Device Tree: hardware description for everybody!

Thomas Petazzoni
thomas.petazzoni@bootlin.com

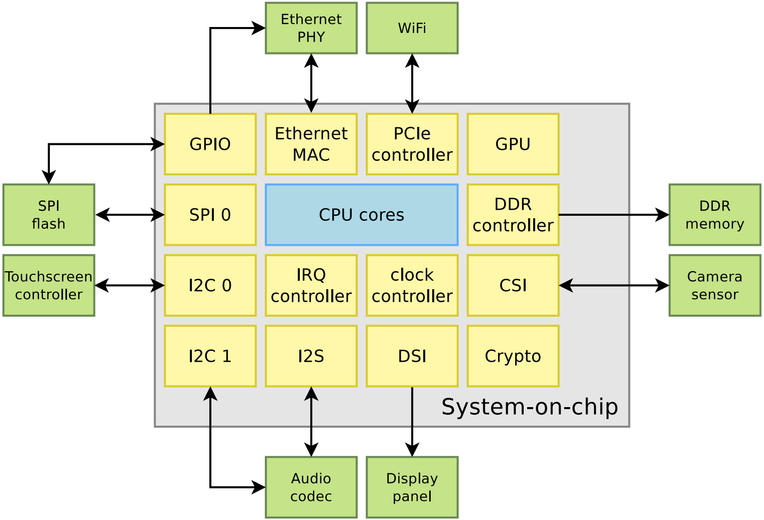
Thomas Petazzoni

- 12+ years CTO/Embedded Linux engineer at Bootlin
  - Embedded Linux **expertise**
  - **Development**, consulting and training
  - Bootloader, Linux kernel, Yocto Project, Buildroot
  - Complete Linux BSP development
  - Hardware support in bootloader/Linux
  - Strong open-source focus: upstreaming and contributions
  - Freely available training materials
- Co-maintainer of **Buildroot**
- Living in **Toulouse**, France

Agenda

- This talk is an update from the *Device Tree for Dummies* talk given in 2013/2014
- Why the Device Tree ?
- Basic Device Tree syntax
- Device Tree inheritance
- Device Tree specifications and bindings
- Building and validating Device Trees
- Common properties and examples

Your typical embedded platform



Discoverable vs. non-discoverable hardware

- Some hardware busses provide **discoverability** mechanisms
  - E.g: PCI(e), USB
  - One does not need to know ahead of time what will be connected on these busses
  - Devices can be enumerated and identified at runtime
  - Concept of *vendor ID*, *product ID*, *device class*, etc.
- But many hardware busses **do not provide discoverability** mechanisms
  - E.g: I2C, SPI, 1-wire, memory-mapped, etc.
  - One needs to know what is connected on those busses, and how they are connected to the rest of the system
  - Embedded systems typically make extensive use of such busses

Hardware description for non-discoverable hardware

Allows the operating system or bootloader to **know things like**:

- This **system-on-chip** has:
  - 2 Cortex-A9 CPU cores
  - 2 memory-mapped UART controllers of *this* variant, one with registers at `0xF1000000` and IRQ *23*, and another with registers at `0xF1001000` and IRQ *24*
  - 3 I2C controllers of *that* variant, with registers at *those* memory-mapped addresses, *those* IRQs and taking their input clock from *this* source
- This **board** has an CS4234 audio codec
  - Connected on the I2C bus 0 of the SoC, at slave address *0x45*
  - Connected to the I2S interface 2 of the SoC, with the codec providing the clocks
  - With its reset signal connected to GPIO *67* of the SoC

These details **cannot be guessed** by the operating system/bootloader.

Describing non-discoverable hardware

- Directly in the **OS/bootloader code**, using compiled data structures, typically in C
  - How it was done on most embedded platforms in Linux, U-Boot.
  - Considered not maintainable/sustainable on ARM32, which motivated the move to another solution.
- Using **ACPI** tables
  - On *x86* systems, but also on a subset of ARM64 platforms
  - Tables provided by the firmware
- Using a **Device Tree**
  - On most embedded-oriented CPU architectures that run Linux: ARC, ARM64, RISC-V, ARM32, PowerPC, Xtensa, MIPS, etc.
  - Originates from the PowerPC world, not Linux specific
  - Now used by Linux, U-Boot, Barebox, TF-A, FreeBSD, etc.
  - Writing/tweaking a DT is now always necessary when porting Linux to a new board.
  - The topic of this talk !

