Making Everything Easier!™

Novelty Edition

Device Tree

FOR DUMMIES®

Learn to:

- Boot your system with a Device Tree
- Understand the basic syntax of the Device Tree
- Learn about Device Tree bindings and their rules

Thomas Petazzoni

Thomas Petazzoni

- CTO and Embedded Linux engineer at Free Electrons
  - Embedded Linux **development**: kernel and driver development, system integration, boot time and power consumption optimization, consulting, etc.
  - Embedded Linux **training**, Linux driver development training and Android system development training, with materials freely available under a Creative Commons license.
  - `http://free-electrons.com`
- Contributions
  - **Kernel support for the new Armada 370 and Armada XP** ARM SoCs from Marvell
  - Major contributor to **Buildroot**, an open-source, simple and fast embedded Linux build system
- Living in **Toulouse**, south west of France

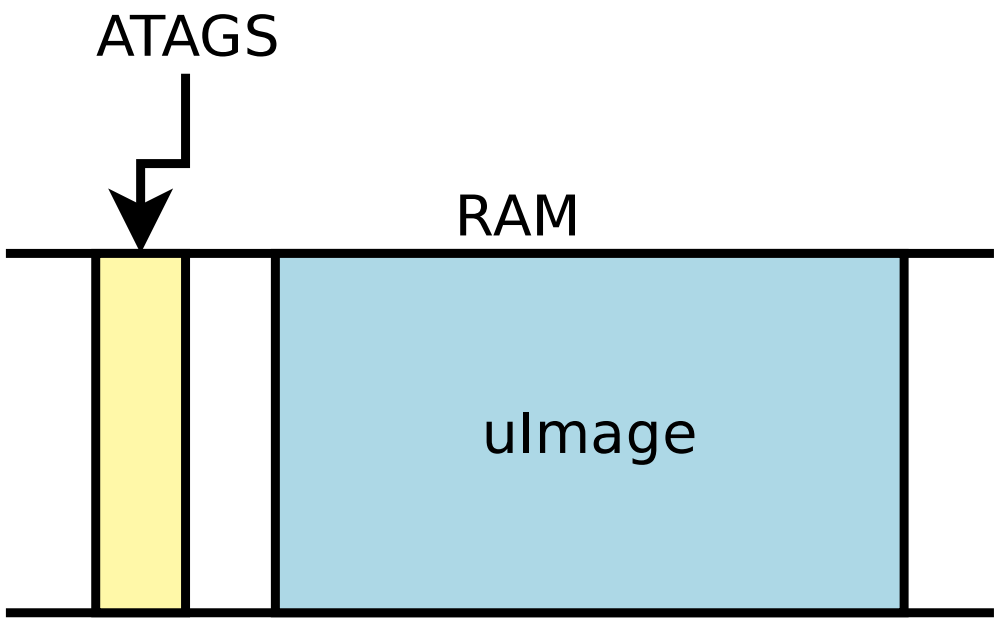
Agenda

- User perspective: booting with the Device Tree
- Basic Device Tree syntax and compilation
- Simple example of Device Tree fragment
- Overall organization of a Device Tree
- Examples of Device Tree usage
- General considerations about the Device Tree in Linux

User perspective: before the Device Tree

- The kernel contains the entire description of the hardware.
- The bootloader loads a single binary, the kernel image, and executes it.
  - `uImage` or `zImage`
- The bootloader prepares some additional information, called `ATAGS`, which address is passed to the kernel through register `r2`
  - Contains information such as memory size and location, kernel command line, etc.
- The bootloader tells the kernel on which board it is being booted through a *machine type* integer, passed in register `r1`.
- U-Boot command: `bootm <kernel img addr>`
- Barebox variable: `bootm.image`

User perspective: before the Device Tree




```
r1 = <machine type>
r2 = <pointer to ATAGS>
```

User perspective: booting with a Device Tree

- The kernel no longer contains the description of the hardware, it is located in a separate binary: the *device tree blob*
- The bootloader loads two binaries: the kernel image and the *DTB*
  - Kernel image remains `uImage` or `zImage`
  - DTB located in `arch/arm/boot/dts`, one per board
- The bootloader passes the DTB address through `r2`. It is supposed to adjust the DTB with memory information, kernel command line, and potentially other info.
- No more *machine type*.
- U-Boot command:

  ```
  bootm <kernel img addr> - <dtb addr>
  ```

- Barebox variables: `bootm.image`, `bootm.oftree`

User perspective: booting with a Device Tree

User perspective: compatibility mode for DT booting

- Some bootloaders have no specific support for the Device Tree, or the version used on a particular device is too old to have this support.
- To ease the transition, a *compatibility* mechanism was added: `CONFIG_ARM_APPENDED_DTB`.
  - It tells the kernel to look for a DTB right *after* the kernel image.
  - There is no built-in Makefile rule to produce such kernel, so one must manually do:

```
cat arch/arm/boot/zImage arch/arm/boot/dts/myboard.dtb > my-zImage
mkimage ... -d my-zImage my-uImage
```

- In addition, the additional option `CONFIG_ARM_ATAG_DTB_COMPAT` tells the kernel to read the *ATAGS* information from the bootloader, and update the DT using them.

