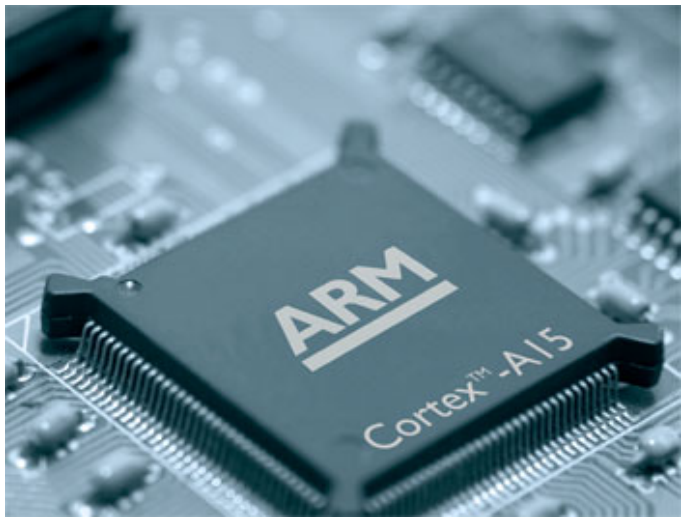
# Your new ARM SoC Linux support check-list!




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

# Thomas Petazzoni

- Embedded Linux engineer and trainer at Free Electrons since 2008
  - 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://www.free-electrons.com`
- Contributing the **kernel support for the new Armada 370 and Armada XP** ARM SoCs from Marvell, under contract with Marvell.
- Major contributor to **Buildroot**, an open-source, simple and fast embedded Linux build system
- Living in **Toulouse**, south west of France

# Background

- March 2011, Linus Torvalds, *Gaah. Guys, this whole ARM thing is a f*cking pain in the ass*
- More and more ARM SoCs have apperead and are appearing
- The historical maintainer, Russell King, through which all ARM code was initially going got **overflowed by the amount of code** to review
- Code started to flow from sub-architectures maintainers directly to Linus
- Focus of each sub-architecture teams on **their own problems**, no vision of the other sub-architectures.
- Consequences: lot of **code duplication**, **missing common infrastructures**, maintenability problems, etc.

# Why?

- In order to solve these problems, the ARM Linux kernel community has done major changes since the last 2 years
  - In the **maintenance process**
  - In the **code infrastructure and subsystems**
- **Not necessarily easy to follow** all those changes and know what are the current best practices
- This talk is an **attempt** to **summarize some of the most important changes**, and **provide guidelines** for developers willing to add support for new ARM SoCs in the mainline Linux kernel
  - but it might be useful for people porting Linux on new boards as well

# Know who are the maintainers

- **Russell King** is now the maintainer for the core ARM support (i.e anything in `arch/arm` except the SoC specific code in `arch/arm/mach-<foo>` and `arch/arm/plat-<bar>`).
- **Arnd Bergmann and Olof Johansson** are the *arm-soc* maintainers. All the ARM SoC code must go through them. They ensure a coherence between how the various SoC families handle similar problems.
- Also need to interact with the subsystem maintainers
  - `drivers/clocksource`, `drivers/irqchip`, **Thomas Gleixner**
  - `drivers/pinctrl`, **Linus Walleij**, **Stephen Warren**
  - `drivers/gpio`, **Grant Likely**, **Linus Walleij**
  - `drivers/clk`, **Mike Turquette**
- Primary mailing list: `linux-arm-kernel@lists.infradead.org`

# Where is the code, when to submit?

- The **arm-soc** Git tree is at `https://git.kernel.org/?p=linux/kernel/git/arm/arm-soc.git;a=summary`
- Watch the **for-next** branch that contains what will be submitted by the ARM SoC maintainers during the next merge window.
- Generally, the ARM SoC maintainers want to have integrated all the code from the different ARM sub-architectures a few (two?) weeks before Linus opens the merge window.
- If you submit your code *during* the Linus merge window, there is no way it will get integrated at this point: it will have to wait for the next merge window.
- Usual Linux contribution guidelines apply: people will make comments on your code, take them into account, repost. Find the good balance between **patience** and **perseverance**.

