设备树使用手册

This page walks through how to write a device tree for a new machine. It is intended to provide an overview of device tree concepts and how they are used to describe a machine.
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念的概述和如何使用这些设备树来描述一个机器。
For a full technical description of device tree data format, refer to the ePAPR specification. The ePAPR specification covers a lot more detail than the basic topics covered on this page, please refer to it for more advanced usage that isn't covered by this page.
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。

Contents

Basic Data Format

基本数据格式

The device tree is a simple tree structure of nodes and properties. Properties are key-value pairs, and node may contain both properties and child nodes. For example, the following is a simple tree in the .dts format:
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：

```
/ {
    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 {
            };
        };
    };
```
This tree is obviously pretty useless because it doesn't describe anything, but it does show the structure of nodes an properties. There is:
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：

- a single root node: "/"
  一个单独的根节点： “/”
- a couple of child nodes: "node1" and "node2"
  两个子节点： “node1” 和 “node2”
- a couple of children for node1: "child-node1" and "child-node2"
  两个 node1 的子节点： “child-node1” 和 “child-node2”
- a bunch of properties scattered through the tree.
  一堆分散在树里的属性。

Properties are simple key-value pairs where the value can either be empty or contain an arbitrary byte stream. While data types are not encoded into the data structure, there are a few fundamental data representations that can be expressed in a device tree source file.
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。

- Text strings (null terminated) are represented with double quotes:
  文本字符串（无结束符）可以用双引号表示：
  string-property = "a string"
- 'Cells' are 32 bit unsigned integers delimited by angle brackets:
  ‘Cells’ 是 32 位无符号整数，用尖括号限定：
  cell-property = <0xbeef 123 0xabcd1234>
- binary data is delimited with square brackets:
  二进制数据用方括号限定：
  binary-property = [0x01 0x23 0x45 0x67];
- Data of differing representations can be concatenated together using a comma:
  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>;
- Commas are also used to create lists of strings:
  逗号也可用于创建字符串列表：
  string-list = "red fish", "blue fish";

Basic Concepts

基本概念

To understand how the device tree is used, we will start with a simple machine and build up a device tree to describe it step by step.
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。

Sample Machine

模型机
Consider the following imaginary machine (loosely based on ARM Versatile), manufactured by "Acme" and named "Coyote's Revenge":
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”：

- One 32bit ARM CPU
  一个 32 位 ARM CPU
- processor local bus attached to memory mapped serial port, spi bus controller, i2c controller, interrupt controller, and external bus bridge
  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 of SDRAM based at 0
  256MB SDRAM 起始地址为 0
- 2 Serial ports based at 0x101F1000 and 0x101F2000
  两个串口起始地址：0x101F0000 和 0x101F2000
- GPIO controller based at 0x101F3000
  GPIO 控制器起始地址：0x101F3000
- SPI controller based at 0x10170000 with following devices
  带有以下设备的 SPI 控制器起始地址：0x10170000
  - MMC slot with SS pin attached to GPIO #1
    MMC 插槽的 SS 管脚连接至 GPIO #1
- External bus bridge with following devices
  外部总线桥挂载以下设备
  - SMC SMC91111 Ethernet device attached to external bus based at 0x10100000

SMC SMC91111 以太网设备连接到外部总线，起始地址：0x10100000

- i2c controller based at 0x10160000 with following devices
  i2c 控制器起始地址：0x10160000，并挂载以下设备
  - Maxim DS1338 real time clock. Responds to slave address 1101000 (0x58)
    Maxim DS1338 实时时钟。响应至从地址 1101000 (0x58)
- 64MB of NOR flash based at 0x30000000
  64MB NOR 闪存起始地址 0x30000000

Initial structure

初始结构

The first step is to lay down a skeleton structure for the machine. This is the bare minimum structure required for a valid device tree. At this stage you want to uniquely identify the machine.
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。

