设备树使用手册

本文将介绍如何为一个新机器编写设备树。我们准备提供一个有关设备树概念的概述和如何使用这些设备树来描述一个机器。

完整的设备树数据格式的技术说明书请参考 ePAPR 规范。ePAPR 规范涵盖了比本文基本主题更丰富的细节，要查阅本文没有涉及到的高级用法请参考该规范。

基本数据格式

设备树是一个包含节点和属性的简单树状结构。属性就是键－值对，而节点可以同时包含属性和子节点。例如，以下就是一个 .dts 格式的简单树：

```
1   / {
2           node1 {
3                   a-string-property = "A string";
4                   a-string-list-property = "first string", "second string";
5                   a-byte-data-property = [0x01 0x23 0x34 0x56];
6                   child-node1 {
7                           first-child-property;
8                           second-child-property = <1>;
9                           a-string-property = "Hello, world";
10                  };
11                  child-node2 {
12                  };
13          };
14          node2 {
15                  an-empty-property;
16                  a-cell-property = <1 2 3 4>; /* each number (cell) is a uint32 */
17                  child-node1 {
18                  };
19          };
20  };
```

这棵树显然是没什么用的，因为它并没有描述任何东西，但它确实体现了节点的一些属性：

一个单独的根节点：“/”

两个子节点：“node1”和“node2”

两个 node1 的子节点：“child-node1”和“child-node2”

一堆分散在树里的属性。

属性是简单的键－值对，它的值可以为空或者包含一个任意字节流。虽然数据类型并没有编码进数据结构，但在设备树源文件中任有几个基本的数据表示形式。

文本字符串（无结束符）可以用双引号表示：

string-property = "a string"

‘Cells’是 32 位无符号整数，用尖括号限定：

cell-property = <0xbeef 123 0xabcd1234>

二进制数据用方括号限定：

binary-property = [0x01 0x23 0x45 0x67];

不同表示形式的数据可以使用逗号连在一起：

mixed-property = "a string", [0x01 0x23 0x45 0x67], <0x12345678>;

逗号也可用于创建字符串列表：

string-list = "red fish", "blue fish";

基本概念

我们将以一个简单机开始，然后通过一步步的建立一个描述这个简单机的设备树，来了解如何使用设备树。

模型机

考虑下面这个假想的机器（大致基于 ARM Versatile），制造商为“Acme”，并命名为“Coyote's Revenge”：

- 一个 32 位 ARM CPU
- 处理器本地总线连接到内存映射的串行口、spi 总线控制器、i2c 控制器、中断控制器和外部总线桥
- 256MB SDRAM 起始地址为 0
- 两个串口起始地址：0x101F1000 和 0x101F2000
- GPIO 控制器起始地址：0x101F3000
- 带有以下设备的 SPI 控制器起始地址：0x10170000
  - MMC 插槽的 SS 管脚连接至 GPIO #1
- 外部总线桥挂载以下设备
  - SMC SMC91111 以太网设备连接到外部总线，起始地址：0x10100000
  - i2c 控制器起始地址：0x10160000，并挂载以下设备
    - Maxim DS1338 实时时钟。响应至从地址 1101000 (0x58)
  - 64MB NOR 闪存起始地址 0x30000000

初始结构

第一步就是要为这个模型机构建一个基本结构，这是一个有效的设备树最基本的结构。在这个阶段你需要唯一的标识该机器。

```
1  / {
2        compatible = "acme,coyotes-revenge";
3  };
```

compatible 指定了系统的名称。它包含了一个“<制造商>,<型号>”形式的字符串。重要的是要指定一个确切的设备，并且包括制造商的名子， 以避免命名空间冲突。由于操作系统会使用 compatible 的值来决定如何在机器上运行，所以正确的设置这个属性变得非常重要。

中央处理器

接下来就应该描述每个 CPU 了。先添加一个名为“cpus”的容器节点，然后为每个 CPU 分别添加子节点。具体到我们的情况是一个 ARM 的 双核 Cortex A9 系统。

```
1   / {
2           compatible = "acme,coyotes-revenge";
3           cpus {
4                   cpu@0 {
5                           compatible = "arm,cortex-a9";
6                   };
7                   cpu@1 {
8                           compatible = "arm,cortex-a9";
9                   };
10          };
11  };
```