# Existing code?

- You have existing Linux kernel code to support your SoC?
- There are **99% chances that you should throw it away completely**
  - Most of the time, SoC support code written by SoC vendors do not comply with the Linux coding rules, the Linux infrastructures, has major design issues, is ugly, etc.
  - With the recent major changes in the way to support ARM SoC, all existing code has become **basically irrelevant**.
- Of course, existing code useful as a reference to know how the hardware works. But the code to be submitted should most likely be **written from scratch**.

# Step 1: start minimal

Device Tree
(SoC and board)

`arch/arm/boot/dts/`

Basic
"initialization"
C file
+
basic header
files

`arch/arm/mach-<foo>/`

Timer
driver

`drivers/clocksource/`

IRQ controller
driver

`drivers/irqchip/`

earlyprintk
support

`arch/arm/include/debug`

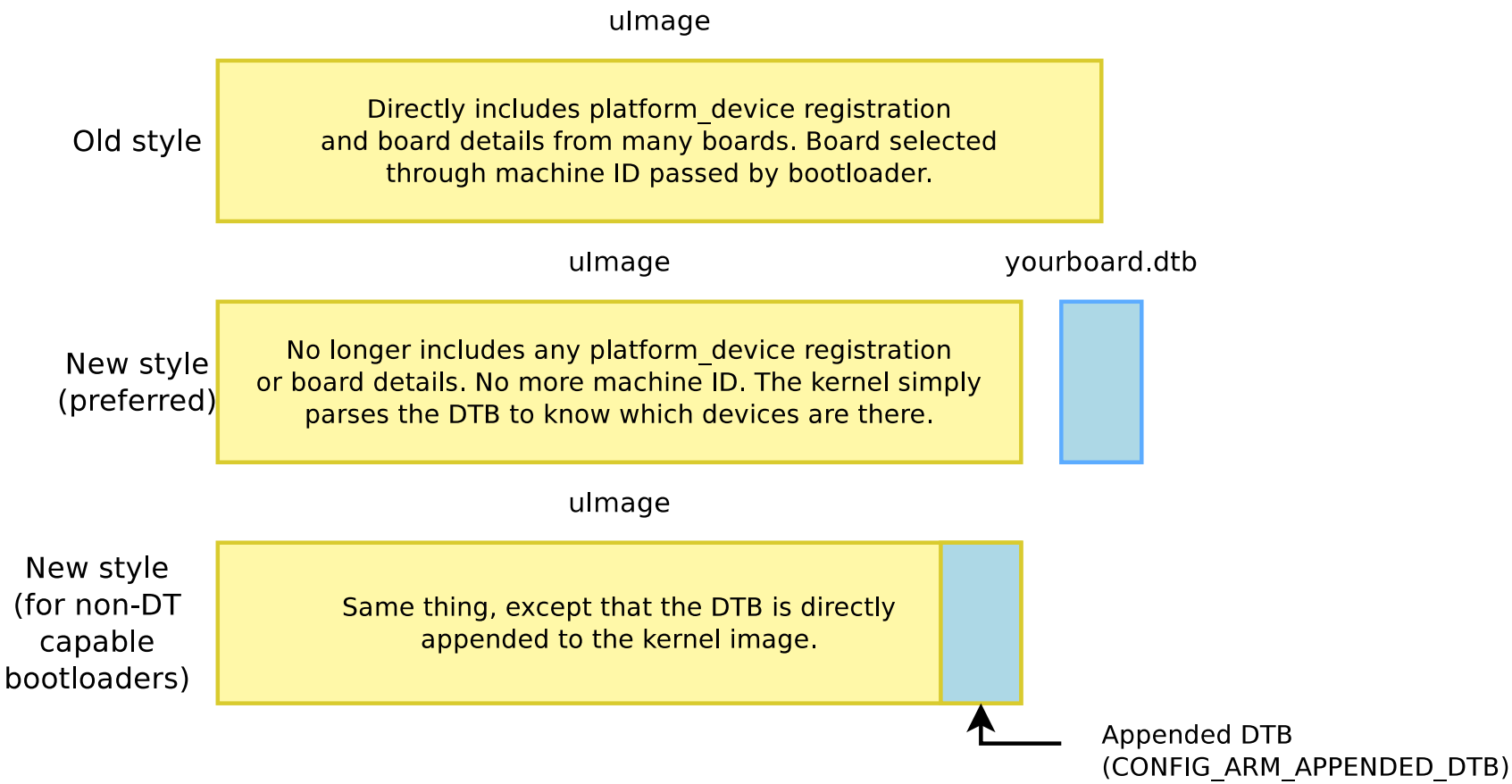
Serial port
driver

`drivers/tty/serial/`

# Device Tree

- The purpose of the Device Tree is to move a significant part of the **hardware description** into a data structure that is no longer part of the kernel binary itself.
- This data structure, the **Device Tree Source** is compiled into a binary **Device Tree Blob**
- The **Device Tree Blob** is loaded into memory by the bootloader, and passed to the kernel.
- It replaces all the `board-*.c` files, and removes all the manual registration of `platform_device`. Also, no longer needed to have `Kconfig` options for each board.
- Usage of the Device Tree is **mandatory** for all new ARM SoCs. No way around it.

# Boot process with a Device Tree Blob



# Writing your Device Tree

- Add one `<soc>.dtsi` file in `arch/arm/boot/dts/` that describes the devices in your SoC
  - You can also have multiple `<soc>.dtsi` files including each other with the `/include/` directive, if you have a SoC family with multiple SoCs having common things, but also specific things.
- Add one `<board>.dts` file in `arch/arm/boot/dts/` for each of the board you support. It shoud `/include/` the appropriate `.dtsi` file.
- Add a `dtb-$(CONFIG_ARCH_<yourarch>)` line in `arch/arm/boot/dts/Makefile` for all your board `.dts` files so that all the `.dtb`s are automatically built.

# Example on Armada 370/XP support

- armada-370-xp.dtsi
  - armada-370.dtsi