Device Tree: principle

- A tree data structure describing the hardware is written by a developer in a *Device Tree Source* file, `.dts`
- Gets compiled to a more efficient *Device Tree Blob* representation, `.dtb` by the *Device Tree Compiler*, `dtc`
- The resulting `.dtb` accurately describes the hardware platform in an **OS-agnostic** way and:
  - Can be **linked directly** inside a bootloader binary (U-Boot, Barebox)
  - Can be **passed** to the operating system by the bootloader (Linux)
  - U-Boot: `bootz <kernel-addr> - <dtb-addr>`

- Tree of **nodes**
- Nodes with **properties**
- A node ≈ a device or IP block
- Properties ≈ device characteristics
- `dtc` only does syntax checking, no semantic validation

```
/ {
    node@0 {
        a-string-property = "A string";
        a-string-list-property = "first string", "second string";
        a-byte-data-property = [0x01 0x23 0x34 0x56];

        child-node@0 {
            first-child-property;
            second-child-property = <1>;
            a-reference-to-something = <&node1>;
        };

        child-node@1 {
        };
    };

    node1: node@1 {
        an-empty-property;
        a-cell-property = <1 2 3 4>;

        child-node@0 {
        };
    };
};
```

Node name

Unit address

Property name

Property value

Properties of node@0

Bytestring

A phandle (reference to another node)

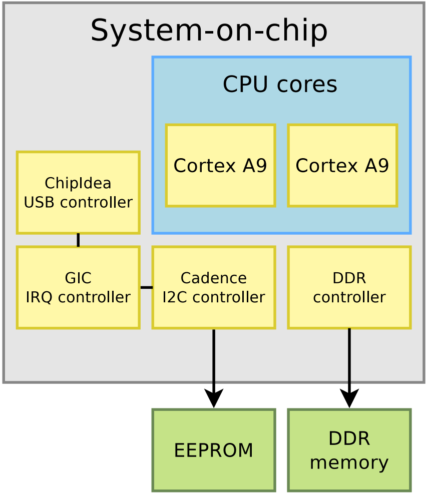
Label

Four cells (32 bits values)

Simplified example

```
/ {
    #address-cells = <1>;
    #size-cells = <1>;
    compatible = "vendor1,board", "vendor2,soc";

    cpus { ... };
    memory@0 { ... };
    chosen { ... };
    soc {
       intc: interrupt-controller@f8f01000 { ... };
       i2c0: i2c@e0004000 { ... };
       usb0: usb@e0002000 { ... };
    };
};
```



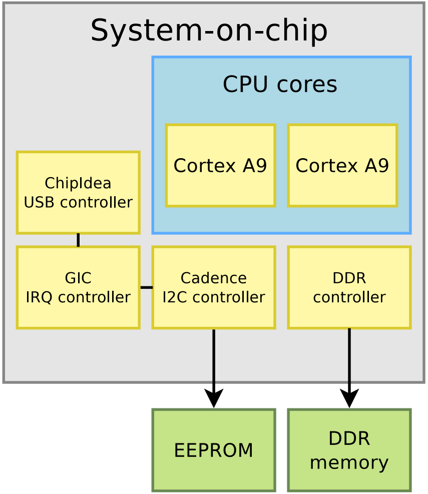
Simplified example

```
/ {
   cpus {
     #address-cells = <1>;
     #size-cells = <0>;

     cpu0: cpu@0 {
         compatible = "arm,cortex-a9";
         device_type = "cpu";
         reg = <0>;
     };

     cpu1: cpu@1 {
         compatible = "arm,cortex-a9";
         device_type = "cpu";
         reg = <1>;
     };
   };

   memory@0 { ... };
   chosen { ... };
   soc {
      intc: interrupt-controller@f8f01000 { ... };
      i2c0: i2c@e0004000 { ... };
      usb0: usb@e0002000 { ... };
   };
};
```