每个 cpu 节点的 compatible 属性是一个“<制造商>,<型号>”形式的字符串，并指定了确切的 cpu，就像顶层的 compatible 属性一样。

节点名称

现在应该花点时间来讨论命名约定了。每个节点必须有一个“<名称>[@<设备地址>]”形式的名字。

<名称> 就是一个不超过31位的简单 ascii 字符串。通常，节点的命名应该根据它所体现的是什么样的设备。比如一个 3com 以太网适配器的节点就应该命名为 ethernet，而不应该是 3com509。

如果该节点描述的设备有一个地址的话就还应该加上设备地址（unit-address）。通常，设备地址就是用来访问该设备的主地址，并且该地址也在节点的 reg 属性中列出。本文档中我们将在稍后涉及到 reg 属性。

同级节点命名必须是唯一的，但只要地址不同，多个节点也可以使用一样的通用名称（例如 serial@101f1000 和 serial@101f2000）。关于节点命名的更多细节请参考 ePAPR 规范 2.2.1 节。

设备

系统中每个设备都表示为一个设备树节点。所以接下来就应该为这个设备树填充设备节点。现在，知道我们讨论如何进行寻址和中断请求如何处理之前这些新节点将一直为空。

```
1   / {
2           compatible = "acme,coyotes-revenge";
3
4           cpus {
5                   cpu@0 {
6                           compatible = "arm,cortex-a9";
7                   };
8                   cpu@1 {
9                           compatible = "arm,cortex-a9";
```

```
10              };
11      };
12
13      serial@101F0000 {
14              compatible = "arm,pl011";
15      };
16
17      serial@101F2000 {
18              compatible = "arm,pl011";
19      };
20
21      gpio@101F3000 {
22              compatible = "arm,pl061";
23      };
24
25      interrupt-controller@10140000 {
26              compatible = "arm,pl190";
27      };
28
29      spi@10115000 {
30              compatible = "arm,pl022";
31      };
32
33      external-bus {
34              ethernet@0,0 {
35                      compatible = "smc,smc91c111";
36              };
37
38              i2c@1,0 {
39                      compatible = "acme,a1234-i2c-bus";
40                      rtc@58 {
41                              compatible = "maxim,ds1338";
42                      };
43              };
44
45              flash@2,0 {
46                      compatible = "samsung,k8f1315ebm", "cfi-flash";
47              };
48      };
49 };
```

在此树中，已经为系统中的每个设备添加了节点，而且这个·层次结构也反映了设备与系统的连接方式。例如，外部总线上的设备就是外部总线节点的子节点，i2c 设备就是 i2c 总线节点的子节点。通常，这个层次结构表现的是 CPU 视角的系统视图。

现在这棵树还是无效的，因为它缺少关于设备之间互联的信息。稍后将添加这些信息。

在这颗树中，应该注意这些事情：

- 每个设备节点都拥有一个 compatible 属性。
- 闪存（flash）节点的 compatible 属性由两个字符串构成。欲知为何，请阅读下一节。
- 正如前面所述，节点的命名应当反映设备的类型而不是特定的型号。请查阅 ePAPR 规范第 2.2.2 节里定义的通用节点名，应当优先使用这些节点名。

理解 compatible 属性

树中每个表示一个设备的节点都需要一个 compatible 属性。compatible 属性是操作系统用来决定使用哪个设备驱动来绑定到一个设备上的关键因素。

compatible 是一个字符串列表，之中第一个字符串指定了这个节点所表示的确切的设备，该字符串的格式为："<制造商>,<型号>"。剩下的字符串的则表示其它与之相兼容的设备。

例如，Freescale MPC8349 片上系统（SoC）拥有一个实现了美国国家半导体 ns16550 的寄存器接口的串行设备，那么 MPC8349 的串行设备的 compatible 属性就应该是：compatible = "fsl,mpc8349-uart", "ns16550"。在这里，mpc8349-uart 指定了确切的设备，而 ns16550 则说明这是与美国国家半导体 ns16550 UART 的寄存器级兼容。
注：ns16550 并没有制造商前缀，这仅仅是历史原因造成的。所有的新 compatible 值都应该使用制造商前缀。

这种做法可以使现有的设备驱动能够绑定到新设备上，并仍然唯一的指定确切的设备。

警告：不要使用带通配符的 compatible 值，比如“fsl,mpc83xx-uart”或类似情况。芯片提供商无不会做出一些能够轻易打破你通配符猜想的变化，这时候在修改已经为时已晚了。相反，应该选择一个特定的芯片然后是所有后续芯片都与之兼容。

