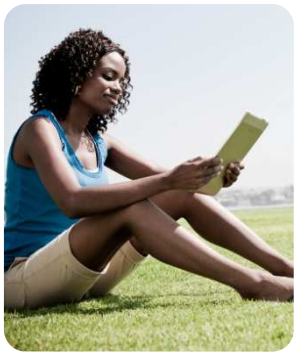
Open Source. Open Possibilities.



Device Tree Experiences

Presented by: Michael Bohan

Presentation Date: 07/03/2012

QuIC
QUALCOMM INNOVATION CENTER, INC.

Agenda

- Overview
- Pros & Cons
- Device Tree Description
- Device Tree Includes
- Device Tree Conversion
- Device Matching
- Booting
- Handling Different H/W Revs
- Device Tree Devices vs. Linux Devices
- Representing Real Devices in Device Tree
- Handling Device Dependencies
- Gerrit Code Review Requirements
- What's our Customer Story?
- References
- Discussion Forums

Overview

- What is Device Tree?
  - Data structure for describing hardware
    - Tree of nodes
    - Nodes can contain properties and other nodes
    - Properties are key/value pairs
  - OS independent
    - ARM Linux is a late adopter
  - Supported in Linux by a framework to easily access node properties
  - Similar to the Windows Registry
- History of Device Tree
  - Device Tree first appeared in Sun's Open Firmware (IEEE1275)
  - PPC Linux had problems supporting both systems with Open Firmware and those without
  - Power.org developed the Embedded Power Architecture Platform Requirements (ePAPR)
    - Specifies the Flattened Device Tree format
  - Circa 2011, Linus Torvalds refuses to accept any more board file changes
    - ARM-Linux moves to Flattened Device Tree in response

Pros & Cons

- Pros
  - Formal and clear hardware description
    - Based on standards
    - Old platform data driven model was inconsistent and lacked versioning of data structures used to pass data
  - Multiplatform kernels now possible
  - Less board-specific code, more efficient device driver binding
  - Reduced effort to bring up new boards
  - Upstream ARM/Linux now requiring Device Tree for all new code submissions

- Cons
  - No complete built-in dependency solution
  - Slower boot time

Device Tree Description

- Device Tree Source (.dts)
  - Used to express device tree in human editable format
  - Organized as a tree structure of nodes and properties
  - Properties are key / value pairs, and node may contain both properties and child nodes

```
/ {
        property1 = "string_value";                 /* define a property containing a 0 terminated
                                                       string */
        property2 = <1234abcd>;                     /* define a property containing a unsigned
                                                       32 bits value (defaults to decimal)*/
        property3 = <12345678 12345678 deadbeef>; /* define a property containing 3 numerical 32 bits
                                                       values (cells) (defaults to decimal) */
        property4 = [0a 0b 0c 0d de ea ad be ef];   /* define a property whose content is an arbitrary
                                                       array of bytes (defaults to hexidecimal)*/
        property5;                                  /* bool property – exists or does not exist */

        childnode@addresss {                        /* define a child node named "childnode" whose
                                                       unit name is "childnode at address" */
            childprop = "hello\n";                  /* define a property "childprop" of childnode
                                                       (in this case, a string) */
        };
    };
```

Device Tree Description

- Reading properties from the Linux framework
  - Strings
    - rc = of_property_count_strings(np, propname);
    - rc = of_property_read_string(node, propname, out_string);
    - rc = of_property_read_string_index(node, propname, idx, out_string);
  - Unsigned integers
    - rc = of_property_read_u32(node, propname, out_values);
    - rc = of_property_read_u32_array(node, propname, out_values, size);
  - Boolean
    - bool = of_property_read_bool(node, propname)
  - Binary data
    - prop = of_get_property(node, name, lenp)
  - Check error codes

Device Tree Description

- Sample Device Tree Source (.dts) for MSM device

```
/dts-v1/;
/include/ "skeleton.dtsi"
/ {
        model = "Qualcomm MSM8660 SURF";
        compatible = "qcom,msm8660-surf", "qcom,msm8660";
        interrupt-parent = <&intc>;
        #address-cells = <1>;
        #size-cells = <1>;

        intc: interrupt-controller@02080000 {
                compatible = "qcom,msm-8660-qgic";
                interrupt-controller;
                #interrupt-cells = <1>;
                reg = < 0x02080000 0x1000 >,
                      < 0x02081000 0x1000 >;
        };

        qcom,fpga@1d000000 {
                compatible = "qcom,turaco3-surf-fpga";
                reg = < 0x1d000000 0x1000 >;
         };
};
```