What is the Device Tree ?

- Quoted from the *Power.org Standard for Embedded Power Architecture Platform Requirements (ePAPR)*
  - The ePAPR specifies a concept called a device tree to describe system hardware. A boot program loads a device tree into a client program's memory and passes a pointer to the device tree to the client.
  - A device tree is a tree data structure with nodes that describe the physical devices in a system.
  - An ePAPR-compliant device tree describes device information in a system that cannot be dynamically detected by a client program.

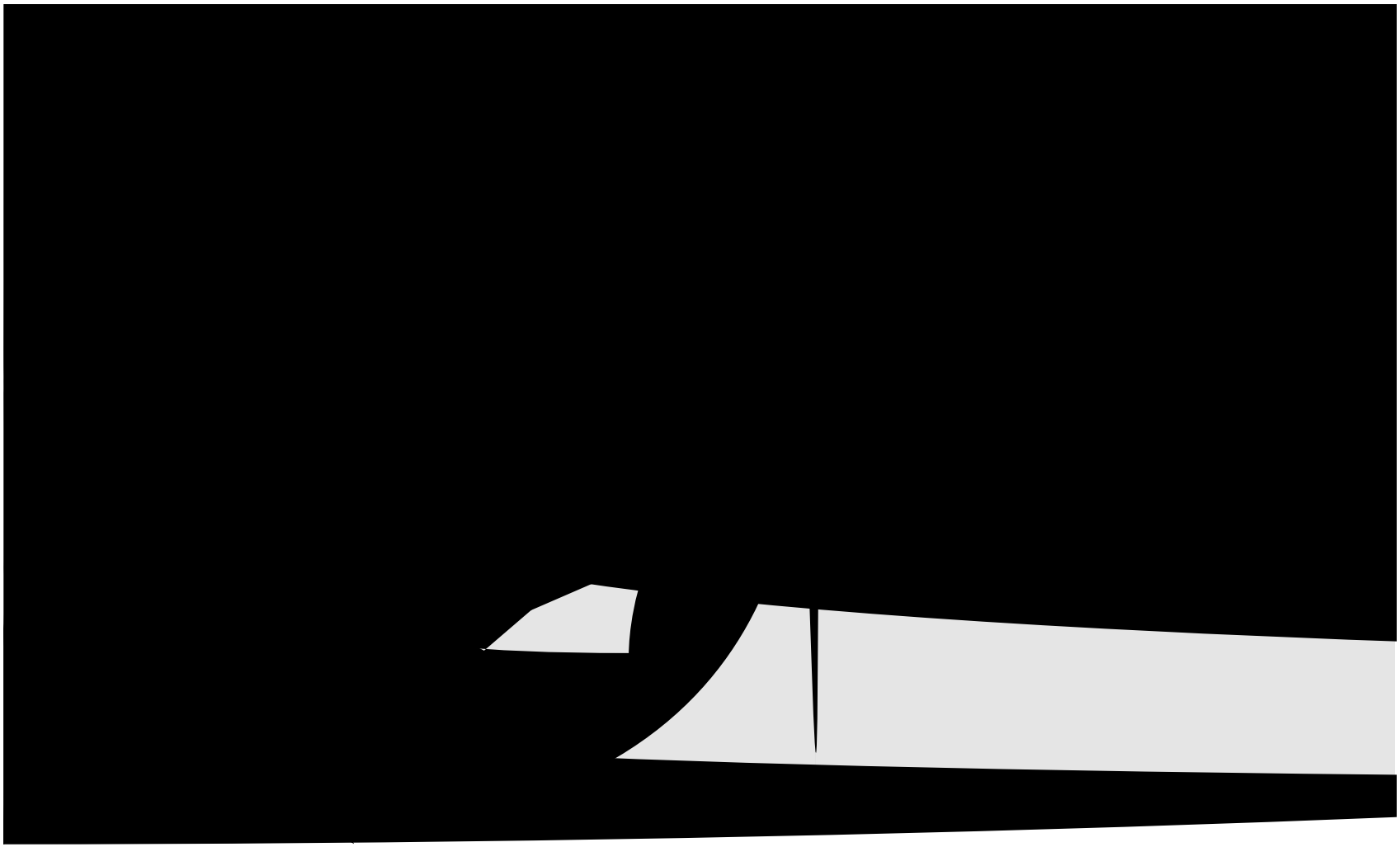
Basic Device Tree syntax

From source to binary

- On ARM, all **Device Tree Source** files (DTS) are for now located in `arch/arm/boot/dts`
  - `.dts` files for board-level definitions
  - `.dtsi` files for included files, generally containing SoC-level definitions
- A tool, the **Device Tree Compiler** compiles the source into a binary form.
  - Source code located in `scripts/dtc`
- The **Device Tree Blob** is produced by the compiler, and is the binary that gets loaded by the bootloader and parsed by the kernel at boot time.
- `arch/arm/boot/dts/Makefile` lists which DTBs should be generated at build time.

```
dtb-$(CONFIG_ARCH_MVEBU) += armada-370-db.dtb \
        armada-370-mirabox.dtb \
...
```

A simple example, DT side



A simple example, driver side (1)

The compatible string used to bind a device with the driver

```
static struct of_device_id mxs_auart_dt_ids[] = {
        {
                .compatible = "fsl,imx28-auart",
                .data = &mxs_auart_devtype[IMX28_AUART]
        }, {
                .compatible = "fsl,imx23-auart",
                .data = &mxs_auart_devtype[IMX23_AUART]
        }, { /* sentinel */ }
};
MODULE_DEVICE_TABLE(of, mxs_auart_dt_ids);
[...]
static struct platform_driver mxs_auart_driver = {
        .probe = mxs_auart_probe,
        .remove = mxs_auart_remove,
        .driver = {
                .name = "mxs-auart",
                .of_match_table = mxs_auart_dt_ids,
        },
};
```

Code from `drivers/tty/serial/mxs-auart.c`

A simple example, driver side (2)

- `of_match_device` allows to get the matching entry in the `mxs_auart_dt_ids` table.
- Useful to get the driver-specific `data` field, typically used to alter the behavior of the driver depending on the variant of the detected device.

```
static int mxs_auart_probe(struct platform_device *pdev)
{
        const struct of_device_id *of_id =
                        of_match_device(mxs_auart_dt_ids, &pdev->dev);

        if (of_id) {
                /* Use of_id->data here */
                [...]
        }
        [...]
}
```

A simple example, driver side (3)