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

compatible specifies the name of the system. It contains a string in the form "<manufacturer>,<model>. It is important to specify the exact device, and to include the manufacturer name to avoid namespace collisions. Since the operating system will use the compatible value to make decisions about how to run on the machine, it is very important to put correct data into this property.
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。
Theoretically, compatible is all the data an OS needs to uniquely identify a machine. If all the machine details are hard coded, then the OS could look specifically for "acme,coyotes-revenge" in the top level compatible property.
理论上讲，兼容性（compatible）就是操作系统需要的所有数据都唯一标识一个机器。如果机器的所有细节都是硬编码的，那么操作系统则可以在顶层的 compatible 属性中具体查看“acme,coyotes-revenge”。

CPUs

中央处理器

Next step is to describe for each of the CPUs. A container node named "cpus" is added with a child node for each CPU. In this case the system is a dual-core Cortex A9 system from ARM.
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。

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

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

The compatible property in each cpu node is a string that specifies the exact cpu model in the form <manufacturer>,<model>, just like the compatible property at the top level.
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。
More properties will be added to the cpu nodes later, but we first need to talk about more of the basic concepts.
稍后将会有更多的属性添加进 cpu 节点，但我们先得讨论一些更多的基本概念。

Node Names

节点名称

It is worth taking a moment to talk about naming conventions. Every node must have a name in the form <name>[@<unit-address>].
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。
<name> is a simple ascii string and can be up to 31 characters in length. In general, nodes are named according to what kind of device it represents. ie. A node for a 3com Ethernet adapter would be use the name ethernet, not 3com509.
<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。
The unit-address is included if the node describes a device with an address. In general, the unit address is the primary address used to access the device, and is listed in the node's reg property. We'll cover the reg property later in this document.
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。
Sibling nodes must be uniquely named, but it is normal for more than one node to use the same generic name so long as the address is different (ie, serial@101f1000 & serial@101f2000).
See section 2.2.1 of the ePAPR spec for full details about node naming.

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。

Devices

设备

Every device in the system is represented by a device tree node. The next step is to populate the tree with a node for each of the devices. For now, the new nodes will be left empty until we can talk about how address ranges and irqs are handled.

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。

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

In this tree, a node has been added for each device in the system, and the hierarchy reflects the how devices are connected to the system. ie. devices on the extern bus are children of the external bus node, and i2c devices are children of the i2c bus controller node. In general, the hierarchy represents the view of the system from the perspective of the CPU.

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。

This tree isn't valid at this point. It is missing information about connections between devices. That data will be added later.

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。

Some things to notice in this tree:

在这颗树中，应该注意这些事情：

- Every device node has a compatible property.

每个设备节点都拥有一个 compatible 属性。

- The flash node has 2 strings in the compatible property. Read on to the next section to learn why.

  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。

- As mentioned earlier, node names reflect the type of device, not the particular model. See section 2.2.2 of the ePAPR spec for a list of defined generic node names that should be used wherever possible.

  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。

Understanding the compatible Property

理解 *compatible* 属性

Every node in the tree that represents a device is required to have the compatible property. compatible is the key an operating system uses to decide which device driver to bind to a device.
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 is a list of strings. The first string in the list specifies the exact device that the node represents in the form "<manufacturer>,<model>". The following strings represent other devices that the device is compatible with.
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。
For example, the Freescale MPC8349 System on Chip (SoC) has a serial device which implements the National Semiconductor ns16550 register interface. The compatible property for the MPC8349 serial device should therefore be: compatible = "fsl,mpc8349-uart", "ns16550". In this case, fsl,mpc8349-uart specifies the exact device, and ns16550 states that it is register-level compatible with a National Semiconductor 16550 UART.
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。
Note: ns16550 doesn't have a manufacturer prefix purely for historical reasons. All new compatible values should use the manufacturer prefix.
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。
This practice allows existing device drivers to be bound to a newer device, while still uniquely identifying the exact hardware.
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。
Warning: Don't use wildcard compatible values, like "fsl,mpc83xx-uart" or similar. Silicon vendors will invariably make a change that breaks your wildcard assumptions the moment it is too late to change it. Instead, choose a specific silicon implementations and make all subsequent silicon compatible with it.
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。

