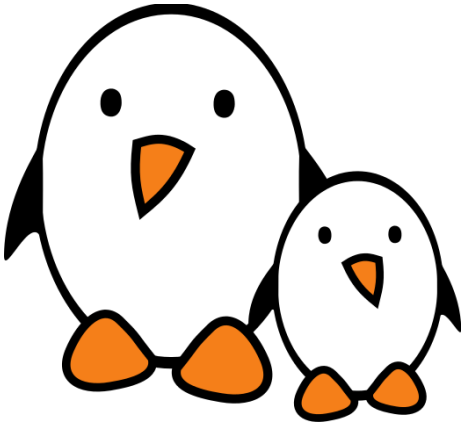
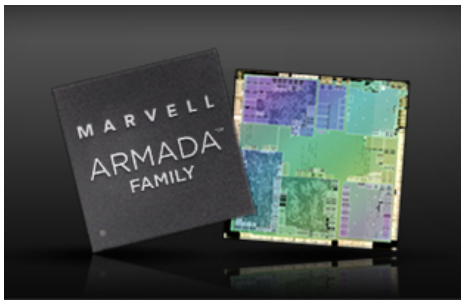
Device Tree as a stable ABI: a fairy tale?

Thomas Petazzoni
Free Electrons
thomas.petazzoni@free-electrons.com

Thomas Petazzoni

- CTO and Embedded Linux engineer at **Free Electrons**
  - Embedded Linux specialists.
  - Development, consulting and training.
  - `http://free-electrons.com`
- Contributions
  - **Kernel support for the Marvell Armada** 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





Context

- ARM platforms have been switching to a *Device Tree* based hardware representation for a few years.
- Intended goals
  - get rid of numerous board files
  - encourage the usage of generic subsystems instead of arch-specific solutions
  - and make multiplatform support easier
  - $\rightarrow$ quite successful in those goals!
- But the usage of the *Device Tree* on ARM also came with a requirement: it should be a **stable ABI**
  - An old Device Tree for a given hardware platform must continue to work with newer kernel versions.

Assumptions

- Handling backward compatibility is easy
  - "*Just handle a default behavior to be compatible with the old situation*"
- Having stable Device Tree is needed by the users: hardware vendors, distributions
- Good review by a team of Device Tree bindings maintainers will make sure the bindings are good enough to be stable.

Backward compatibility is easy?

- Yes, if the change in your binding is just to add a new property.
- But what if
  - you need to add new nodes?
  - *phandles* pointing to them?
  - re-organize nodes to use `simple-mfd`?
  - ...
- Causes:
  - Undocumented or poorly documented hardware: no datasheet, only vendor BSP code available
  - Badly understood hardware: information was available, but was somehow overlooked.
  - Simply badly designed bindings.
- We'll go through a number of examples to illustrate those cases.

Marvell Berlin BG2Q system control IP (1)

- No datasheet available, work is done based on vendor kernel code + datasheets of older SoCs that are sometimes similar, sometimes not.
- A set of registers called *System control* that controls the muxing of a number of pins.
- A classical DT representation:
- And a *pinctrl* `platform_driver` matching against this compatible string.
- So far, so good.

```
sysctrl: pin-controller@d000 {
    compatible = "marvell,berlin2q-system-ctrl";
    reg = <0xd000 0x100>;

    uart0_pmux: uart0-pmux {
        groups = "GSM12";
        function = "uart0";
    };

    uart1_pmux: uart1-pmux {
        groups = "GSM14";
        function = "uart1";
    };
};
```

Marvell Berlin BG2Q system control IP (2)

- Much later, asked to work on supporting the ADC.
- Turns out the ADC registers are part of the same *System control* registers.
- Ah, great there is new `simple-mfd` Device Tree binding proposed by Linus Walleij. Exactly does what we need.
- Works fine, but how to handle *backward compatibility* when DT nodes are re-organized this way?

```
sysctrl: system-controller@d000 {
    compatible = "marvell,berlin2q-system-ctrl",
                 "simple-mfd", "syscon";
    reg = <0xd000 0x100>;

    sys_pinctrl: pin-controller {
        compatible =
            "marvell,berlin2q-system-pinctrl";

        uart0_pmux: uart0-pmux {
            groups = "GSM12";
            function = "uart0";
        };
        [...]
    };

    adc: adc {
       compatible = "marvell,berlin2-adc";
       interrupt-parent = <&sic>;
       interrupts = <12>, <14>;
       interrupt-names = "adc", "tsen";
    };
};
```

- Similar, but even more complicated situation with the *Chip Control* registers: pin-muxing, reset, clocks.