    - Board armada-370-db.dts
  - armada-xp.dtsi
    - armada-xp-mv78230.dtsi
    - armada-xp-mv78260.dtsi
      - Board openblocks-ax3-4.dts
    - armada-xp-mv78460.dtsi
      - Board armada-xp-db.dts

# bcm2835.dtsi

```
/include/ "skeleton.dtsi"
/ {
        compatible = "brcm,bcm2835";
        model = "BCM2835";
        interrupt-parent = <&intc>;

        chosen {
                bootargs = "earlyprintk console=ttyAMA0";
        };

        soc {
                compatible = "simple-bus";
                #address-cells = <1>;
                #size-cells = <1>;
                ranges = <0x7e000000 0x20000000 0x02000000>;

                [...]
                intc: interrupt-controller {
                        compatible = "brcm,bcm2835-armctrl-ic";
                        reg = <0x7e00b200 0x200>;
                        interrupt-controller;
                        #interrupt-cells = <2>;
                };
                uart@20201000 {
                        compatible = "brcm,bcm2835-pl011", "arm,pl011", "arm,primecell";
                        reg = <0x7e201000 0x1000>;
                        interrupts = <2 25>;
                        clock-frequency = <3000000>;
                        status = "disabled";
                };
        };
};
```

# bcm2835-rpi-b.dts

```
/dts-v1/;
/memreserve/ 0x0c000000 0x04000000;
/include/ "bcm2835.dtsi"

/ {
        compatible = "raspberrypi,model-b", "brcm,bcm2835";
        model = "Raspberry Pi Model B";

        memory {
                reg = <0 0x10000000>;
        };
        soc {
                uart@20201000 {
                      status = "okay";
                };
        };
};
```

# Basic things in arch/arm/mach-<yourarch>

- Kconfig, describing your ARCH_<yourarch> option, as well as sub-options for each SoC if your family has multiple SoC. No per-board options: your kernel image is independent from the board details.
- Makefile, compiling just one file, <yoursoc>.c
- <yoursoc>.c, that contains a DT_MACHINE_START definition, either per SoC or per family of SoC
- <yoursoc>.h, that contains a minimal set of constants used to implement a static mapping for the registers used to access the UARTs. No constants with addresses or IRQ numbers for any other devices: they will be provided through the Device Tree.