Device Tree Description

- Sample Device Tree Source (.dts) for MSM device

```
/dts-v1/;
/include/ "skeleton.dtsi"
/ {
        model = "Qualcomm MSM8660 SURF";
        compatible = "qcom,msm8660-surf", "qcom,msm8660";
        interrupt-parent = <&intc>;
        #address-cells = <1>;
        #size-cells = <1>;

        intc: interrupt-controller@02080000 {
                compatible = "qcom,msm-8660-qgic";
                interrupt-controller;
                #interrupt-cells = <1>;
                reg = < 0x02080000 0x1000 >,
                      < 0x02081000 0x1000 >;
        };

        qcom,fpga@1d000000 {
                compatible = "qcom,turaco3-surf-fpga";
                reg = < 0x1d000000 0x1000 >;
        };
};
```

Root node

Root node's properties

Node 1 properties

Node 2 properties

Device Tree Description

- Representing a Device Tree Node in Linux

```
struct device_node {
        const char *name;
        const char *type
        phandle phandle;
        char *full_name;
        struct property *properties;
        ...
};
```

Device Tree Description

- Bindings
  - Description of how a device is described in the device tree
  - Documentation of device tree bindings can be found at kernel/Documentation/devicetree/bindings/…

Device Tree Description

- Node Names
  - Most nodes should have a name in the form <name>[@<unit-address>]
    - Some nodes do not have an address
  - <name> is an ascii string
    - **Node names should be somewhat generic, reflecting the function of the device and not the precise programming model**
  - <unit-address> is included if the node describes a device with an address
    - Address defined within the range of the parent
  - Sibling nodes must be uniquely named, it can have the same generic name as long as the address is different (i.e., serial@19c40000 and serial@19d40000)
  - Vendor specific names should start with a prefix and a comma (all names we create that aren't documented should start with "qcom", or perhaps another vendor prefix if we're using other hardware)
  - **If the device has no address, the name should be as generic as possible, with a suffix that disambiguates the device from other devices at that level**

Device Tree Description

- “compatible” property
  - Used by OS to decide which device driver to bind to a device
  - List of strings starting with most specific, and then a generic name , if that makes sense

```
/dts-v1/;
/include/ "skeleton.dtsi"
/ {
        model = "Qualcomm MSM8660 SURF";
        compatible = "qcom,msm8660-surf", "qcom,msm8660";
        interrupt-parent = <&intc>;
        #address-cells = <1>;
        #size-cells = <1>;

        intc: interrupt-controller@02080000 {
                compatible = "qcom,msm-8660-qgic";
                interrupt-controller;
                #interrupt-cells = <1>;
                reg = < 0x02080000 0x1000 >,
                      < 0x02081000 0x1000 >;
        };
};
```

“compatible” strings should be of the form <manufacturer>,<part-num>

Device Tree Description

- Addressing
  - Specified with the “reg” property, which is a list of address tuples
  - Each tuple consists of base address of region and the region size
  - “#address-cells” and “#size-cells” indicate the number of cells (32-bit fields) required to specify an address (or size) in the child node

```
/dts-v1/;
/include/ "skeleton.dtsi"
/ {
        model = "Qualcomm MSM8660 SURF";
        compatible = "qcom,msm8660-surf", "qcom,msm8660";
        interrupt-parent = <&intc>;
        #address-cells = <1>;
        #size-cells = <1>;

        intc: interrupt-controller@02080000 {
                compatible = "qcom,msm-8660-qgic";
                interrupt-controller;
                #interrupt-cells = <1>;
                reg = < 0x02080000 0x1000 >,
                      < 0x02081000 0x1000 >;
        };
};
```

Number of cells used to specify a base address / region size

Number of cells used by reg must be a multiple of #address-cells plus #size-cells

Device Tree Description

- Addressing (Cont.)
  - “reg” defines address that is local to the parent of the node
  - Mapping addresses up to their parent can be done using “ranges” property
  - Format of “ranges” property is:

```
ranges = <addr1 parent1 size1 [...]>;
```

  - Usage of “ranges” property:

```
…
ranges = <0x0 0x00100000 0x00100000>
…

childnode@address {
        …
        reg = <0x1000 0x1000>
        …
};
…
```