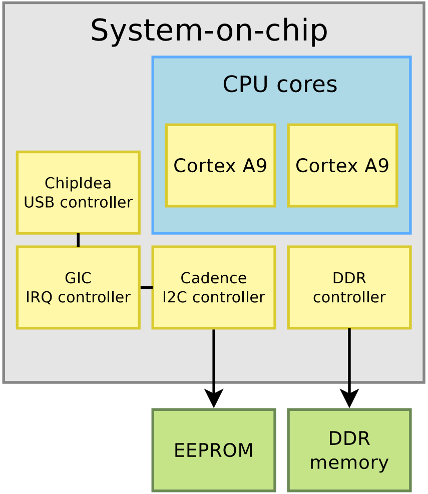


Simplified example

```
/ {
   cpus { ... };
   memory@0 {
       device_type = "memory";
       reg = <0x0 0x20000000>;
   };

   chosen {
       bootargs = "";
       stdout-path = "serial0:115200n8";
   };

   soc {
      intc: interrupt-controller@f8f01000 { ... };
      i2c0: i2c@e0004000 { ... };
      usb0: usb@e0002000 { ... };
   };
};
```



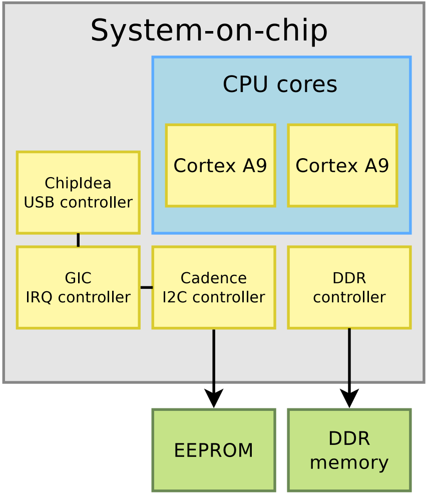
Simplified example

```
/ {
   cpus { ... };
   memory@0 { ... };
   chosen { ... };

   soc {
      compatible = "simple-bus";
      #address-cells = <1>;
      #size-cells = <1>;
      interrupt-parent = <&intc>;

      intc: interrupt-controller@f8f01000 {
              compatible = "arm,cortex-a9-gic";
              #interrupt-cells = <3>;
              interrupt-controller;
              reg = <0xF8F01000 0x1000>,
                    <0xF8F00100 0x100>;
      };

      i2c0: i2c@e0004000 { ... };
      usb0: usb@e0002000 { ... };
   };
};
```



Simplified example

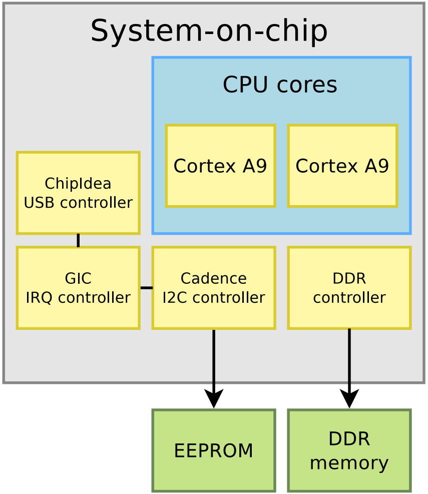
```
/ {
   cpus { ... };
   memory@0 { ... };
   chosen { ... };

   soc {
      intc: interrupt-controller@f8f01000 { ... };

      i2c0: i2c@e0004000 {
              compatible = "cdns,i2c-r1p10";
              status = "okay";
              clocks = <&clkc 38>;
              interrupts = <GIC_SPI 25 IRQ_TYPE_LEVEL_HIGH>;
              reg = <0xe0004000 0x1000>;
              #address-cells = <1>;
              #size-cells = <0>;
              clock-frequency = <400000>;

              eeprom0: eeprom@52 {
                     compatible = "atmel,24c02";
                     reg = <0x52>;
              };
      };

      usb0: usb@e0002000 { ... };
   };
};
```



