Device Tree Usage

From eLinux.org

Top Device Tree page

(This page was previously located at htttp://devicetree.org/Device_Tree_Usage)

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.

For a full technical description of device tree data format, refer to the ePAPR v1.1 (https://www.power.org/documentation/power-org-standard-for-embedded-power-architecture-platform-requirements-epapr-v1-1-2/) 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. The ePAPR is currently being updated with a new name of Devicetree Specification Documentation.

Contents

Basic Data Format

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:

```
/dts-v1/;

/ {
    node1 {
        a-string-property = "A string";
        a-string-list-property = "first string", "second string";
        // hex is implied in byte arrays. no '0x' prefix is required
        a-byte-data-property = [01 23 34 56];
        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 and properties. There is:

- a single root node: "`/`"
- a couple of child nodes: "`node1`" and "`node2`"
- a couple of children for node1: "`child-node1`" and "`child-node2`"
- a bunch of properties scattered through the tree.

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.

- Text strings (null terminated) are represented with double quotes:
  - `string-property = "a string";`
- 'Cells' are 32 bit unsigned integers delimited by angle brackets:
  - `cell-property = <0xbeef 123 0xabcd1234>;`
- Binary data is delimited with square brackets:
  - `binary-property = [0x01 0x23 0x45 0x67];`
- Data of differing representations can be concatenated together using a comma:
  - `mixed-property = "a string", [0x01 0x23 0x45 0x67], <0x12345678>;`
- Commas are also used to create lists of strings:
  - `string-list = "red fish", "blue fish";`

Basic Concepts

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.

Sample Machine

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

- One 32bit ARM CPU
- processor local bus attached to memory mapped serial port, spi bus controller, i2c controller, interrupt controller, and external bus bridge
- 256MB of SDRAM based at 0
- 2 Serial ports based at 0x101F1000 and 0x101F2000
- GPIO controller based at 0x101F3000
- SPI controller based at 0x10170000 with following devices
    - MMC slot with SS pin attached to GPIO #1
- External bus bridge with following devices
    - SMC SMC91111 Ethernet device attached to external bus based at 0x10100000
    - i2c controller based at 0x10160000 with following devices
        - Maxim DS1338 real time clock. Responds to slave address 1101000 (0x58)
    - 64MB of NOR flash based at 0x30000000

Initial structure

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.

```
/dts-v1/;

/ {
    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.

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.

CPUs

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.

```
/dts-v1/;

/ {
    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.

More properties will be added to the cpu nodes later, but we first need to talk about more of the basic concepts.

Node Names

It is worth taking a moment to talk about naming conventions. Every node must have a name in the form `<name>[@<unit-address>]`.

`<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`.

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.

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.

Devices

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.

```
/dts-v1/;

/ {
    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.

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

Some things to notice in this tree:

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

Understanding the `compatible` Property

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.

`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.

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.

Note: `ns16550` doesn't have a manufacturer prefix purely for historical reasons. All new compatible values should use the manufacturer prefix.

This practice allows existing device drivers to be bound to a newer device, while still uniquely identifying the exact hardware.

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.

How Addressing Works

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

- `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.

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.

CPU addressing

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.

```
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.

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.

Memory Mapped Devices

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.

```
/dts-v1/;

/ {
    #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.

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.

```
    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.

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.

Non Memory Mapped Devices

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.

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.

```
        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)

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.

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.

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.

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

```
/dts-v1/;

/ {
    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` 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:

- Offset 0 from chip select 0 is mapped to address range 0x10100000..0x1010ffff
- Offset 0 from chip select 1 is mapped to address range 0x10160000..0x1016ffff
- Offset 0 from chip select 2 is mapped to address range 0x30000000..0x30ffffff

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.

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.

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.

How Interrupts Work

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:

- `interrupt-controller` - An empty property declaring a node as a device that receives interrupt signals

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

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.

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

```
/dts-v1/;

/ {
    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:

- The machine has a single interrupt controller, 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.
- Each device uses an interrupt property to specify a different interrupt input line.
- #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.