# Multi-SoC, multi-platform kernel

- Originally a given Linux kernel image could only support multiple boards using a given SoC, boards being “detected” thanks to the machine ID.
- Now, a given `mach-foo` directory must allow the support for all the SoCs it supports to be compiled into a single kernel image. No more `#ifdef` conditionals depending on SoCs, it must all be runtime detected.
- And further than that, now all new sub-architectures must be compatible with the `CONFIG_ARCH_MULTIPLATFORM` mechanism, which allows the support for all SoCs to be built into a single binary kernel image.

# arch/arm/mach-<soc>/<soc>.c (1)

```
static struct map_desc mysoc_io_desc[] __initdata = {
        {
                .virtual        = (unsigned long) MYSOC_REGS_VIRT_BASE,
                .pfn            = __phys_to_pfn(MYSOC_REGS_PHYS_BASE),
                .length         = MYSOC_XP_REGS_SIZE,
                .type           = MT_DEVICE,
        },
};

void __init mysoc_map_io(void)
{
        iotable_init(mysoc_io_desc, ARRAY_SIZE(mysoc_io_desc));
}

struct sys_timer mysoc_timer = {
        .init           = mysoc_timer_init,
};
```

# arch/arm/mach-<soc>/<soc>.c (2)

```
static void __init mysoc_dt_init(void)
{
        of_platform_populate(NULL, of_default_bus_match_table,
                             NULL, NULL);
}

static const char * const mysoc_dt_compat[] = {
        "vendor,soc-model1",
        "vendor,soc-model2",
        NULL,
};

DT_MACHINE_START(MYSOC_DT, "Company Wonderful SoC (Device Tree)")
        .init_machine   = mysoc_dt_init,
        .map_io         = mysoc_map_io,
        .init_irq       = irqchip_init,
        .timer          = &mysoc_timer,
        .restart        = mysoc_restart,
        .dt_compat      = mysoc_dt_compat,
MACHINE_END
```

# arch/arm/mach-<soc>/<soc>.h

```
#ifndef __MYSOC_H
#define __MYSOC_H

#define MYSOC_REGS_PHYS_BASE    0xd0000000
#define MYSOC_REGS_VIRT_BASE    IOMEM(0xfeb00000)
#define MYSOC_REGS_SIZE         SZ_1M

#endif /* __MYSOC_H */
```

# arch/arm/mach-<soc>/Kconfig

```
config ARCH_MYSOC
        bool "Wonderful SoC" if ARCH_MULTI_V7
        select CLKSRC_MMIO
        select COMMON_CLK
        select GENERIC_CLOCKEVENTS
        select GENERIC_IRQ_CHIP
        select IRQ_DOMAIN
        select MULTI_IRQ_HANDLER
        select PINCTRL
        select SPARSE_IRQ
```

# Earlyprintk support

- The first thing to have is obviously the *earlyprintk* support, to get early messages from the kernel.
- In `arch/arm/Kconfig.debug`, in the choice of `DEBUG_LL` UARTs, add an entry for your platform, and add an entry to `CONFIG_DEBUG_LL_INCLUDE` that references `arch/arm/include/debug/<yoursoc>.S`
- In `arch/arm/include/debug/`, implement the `addruart`, `senduart`, `waituart` and `busyuart` assembly macros. On many platforms (using 8250 compatible or the PL011 serial IP), existing code can be re-used and only `addruart` needs to be implemented.