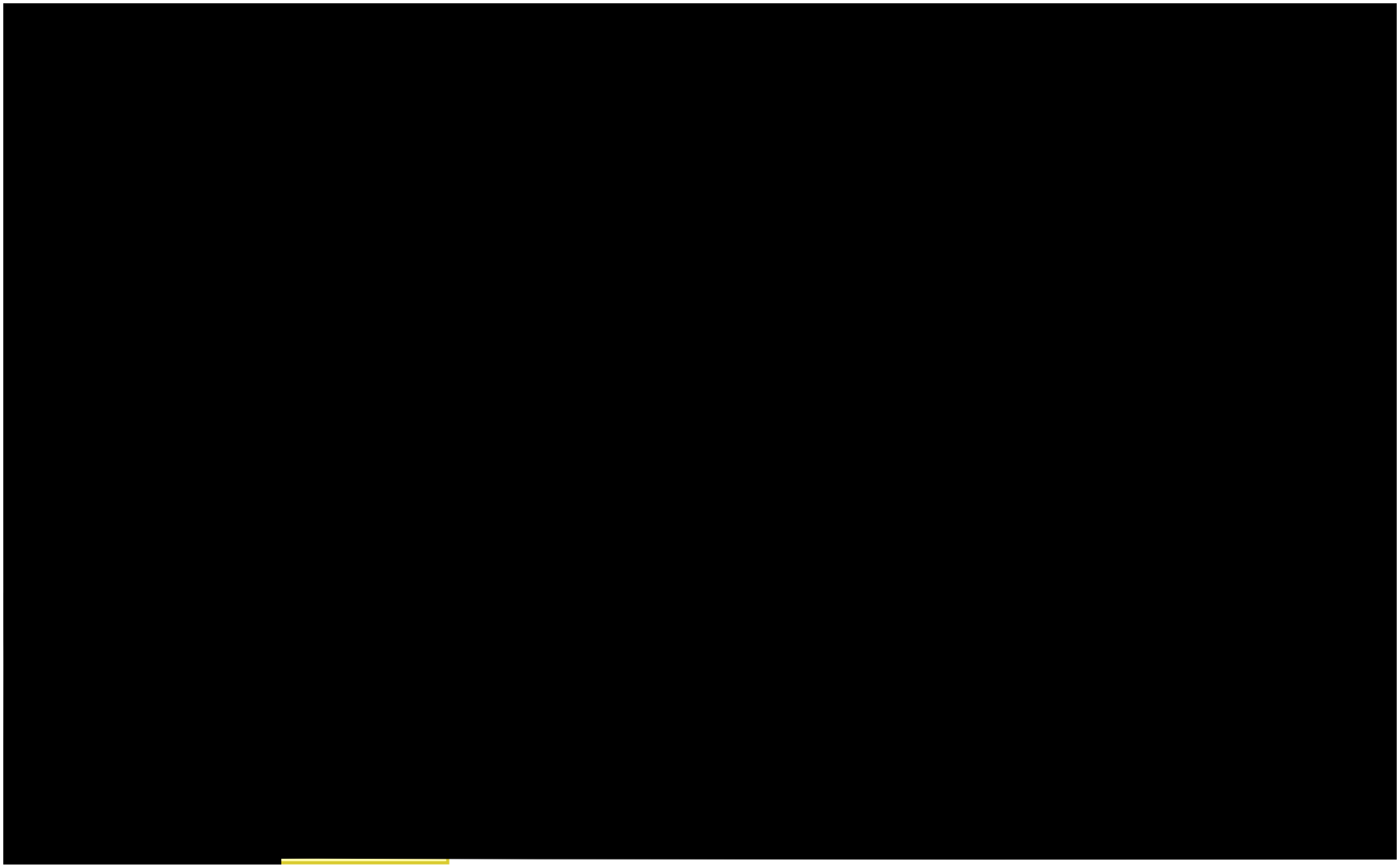
- Getting a reference to the clock
  - described by the `clocks` property
  - `s->clk = clk_get(&pdev->dev, NULL);`
- Getting the I/O registers *resource*
  - described by the `reg` property
  - `r = platform_get_resource(pdev, IORESOURCE_MEM, 0);`
- Getting the interrupt
  - described by the `interrupts` property
  - `s->irq = platform_get_irq(pdev, 0);`
- Get a DMA channel
  - described by the `dmas` property
  - `s->rx_dma_chan = dma_request_slave_channel(s->dev, "rx");`
  - `s->tx_dma_chan = dma_request_slave_channel(s->dev, "tx");`
- Check some custom property
  - `struct device_node *np = pdev->dev.of_node;`
  - `if (of_get_property(np, "fsl,uart-has-rtscts", NULL))`

Device Tree inclusion

- Device Tree files are not monolithic, they can be split in several files, including each other.
- `.dtsi` files are included files, while `.dts` files are *final* Device Trees
- Typically, `.dtsi` will contain definition of SoC-level information (or sometimes definitions common to several almost identical boards).
- The `.dts` file contains the board-level information.
- The inclusion works by **overlaying** the tree of the including file over the tree of the included file.
- Inclusion using the DT operator `/include/`, or since a few kernel releases, the DTS go through the C preprocessor, so `#include` is recommended.

Device Tree inclusion example

Device Tree inclusion example (2)



Concept of Device Tree binding

- Quoting the *ePAPR*:
  - This chapter contains requirements, known as **bindings, for how specific types and classes of devices are represented in the device tree**.
  - The `compatible` property of a device node describes the specific binding (or bindings) to which the node complies.
  - When creating a new device tree representation for a device, a **binding should be created that fully describes the required properties and value of the device**. This set of properties shall be sufficiently descriptive to provide device drivers with needed attributes of the device.

Documentation of Device Tree bindings

- All Device Tree bindings recognized by the kernel are documented in `Documentation/devicetree/bindings`.
- Each binding documentation described which properties are accepted, with which values, which properties are mandatory vs. optional, etc.
- All new Device Tree bindings must be reviewed by the *Device Tree maintainers*, by being posted to `devicetree@vger.kernel.org`. This ensures correctness and consistency across bindings.

```
OPEN FIRMWARE AND FLATTENED DEVICE TREE BINDINGS
M:      Rob Herring <rob.herring@calxeda.com>
M:      Pawel Moll <pawel.moll@arm.com>
M:      Mark Rutland <mark.rutland@arm.com>
M:      Stephen Warren <swarren@wwwdotorg.org>
M:      Ian Campbell <ijc+devicetree@hellion.org.uk>
L:      devicetree@vger.kernel.org
```