Local address 0x0 is mapped to address 1MB on the parent and size is 1MB for this range

Local address 0x1000 translates to 0x00101000 in the parent address space

Device Tree Description

- Interrupt Controllers
  - *phandle* is a label that allows for a reference from one node to another node
  - “interrupt-parent” gives a *phandle* to a node that describes interrupt controller
  - “interrupts” is a list of interrupt signals that the device can raise
  - “#interrupt-cells” specifies number of cells required to specify an interrupt signal

```
…
compatible = "qcom,msm8660-surf", "qcom,msm8660";
interrupt-parent = <&intc>;

intc: interrupt-controller@02080000 {
        compatible = "qcom,msm-8660-qgic";
        interrupt-controller;
        #interrupt-cells = <3>;
        reg = < 0x02080000 0x1000 >,
              < 0x02081000 0x1000 >;
};
serial@19c40000 {
        compatible = "qcom,msm-hsuart";
        reg = <0x19c40000 0x1000>,
              <0x19c00000 0x1000>;
        interrupts = <0 195 0>;
};
```

“interrupt-parent” defines the link between node and its interrupt parent

Interrupt controller nodes must define an empty property called “interrupt-controller”

“interrupts” property defines the specific interrupt identifier

Device Tree Description

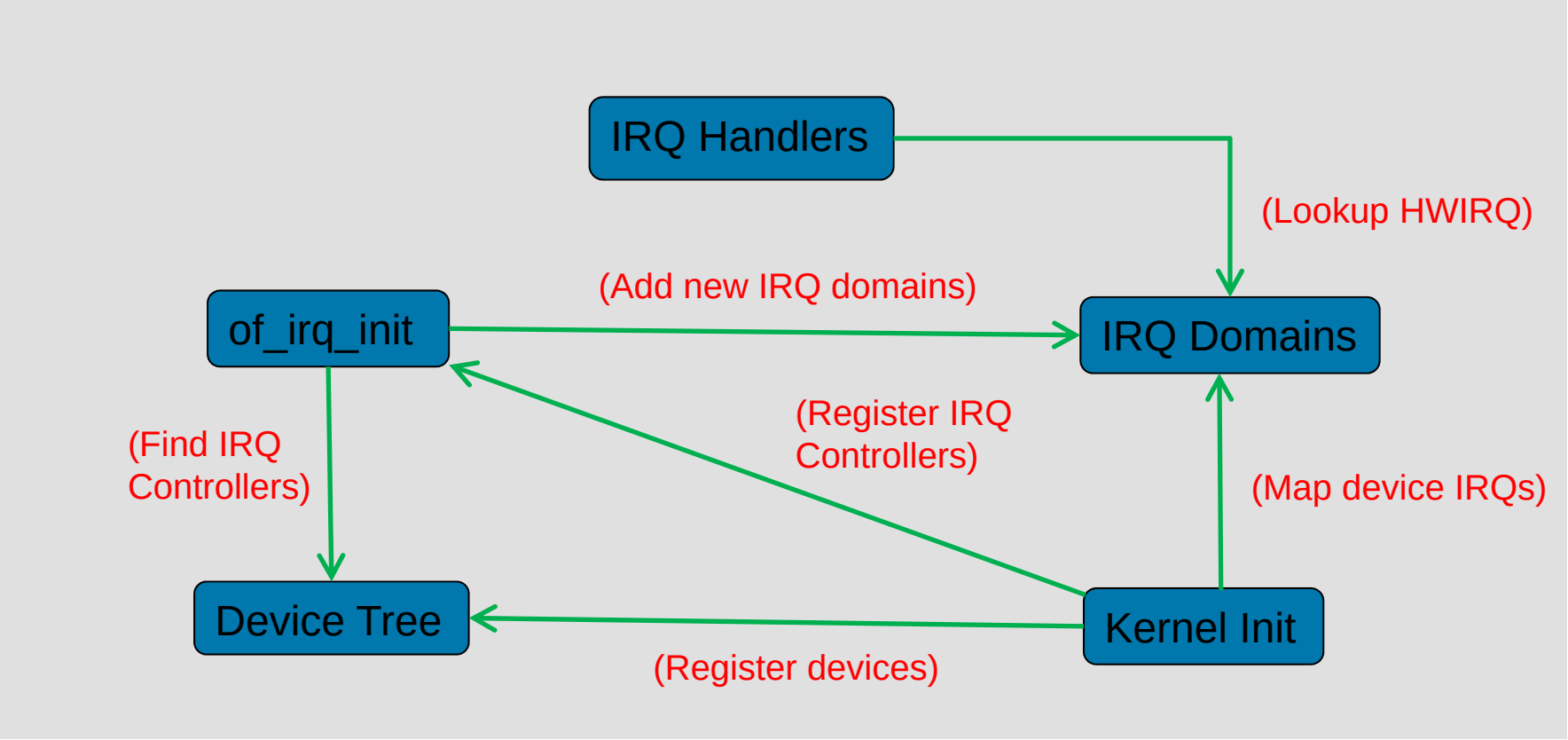
- Interrupt Controllers (Cont.)
  - IRQ numbers are passed in struct resource.
    - They are logical numbers allocated by the kernel
    - Drivers should call platform_get_irq() or platform_get_irq_byname() to get the resources
  - Interrupt parent domains in Device Tree
    - The set of nodes all sharing the same interrupt-parent
      - For a given node, the interrupt parent can be found by traversing upwards to find the lowest node that has the “interrupt-parent” binding.
    - It’s necessary to know your interrupt parent since the format of the ‘interrupts’ binding is IRQ controller specific.
      - Each node can only have one interrupt-parent
  - interrupt-map / interrupt-mask properties
    - We don’t currently use them
    - Useful for PCI type configurations