# IRQ controller support

- If your platform uses the GIC or VIC interrupt controllers, there are already drivers in `arch/arm/common`
- Otherwise, implement a new one in `drivers/irqchip/` (rather new rule, most of them still leave in the `arch/arm/mach-<foo>` directory).
- It must support the `SPARSE_IRQ` and `irqdomain` mechanisms: no more fixed number of IRQs `NR_IRQS`: an *IRQ domain* is dynamically allocated for each interrupt controller.
- It must support the `MULTI_IRQ_HANDLER` mechanism, where your `DT_MACHINE_START` structure references the *IRQ controller handler* through its `->handle_irq()` field.
- In your `DT_MACHINE_START` structure, also call the initialization function of your IRQ controller driver using the `->init_irq()` field.
- Instantiated from your Device Tree `.dtsi` file.

# Timer driver

- Should be implemented in `drivers/clocksource`
- It must register
  - A *clocksource* device, which using a free-running timer, provides a way for the kernel to keep track of time passing. See `clocksource_mmio_init()` if your timer value can be read from a simple memory-mapped register, or `clocksource_register_hz()` for a more generic solution.
  - A *clockevents* device, which allows the kernel to program a timer for one-shot or periodic events notified by an interrupt. See `clockevents_config_and_register()`
- Driver must have a Device Tree binding, and the device be instantiated from your Device Tree `.dtsi` file.