如何编址

可编址设备使用以下属性将地址信息编码进设备树：

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

每个可编址设备都有一个元组列表的 reg，元组的形式为：reg = <地址 1 长度 1 [地址 2 长度 2] [地址 3 长度 3] ... >。每个元组都表示一个该设备使用的地址范围。每个地址值是一个或多个 32 位整型数列表，称为 cell。同样，长度值也可以是一个 cell 列表或者为空。

由于地址和长度字段都是可变大小的变量，那么父节点的 #address-cells 和 #size-cells 属性就用来声明各个字段的 cell 的数量。换句话说，正确解释一个 reg 属性需要用到父节点的 #address-cells 和 #size-cells 的值。要知道这一切是如何运作的，我们将给模型机添加编址属性，就从 CPU 开始。

CPU 编址

CPU 节点表示了一个关于编址的最简单的例子。每个 CPU 都分配了一个唯一的 ID，并且没有 CPU id 相关的大小信息。

```
1       cpus {
2           #address-cells = <1>;
3           #size-cells = <0>;
4           cpu@0 {
5               compatible = "arm,cortex-a9";
6               reg = <0>;
7           };
8           cpu@1 {
9               compatible = "arm,cortex-a9";
10              reg = <1>;
11          };
12      };
```

在 cpu 节点中，#address-cells 设置为 1，#size-cells 设置为 0。这意味着子节点的 reg 值是一个单一的 uint32，这是一个不包含大小字段的地址，为这两个 cpu 分配的地址是 0 和 1。cpu 节点的 #size-cells 为 0 是因为只为每个 cpu 分配一个单独的地址。

你可能还会注意到 reg 的值和节点名字是相同的。按照惯例，如果一个节点有 reg 属性，那么该节点的名字就必须包含设备地址，这个设备地址就是 reg 属性里第一个地址值。

内存映射设备

与 cpu 节点里单一地址值不同，应该分配给内存映射设备一个地址范围。#size-cells 声明每个子节点的 reg 元组中长度字段的大小。在接下来的例子中，每个地址值是 1 cell（32 位），每个长度值也是 1 cell，这是典型的 32 位系统。64 位的机器则可以使用值为 2 的 #address-cells 和 #size-cells 来获得在设备树中的 64 位编址。

```
1   / {
2       #address-cells = <1>;
3       #size-cells = <1>;
4
5       ...
6
7       serial@101f0000 {
```

```
8               compatible = "arm,pl011";
9               reg = <0x101f0000 0x1000 >;
10      };
11
12      serial@101f2000 {
13              compatible = "arm,pl011";
14              reg = <0x101f2000 0x1000 >;
15      };
16
17      gpio@101f3000 {
18              compatible = "arm,pl061";
19              reg = <0x101f3000 0x1000
20                      0x101f4000 0x0010>;
21      };
22
23      interrupt-controller@10140000 {
24              compatible = "arm,pl190";
25              reg = <0x10140000 0x1000 >;
26      };
27
28      spi@10115000 {
29              compatible = "arm,pl022";
30              reg = <0x10115000 0x1000 >;
31      };
32
33      ...
34  };
```

每个设备都被分配了一个基址以及该区域的大小。这个例子中为 GPIO 分配了两个地址范围：0x101f3000...0x101f3fff 和 0x101f4000..0x101f400f。

一些挂在总线上的设备有不同的编址方案。例如一个带独立片选线的设备也可以连接至外部总线。由于父节点会为其子节点定义地址域，所以可以选择不同的地址映射来最恰当的描述该系统。下面的代码展示了设备连接至外部总线并将其片选号编码进地址的地址分配。

```
1   external-bus {
2               #address-cells = <2>
3               #size-cells = <1>;
4
5               ethernet@0,0 {
6                       compatible = "smc,smc91c111";
7                       reg = <0 0 0x1000>;
8               };
9
10              i2c@1,0 {
```

```
11                  compatible = "acme,a1234-i2c-bus";
12                  reg = <1 0 0x1000>;
13                  rtc@58 {
14                      compatible = "maxim,ds1338";
15                  };
16              };
17
18              flash@2,0 {
19                  compatible = "samsung,k8f1315ebm", "cfi-flash";
20                  reg = <2 0 0x4000000>;
21              };
22          };
```

外部总线的地址值使用了两个 cell，一个用于片选号；另一个则用于片选基址的偏移量。而长度字段则还是单个 cell，这是因为只有地址的偏移部分才需要一个范围量。所以，在这个例子中，每个 reg 项都有三个 cell：片选号、偏移量和长度。