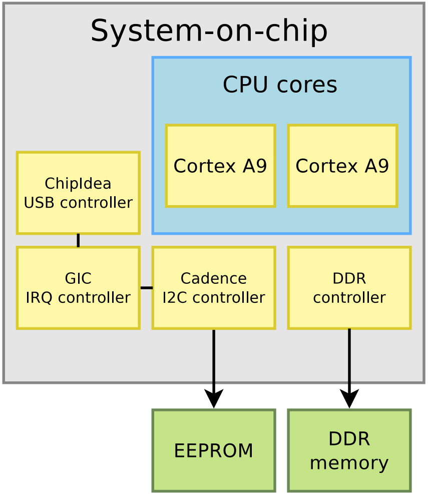
Simplified example

```
/ {
   cpus { ... };
   memory@0 { ... };
   chosen { ... };

   soc {
      compatible = "simple-bus";
      #address-cells = <1>;
      #size-cells = <1>;
      interrupt-parent = <&intc>;

      intc: interrupt-controller@f8f01000 { ... };
      i2c0: i2c@e0004000 { ... };

      usb0: usb@e0002000 {
              compatible = "xlnx,zynq-usb-2.20a", "chipidea,usb2";
              status = "okay";
              clocks = <&clkc 28>;
              interrupt-parent = <&intc>;
              interrupts = <GIC_SPI 21 IRQ_TYPE_LEVEL_HIGH>;
              reg = <0xe0002000 0x1000>;
              phy_type = "ulpi";
              dr_mode = "host";
              usb-phy = <&usb_phy0>;
      };
   };
};
```



Where are Device Tree Sources located ?

- Even though they are OS-agnostic, no central and OS-neutral place to host Device Tree sources and share them between projects
  - Often discussed, never done
- In practice, the Linux kernel sources can be considered as the canonical location for Device Tree Source files
  - `arch/<ARCH>/boot/dts`
  - $\approx$ 4700 Device Tree Source files in Linux as of 5.10
- Duplicated/synced in various projects
  - U-Boot, Barebox

Device Tree inheritance

- 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
  - Only `.dts` files are accepted as input to `dtc`
- 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.
- Uses the C pre-processor `#include` directive
  - Using the C pre-processor also allows to use `#define` to replace hardcoded values by human readable definitions

Device Tree inheritance example

Definition of the AM33xx SoC

```
/ {
	compatible = "ti,am33xx";
	[...]
	ocp {
		uart0: serial@44e09000 {
			compatible = "ti,omap3-uart";
			reg = <0x44e09000 0x2000>;
			interrupts = <72>;
			status = "disabled";
		};
	};
};
```

am33xx.dtsi

+

Definition of the BeagleBone board

```
#include "am33xx.dtsi"

/ {
	compatible = "ti,am335x-bone", "ti,am33xx";
	[...]
	ocp {
		uart0: serial@44e09000 {
			pinctrl-names = "default";
			pinctrl-0 = <&uart0_pins>;
			status = "okay";
		};
	};
};
```

am335x-bone.dts

=

Compiled DTB

```
/ {
	compatible = "ti,am335x-bone", "ti,am33xx";
	[...]
	ocp {
		uart0: serial@44e09000 {
			compatible = "ti,omap3-uart";
			reg = <0x44e09000 0x2000>;
			interrupts = <72>;
			pinctrl-names = "default";
			pinctrl-0 = <&uart0_pins>;
			status = "okay";
		};
	};
};
```

am335x-bone.dtb

Note: the real DTB is in binary format. Here we show the text equivalent of the DTB contents;

Building Device Trees in Linux

- On ARM/ARM64, `arch/<ARCH>/boot/dts/Makefile` or `arch/<ARCH>/boot/dts/<vendor>/Makefile` indicates which DT to build depending on the platform

arch/arm64/boot/dts/marvell/Makefile

```
dtb-$(CONFIG_ARCH_MVEBU) += armada-3720-db.dtb
dtb-$(CONFIG_ARCH_MVEBU) += armada-3720-espressobin.dtb
```

- Building the kernel with `make` will also build the Device Trees on most architectures
- Explicit `make dtbs` target also available

```
  DTC   armada-3720-db.dtb
  DTC   armada-3720-espressobin.dtb
```