Allwinner MMC clock phase (1)

- No datasheet, only vendor code.
- Originally: one clock for the MMC, plus some *magic bits* controlling two phases: *output phase* and *sample phase*.
- Represented as one clock `mmcX_clk`

```
clocks = <&ahb_gates 8>, <&mmc0_clk>;
clock-names = "ahb", "mmc";
```

- Custom clock API used to control both the *output phase* and *sample phase*, used by the MMC driver.

```
clk_sunxi_mmc_phase_control(host->clk_mmc, sclk_dly, oclk_dly);
```

Allwinner MMC clock phase (2)

- Contacted the vendor, and after lots of efforts, finally got some details.
- In fact, the *output* and *sample* parameters are clocks by themselves: so there are three clocks and not one.
- Moved to a three clocks model:

```
clocks = <&ahb_gates 8>, <&mmc0_clk 0>, <&mmc0_clk 1>, <&mmc0_clk 2>;
clock-names = "ahb", "mmc", "output", "sample";
```

- And a generic clock framework API:

```
clk_set_phase(host->clk_sample, sclk_dly);
clk_set_phase(host->clk_output, oclk_dly);
```

- To keep DT backward compatibility, options are:
  - Keep an ugly non-accurate HW representation
  - Carry lots of legacy (and never tested) code in the MMC and clock drivers

Marvell window 13 issue (1)

- Marvell processors have a concept of *configurable windows*
- You define a base address, a size, and a target device (NOR, PCIe device, BootROM, SRAM, etc.) and the device appears in physical memory.
- Some of the windows have a *remap* capability. In addition to the *base address* visible from the CPU side, you can define a *remap address* which is the device visible address.
- Each window has *control* and *base* registers
- Remappable windows also have *remap high* and *remap low registers*
- Not configuring those *remap* registers when available lead to an unusable window.

Marvell window 13 issue (2)

- At first sight, Armada 370, 375, 38x and XP looked the same.
- They have 20 windows, of which the first 8 have the remap capability.
- So, in all Device Tree files, we used:

```
compatible = "marvell,armada380-mbus", "marvell,armada370-mbus", "simple-bus";
```

```
compatible = "marvell,armada375-mbus", "marvell,armada370-mbus", "simple-bus";
```

- Originally, only the `marvell,armada370-mbus` was matching.
- Later, we discovered that Armada XP, 375 and 38x had a *remappable* window 13.
- You need a different handling of window 13 between Armada 370 on one side, and Armada XP/375/38x on the other side.

Marvell window 13 issue (3)

- Thanks to the provision of a more SoC-specific compatible string, we could override the behavior for Armada XP, Armada 375, Armada 38x.

```
compatible = "marvell,armada375-mbus", "marvell,armada370-mbus", "simple-bus";
```

- However, `marvell,armada370-mbus` remains in the compatible list, which is incorrect.
  - Armada XP/375/38x functionality is **not** a super set of Armada 370 functionality.
  - Using the Armada 370 *MBus* behavior on Armada XP, 375 and 38x is a bug.
- Not a backward compatibility problem, but shows that one can easily overlook minor differences between SoC revisions that make similar hardware blocks incompatible.

Marvell CPU reset IP

- On Armada XP, enabling SMP requires fiddling with the *Power Management Service Unit* (PMSU) and CPU reset registers.
- So, we created a Device Tree node:

```
armada-370-xp-pmsu@22000 {
    compatible = "marvell,armada-370-xp-pmsu";
    reg = <0x22100 0x400>, <0x20800 0x20>;
};
```

- The first register area is the PMSU, the second are the CPU reset bits.
- One *driver*, `pmsu.c`, was mapping those registers, and providing the necessary functions for SMP enabling.
- We introduced this in Linux 3.8, at a time where there was no *reset framework* $\rightarrow$ no use of the *reset* DT binding.