由于地址域是包含于一个节点及其子节点的，所以父节点可以自由的定义任何对于该总线来说有意义的编址方案。那些在直接父节点和子节点以外的节点通常不关心本地地址域，而地址应该从一个域映射到另一个域。

非内存映射设备

其他的设备没有被映射到处理机总线上。虽然这些设备可以有一个地址范围，但他们并不是由 CPU 直接访问。取而代之的是，父设备的驱动程序会代表 CPU 执行简介访问。

以 i2c 设备为例，每个设备都分配了一个地址，但并没有与之关联的长度或范围信息。这看起来和 CPU 的地址分配很像。

```
1               i2c@1,0 {
2                   compatible = "acme,a1234-i2c-bus";
3                   #address-cells = <1>;
4                   #size-cells = <0>;
5                   reg = <1 0 0x1000>;
6                   rtc@58 {
7                       compatible = "maxim,ds1338";
8                       reg = <58>;
9                   };
10              };
```

范围（地址转换）

我们已经讨论了如何给设备分配地址，但目前来说这些地址还只是设备节点的本地地址，我们还没有描述如何将这些地址映射成 CPU 可使用的地址。

根节点始终描述的是 CPU 视角的地址空间。根节点的子节点已经使用的是 CPU 的地址域，所以它们不需要任何直接映射。例如，serial@101f0000 设备就是直接分配的 0x101f0000

地址。

那些非根节点直接子节点的节点就没有使用 CPU 地址域。为了得到一个内存映射地址，设备树必须指定从一个域到另一个域地址转换地方法，而 ranges 属性就为此而生。

下面就是一个添加了 ranges 属性的示例设备树。

```
1  / {
2          compatible = "acme,coyotes-revenge";
3          #address-cells = <1>;
4          #size-cells = <1>;
5          ...
6          external-bus {
7                  #address-cells = <2>
8                  #size-cells = <1>;
9                  ranges = <0 0    0x10100000      0x10000          // Chipselect 1, Ethernet
10                                 1 0    0x10160000      0x10000          // Chipselect 2, i2c controller
11                                 2 0    0x30000000      0x1000000>; // Chipselect 3, NOR Flash
12
13                 ethernet@0,0 {
14                         compatible = "smc,smc91c111";
15                         reg = <0 0 0x1000>;
16                 };
17
18                 i2c@1,0 {
19                         compatible = "acme,a1234-i2c-bus";
20                         #address-cells = <1>;
21                         #size-cells = <0>;
22                         reg = <1 0 0x1000>;
23                         rtc@58 {
24                                 compatible = "maxim,ds1338";
25                                 reg = <58>;
26                         };
27                 };
28
29                 flash@2,0 {
30                         compatible = "samsung,k8f1315ebm", "cfi-flash";
31                         reg = <2 0 0x4000000>;
32                 };
33         };
34 };
```

ranges 是一个地址转换列表。ranges 表中的每一项都是一个包含子地址、父地址和在子地址空间中区域大小的元组。每个字段的值都取决于子节点的 #address-cells 、父节点的

#address-cells 和子节点的 #size-cells。以本例中的外部总线来说，子地址是 2 cell、父地址是 1 cell、区域大小也是 1 cell。那么三个 ranges 被翻译为：

从片选 0 开始的偏移量 0 被映射为地址范围：0x10100000..0x1010ffff
从片选 0 开始的偏移量 1 被映射为地址范围：0x10160000..0x1016ffff
从片选 0 开始的偏移量 2 被映射为地址范围：0x30000000..0x10000000

另外，如果父地址空间和子地址空间是相同的，那么该节点可以添加一个空的 range 属性。一个空的 range 属性意味着子地址将被 1:1 映射到父地址空间。

你有可能会问当全都可以设计成 1:1 映射的时候为何还要使用地址转换。答案就是，有一些具有完全不同地址空间的总线（比如 PCI），而它们的细节需要暴露给操作系统。另外一些带有 DMA 引擎的设备需要知道总线上的真实地址。有时有需要将设备组合到一块，因为他们共享相同的软件可编程物理地址映射。是否应该使用 1:1 映射在很大程度上取决于来自操作系统的信息以及硬件设计。