Validating Device Tree in Linux

- `dtc` only does syntaxic validation
- YAML bindings allow to do semantic validation
  - `make dt_bindings_check`
    verify that YAML bindings are valid
  - `make dtbs_check`
    validate DTs currently enabled against YAML bindings

Exploring the DT on the target

- In /sys/firmware/devicetree/base, there is a directory/file representation of the Device Tree contents

```
# ls -l /sys/firmware/devicetree/base/
total 0
-r--r--r--    1 root     root      4 Jan  1 00:00 #address-cells
-r--r--r--    1 root     root      4 Jan  1 00:00 #size-cells
drwxr-xr-x    2 root     root      0 Jan  1 00:00 chosen
drwxr-xr-x    3 root     root      0 Jan  1 00:00 clocks
-r--r--r--    1 root     root     34 Jan  1 00:00 compatible
[...]
-r--r--r--    1 root     root      1 Jan  1 00:00 name
drwxr-xr-x   10 root     root      0 Jan  1 00:00 soc
```

- If dtc is available on the target, possible to "unpack" the Device Tree using:
  dtc -I fs /sys/firmware/devicetree/base

Modifying the Device Tree at runtime

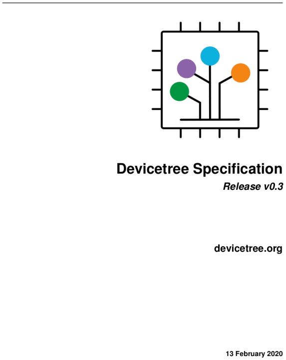
- U-Boot automatically patches the *Device Tree Blob* passed to Linux
  - Sets the RAM base address and size
  - Sets the kernel command line
  - Sets MAC address for network interfaces
- Additional *Device Tree Blob* patching in U-Boot can be done
  - Using `fdt` commands: `fdt set`, `fdt mknode`, `fdt rm`
  - Using *Device Tree Overlays*

Device Tree Overlays

- A number of platforms have some flexibility aspects that are difficult to describe in a static Device Tree
  - Base boards to which an arbitrary number of expansion boards can be connected: BeagleBoard capes, RaspberrPi hats, etc.
  - FPGA with arbitrary IP blocks synthetized
- A *Device Tree Overlay* is a small snippet of Device Tree that acts as a *patch* to a Device Tree
  - For example to describe additional devices provided by an expansion board
- U-Boot supports applying DT overlays
- No support in Linux for applying DT overlays however
- Examples: `https://github.com/raspberrypi/linux/tree/rpi-5.4.y/arch/arm/boot/dts/overlays/`

Device Tree specifications

- How does one know how to write the correct nodes/properties to describe a given hardware platform ?
- The **DeviceTree Specifications** at https://www.devicetree.org/specifications/ gives the base Device Tree syntax and specifies a number of standard properties.
  - Far from being sufficient, though.
- The **Device Tree Bindings** are documents that each describe how a particular piece of hardware.
  - `Documentation/devicetree/bindings/` in Linux kernel sources
  - Reviewed by DT bindings maintainer team
  - Legacy: human readable documents
  - New norm: YAML-written specifications



Device Tree binding: old style

```
I2C for Atmel platforms

Required properties :
- compatible : Must be one of:
        "atmel,at91rm9200-i2c",
        "atmel,at91sam9261-i2c",
        "atmel,at91sam9260-i2c",
        "atmel,at91sam9g20-i2c",
        "atmel,at91sam9g10-i2c",
        "atmel,at91sam9x5-i2c",
        "atmel,sama5d4-i2c",
        "atmel,sama5d2-i2c",
        "microchip,sam9x60-i2c".
- reg: physical base address of the controller and length of memory mapped
     region.
- interrupts: interrupt number to the cpu.
- #address-cells = <1>;
- #size-cells = <0>;
- clocks: phandles to input clocks.

Optional properties:
- clock-frequency: Desired I2C bus frequency in Hz, otherwise defaults
  to 100000
- dmas: A list of two dma specifiers, one for each entry in
  dma-names.
- dma-names: should contain "tx" and "rx".
- scl-gpios: specify the gpio related to SCL pin
- sda-gpios: specify the gpio related to SDA pin
[...]
```

```
Examples :

i2c0: i2c@fff84000 {
        compatible = "atmel,at91sam9g20-i2c";
        reg = <0xfff84000 0x100>;
        interrupts = <12 4 6>;
        #address-cells = <1>;
        #size-cells = <0>;
        clocks = <&twi0_clk>;
        clock-frequency = <400000>;

        24c512@50 {
                compatible = "atmel,24c512";
                reg = <0x50>;
                pagesize = <128>;
        }
}
```