Device Tree binding documentation example

```
* Freescale MXS Application UART (AUART)

Required properties:
- compatible : Should be "fsl,<soc>-auart". The supported SoCs include
  imx23 and imx28.
- reg : Address and length of the register set for the device
- interrupts : Should contain the auart interrupt numbers
- dmas: DMA specifier, consisting of a phandle to DMA controller node
  and AUART DMA channel ID.
  Refer to dma.txt and fsl-mxs-dma.txt for details.
- dma-names: "rx" for RX channel, "tx" for TX channel.

Example:
auart0: serial@8006a000 {
        compatible = "fsl,imx28-auart", "fsl,imx23-auart";
        reg = <0x8006a000 0x2000>;
        interrupts = <112>;
        dmas = <&dma_apbx 8>, <&dma_apbx 9>;
        dma-names = "rx", "tx";
};

Note: Each auart port should have an alias correctly numbered in "aliases"
node.

Example:
[...]
```

Documentation/devicetree/bindings/tty/serial/fsl-mxs-auart.txt

Device Tree organization: top-level nodes

Under the root of the Device Tree, one typically finds the following top-level nodes:

- A `cpus` node, which sub-nodes describing each CPU in the system.
- A `memory` node, which defines the location and size of the RAM.
- A `chosen` node, which defines *parameters chosen or defined by the system firmware at boot time*. In practice, one of its usage is to pass the kernel command line.
- A `aliases` node, to define shortcuts to certain nodes.
- One or more nodes defining the *buses* in the SoC.
- One or mode nodes defining on-board devices.

Device Tree organization: imx28.dtsi

arch/arm/boot/dts/imx28.dtsi

```
/ {
        aliases { ... };
        cpus { ... };

        apb@80000000 {
                apbh@80000000 {
                        /* Some devices */
                };

                apbx@80040000 {
                        /* Some devices */
                };
        };

        ahb@80080000 {
                /* Some devices */
        };
};
```