你还应该注意到在 i2c@1,0 节点中并没有 range 属性。不同于外部总线，这里的原因是 i2c 总线上的设备并没有被内存映射到 CPU 的地址域。相反，CPU 将通过 i2c@1,0 设备间接访问 rtc@58 设备。缺少 ranges 属性意味着这个设备将不能被出他的父设备之外的任何设备直接访问。

中断如何工作

与遵循树的自然结构而进行的地址转换不同，机器上的任何设备都可以发起和终止中断信号。另外地址的编址也不同于中断信号，前者是设备树的自然表示，而后者者表现为独立于设备树结构的节点之间的链接。描述中断连接需要四个属性：

- interrupt-controller - 一个空的属性定义该节点作为一个接收中断信号的设备。
- #interrupt-cells - 这是一个中断控制器节点的属性。它声明了该中断控制器的中断指示符中 cell 的个数（类似于 #address-cells 和 #size-cells）。
- interrupt-parent - 这是一个设备节点的属性，包含一个指向该设备连接的中断控制器的 phandle。那些没有 interrupt-parent 的节点则从它们的父节点中继承该属性。
- **interrupts** - 一个设备节点属性，包含一个中断指示符的列表，对应于该设备上的每个中断输出信号。

中断指示符是一个或多个 cell 的数据（由 #interrupt-cells 指定），这些数据指定了该设备连接至哪些输入中断。在以下的例子中，大部分设备都只有一个输出中断，但也有可能在一个设备上有多个输出中断。一个中断指示 符的意义完全取决于与中断控制器设备的 binding。每个中断控制器可以决定使用几个 cell 来唯一的定义一个输入中断。

下面的代码为我们 Coyote's Revenge 模型机添加了中断连接：

```
1  / {
```

```
2       compatible = "acme,coyotes-revenge";
3       #address-cells = <1>;
4       #size-cells = <1>;
5       interrupt-parent = <&intc>;
6
7       cpus {
8           #address-cells = <1>;
9           #size-cells = <0>;
10          cpu@0 {
11              compatible = "arm,cortex-a9";
12              reg = <0>;
13          };
14          cpu@1 {
15              compatible = "arm,cortex-a9";
16              reg = <1>;
17          };
18      };
19
20      serial@101f0000 {
21          compatible = "arm,pl011";
22          reg = <0x101f0000 0x1000 >;
23          interrupts = < 1 0 >;
24      };
25
26      serial@101f2000 {
27          compatible = "arm,pl011";
28          reg = <0x101f2000 0x1000 >;
29          interrupts = < 2 0 >;
30      };
31
32      gpio@101f3000 {
33          compatible = "arm,pl061";
34          reg = <0x101f3000 0x1000
35                  0x101f4000 0x0010>;
36          interrupts = < 3 0 >;
37      };
38
39      intc: interrupt-controller@10140000 {
40          compatible = "arm,pl190";
41          reg = <0x10140000 0x1000 >;
42          interrupt-controller;
43          #interrupt-cells = <2>;
44      };
45
```

```
46        spi@10115000 {
47              compatible = "arm,pl022";
48              reg = <0x10115000 0x1000 >;
49              interrupts = < 4 0 >;
50        };
51
52        external-bus {
53              #address-cells = <2>
54              #size-cells = <1>;
55              ranges = <0 0    0x10100000      0x10000          // Chipselect 1, Ethernet
56                        1 0    0x10160000      0x10000          // Chipselect 2, i2c controller
57                        2 0    0x30000000      0x1000000>; // Chipselect 3, NOR Flash
58
59              ethernet@0,0 {
60                    compatible = "smc,smc91c111";
61                    reg = <0 0 0x1000>;
62                    interrupts = < 5 2 >;
63              };
64
65              i2c@1,0 {
66                    compatible = "acme,a1234-i2c-bus";
67                    #address-cells = <1>;
68                    #size-cells = <0>;
69                    reg = <1 0 0x1000>;
70                    interrupts = < 6 2 >;
71                    rtc@58 {
72                          compatible = "maxim,ds1338";
73                          reg = <58>;
74                          interrupts = < 7 3 >;
75                    };
76              };
77
78              flash@2,0 {
79                    compatible = "samsung,k8f1315ebm", "cfi-flash";
80                    reg = <2 0 0x4000000>;
81              };
82        };
83  };
```

需要注意的事情：

- 这个机器只有一个中断控制器：interrupt-controller@10140000。
- 中断控制器节点上添加了‘inc:’标签，该标签用于给根节点的 interrupt-parent 属性分配一个 phandle。这个 interrupt-parent 将成为本系统的默认值，因为所有的子节点都将继承它，除非显示覆写这个属性。
- 每个设备使用 **interrupts** 属性来不同的中断输入线。