Device Tree Description

- IRQ Domains
  - Framework to map a interrupt controller hwirq to a Linux virq number
    - Allocates interrupt descriptors when irq is mapped
    - Removes the need for platform code to hard code IRQ numbers.
  - Provides four lookup methods to convert from hwirq to virq
    - Legacy
    - Linear
    - Radix Tree
    - No Map
  - Invoked by Device Tree implementation to map IRQs and pass them as resources to client devices.
  - Invoked by interrupt service routines to lookup a hwirq to pass it to Linux.
  - Used by some drivers that are intrinsically tied to the board design.

Device Tree Description

- Call graph for IRQ Domains



Device Tree Description

- “reg-names” and “interrupt-names” properties solve resource indexing into Device Tree so maintain compatibility for platform_get_resource_byname() and platform_get_irq_byname()

```
…
serial@19c40000 {
                compatible = "qcom,msm-hsuart";
                reg = <0x19c40000 0x1000>,
                      <0x19c00000 0x1000>;
                reg-names = “primary”, “secondary”;
                interrupts = <0 194 0 0 195 0>;
                interrupt-names = “rx_int”, “tx_int”;
};
```

Device Tree Description

- GPIOs

```
…
msmgpio_8974:  gpio@abc123 {
        reg = <0xabc123 0x1000>;
        compatible = “qcom,msm-gpio”;
        gpio-controller;
        #gpio-cells = <2>;
};
qcom,spmi@fc4c0000 {
        compatible = “qcom,spmi-pmic-arb”;
        #address-cells = 1;
        #size-cells = 0;
        pm8941_gpios: gpio@0 {
                reg = <0>;
                gpio-controller;
                #gpio-cells = <1>;
        };
        rtc@1 {
                reg = <1>;
                gpios = <&pm8941_gpios 1 &msmgpio_8974 5 0>
        } ;
};
```

- “gpios” is assigned to a list of tuples containing a gpio-controller phandle and the associated gpio number
- “gpio-controller” must be specified as an empty property for gpio controller nodes
- “#gpio-cells” is the number of 32-bit cells that specify one gpio number on gpio controller nodes
- Client code can get its gpios using the of_gpio_count() and of_get_gpio_flags() APIs.

Device Tree Description

- Regulators

```
…
fb@abc123 {
        reg = <0xabc123 0x1000>;
        compatible = “qcom,msm-fb”;
        mdp-supply = <&pm8941_l1>;
        mddi-supply = <&pm8941_l2>;
};

qcom,spmi@fc4c0000 {
        compatible = “qcom,spmi-pmic-arb”;
        #address-cells = 1;
        #size-cells = 0;
        pm8941_l1: regulator@0 {
                reg = <0>;
         };
};
```

- Consumer regulator supplies define a consumer specific name for the supply, then use the binding with “-supply” tacked on.
- The regulator framework will automatically find such bindings upon regulator_get(), so no driver modifications are necessary.