i.MX28 buses organization

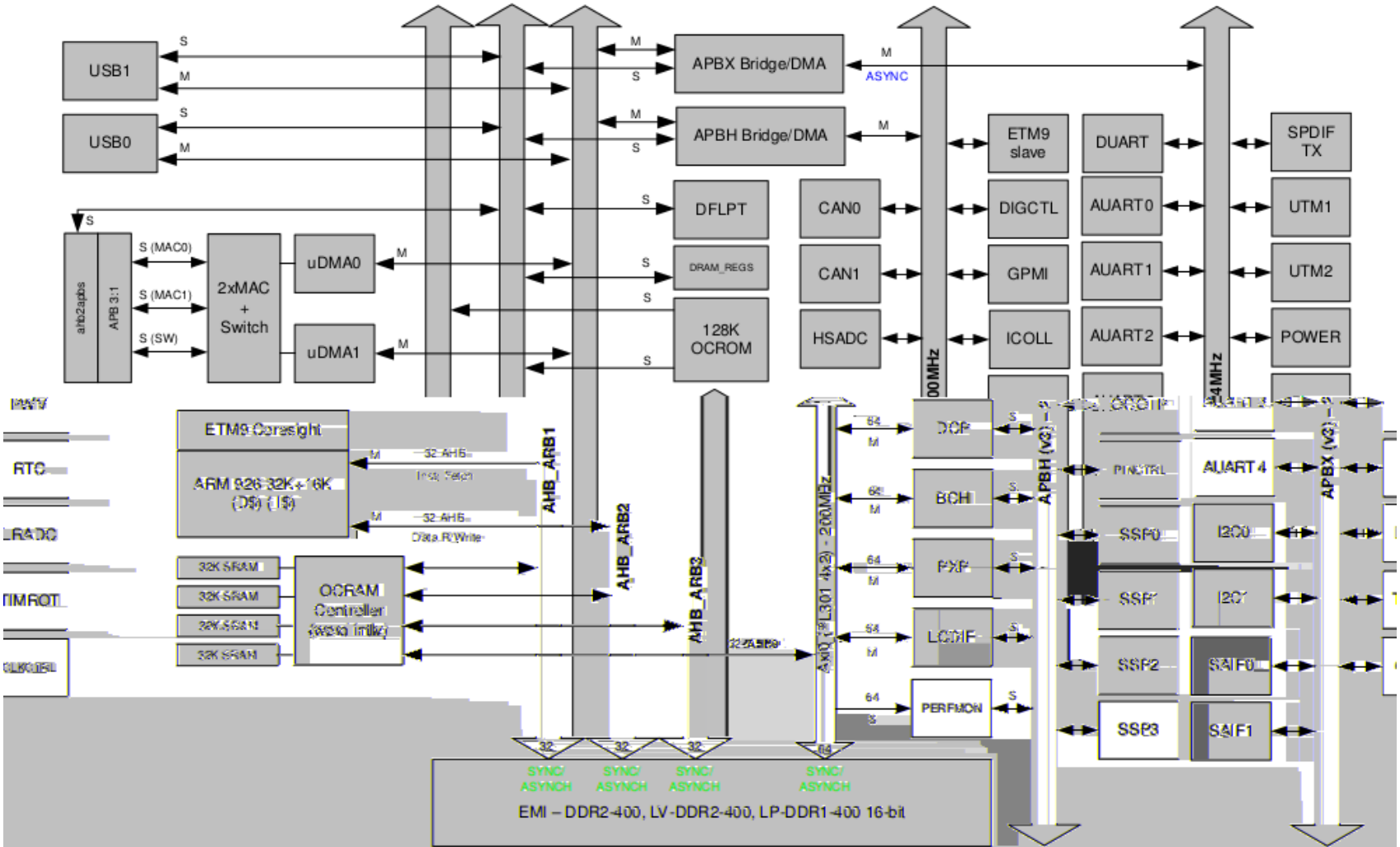


Figure 1-2. i.MX28 SOC System Buses

Device Tree organization: imx28-evk.dts

arch/arm/boot/dts/imx28-evk.dts

```
/ {
        model = "Freescale i.MX28 Evaluation Kit";
        compatible = "fsl,imx28-evk", "fsl,imx28";

        memory {
                reg = <0x40000000 0x08000000>;
        };


        apb@80000000 {
                apbh@80000000 { ... };
                apbx@80040000 { ... };
        };

        ahb@80080000 { ... };

        sound { ... };
        leds { ... };
        backlight { ... };
};
```

Top-level compatible property

- The top-level `compatible` property typically defines a compatible string for the board, and then for the SoC.
- Values always given with the most-specific first, to least-specific last.
- Used to match with the `dt_compat` field of the `DT_MACHINE` structure:

```
static const char *mxs_dt_compat[] __initdata = {
        "fsl,imx28",
        "fsl,imx23",
        NULL,
};

DT_MACHINE_START(MXS, "Freescale MXS (Device Tree)")
        .dt_compat      = mxs_dt_compat,
        [...]
MACHINE_END
```

- Can also be used within code to test the machine:

```
if (of_machine_is_compatible("fsl,imx28-evk"))
        imx28_evk_init();
```

Bus, address cells and size cells

Inside a bus, one typically needs to define the following properties:

- A `compatible` property, which identifies the bus controller (in case of I2C, SPI, PCI, etc.). A special value `compatible = "simple-bus"` means a simple memory-mapped bus with no specific handling or driver. Child nodes will be registered as *platform devices*.
- The `#address-cells` property indicate how many cells (i.e 32 bits values) are needed to form the base address part in the `reg` property.