Device Tree binding: YAML style

```yaml
# SPDX-License-Identifier: (GPL-2.0 OR BSD-2-Clause)
# Copyright 2019 BayLibre, SAS
%YAML 1.2
---
$id: "http://devicetree.org/schemas/i2c/amlogic,meson6-i2c.yaml#"
$schema: "http://devicetree.org/meta-schemas/core.yaml#"

title: Amlogic Meson I2C Controller

maintainers:
  - Neil Armstrong <narmstrong@baylibre.com>
  - Beniamino Galvani <b.galvani@gmail.com>

allOf:
  - $ref: /schemas/i2c/i2c-controller.yaml#

properties:
  compatible:
    enum:
      - amlogic,meson6-i2c # Meson6, Meson8 and compatible SoCs
      - amlogic,meson-gxbb-i2c # GXBB and compatible SoCs

  reg:
    maxItems: 1

  interrupts:
    maxItems: 1
```

```yaml
  clocks:
    minItems: 1

required:
  - compatible
  - reg
  - interrupts
  - clocks

unevaluatedProperties: false

examples:
  - |
    i2c@c8100500 {
        compatible = "amlogic,meson6-i2c";
        reg = <0xc8100500 0x20>;
        interrupts = <92>;
        clocks = <&clk81>;
        #address-cells = <1>;
        #size-cells = <0>;

        eeprom@52 {
            compatible = "atmel,24c32";
            reg = <0x52>;
        };
    };
```

Device Tree design principles

- Describe hardware (how the hardware is), not configuration (how I choose to use the hardware)
- OS-agnostic
  - For a given piece of HW, Device Tree should be the same for U-Boot, FreeBSD or Linux
  - There should be no need to change the Device Tree when updating the OS
- Describe integration of hardware components, not the internals of hardware components
  - The details of how a specific device/IP block is working is handled by code in device drivers
  - The Device Tree describes how the device/IP block is connected/integrated with the rest of the system: IRQ lines, DMA channels, clocks, reset lines, etc.
- Like all beautiful design principles, these principles are not sometimes violated.

The compatible property

- Is a list of strings
  - From the most specific to the less specific
- Describes the specific binding to which the node complies.
- It uniquely identifies the *programming model* of the device.
- Practically speaking, it is used by the operating system to find the appropriate driver for this device.
- Special value: `simple-bus` indicates a bus where all sub-nodes are memory-mapped devices. Generally used for devices inside the SoC.
- When describing real hardware, typical form is `vendor,model`
- Examples:
  - `compatible = "arm,armv8-timer";`
  - `compatible = "actions,s900-uart", "actions,owl-uart";`
  - `compatible = "regulator-fixed";`
  - `compatible = "gpio-keys";`

Matching with drivers in Linux: platform driver

drivers/tty/serial/imx.c

```
static const struct of_device_id imx_uart_dt_ids[] = {
        { .compatible = "fsl,imx6q-uart", .data = ... },
        { .compatible = "fsl,imx53-uart", .data = ... },
        { .compatible = "fsl,imx1-uart", .data = ... },
        { .compatible = "fsl,imx21-uart", .data = ... },
        { /* sentinel */ }
};
MODULE_DEVICE_TABLE(of, imx_uart_dt_ids);

static struct platform_driver imx_uart_platform_driver = {
        .probe = imx_uart_probe,
        .remove = imx_uart_remove,

        .id_table = imx_uart_devtype,
        .driver = {
                .name = "imx-uart",
                .of_match_table = imx_uart_dt_ids,
                .pm = &imx_uart_pm_ops,
        },
};
```