How Addressing Works

如何编址

Devices that are addressable use the following properties to encode address information into the device tree:
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

Each addressable device gets a reg which is a list of tuples in the form reg = <address1 length1 [address2 length2] [address3 length3] ... >. Each tuple represents an address range used by the device. Each address value is a list of one or more 32 bit integers called cells. Similarly, the length value can either be a list of cells, or empty.
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。
Since both the address and length fields are variable of variable size, the #address-cells and #size-cells properties in the parent node are used to state how many cells are in each field. Or in other words, interpreting a reg property correctly requires the parent node's #address-cells and #size-cells values. To see how this all works, lets add the addressing properties to the sample device tree, starting with the CPUs.
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 addressing

CPU 编址

The CPU nodes represent the simplest case when talking about addressing. Each CPU is assigned a single unique ID, and there is no size associated with CPU ids.
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。

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

```
            cpu@1 {
                compatible = "arm,cortex-a9";
                reg = <1>;
            };
        };
```
In the cpus node, #address-cells is set to 1, and #size-cells is set to 0. This means that child reg values are a single uint32 that represent the address with no size field. In this case, the two cpus are assigned addresses 0 and 1. #size-cells is 0 for cpu nodes because each cpu is only assigned a single address.

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。

You'll also notice that the reg value matches the value in the node name. By convention, if a node has a reg property, then the node name must include the unit-address, which is the first address value in the reg property.

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。

Memory Mapped Devices

内存映射设备

Instead of single address values like found in the cpu nodes, a memory mapped device is assigned a range of addresses that it will respond to. #size-cells is used to state how large the length field is in each child reg tuple. In the following example, each address value is 1 cell (32 bits), and each length value is also 1 cell, which is typical on 32 bit systems. 64 bit machines may use a value of 2 for #address-cells and #size-cells to get 64 bit addressing in the device tree.

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。

```
    / {
        #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 >;
        };

        ...
    };
```
Each device is assigned a base address, and the size of the region it is assigned. The GPIO device address in this example is assigned two address ranges; 0x101f3000...0x101f3fff and 0x101f4000..0x101f400f.

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。

Some devices live on a bus with a different addressing scheme. For example, a device can be attached to an external bus with discrete chip select lines. Since each parent node defines the addressing domain for its children, the address mapping can be chosen to best describe the system. The code below show address assignment for devices attached to the external bus with the chip select number encoded into the address.

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。

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

The external-bus uses 2 cells for the address value; one for the chip select number, and one for the offset from the base of the chip select. The length field remains as a single cell since only the offset portion of the address needs to have a range. So, in this example, each reg entry contains 3 cells; the chipselect number, the offset, and the length.

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。

Since the address domains are contained to a node and its children, parent nodes are free to define whatever addressing scheme makes sense for the bus. Nodes outside of the immediate parent and child nodes do not normally have to care about the local addressing domain, and addresses have to be mapped to get from one domain to another.

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。

Non Memory Mapped Devices

非内存映射设备

Other devices are not memory mapped on the processor bus. They can have address ranges, but they are not directly accessible by the CPU. Instead the parent device's driver would perform indirect access on behalf of the CPU.

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 执行间接访问。

To take the example of i2c devices, each device is assigned an address, but there is no length or range associated with it. This looks much the same as CPU address assignments.

