实现的。中断映射的详细程序在[3]中有描述。因为中断号（#INTA等）不足以在单个PCI总线上区分若干PCI设备，我们也必须表明哪个PCI设备被中断线出发。幸运的是，每个PCI设备都有一个唯一的设备号。为了区分不同PCI设备的中断，我们需要一个包含PCI设备号和PCI中断号的元组。更一般来说，我们构造了一个有四个单元的单位中断说明符：3个#address-cells包括phys.hi, phys.mid, phys.low，和一个#interrupt-cell (#INTA, #INTB, #INTC, #INTD)。
因为我们只设备号（部分PCI地址），interrupt-map-mask属性就发挥作用了。interrupt-map-mask属性也是一个四元组，如单位中断符。第一位表该单元中断符的一部分应该予以考虑。在我们的例子中，我们可以看到，只有phys.hi的唯一设备号部分是必须的，我们需要3位来区分4个中断线。（计数PCI中断线从1开始，而不是0！）

现在我们可以构建interrupt-map属性。该属性是一个表，表中的每个条目由一个子单位（PCI总线）中断符，一个父句柄（中断控制器负责提供中断服务）和父中断说明符。因此在第一行中，我们可以读取到PCI中断#INTA被映射到IRQ9，低电平敏感的中断控制器[4]。

现在唯一缺少的部分是奇怪的数字诠释了PCI总线的单位中断符。单位中断符的重要组成部分是phys.hi位域中的设备号。设备号是板子特有的，它取决于每个PCI主机控制器怎样激活每个设备上的IDSEL引脚。在本示例中， PCI插槽1被分配了设备ID 24(0x18)，PCI插槽2被分配了设备ID 25(0x19)。每个槽的phys.hi值是由设备号偏移11位多达确定
插槽1的phys.hi是0xC000, 插槽2的phys.hi是0xC800.
全部放在一起interrupt-map属性显示为：

- 主中断控制器上的插槽1的#INTA是IRQ9, 低电平敏感
- 主中断控制器上的插槽1的#INTB是IRQ10, 低电平敏感
- 主中断控制器上的插槽1的#INTC是IRQ11, 低电平敏感
- 主中断控制器上的插槽1的#INTD是IRQ12, 低电平敏感

- 主中断控制器上的插槽2的#INTA是IRQ10, 低电平敏感
- 主中断控制器上的插槽2的#INTB是IRQ11, 低电平敏感
- 主中断控制器上的插槽2的#INTC是IRQ12, 低电平敏感
- 主中断控制器上的插槽2的#INTD是IRQ9, 低电平敏感

interrupts = <8 0>; 属性描述了host/PCI-bridge控制器本身可能触发的中断。不要将这些中断与PCI设备可能会触发的中断(INTA, INTB, ...)混淆。
最后要注意一件事。就像interrupt-parent属性，节点上的interrupt-map属性是否存在将改变所有子节点和孙子节点的默认中断控制器。在本PCI例子中，这意味着PCI host bridge将成为默认中断控制器。如果通过PCI总线连接的设备直接连接到另一个中断控制器上，那它也需要指定它自己的interrupt-parent属性。

返回

【上篇】自学网站
【下篇】linux free

作者: mxgongsi

- 该日志由 mxgongsi 于6年前发表在综合分类下，最后更新于 2014年09月05日.
- 转载请注明: 设备树的用法（Device Tree Usage） | 学步园 +复制链接