Common properties

- `reg`
  - Memory-mapped devices: base address and size of the registers. Can have several entries.
  - I2C devices: address on the I2C bus
  - SPI devices: chip select number
- `interrupts`, `interrupt-parent`, `interrupts-extended`: interrupts lines used by the device, and which interrupt controller they are connected to.
- `clocks`: which clock(s) are used by the device, from which clock controller
- `dmas`: which DMA controller and channels are used by the device
- `status`: `okay` means the device is present and should be enabled, otherwise, the device is left unused
- `pinctrl-*`: indicates the pin-muxing configuration requested by the device

Cells concept

- Integer values represented as 32-bit integers called cells

```
soc {
    /* This property has 1 cell */
    foo = <0xdeadbeef>;
};
```

Cells concept

- Integer values represented as 32-bit integers called cells
- Encoding a 64-bit value requires two cells

```
soc {
    /* This property has 2 cells */
    foo = <0xdeadbeef 0xbadcafe>;
};
```

Cells concept

- Integer values represented as 32-bit integers called cells
- Encoding a 64-bit value requires two cells
- `#address-cells` and `#size-cells`: how many cells are used in sub-nodes to encode the address and size in the `reg` property

```
soc {
    compatible = "simple-bus";
    #address-cells = <1>;
    #size-cells = <1>;

    i2c@f1001000 {
        reg = <0xf1001000 0x1000>;
        #address-cells = <1>;
        #size-cells = <0>;

        eeprom@52 {
            reg = <0x52>;
        };
    };
};
```

Cells concept

- Integer values represented as 32-bit integers called cells
- Encoding a 64-bit value requires two cells
- `#address-cells` and `#size-cells`: how many cells are used in sub-nodes to encode the address and size in the `reg` property
- `#interrupts-cells`: how many cells are used to encode interrupt specifiers for this interrupt controller

```
soc {
    intc: interrupt-controller@f1002000 {
        compatible = "foo,bar-intc";
        reg = <0xf1002000 0x1000>;
        interrupt-controller;
        #interrupt-cells = <2>;
    };

    i2c@f1001000 {
        interrupt-parent = <&intc>;
        /* Must have two cells */
        interrupts = <12 24>;
    };
};
```

Cells concept

- Integer values represented as 32-bit integers called cells
- Encoding a 64-bit value requires two cells
- `#address-cells` and `#size-cells`: how many cells are used in sub-nodes to encode the address and size in the `reg` property
- `#interrupts-cells`: how many cells are used to encode interrupt specifiers for this interrupt controller
- Ditto `#clock-cells`, `#gpio-cells`, `#phy-cells`, `#pwm-cells`, `#dma-cells`, etc.

```
soc {
    clkc: clock@f1003000 {
        compatible = "foo,bar-clock";
        reg = <0xf1003000 0x1000>;
        #clock-cells = <3>;
    };

    i2c@f1001000 {
        /* Must have three cells */
        clocks = <&clkc 12 24 32>;
    };
};
```

-names properties

- Some properties are associated to a corresponding `<prop>-names` property
- Gives some human-readable names to entries of the corresponding `<prop>` properties

```
interrupts      = <0 59 0>, <0 70 0>;
interrupt-names = "macirq", "macpmt";
clocks          = <&car 39>, <&car 45>, <&car 86>, <&car 87>;
clock-names     = "gnssm_rgmii", "gnssm_gmac", "rgmii", "gmac";
```

- Such names can be typically be used by the driver
  - `platform_get_irq_byname(pdev, "macirq");`

Conclusion

- Representation of non-discoverable hardware
- Tree of nodes, with properties
- Standardization based on *Device Tree bindings*
- New description language with lots of properties and sometimes complex bindings
- Used for numerous CPU architectures
- Now widely used outside of Linux
- A must know for all embedded Linux developers!

Questions? Suggestions? Comments?

Thomas Petazzoni

thomas.petazzoni@bootlin.com

https://bootlin.com/pub/conferences/2020/lee/petazzoni-dt-hw-description-everybody