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。

```
        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 (Address Translation)

范围（地址转换）

We've talked about how to assign addresses to devices, but at this point those addresses are only local to the device node. It doesn't yet describe how to map from those address to an address that the CPU can use.

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。

The root node always describes the CPU's view of the address space. Child nodes of the root are already using the CPU's address domain, and so do not need any explicit mapping. For example, the serial@101f0000 device is directly assigned the address 0x101f0000.

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。

Nodes that are not direct children of the root do not use the CPU's address domain. In order to get a memory mapped address the device tree must specify how to translate addresses from one domain to another. The ranges property is used for this purpose.

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。

Here is the sample device tree with the ranges property added.

下面就是一个添加了 ranges 属性的示例设备树。

```
    / {
        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   0x10000000>; // 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'属性的每个元素是三元组，按照前后顺序分别是(子总线地址，父总线地址，大小)。

子总线地址需要几个u32表示由'ranges'属性所在节点的'#address-cells'属性决定，

父总线地址需要几个u32表示由上一级节点的'#address-cells'属性决定，

大小需要几个u32表示由当前节点的'#size- cells'属性确定。

ranges is a list of address translations. Each entry in the ranges table is a tuple containing the child address, the parent address, and the size of the region in the child address space. The size of each field is determined by taking the child's #address-cells value, the parent's #address-cells value, and the child's #size-cells value. For the external bus in our example, the child address is 2 cells, the parent address is 1 cell, and the size is also 1 cell. Three ranges are being translated:

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

Offset 0 from chip select 0 is mapped to address range 0x10100000..0x1010ffff

从片选 0 开始的偏移量 0 被映射为地址范围：0x10100000..0x1010ffff

Offset 0 from chip select 1 is mapped to address range 0x10160000..0x1016ffff

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

Offset 0 from chip select 2 is mapped to address range 0x30000000..0x10000000

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

```
ranges = <
0 0 0x10100000 0x10000 // Chipselect 1, Ethernet
1 0 0x10160000 0x10000 // Chipselect 2, i2c controller
2 0 0x30000000 0x10000000 // Chipselect 3, NOR Flash
>;
```

Alternately, if the parent and child address spaces are identical, then a node can instead add an empty ranges property. The presence of an empty ranges property means addresses in the child address space are mapped 1:1 onto the parent address space.

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。

You might ask why address translation is used at all when it could all be written with 1:1 mapping. Some busses (like PCI) have entirely different address spaces whose details need to be exposed to the operating system. Others have DMA engines which need to know the real address on the bus. Sometimes devices need to be grouped together because they all share the same software programmable physical address mapping. Whether or not 1:1 mappings should be used depends a lot on the information needed by the Operating system, and on the hardware design.

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。有时又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。

You should also notice that there is no ranges property in the i2c@1,0 node. The reason for this is that unlike the external bus, devices on the i2c bus are not memory mapped on the CPU's address domain. Instead, the CPU indirectly accesses the rtc@58 device via the i2c@1,0 device. The lack of a ranges property means that a device cannot be directly accessed by any device other than it's parent.

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。

How Interrupts Work

中断如何工作

Unlike address range translation which follows the natural structure of the tree, Interrupt signals can originate from and terminate on any device in a machine. Unlike device addressing which is naturally expressed in the device tree, interrupt signals are expressed as links between nodes independent of the tree. Four properties are used to describe interrupt connections:

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 - An empty property declaring a node as a device that receives interrupt signals 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 - This is a property of the interrupt controller node. It states how many cells are in an interrupt specifier for this interrupt controller (Similar to #address-cells and #size-cells).

#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 - A property of a device node containing a phandle to the interrupt controller that it is attached to. Nodes that do not have an interrupt-parent property can also inherit the property from their parent node.

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 - A property of a device node containing a list of interrupt specifiers, one for each interrupt output signal on the device.

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。

An interrupt specifier is one or more cells of data (as specified by #interrupt-cells) that specifies which interrupt input the device is attached to. Most devices only have a single interrupt output as shown in the example below, but it is possible to have multiple interrupt outputs on a device. The meaning of an interrupt specifier depends entirely on the binding for the interrupt controller device. Each interrupt controller can decide how many cells it need to uniquely define an interrupt input.