Marvell CPU reset IP (2)

- Then came the Armada 375.
  - It has the exact same CPU reset bits, but does not have a PMSU.
- Clearly, one DT node for both registers areas was a mistake, so we splitted in two nodes.
- We kept backward compatibility code in the new *CPU reset* driver, by looking for the register addresses in the second *reg* of `marvell,armada-370-xp-pmsu`.

```
pmsu@22000 {
    compatible =
      "marvell,armada-370-pmsu";
    reg = <0x22000 0x1000>;
};

cpurst@20800 {
    compatible =
      "marvell,armada-370-cpu-reset";
    reg = <0x20800 0x20>;
};
```

- This example is more a design mistake from the developers, but doesn't this happens?
- Shouldn't the review have pointed out the stupidity of one DT nodes for two different hardware blocks?

Marvell CPU reset IP (3)

- Ideally, we would like to use the CPU reset framework and its Device Tree binding, now that it exists?
- Refactoring code to use more modern framework is what good kernel citizens should do, no?
- Except that using the CPU reset framework involves creating `resets` *phandles* pointing to the reset controller node.
- And in our case, the *reset controller* node did not exist in the old DT, since it was mistakenly mixed with the PMSU node.
- Moving to the CPU reset framework *and* supporting backward compatibility would have required:
  - Dynamically creating the CPU reset node, with at least a `#reset-cells` property.
  - Dynamically adding the `resets` properties to each CPU, with a phandle pointing to the CPU reset node.
- $\rightarrow$ in the end we gave up and simply did not use the *reset framework*

Marvell CPU reset IP (4)

Old DT

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

        cpu@1 {
                device_type = "cpu";
                compatible = "marvell,sheeva-v7";
                reg = <1>;
        };
};

armada-370-xp-pmsu@22000 {
        compatible = "marvell,armada-370-xp-pmsu";
        reg = <0x22100 0x400>, <0x20800 0x20>;
};
```

New non-ideal DT

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

        cpu@1 {
                device_type = "cpu";
                compatible = "marvell,sheeva-v7";
                reg = <1>;
        };
};

pmsu@22000 {
        compatible = "marvell,armada-370-pmsu";
        reg = <0x22000 0x1000>;
};

cpurst@20800 {
        compatible = "marvell,armada-370-cpu-reset";
        reg = <0x20800 0x20>;
};
```

Marvell CPU reset IP (5)

Old DT

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

        cpu@1 {
                device_type = "cpu";
                compatible = "marvell,sheeva-v7";
                reg = <1>;
        };
};

armada-370-xp-pmsu@22000 {
        compatible = "marvell,armada-370-xp-pmsu";
        reg = <0x22100 0x400>, <0x20800 0x20>;
};
```

Ideal DT

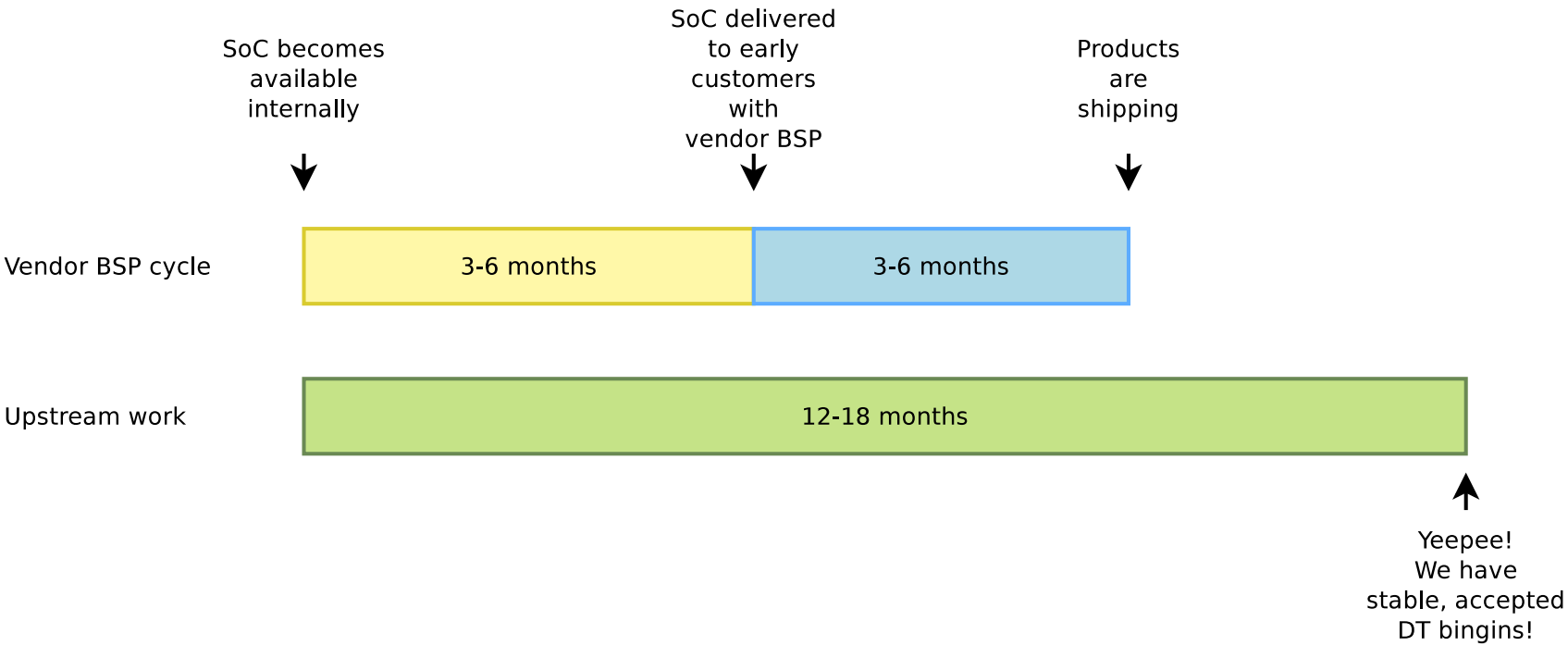
```
cpus {
        cpu@0 {
                device_type = "cpu";
                compatible = "marvell,sheeva-v7";
                reg = <0>;
                resets = <&cpurst 0>;
        };