Device Specific Data

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.

First, new device-specific property names should use a manufacture prefix so that they don't conflict with existing standard property names.

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.

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.

Special Nodes

`aliases` Node

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:

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

The operating system is welcome to use the aliases when assigning an identifier to a device.

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.

`chosen` Node

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.

In our example system, firmware might add the following to the chosen node:

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

Advanced Topics

Advanced Sample Machine

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.

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.

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.

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

```
            interrupts = <8 0>;
        };
```

PCI Host Bridge

This section describes the Host/PCI bridge node.

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 v1.1 (https://www.power.org/documentation/power-org-standard-for-embedded-power-architecture-platform-requirements-epapr-v1-1-2/) or the PCI Bus Binding to Open Firmware (http://playground.sun.com/1275/bindings/pci/pci2_1.pdf). A complete working example for a Freescale MPC5200 can be found here.

PCI Bus numbering

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.

The sample machine has a single pci bus, so both cells are 0.

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

PCI Address Translation

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.

```
        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].

- `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:

- `n`: relocatable region flag (doesn't play a role here)
- `p`: prefetchable (cacheable) region flag
- `t`: aliased address flag (doesn't play a role here)
- `ss`: space code
  - 00: configuration space
  - 01: I/O space
  - 10: 32 bit memory space
  - 11: 64 bit memory space
- `bbbbbbbb`: The PCI bus number. PCI may be structured hierarchically. So we may have PCI/PCI bridges which will define sub busses.
- `ddddd`: The device number, typically associated with IDSEL signal connections.
- `fff`: The function number. Used for multifunction PCI devices.
- `rrrrrrrr`: Register number; used for configuration cycles.

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:

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

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.

Advanced Interrupt Mapping

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. A single-function device is obligated to use #INTA for interrupts. A multi-function device must use #INTA if it uses a single interrupt pin, #INTA and #INTB if it uses two interrupt pins, etc. Due to these rules, #INTA is normally used by more functions than #INTB, #INTC, and #INTD. To distribute the load across the four IRQ lines backing #INTA through #INTD, each PCI slot or device is typically wired to different inputs on the interrupt controller in rotating manner so as to avoid having all #INTA clients connected to the same incoming interrupt line. This procedure is referred to as *swizzling* the interrupts. 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.

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.

```
        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.

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] .

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:

- three #address-cells consisting of phys.hi, phys.mid, phys.low, and
- one #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!).

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].

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:

- phys.hi for slot 1 is 0xC000, and
- phys.hi for slot 2 is 0xC800.

Putting it all together the interrupt-map property show:

- #INTA of slot 1 is IRQ9, level low sensitive on the primary interrupt controller
- #INTB of slot 1 is IRQ10, level low sensitive on the primary interrupt controller
- #INTC of slot 1 is IRQ11, level low sensitive on the primary interrupt controller
- #INTD of slot 1 is IRQ12, level low sensitive on the primary interrupt controller

and

- #INTA of slot 2 is IRQ10, level low sensitive on the primary interrupt controller
- #INTB of slot 2 is IRQ11, level low sensitive on the primary interrupt controller
- #INTC of slot 2 is IRQ12, level low sensitive on the primary interrupt controller
- #INTD of slot 2 is IRQ9, level low sensitive on the primary interrupt controller

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, ...).

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.

Notes

1. ↑ Tom Shanley / Don Anderson: PCI System Architecture. Mindshare Inc. (http://www.mindshare.com/)
2. ↑ PCI Bus Bindings to Open Firmware. (http://playground.sun.com/1275/bindings/pci/pci2_1.pdf)
3. ↑ Open Firmware Recommended Practice: Interrupt Mapping (http://playground.sun.com/1275/practice/imap/imap0_9d.pdf)
4. ↑ PCI interrupts are always level low sensitive.

Retrieved from "https://elinux.org/index.php?title=Device_Tree_Usage&oldid=449051"

Category: Device tree