■ #interrupt-cells 是 2，所以每个中断指示符都有 2 个 cell。本例使用一种通用的模式，也就是用第一个 cell 来编码中断线号；然后用第二个 cell 编码标志位，比如高电平/低电平有效，或者边缘/水平触发。对于任何给定的中断控制器，请参考该控制器的 binding 文档以了解指示符如何编码。

设备特定数据

除了通用属性以外，一个节点中可以添加任何属性和子节点。只要遵循一些规则，可以添加任何操作系统所需要的数据。

首先，新的设备特定属性的名字都应该使用制造商前缀，以避免和现有标准属性名相冲突。

其次，属性和子节点的含义必须存档在 binding 文档中，以便设备驱动程序的程序员知道如何解释这些数据。一个 binding 记录了一个特定 compatible 值的意义、应该包含什么样的属性、有可能包含那些子节点、以及它代表了什么样的设备。每个特别的 compatible 值都应该有一个它自己的 binding（或者要求与其他 compatible 值兼容）。新设备的 binding 存档在本 wiki 中。请查看主页上的文档格式描述和审核流程
第三，使用邮件列表 devicetree-discuss@lists.ozlabs.org 发送新的 binding 以进行审核。新 binding 的审核可以捕获很多可能在以后导致问题的常见错误。

特殊节点

aliases 节点

引用一个特定的节点通常使用全路径，如 /external-bus/ethernet@0,0，但当用户真真想知道的只是“那个设备是 eth0？”时，这样的全路径就变得很冗长。这时，aliases 节点就可以用于指定一个设备全路径的别名。例如：

```
1       aliases {
2           ethernet0 = &eth0;
3           serial0 = &serial0;
4       };
```

当给一个设备分配一个识别符是操作系统将非常乐意使用别名。

在这里你会发现一个新语法。property = &label;，将作为字符串属性并通过引用标签来指定一个节点的全路径。这与之前的 phandle = < &label >; 形式不同，这是把一个 phandle 值插入进一个 cell。

chosen 节点

chosen 节点并不代表一个真正的设备，只是作为一个为固件和操作系统之间传递数据的地方，比如引导参数。chosen 节点里的数据也不代表硬件。通常，chosen 节点在 .dts 源文件中为空，并在启动时填充。

在我们的示例系统中，固件可以往 chosen 节点添加以下信息：

```
1        chosen {
2            bootargs = "root=/dev/nfs rw nfsroot=192.168.1.1 console=ttyS0,115200";
3        };
```

高级主题

高级模型机

现在，我们已经掌握了基本的定义，接下来让我们往模型机里添加一些硬件，以讨论一些更复杂的用例。

高级模型机添加了一个 PCI 主桥，其控制寄存器映射到内存 0x10180000，并且 BARs 编程至以地址 0x80000000 为起始。
既然关于设备树我们已经有所了解了，那么我们就从以下所示新增加的节点来介绍 PCI 主桥。

```
1            pci@10180000 {
2                compatible = "arm,versatile-pci-hostbridge", "pci";
3                reg = <0x10180000 0x1000>;
4                interrupts = <8 0>;
5            };
```

PCI 主桥

注，本节将假定读者了解 PCI 的一些基本知识。本文并不是 PCI 教程，想要了解更深入的信息，请阅读 [1]。你也可以参考 ePAPR 或还可以访问 http://devicetree.org/MPC5200:PCI，这里可以找到 Freescale MPC5200 的一个完整工作的例子。

PCI 总线编号

每个 PCI 总线段都是唯一编号的，并且总线的编号是通过使用 bus-ranges 属性在 pci 节点中暴露出来的，这个属性有两个 cell。第一个 cell 给出分配给该节点的总线号；第二个 cell 给出任何次级 PCI 总线最大总线号。
模型机只有一个 pci 总线，所以两个 cell 都是 0。

```
1            pci@0x10180000 {
2                compatible = "arm,versatile-pci-hostbridge", "pci";
3                reg = <0x10180000 0x1000>;
4                interrupts = <8 0>;
5                bus-ranges = <0 0>;
6            };
```