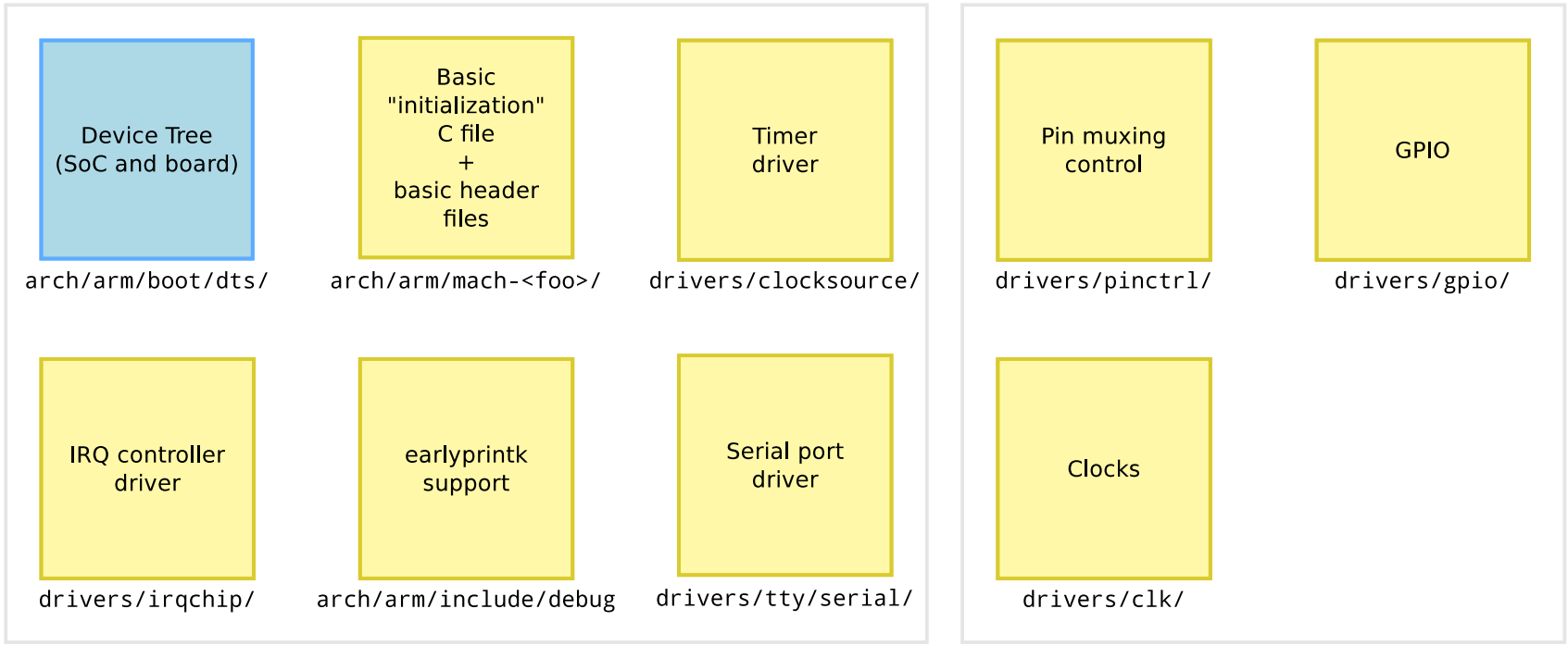
# Serial port driver

- These days, many ARM SoC use either a 8250-compatible UART, or the PL011 UART controller from ARM. In both cases, Linux already has a driver
  - Just need to instantiate devices in your `.dtsi` file, and mark those that are available on a particular board with `status = "okay"` in the `.dts` file.
- If you have a custom UART controller, then get ready for more fun. You'll have to write a complete driver in `drivers/tty/serial`
  - A platform driver, with Device Tree binding, integrated with the *uart* and *console* subsystems
  - Maintainer is Alan Cox, also Cc Greg Kroah-Hartmann who looks after the overall TTY layer.

# End of step 1

- At this point, your system should boot all the way to a shell
- You don't have any storage device driver for now, but you can boot into a minimal root filesystem embedded inside an *initramfs*
- Time to submit your basic ARM SoC support. Don't wait to have all the drivers and all the features: submit something minimal **as soon as possible**.

# Step 2: more core infrastructure



# The clock framework

- A proper *clock framework* has been added in kernel 3.4, released in May 2012
- Initially from Jeremy Kerr (Canonical), finally implemented and merged by Mike Turquette (Texas Instruments)
- This framework:
  - Implements the `clk_get`, `clk_put`, `clk_prepare`, `clk_unprepare`, `clk_enable`, `clk_disable`, `clk_get_rate`, etc. **API for usage by device drivers**
  - Implements **some basic clock drivers** (fixed rate, gatable, divider, fixed factor, etc.) and allows the implementation of **custom clock drivers** using `struct clk_hw` and `struct clk_ops`
  - Allows to declare the available clocks and their association to devices in the Device Tree (preferred) or statically in the source code (old method)
  - Provides a *debugfs* representation of the clock tree
  - Is implemented in `drivers/clk`
  - See `Documentation/clk.txt`

# Clock framework, the driver side

From `drivers/serial/tty/amba-pl011.c`.

```
pl011_startup()
{
        [...]
        clk_prepare_enable(uap->clk);
        uap->port.uartclk = clk_get_rate(uap->clk);
        [...]
}

pl011_shutdown()
{
        [...]
        clk_disable_unprepare(uap->clk);
}

pl011_probe()
{
        [...]
        uap->clk = clk_get(&dev->dev, NULL);
        [...]
}

pl011_remove()
{
        [...]
        clk_put(uap->clk);
        [...]
}
```