        cpu@1 {
                device_type = "cpu";
                compatible = "marvell,sheeva-v7";
                reg = <1>;
                resets = <&cpurst 1>;
        };
};

cpurst: cpurst@20800 {
        compatible = "marvell,armada-370-cpu-reset";
        reg = <0x20800 0x20>;
        #reset-cells = <1>;
};

pmsu@22000 {
        compatible = "marvell,armada-370-pmsu";
        reg = <0x22000 0x1000>;
};
```

A timing problem (1)

Idea behind stable DT bindings:

Hardware vendors can ship the hardware with a DT describing the platform, and any recent kernel version will work with it, thanks to the DT bindings being a stable ABI.

A timing problem (2)

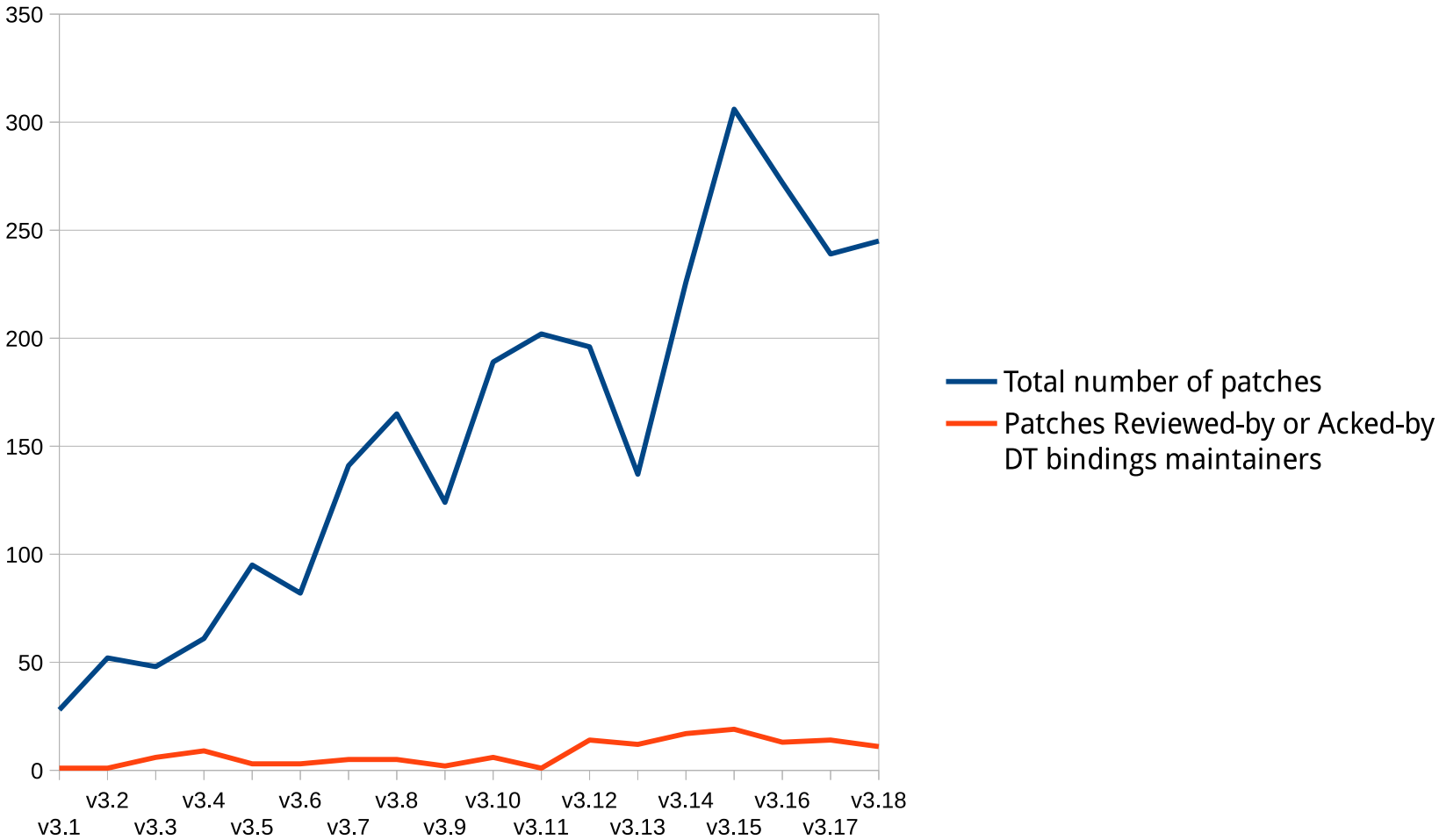


A timing problem (3)

- Do we really want to let hardware vendors define what DT bindings should be?
- Without review from the community?
- Really?
- Or should we speed up the upstreaming process?
- But how is this compatible with taking the time to define proper DT bindings that will be stable?

Enough review?

- Stability of the system call ABI is achieved by careful review of the proposed changes.
- What amount of review do we have for DT bindings?



Testing?

- Maintaining backward compatibility requires a **lot** of code in device drivers to handle old DT bindings.
- No, not just a simple fall back to a default behavior when property *foo* is not here.
- Who will test this code?
- Most likely nobody, it will be dead, untested, unmaintained code.
- At a time where `devm_*` functions are generalized to avoid mistakes in untested error handling, we're adding huge amounts of backward compatibility code.
- Is that really reasonable?

Usefulness

- For Linux distros?
  - Fedora is installing DTBs in `/boot/dtb-<kernel-version>/`