- The `#size-cells` is the same, for the size part of the `reg` property.
- The `ranges` property can describe an *address translation* between the child bus and the parent bus. When simply defined as `ranges;`, it means that the translation is an identity translation.

simple-bus, address cells and size cells

```
apbh@80000000 {
        compatible = "simple-bus";
        #address-cells = <1>;
        #size-cells = <1>;
        reg = <0x80000000 0x3c900>;
        ranges;

        [...]

        hsadc: hsadc@80002000 {
                reg = <0x80002000 0x2000>;
                interrupts = <13>;
                dmas = <&dma_apbh 12>;
                dma-names = "rx";
                status = "disabled";
        };

        [...]
};
```

I2C bus, address cells and size cells

```
i2c0: i2c@80058000 {
        #address-cells = <1>;
        #size-cells = <0>;
        compatible = "fsl,imx28-i2c";
        reg = <0x80058000 0x2000>;
        interrupts = <111>;
        [...]

        sgtl5000: codec@0a {
                compatible = "fsl,sgtl5000";
                reg = <0x0a>;
                VDDA-supply = <&reg_3p3v>;
                VDDIO-supply = <&reg_3p3v>;
                clocks = <&saif0>;
        };

        at24@51 {
                compatible = "at24,24c32";
                pagesize = <32>;
                reg = <0x51>;
        };
};
```

Interrupt handling

- `interrupt-controller;` is a boolean property that indicates that the current node is an interrupt controller.
- `#interrupt-cells` indicates the number of cells in the `interrupts` property for the interrupts managed by the selected interrupt controller.
- `interrupt-parent` is a *phandle* that points to the interrupt controller for the current node. There is generally a top-level `interrupt-parent` definition for the main interrupt controller.