# Clock framework, declaration of clocks in DT

## From arch/arm/boot/dts/highbank.dts

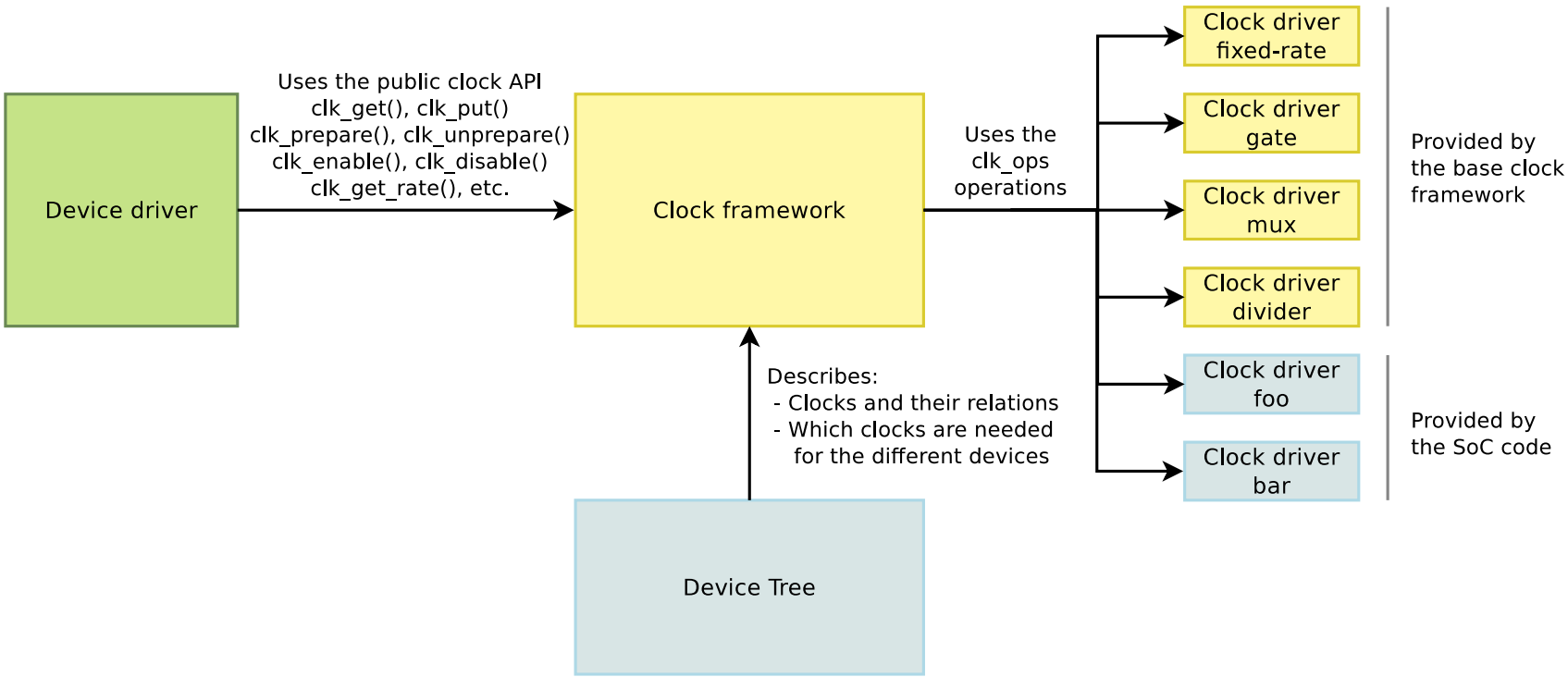
```
clocks {
        #address-cells = <1>;
        #size-cells = <0>;

        osc: oscillator {
                #clock-cells = <0>;
                compatible = "fixed-clock";
                clock-frequency = <33333000>;
        };

        [...]

        emmcpll: emmcpll {
                #clock-cells = <0>;
                compatible = "calxeda,hb-pll-clock";
                clocks = <&osc>;
                reg = <0x10C>;
        };

        [...]

        pclk: pclk {
                #clock-cells = <0>;
                compatible = "fixed-clock";
                clock-frequency = <150000000>;
        };
};
```

# Clock framework, devices referencing their clocks

From arch/arm/boot/dts/highbank.dts

```
[...]
serial@fff36000 {
        compatible = "arm,pl011", "arm,primecell";
        reg = <0xfff36000 0x1000>;
        interrupts = <0 20 4>;
        clocks = <&pclk>;
        clock-names = "apb_pclk";
};
[...]
```