  - E.g: `/boot/dtb-3.11.10-301.fc20.armv7hl/vexpress-v2p-ca9.dtb`
- For hardware vendors?
  - From `Documentation/arm/Atmel/README`: *Device Tree files and Device Tree bindings that apply to AT91 SoCs and boards are considered as "Unstable". To be completely clear, any at91 binding can change at any time. So, be sure to use a Device Tree Binary and a Kernel Image generated from the same source tree.*
  - Atmel is one of the most well supported platform in mainline, with excellent fully public datasheets, and slow evolution of the SoC family.
  - And still they want their bindings to not be stable.

Comparing to the system call ABI

- Kernel developers already consider maintaining the system call backward compatibility to be a difficult exercise.
- Lot of review and discussion before introducing a new system call.
- Lots of testing naturally happening, since mixing userspace / kernel versions is the default way of using Linux.
- The *surface* of the system call ABI is much more narrow, and not directly tied to constantly changing hardware details.

Reality

"So embedded people are going to ship with unfinished DT and upgrade later. They have to. There is no choice. Stable DT doesn't change anything unless you can create perfect stable bindings for a new SOC instantaneously."

— Jason Gunthorpe

Conclusion

- Maintaining Device Tree backward compatibility
  - has a high cost in maintenance and testing effort
  - prevents refactoring code to use new generic kernel frameworks
  - is not used by Linux distributions, not wanted by hardware vendors
  - has no chance to work due to the timing of product development vs. speed of upstreaming
- So: should we really care?

thomas.petazzoni@free-electrons.com

http://free-electrons.com/pub/conferences/2015/elc/petazzoni-dt-as-stable-abi-fairy-tale/