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。

The following code adds interrupt connections to our Coyote's Revenge example machine:

下面的代码为我们 Coyote's Revenge 模型机添加了中断连接：

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

Some things to notice:
需要注意的事情：

- The machine has a single interrupt controller, interrupt-controller@10140000.
  这个机器只有一个中断控制器：interrupt-controller@10140000。
- The label 'intc:' has been added to the interrupt controller node, and the label was used to assign a phandle to the interrupt-parent property in the root node. This interrupt-parent value becomes the default for the system because all child nodes inherit it unless it is explicitly overridden.
  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。
- Each device uses an interrupt property to specify a different interrupt input line.
  每个设备使用 interrupts 属性来标识不同的中断输入线。
- #interrupt-cells is 2, so each interrupt specifier has 2 cells. This example uses the common pattern of using the first cell to encode the interrupt line number, and the second cell to encode flags such as active high vs. active low, or edge vs. level sensitive. For any given interrupt controller, refer to the controller's binding documentation to learn how the specifier is encoded.
  #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。

Device Specific Data

设备特定数据

Beyond the common properties, arbitrary properties and child nodes can be added to nodes. Any data needed by the operating system can be added as long as some rules are followed.
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。
First, new device-specific property names should use a manufacture prefix so that they don't conflict with existing standard property names.
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。
Second, the meaning of the properties and child nodes must be documented in a binding so that a device driver author knows how to interpret the data. A binding documents what a particular compatible value means, what properties it should have, what child nodes it might have, and what device it represents. Each unique compatible value should have its own binding (or claim compatibility with another compatible value). Bindings for new devices are documented in this wiki. See the Main Page for a description of the documentation format and review process.
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。
Third, post new bindings for review on the devicetree-discuss@lists.ozlabs.org mailing list. Reviewing new bindings catches a lot of common mistakes that will cause problems in the future.
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。

Special Nodes

特殊节点
----

aliases Node

aliases 节点
A specific node is normally referenced by the full path, like /external-bus/ethernet@0,0, but that gets cumbersome when what a user really wants to know is, "which device is eth0?" The aliases node can be used to assign a short alias to a full device path. For example:
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：

```
        aliases {
            ethernet0 = &eth0;
            serial0 = &serial0;
        };
```

The operating system is welcome to use the aliases when assigning an identifier to a device.
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。
You'll notice a new syntax used here. The property = &label; syntax assigns the full node path referenced by the label as a string property. This is different from the phandle = < &label >; form used earlier which inserts a phandle value into a cell.
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 Node

chosen 节点
The chosen node doesn't represent a real device, but serves as a place for passing data between firmware and the operating system, like boot arguments. Data in the chosen node does not represent the hardware. Typically the chosen node is left empty in .dts source files and populated at boot time.
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。
In our example system, firmware might add the following to the chosen node:
在我们的示例系统中，固件可以往 chosen 节点添加以下信息：

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

Advanced Topics

高级主题

Advanced Sample Machine

高级模型机
Now that we've got the basics defined, let's add some hardware to the sample machine to discuss some of the more complicated use cases.
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。
The advanced sample machine adds a PCI host bridge with control registers memory mapped to 0x10180000, and BARs programmed to start above the address 0x80000000.
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。
Given what we already know about the device tree, we can start with the addition of the following node to describe the PCI host bridge.
既然关于设备树我们已经有所了解，那么我们就从以下所示新增加的节点来介绍 PCI 主桥。

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

PCI Host Bridge

PCI 主桥
This section describes the Host/PCI bridge node.
本节介绍 Host/PCI 桥节点。
Note, some basic knowledge of PCI is assumed in this section. This is NOT a tutorial about PCI, if you need some more in depth information, please read[1]. You can also refer to either ePAPR or the PCI Bus Binding to Open Firmware. A complete working example for a Freescale MPC5200 can be found here.