Interrupt example: imx28.dtsi

```
/ {
    interrupt-parent = <&icoll>;
    apb@80000000 {
        apbh@80000000 {
            icoll: interrupt-controller@80000000 {
                compatible = "fsl,imx28-icoll", "fsl,icoll";
                interrupt-controller;
                #interrupt-cells = <1>;
                reg = <0x80000000 0x2000>;
            };

            ssp0: ssp@80010000 {
                [...]
                interrupts = <96>;
            };
        };
    };
};
```

A more complicated example on Tegra 20

Interrupt example: tegra20.dtsi

```
/ {
  interrupt-parent = <&intc>;

  intc: interrupt-controller {
    compatible = "arm,cortex-a9-gic";
    reg = <0x50041000 0x1000 0x50040100 0x0100>;
    interrupt-controller;
    #interrupt-cells = <3>;
  };

  i2c@7000c000 {
    compatible = "nvidia,tegra20-i2c";
    reg = <0x7000c000 0x100>;
    interrupts = <GIC_SPI 38 IRQ_TYPE_LEVEL_HIGH>;
    #address-cells = <1>;
    #size-cells = <0>;
    [...]
  };

  gpio: gpio {
      compatible = "nvidia,tegra20-gpio";
      reg = <0x6000d000 0x1000>;
      interrupts = <GIC_SPI 32 IRQ_TYPE_LEVEL_HIGH>, <GIC_SPI 33 IRQ_TYPE_LEVEL_HIGH>,
           [...], <GIC_SPI 89 IRQ_TYPE_LEVEL_HIGH>;
      #gpio-cells = <2>;
      gpio-controller;
      #interrupt-cells = <2>;
      interrupt-controller;
  };
};
```

Interrupt example: tegra20-harmony.dts

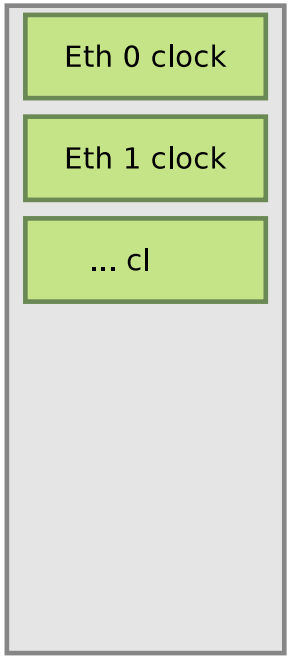
```
i2c@7000c000 {
  status = "okay";
  clock-frequency = <400000>;

  wm8903: wm8903@1a {
    compatible = "wlf,wm8903";
    reg = <0x1a>;
    interrupt-parent = <&gpio>;
    interrupts = <TEGRA_GPIO(X, 3) IRQ_TYPE_LEVEL_HIGH>;

    gpio-controller;
    #gpio-cells = <2>;

    micdet-cfg = <0>;
    micdet-delay = <100>;
    gpio-cfg = <0xffffffff 0xffffffff 0 0xffffffff 0xffffffff>;
  };
};
```

Clock tree example, Marvell Armada XP



Clock examples: instantiating clocks

```
soc {
    coreclk: mvebu-sar@18230 {
        compatible = "marvell,armada-xp-core-clock";
        reg = <0x18230 0x08>;
        #clock-cells = <1>;
    };

    cpuclk: clock-complex@18700 {
        #clock-cells = <1>;
        compatible = "marvell,armada-xp-cpu-clock";
        reg = <0x18700 0xA0>;
        clocks = <&coreclk 1>;
    };

    gateclk: clock-gating-control@18220 {
        compatible = "marvell,armada-xp-gating-clock";
        reg = <0x18220 0x4>;
        clocks = <&coreclk 0>;
        #clock-cells = <1>;
    };
}

clocks {
    /* 25 MHz reference crystal */
    refclk: oscillator {
          compatible = "fixed-clock";
          #clock-cells = <0>;
          clock-frequency = <25000000>;
    };
};
```

Clock examples: consuming clocks

CPU, using a cpuclk

```
cpu@0 {
        device_type = "cpu";
        compatible = "marvell,sheeva-v7";
        reg = <0>;
        clocks = <&cpuclk 0>;
};
```

Timer, using either a coreclk or refclk

```
timer@20300 {
        compatible = "marvell,armada-xp-timer";
        clocks = <&coreclk 2>, <&refclk>;
        clock-names = "nbclk", "fixed";
};
```