PCI 地址转换

类似于前面所描述的本地总线，PCI 地址空间和 CPU 地址空间是完全分离的，所以需要一个从 PCI 地址到 CPU 地址的转换。同样，完成这样的转换将使用 ranges、#address-cells 和 #size-cells 属性。

```
1           pci@0x10180000 {
2               compatible = "arm,versatile-pci-hostbridge", "pci";
3               reg = <0x10180000 0x1000>;
4               interrupts = <8 0>;
5               bus-ranges = <0 0>;
6
7               #address-cells = <3>
8               #size-cells = <2>;
9               ranges = <0x42000000 0 0x800000000x800000000 0x20000000
10                          0x02000000 0 0xa0000000 0xa0000000 0 0x10000000
11                          0x01000000 0 0x00000000 0xb0000000 0 0x01000000>;
12          };
```

正如你所看到的，子地址（PCI 地址）使用 3 个 cell，同时 PCI rangs 被编码为 2 个 cell。那么第一个问题就可能是，为什么我们要用三个 32 位 cell 去指定一个 PCI 地址？这三个 cell 分别标记了 phys.hi、phys.mid 和 phys.low[2]。

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

PCI 地址是 64 位的，并编码进了 phys.mid 和 phys.low。然而真正有意思的是在 phys.high 里棉面，这是一个位域。

- n：重定位区域标志（在这里不起作用）
- p：预取（可缓存）区标志
- t：地址别名标志（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 总线号。PCI 可以是分层结构，所以我们可能有 PCI/PCI 桥，这可以定义子总线。
- ddddd：设备号，通常与 IDSEL 型号相关联。
- fff：功能号。用于多功能 PCI 设备。
- rrrrrrrr：寄存器号，用于配置周期。

对于 PCI 地址转换来说，p 和 ss 是最重要的字段。在 phys.hi 里的 p 和 ss 的值决定了访问哪个 PCI 地址空间。因此，通过查找 ranges 属性，我们将得到三个区域。

- 以 PCI 地址 0x80000000 开始的一个 512 MByte 32 位预取存储区，该区域将映射到主机 CPU 地址 0x80000000。
- 以 PCI 地址 0xa0000000 开始的一个 265 MByte 32 位非预取存储区，该区域将映射到主机 CPU 地址 0xa0000000。
- 以 PCI 地址 0x00000000 开始的一个 16 MByte I/O 区，该区域将映射到主机 CPU 地址 0xb0000000。

为阻止这些工作，phys.hi 位域的存在就意味着操作系统必须知道该节点代表了一个 PCI 桥，这样操作系统才能为了地址转换而忽略那些不相关的字段。为了判断应该掩码哪些额外

的字段，操作系统需要在 PCI 总线节点中寻找“pci”字符串。

高级中断映射

现在我们来到了最有趣的部分，PCI 中断映射。一个 PCI 设备可以使用引线 #INTA、#INTB、#INTC 和 #INTD 来触发中断。如果我们没有多功能 PCI 设备，那么设备中断必须使用 #INTA。然而，每个 PCI 插槽或设备通常会连接到中断控制器上不同的输入端。所以设备树需要一种能将各个 PCI 中断信号映射到中断控制器的途径。#interrupt-cells、interrupt-map 和 interrupt-map-mask 属性就被用来描述这个中断映射。

这里所描述的中断映射并不仅仅局限于 PCI 总线，事实上，任何节点都可以指定复杂的中断映射，但 PCI 是最常见的情况。

```
1           pci@0x10180000 {
2               compatible = "arm,versatile-pci-hostbridge", "pci";
3               reg = <0x10180000 0x1000>;
4               interrupts = <8 0>;
5               bus-ranges = <0 0>;
6
7               #address-cells = <3>
8               #size-cells = <2>;
9               ranges = <0x42000000 0 0x80000000    0x80000000    0 0x20000000
10                          0x02000000 0 0xa0000000    0xa0000000    0 0x10000000
11                          0x01000000 0 0x00000000    0xb0000000    0 0x01000000>;
12
13              #interrupt-cells = <1>;
14              interrupt-map-mask = <0xf800 0 0 7>;
15              interrupt-map = <0xc000 0 0 1 &intc    9 3 // 1st slot
16                                      0xc000 0 0 2 &intc 10 3
17                                      0xc000 0 0 3 &intc 11 3
18                                      0xc000 0 0 4 &intc 12 3
19
20                                      0xc800 0 0 1 &intc 10 3 // 2nd slot
21                                      0xc800 0 0 2 &intc 11 3
22                                      0xc800 0 0 3 &intc 12 3
23                                      0xc800 0 0 4 &intc    9 3>;
24          };
```