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 PCI Bus Binding to Open Firmware（http://playground.sun.com/1275/bindings/pci/pci2_1.pdf）。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 Bus numbering

PCI 总线编号

Each PCI bus segment is uniquely numbered, and the bus numbering is exposed in the pci node by using the bus-ranges property, which contains two cells. The first cell gives the bus number assigned to this node, and the second cell gives the maximum bus number of any of the subordinate PCI busses.
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。
The sample machine has a single pci bus, so both cells are 0.
模型机只有一个 pci 总线，所以两个 cell 都是 0。

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

PCI Address Translation

PCI 地址转换

Similar to the local bus described earlier, the PCI address space is completely separate from the CPU address space, so address translation is needed to get from a PCI address to a CPU address. As always, this is done using the range, #address-cells, and #size-cells properties.
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。

```
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>;
};
```

As you can see, child addresses (PCI addresses) use 3 cells, and PCI ranges are encoded into 2 cells. The first question might be, why do we need three 32 bit cells to specify a PCI address. The three cells are labeled phys.hi, phys.mid and phys.low [2].
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 addresses are 64 bits wide, and are encoded into phys.mid and phys.low. However, the really interesting things are in phys.high which is a bit field:
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: relocatable region flag (doesn't play a role here)
  n：重定位区域标志（在这里不起作用）
- p: prefetchable (cacheable) region flag
  p：预取（可缓存）区标志
- t: aliased address flag (doesn't play a role here)
  t：地址别名标志（在这里不起作用）
- ss: space code
  ss：空间代码
  - 00: configuration space
    00：配置空间
  - 01: I/O space
    01：I/O 空间
  - 10: 32 bit memory space
    10：32 位内存空间
  - 11: 64 bit memory space
    11：64 位内存空间
- bbbbbbbb: The PCI bus number. PCI may be structured hierarchically. So we may have PCI/PCI bridges which will define sub busses.
  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: The device number, typically associated with IDSEL signal connections.
  ddddd：设备号，通常与 IDSEL 型号相关联。
- fff: The function number. Used for multifunction PCI devices.

fff：功能号。用于多功能 PCI 设备。

- rrrrrrrr: Register number; used for configuration cycles.

  rrrrrrrr：寄存器号，用于配置周期。

For the purpose of PCI address translation, the important fields are p and ss. The value of p and ss in phys.hi determines which PCI address space is being accessed. So looking onto our ranges property, we have three regions:

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。

- a 32 bit prefetchable memory region beginning on PCI address 0x80000000 of 512 MByte size which will be mapped onto address 0x80000000 on the host CPU.

  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。

- a 32 bit non-prefetchable memory region beginning on PCI address 0xa0000000 of 256 MByte size which will be mapped onto address 0xa0000000 on the host CPU.

  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。

- an I/O region beginning on PCI address 0x00000000 of 16 MByte size which will be mapped onto address 0xb0000000 on the host CPU.

  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。

To throw a wrench into the works, the presence of the phys.hi bitfield means that an operating system needs to know that the node represents a PCI bridge so that it can ignore the irrelevant fields for the purpose of translation. An OS will look for the string "pci" in the PCI bus nodes to determine whether it needs to mask of the extra fields.

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。

Advanced Interrupt Mapping

高级中断映射

Now we come to the most interesting part, PCI interrupt mapping. A PCI device can trigger interrupts using the wires #INTA, #INTB, #INTC and #INTD. If we don't have multifunction PCI devices, a device is obligated to use #INTA for interrupts. However, each PCI slot or device is typically wired to different inputs on the interrupt controller. So, the device tree needs a way of mapping each PCI interrupt signal to the inputs of the interrupt controller. The #interrupt-cells, interrupt-map and interrupt-map-mask properties are used to describe the interrupt mapping.

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。

Actually, the interrupt mapping described here isn't limited to PCI busses, any node can specify complex interrupt maps, but the PCI case is by far the most common.

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。

```
            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>;
            };
```

First you'll notice that PCI interrupt numbers use only one cell, unlike the system interrupt controller which uses 2 cells; one for the irq number, and one for flags. PCI only needs one cell for interrupts because PCI interrupts are specified to always be level-low sensitive.