USB, using a gateclk

```
usb@52000 {
        compatible = "marvell,orion-ehci";
        reg = <0x52000 0x500>;
        interrupts = <47>;
        clocks = <&gateclk 20>;
        status = "disabled";
};
```

pinctrl binding: consumer side

- The *pinctrl* subsystem allows to manage pin muxing.
- In the Device Tree, devices that need pins to be muxed in a certain way must declare the *pinctrl* configuration they need.
- The `pinctrl-<n>` properties allow to give the list of *pinctrl* configuration needed for a certain *state* of the device.
- The `pinctrl-names` property allows to give a name to each state.
- When a device is probed, its `default` *pinctrl* state is requested automatically.

```
ssp0: ssp@80010000 {
    pinctrl-names = "default";
    pinctrl-0 = <&mmc0_8bit_pins_a
                 &mmc0_cd_cfg &mmc0_sck_cfg>;
    [...]
};
```

pinctrl configurations

- A *pinctrl* configuration provides a list of pins and their configuration.
- Such configurations are defined as *sub-nodes* of the *pinctrl device*, either at the SoC-level, or board-level.
- The binding for such configurations is highly dependent on the specific *pinctrl* driver being used.

i.MX28

```
mmc0_8bit_pins_a: mmc0-8bit@0 {
   fsl,pinmux-ids = <
      0x2000 /* MX28_PAD_SSP0_DATA0__SSP0_D0 */
      0x2010 /* MX28_PAD_SSP0_DATA1__SSP0_D1 */
      [...]
      0x2090 /* MX28_PAD_SSP0_DETECT__SSP0_... */
      0x20a0 /* MX28_PAD_SSP0_SCK__SSP0_SCK */
   >;
   fsl,drive-strength = <1>;
   fsl,voltage = <1>;
   fsl,pull-up = <1>;
};
```

Marvell Kirkwood

```
pmx_nand: pmx-nand {
   marvell,pins = "mpp0", "mpp1", "mpp2", "mpp3",
                  "mpp4", "mpp5", "mpp18",
                  "mpp19";
   marvell,function = "nand";
};
```

DT is hardware description, not configuration

- The Device Tree is really a hardware description language.
- It should **describe the hardware layout**, and how it works.
- But it should **not describe which particular hardware configuration** you're interested in.
- As an example:
  - You may describe in the DT whether a particular piece of hardware supports DMA or not.
  - But you may not describe in the DT if you *want* to use DMA or not.

DT bindings as an ABI

- Since the DT is OS independent, it should also be stable.
- The original idea is that DTBs can be flashed on some devices by the manufacturer, so that the user can install whichever operating system it wants.
- Once a Device Tree binding is defined, and used in DTBs, it should no longer be changed anymore. It can only be extended.
- This normally means that **Device Tree bindings become part of the kernel ABI**, and it should be handled with the same care.
- However, kernel developers are realizing that this is really hard to achieve and slowing down the integration of drivers.
  - The ARM Kernel Mini-summit discussions have relaxed those rules.
  - There will be additional discussions during the Kernel Summit, with final conclusions published afterwards.

Basic guidelines for binding design

- **A precise compatible string is better than a vague one**
  - You have a driver that covers both variants T320 and T330 of your hardware. You may be tempted to use `foo,t3xx` as your compatible string.
  - Bad idea: what if T340 is slightly different, in an incompatible way? You'd better use `foo,t320` for both T320 and T330.
- **Do not encode too much hardware details in the DT**
  - When two hardware variants are quite similar, some developers are tempted to encode all the differences in the DT, including register offsets, bit masks or offsets.
  - Bad idea: it makes the binding more complex, and therefore less resilient to future changes. Instead, use two different compatible strings and handle the differences in the driver.

Future directions

- More DT bindings for **various subsystems**
- A **tool to validate DTS against bindings**
  - Currently, the DT compiler only makes syntax checks, no conformance checks against bindings.
  - Proposal from Benoit Cousson and Fabien Parent, `[RFC 00/15] Device Tree schemas and validation`
- **Take out the Device Tree source files from the kernel tree**
  - DTs are OS independent, so they can be used for other purposes than just Linux. They are already used by Barebox or U-Boot.
  - Having them outside of the kernel reduces the amount of *churn* in the kernel source.
  - But is IMO likely to cause a huge number of compatibility issues.

References

- Power.orgTM Standard for Embedded Power Architecture Platform Requirements (ePAPR), `http://www.power.org/resources/downloads/Power_ePAPR_APPROVED_v1.0.pdf`
- DeviceTree.org website, `http://www.devicetree.org`
- Device Tree documentation in the kernel sources, `Documentation/devicetree`
- The Device Tree kernel mailing list, `http://dir.gmane.org/gmane.linux.drivers.devicetree`

Questions?

Thomas Petazzoni

thomas.petazzoni@free-electrons.com

http://free-electrons.com/pub/conferences/2013/elce/petazzoni-device-tree-dummies/