Device Tree Description

- Device Tree Blob (.dtb), aka Flattened Device Tree (FDT)
  - Binary device tree blob format
  - Device Tree Compiler tool (dtc) translates device trees from both .dts to .dtb and .dtb to .dts
    - Unsigned integers stored in Big Endian format
  - Documentation of device tree blob format can be found at: kernel/Documentation/devicetree/booting-without-of.txt
  - ```
    dtc -p 1024 -O dtb -o msm.dtb msm.dts
    ```
    ```

Device Tree Includes

- Files with the .dtsi extension are Device Tree Includes
  - Useful to factor our details that don't change between boards or hardware revisions
- Bindings belonging to the device node at the same path are combined to form the superset if they are unique
- The last binding belonging to the device node at the same path is selected if there is a namespace collision

Device Tree Includes

- File board.dts

```
/dts-v1/;
/include “skeleton.dtsi”
/{
        …
        spmi@abc1234 {
                reg = <0xabc1234 0x1000>;
                pm8841: qcom,pm8841@0 {
                };
                …
        };
        …
};
/include/ “pm8841.dtsi”
/include/ “board-gpios.dtsi”
```

Device Tree Includes

- File pm8841.dtsi
  - The board agnostic PMIC dtsi knows nothing about specific board addresses

```
&pm8841 {
        pm8841_gpios: qcom,pm8841_gpios {
                compatible = “qpnp-gpio”;
                #address-cells = <1>;
                #size-cells = <1>;
                gpio@c000 {
                        reg = <0xc000 0x1000>;
                        status = “disabled”;
                };
                …
        };
        …
};
```

Device Tree Includes

- File board-gpios.dtsi
  - Allows for board specific information to be overwritten without having to modify the board agnostic PMIC dtsi

```
&pm8841_gpios {
        gpio@c000 {
                status = “ok”;
        };
};
```

Device Tree Includes

- Output from ‘dtc board.dts’

```
/{
        model = “Test board”;
        compatible = “qcom,testboard”;
        chosen = {};
        memory = {};

        spmi@abc1234 {
                pm8841_gpios: qcom,pm8841_gpios {
                        compatible = “qpnp-gpio”;
                        #address-cells = <1>;
                        #size-cells = <1>;
                        gpio@c000 {
                                reg = <0xc000 0x1000>;
                                status = “ok”;
                        };
                        …
                };
                …
        };
};
```

Device Tree Conversion

Steps taken to represent MSM serial device in device tree format

1. Provide bindings documentation in Documentation/devicetree/bindings/…

```
* Qualcomm MSM UART

<Description>

Required properties:
- compatible :
        - "qcom,msm-uart"
- reg : offset and length of the register set for the device for the hsuart operating in compatible mode,
        there should be a second pair describing the gsbi registers.
- interrupts : should contain the uart interrupt.

Example:
                serial@19c40000 {
                        compatible = "qcom,msm-hsuart", "qcom,msm-uart";
                        reg = <0x19c40000 0x1000">,
                              <0x19c00000 0x1000">;
                        interrupts = <0 195 0>;
                };
```

Device Tree Conversion

2. Add device tree data for MSM serial device to the device tree source file

```
/dts-v1/;
/include/ "skeleton.dtsi"
/ {
        model = "Qualcomm MSM8660 SURF";
        compatible = "qcom,msm8660-surf", "qcom,msm8660";
        interrupt-parent = <&intc>;
        #address-cells = <1>;
        #size-cells = <1>;

        intc: interrupt-controller@02080000 {
                compatible = "qcom,msm-8660-qgic";
                interrupt-controller;
                #interrupt-cells = <1>;
                reg = < 0x02080000 0x1000 >,
                      < 0x02081000 0x1000 >;
        };

        serial@19c40000 {
                compatible = "qcom,msm-hsuart", "qcom,msm-uart";
                reg = <0x19c40000 0x1000">,
                      <0x19c00000 0x1000">;
                interrupts = <0 195 0>;
        };
};
```

Device Tree Conversion

3. Add match table to platform driver with the appropriate “compatible” string that the device can be matched to

```
+ static struct of_device_id msm_match_table[] = {
+                 {    .compatible = "qcom,msm-hsuart" },
+                 {}
+ };

  static struct platform_driver msm_platform_driver = {
                  .remove = msm_serial_remove,
                  .driver = {
                                  .name = "msm_serial",
                                  .owner = THIS_MODULE,
+                                 .of_match_table = msm_match_table,
                  },
  };
```

Match table can be used to pass data to a particular device version as well. This shouldn’t be abused. Must be NULL terminated.

Device Tree Conversion

4. Clocks are still queried by direct name from the driver until device tree clock support is implemented. Since specific names are not encoded into the device tree, a lookup table can be used to match the device names on a temporary basis until this functionality can be converted over to device tree. The lookup table is specified in the board file, below is one such example:

```
/**
  * struct of_dev_auxdata - lookup table entry for device names & platform_data
  * @compatible: compatible value of node to match against node
  * @phys_addr: Start address of registers to match against node
  * @name: Name to assign for matching nodes
  * @platform_data: platform_data to assign for matching nodes
  */
static struct of_dev_auxdata msm_auxdata_lookup[] __initdata = {
                OF_DEV_AUXDATA("qcom,msm-hsuart", 0x19c40000, "msm_serial.0", NULL),
                {}
 };
```

Device Matching

- Primary difference is what method is used to match devices to drivers

```
static struct of_device_id msm_match_table[] = {
        {   .compatible = "qcom,msm-hsuart" },
        {}
};

static struct platform_driver msm_platform_driver = {
        .remove = msm_serial_remove,
        .driver = {
                .name = "msm_serial",
                .owner = THIS_MODULE,
                .of_match_table = msm_match_table,
        },
};
```

```
serial@19c40000 {
        compatible = "qcom,msm-hsuart", "qcom,msm-uart";
        reg = <0x19c40000 0x1000">,
              <0x19c00000 0x1000">;
        interrupts = <195>;
};
```

Booting

- When booting with Device Tree
  - r1 is the special Machine Type number (0xffffffff) for Device Tree
  - r2 is a pointer to the dtb image
- Bootloader reads the SoC ID / revision and loads the appropriate dtb image.
- Linux reads the top level “compatibility” property and matches that string against all MACHINE_DESC entries populated
- of_platform_populate()
  - Called from the board file at arch initcall
  - Traverses the child nodes from the root node, matching against a passed in match table.
    - Adds all nodes that are children of the passed in (root) node
    - By default, matches against the compatible field “simple-bus”
    - Allows for easy determination of which nodes should be added

Handling different hardware revisions

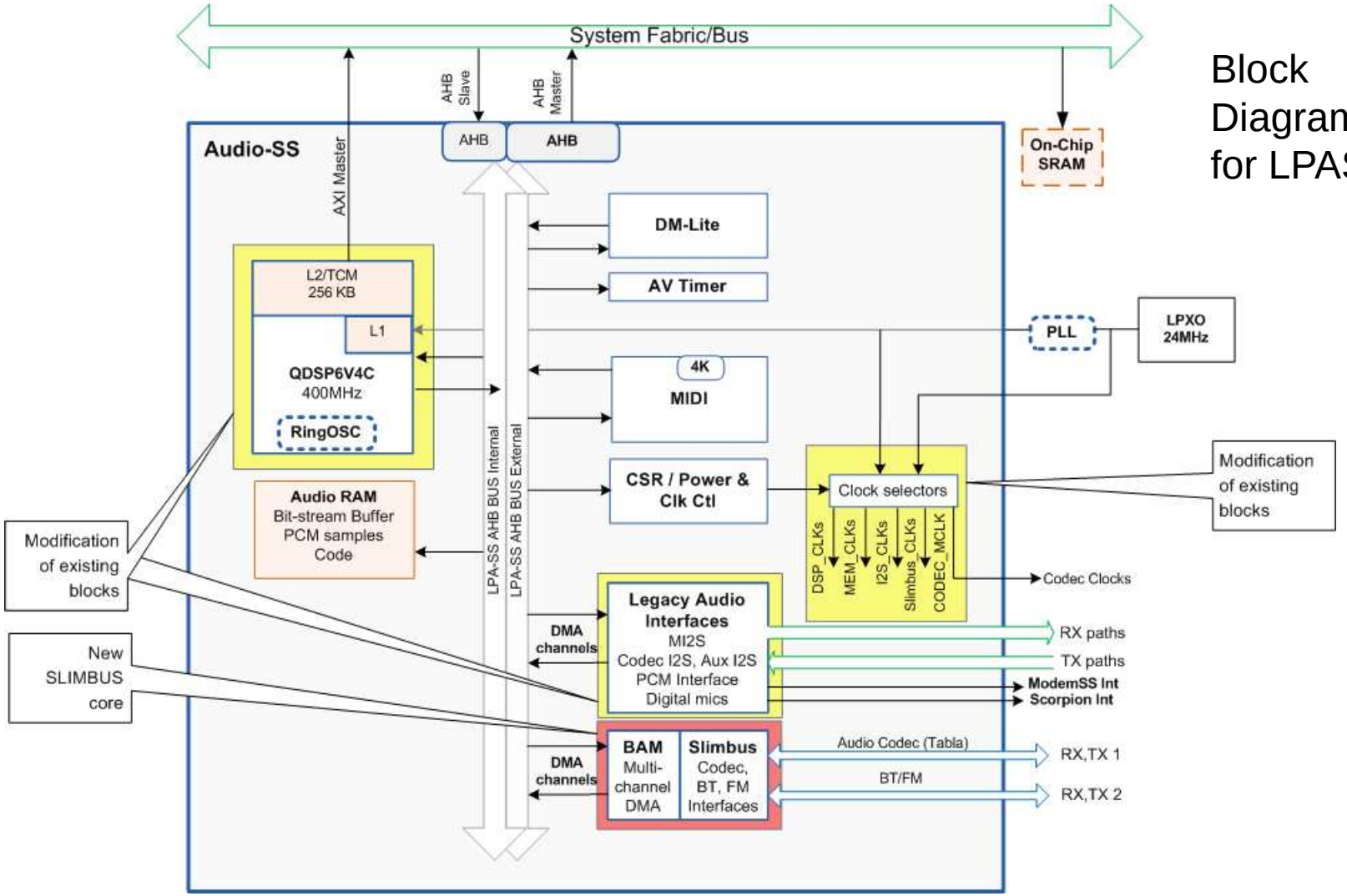
- No macro such as machine_is_msm8974()
- Bootloader will detect different revs of the hardware and load the correct dtb from the boot partition
- Drivers can ‘match’ particular hardware variants