首先你会发现，PCI 中断号只使用了一个 cell，不像系统中断控制器，它使用两个 cell，一个用于中断号，另一个用于标志。PCI 中断只使用了一个 cell，因为 PCI 中断确定为始终是低电平触发。

在这个示例板上，我们有 2 个分别包含 4 个中断线的 PCI 插槽，所以我们需要映射 8 个中断线到中断控制器上。这已经在 interrupt-map 属性中完成了。关于中断映射的具体步骤请参考 [3]。

因为要区分单一 PCI 总线上的若干 PCI 设备中断号（#INA 等）是不够用的，所以我们还需要指出是哪个 PCI 设备触发了中断线。幸运的是我们还可以使用每个设备所拥有的唯

一设备号。为了区分这些 PCI 设备，我们需要一个元组，该元组由 PCI 设备号和 PCI 中断号组成。通俗的说，我们构造了由四个 cell 组成的设备中断指示符。

- 三个 #address-cells 由 phys.hi、phys.mid、phys.low 组成，然后
- 一个 #interrupt-cell（#INTA、#INTB、#INTC、#INTD）

因为我们只需要 PCI 地址中的设备号部分，所以 interrupt-map-mask 发挥了作用。interrupt-map-mask 也是 4 元组，就像设备中断指示符一样。掩码的第一部分指出我们应该考虑设备中断指示符中哪一部分。在本例中，我们可以看到在 phys.hi 中只需要设备号部分，另外我们还需要 3 位来区分四个中断线（PCI 中断线是从 1 开始计数的，不是 0!）。

现在。我们可以构建 interrupt-map 属性了。该属性是一个表，这个表的每一项都由一个子（PCI 总线）设备中断指示符、一个父句柄（用于中断服务的中断控制器）和一个父设备中断指示符组成。因此，在第一行中我们可以知道 PCI 中断 #INTA 将被映射到中断控制器的 IRQ 9，并且是低电平有效。[4]

目前为止，唯一没有讨论的就是 PCI 总线设备中断指示符里 古怪的数字了。来自 phys.hi 位域的设备号是设备中断指示符中的重要组成部分。设备号是平台特定的，并取决于 PCI 主控制器如何激活各个设备的 IDSEL 管脚。在本例中，PCI slot 1 分配设备 id 24（0x18），PCI slot 2 分配设备 id 25（0x19）。每个 slot 的 phys.hi 值是通过将设备号左移 11 位至位域的 ddddd 段得到的，就像下面：

- slot 1 的 phys.hi 就是 0xC000， 并且
- slot 2 的 phys.hi 就是 0xC800。

把这些放在一起之后，interrupt-map 属性就显示为：

- 在主中断控制器上 slot 1 的 #INTA 是 IRQ9，低电平触发
- 在主中断控制器上 slot 1 的 #INTB 是 IRQ10，低电平触发
- 在主中断控制器上 slot 1 的 #INTC 是 IRQ11，低电平触发
- 在主中断控制器上 slot 1 的 #INTD 是 IRQ12，低电平触发
- 在主中断控制器上 slot 2 的 #INTA 是 IRQ10，低电平触发
- 在主中断控制器上 slot 2 的 #INTA 是 IRQ11，低电平触发
- 在主中断控制器上 slot 2 的 #INTA 是 IRQ12，低电平触发
- 在主中断控制器上 slot 2 的 #INTA 是 IRQ9，低电平触发

属性 **interrupts** = <8 0>; 描述了主控制器或 PCI 桥控制器本身有可能触发中断。不要与 PCI 设备触发的中断（使用 INTA，INTB，...）告混了。

最后需要注意的事。就像 interrupt-parent 属性一样，节点中 interrupt-map 属性的存在将改变子节点和孙节点的默认中断控制器。在这个 PCI 示例中，这意味着 PCI 主桥变成了默认中断控制器。如果一个通过 PCI 总线连接的设备同时还直接连接至另一个中断控制器，这时就需要指定它自己的 interrupt-parent 属性。

附注

---

[1] Tom Shanley / Don Anderson: PCI System Architecture. Mindshare Inc.

[2] PCI Bus Bindings to Open Firmware.

[3] Open Firmware Recommended Practice: Interrupt Mapping

[4] PCI **interrupts** are always level low sensitive.

PCI 中断总是低电平触发。

原文： http://devicetree.org/Device_Tree_Usage#How_Interrupts_Work