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。

In our example board, we have 2 PCI slots with 4 interrupt lines, respectively, so we have to map 8 interrupt lines to the interrupt controller. This is done using the interrupt-map property. The exact procedure for interrupt mapping is described in[3] .

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]。

Because the interrupt number (#INTA etc.) is not sufficient to distinguish between several PCI devices on a single PCI bus, we also have to denote which PCI device triggered the interrupt line. Fortunately, every PCI device has a unique device number that we can use for. To distinguish between interrupts of several PCI devices we need a tuple consisting of the PCI device number and the PCI interrupt number. Speaking more generally, we construct a unit interrupt specifier which has four cells:

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。

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

Because we only need the device number part of the PCI address, the interrupt-map-mask property comes into play. interrupt-map-mask is also a 4-tuple like the unit interrupt specifier. The 1's in the mask denote which part of the unit interrupt specifier should be taken into account. In our example we can see that only the device number part of phys.hi is required and we need 3 bits to distinguish between the four interrupt lines (Counting PCI interrupt lines start at 1, not at 0!).
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！）。
Now we can construct the interrupt-map property. This property is a table and each entry in this table consists of a child (PCI bus) unit interrupt specifier, a parent handle (the interrupt controller which is responsible for serving the interrupts) and a parent unit interrupt specifier. So in the first line we can read that the PCI interrupt #INTA is mapped onto IRQ 9, level low sensitive of our interrupt controller. [4].
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]
The only missing part for now are the weird numbers int the PCI bus unit interrupt specifier. The important part of the unit interrupt specifier is the device number from the phys.hi bit field. Device number is board specific, and it depends on how each PCI host controller activates the IDSEL pin on each device. In this example, PCI slot 1 is assigned device id 24 (0x18), and PCI slot 2 is assigned device id 25 (0x19). The value of phys.hi for each slot is determined by shifting the device number up by 11 bits into the ddddd section of the bitfield as follows:
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：

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

Putting it all together the interrupt-map property show:
把这些放在一起之后，interrupt-map 属性就显示为：

- #INTA of slot 1 is IRQ9, level low sensitive on the primary interrupt controller
  在主中断控制器上 slot 1 的 #INTA 是 IRQ9，低电平触发
- #INTB of slot 1 is IRQ10, level low sensitive on the primary interrupt controller
  在主中断控制器上 slot 1 的 #INTB 是 IRQ10，低电平触发
- #INTC of slot 1 is IRQ11, level low sensitive on the primary interrupt controller
  在主中断控制器上 slot 1 的 #INTC 是 IRQ11，低电平触发
- #INTD of slot 1 is IRQ12, level low sensitive on the primary interrupt controller
  在主中断控制器上 slot 1 的 #INTD 是 IRQ12，低电平触发

and

- #INTA of slot 2 is IRQ10, level low sensitive on the primary interrupt controller
  在主中断控制器上 slot 2 的 #INTA 是 IRQ10，低电平触发
- #INTB of slot 2 is IRQ11, level low sensitive on the primary interrupt controller
  在主中断控制器上 slot 2 的 #INTA 是 IRQ11，低电平触发
- #INTC of slot 2 is IRQ12, level low sensitive on the primary interrupt controller
  在主中断控制器上 slot 2 的 #INTA 是 IRQ12，低电平触发
- #INTD of slot 2 is IRQ9, level low sensitive on the primary interrupt controller
  在主中断控制器上 slot 2 的 #INTA 是 IRQ9，低电平触发

The interrupts = <8 0>; property describes the interrupts the host/PCI-bridge controller itself may trigger. Don't mix up these interrupts with interrupts PCI devices might trigger (using INTA, INTB, ...).
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。
One final thing to note. Just like with the interrupt-parent property, the presence of an interrupt-map property on a node will change the default interrupt controller for all child and grandchild nodes. In this PCI example, that means that the PCI host bridge becomes the default interrupt controller. If a device attached via the PCI bus has a direct connection to another interrupt controller, then it also needs to specify its own interrupt-parent property.
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。

Notes

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。