```
…
match = of_match_device(of_match_ptr(msm_match_table), &pdev->dev);
…
 static struct of_device_id msm_match_table[] = {
   {    .compatible = "qcom,msm-hsuart“
        .data = &msm-hsuart
   },
   {    .compatible = "qcom,qrd-hsuart“
        .data = &qrd-hsuart
   },
    {}
  };
```

Device Tree Devices are Not Always Linux Devices

- Linux Device Model exists with or without loading a Flattened Device Tree
  - Linux Device creation can be filtered with the of_platform_populate match table and controlled probes of bus drivers
- Sometimes it doesn't make sense to create a Linux Device per each Device Tree node (sigh)
  - Example: gpios on the 8x41 PMIC
  - Example: Device Nodes sometimes pass configuration in subnodes

Representing Real Devices in Device Tree



Block Diagram for LPASS

QUALCOMM INNOVATION CENTER, INC.

Representing Real Devices in Device Tree

- Representing LPASS in Device Tree (Approach 1)

```
  …
qcom,msm_q6 {
        compatible = “simple-bus”;

        qcom,msm-auxpcm {
                compatible = "qcom,msm-auxpcm-resource";
                qcom,msm-cpudai-auxpcm-clk = "pcm_clk";
                qcom,msm-cpudai-auxpcm-pcm-clk-rate = <2048000>;

                qcom,msm-auxpcm-rx {
                        compatible = "qcom,msm-auxpcm-dev";
                        qcom,msm-auxpcm-dev-id = <4106>;
                 };

                 qcom,msm-auxpcm-tx {
                        compatible = "qcom,msm-auxpcm-dev";
                         qcom,msm-auxpcm-dev-id = <4107>;

                 };
        };
};
```

Representing Real Devices in Device Tree

- Approach 1: Treat each auxpcm subnode as a real Linux device that's probed
  - Consists of two drivers to manage the PCM subtree
    - The 'resource driver' harvests the common bindings that all underlying codec paths (eg. rx / tx) require
    - The 'auxpcm-dev driver' manages each codec path; It uses the common resources passed down through the parent device's (eg. resource) drv_data
  - The subtree for qcom,msm-auxpcm is not scanned automatically by the arch-initcall of_platform_populate() call, since it doesn't match "simple-bus"
  - When the 'resource driver' probes it calls of_platform_populate() after demarshalling all its binding data
  - Forces the children devices to be traversed and added, causing their probe routines to be invoked
  - Handles the dependency of the parent automatically
  - Useful if you want to treat the child devices as *real devices* instead of just abstracting additional hardware details

Representing Real Devices in Device Tree

- Representing LPASS in Device Tree (Approach 2)

```
  …
qcom,msm_q6 {
        compatible = “simple-bus”;

        qcom,msm-auxpcm {
              compatible = "qcom,msm-auxpcm";
              qcom,msm-cpudai-auxpcm-clk = "pcm_clk";
              qcom,msm-cpudai-auxpcm-pcm-clk-rate = <2048000>;

              qcom,msm-auxpcm-rx {
                    /* compatible field N/A */
                    qcom,msm-auxpcm-dev-id = <4106>;
               };

               qcom,msm-auxpcm-tx {
                    /* compatible field N/A */
                    qcom,msm-auxpcm-dev-id = <4107>;
                };
        };
};
```

Representing Real Devices in Device Tree

- Approach 2: Don't instantiate auxpcm nodes as probed Linux devices
  - Child Device Tree devices can be scanned with the for_each_child_of_node() API.
  - The parent node "qcom,msm-auxpcm" is the only real Linux device.
  - Handlings child / parent dependency automatically
  - Useful for modeling real devices that don't deserve their own 'struct device' in due to the nature of the design

```
static int __devinit test_probe(struct platform_device pdev)  {
                for_each_child_of_node(pdev->dev.of_node, child)
                        collect_of_data(child);
…
```

Device Tree and Dependencies

- Device Tree doesn't claim to handle dependencies
  - Doesn't guarantee the ordering of nodes at the same tree level
  - It's an established specification which is not trivial to change
  - Regression from the device tables used in the Linux platform board files, where device registration can be done in an explicit order
- Most dependency information *is* actually embedded in the tree
  - parent / child relationship on nodes
  - phandles

Dependency Solution From the Community

- Keep dependency responsibility in the driver
- Deferred Probe
  - Maintain two lists, pending and active
    - When a device probe fails with –EPROBE_DEFER, add the device to the pending list
    - When a device probe succeeds, move all devices from the pending list to the active list
  - Trigger the probe retry at late_initcall
    - Done when the active list is empty
  - Independent of Device Tree

Implications of Deferred Probe Solution

- Linux developers need to write correct drivers
  - Probe routines will be invoked a multiple number of times
  - Unwind resources properly on probe error
  - Any error code from a framework must be returned from the probe routine
  - All resources the driver requires must be requested in the probe routine
    - Library calls which don't have a corresponding 'get handle' routine and can possibly fail due to missing dependencies need to be modified to support this requirement
    - All frameworks need to return –EPROBE_DEFER

Ensuring Devices Probe in a Timely Manner

- Use case: ensure that a particular device probes in N number of seconds
  - Deferred Probe does not address this
- Possible solutions
  - Register a certain subset of drivers early in a specific order
    - Not ideal since it requires a hard coded list of drivers in the board file that can be abused
    - Each driver needs to export a global symbol so it can be referred to
    - Solution currently used for msm8974
  - Move a subset of drivers to a early initcall level and force a ‘deferred probe trigger’ after each registers its driver
    - Doesn’t scale well due to the rough granularity of a initcall level

Gerrit Code Approval Requirements for Device Tree changes

- A reviewed Device Tree abstraction that portrays the hardware
- Any code that makes use of new bindings should include documentation in the same change
- If you're 'adding Device-Tree support' to an existing driver, that should be done in a separate change than switching boards over to use the new Device Tree support

What's Our Story to Customers?

- For 8974, we are only supporting Device Tree in our drivers
  - Compliance with upstream Linux submissions
- Customers are welcome to port them, but we suspect the added work will give them an incentive to use Device Tree
- **Documentation for device bindings is already in our kernel**
- Customer Engineers need to be familiar with the concepts so they field any customer questions

References

- http://devicetree.org/Main_Page
- http://ozlabs.org/~dgibson/papers/dtc-paper.pdf
- http://elinux.org/Device_Trees
- http://www.linuxsymposium.org/archives/OLS/Reprints-2008/likely2-reprint.pdf
- http://qwiki.qualcomm.com/quic/Kernel/DeviceTree

Discussion Forums

- Internal Mailing List
  - linux.devicetree
- Public devicetree-discuss mailing list
  - devicetree-discuss@lists.ozlabs.org
  - https://lists.ozlabs.org/listinfo/devicetree-discuss