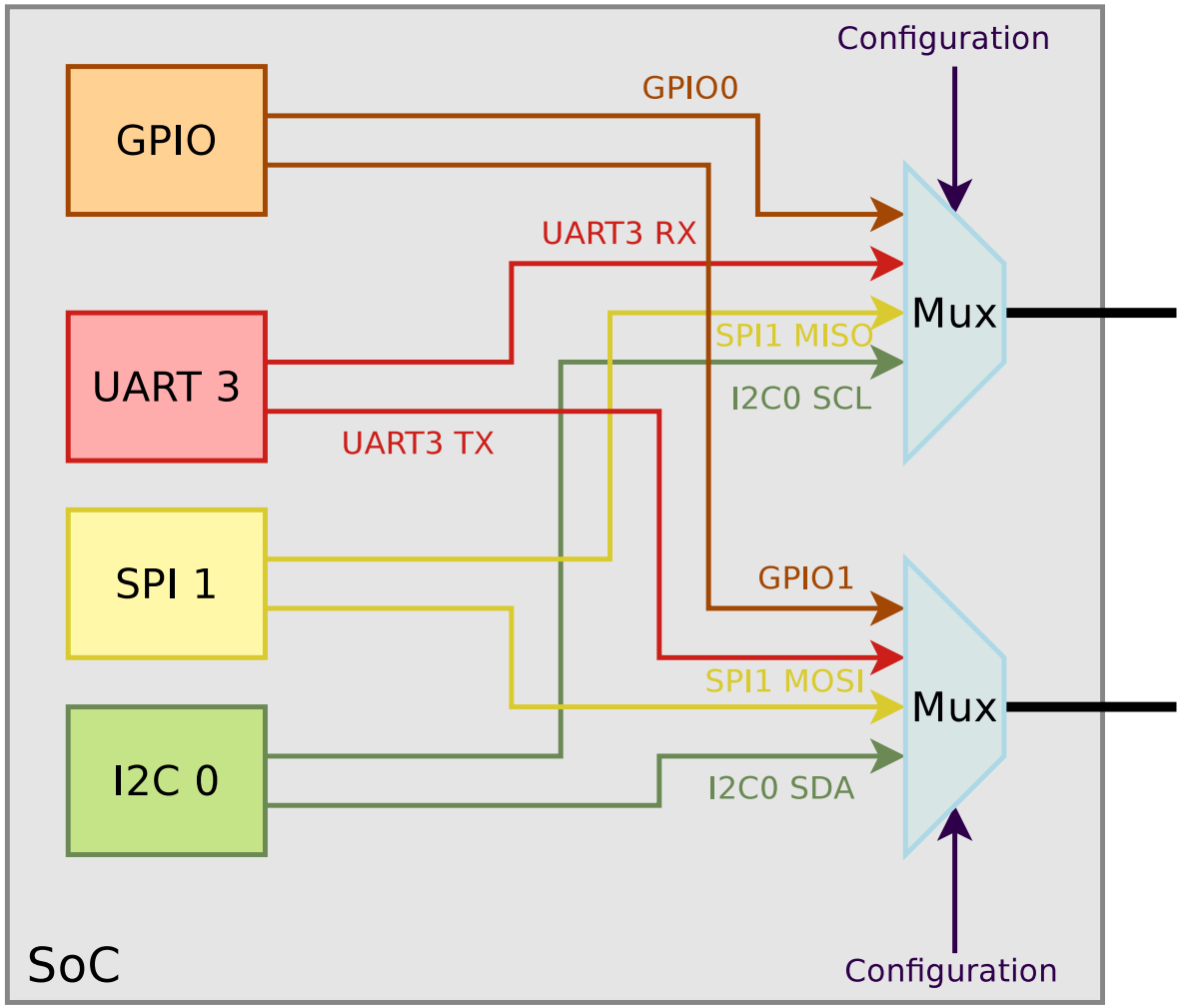
# Clock framework: summary



# Introduction to pin muxing

- SoCs integrate many more peripherals than the number of available pins allows to expose.
- Many of those pins are therefore **multiplexed**: they can either be used as function A, *or* function B, *or* function C, *or* a GPIO
- Example of functions are:
  - parallel LCD lines
  - SDA/SCL lines for I2C busses
  - MISO/MOSI/CLK lines for SPI
  - RX/TX/CTS/DTS lines for UARTs
- This muxing is **software-configurable**, and depends on **how the SoC is used on each particular board**

# Pin muxing: principle



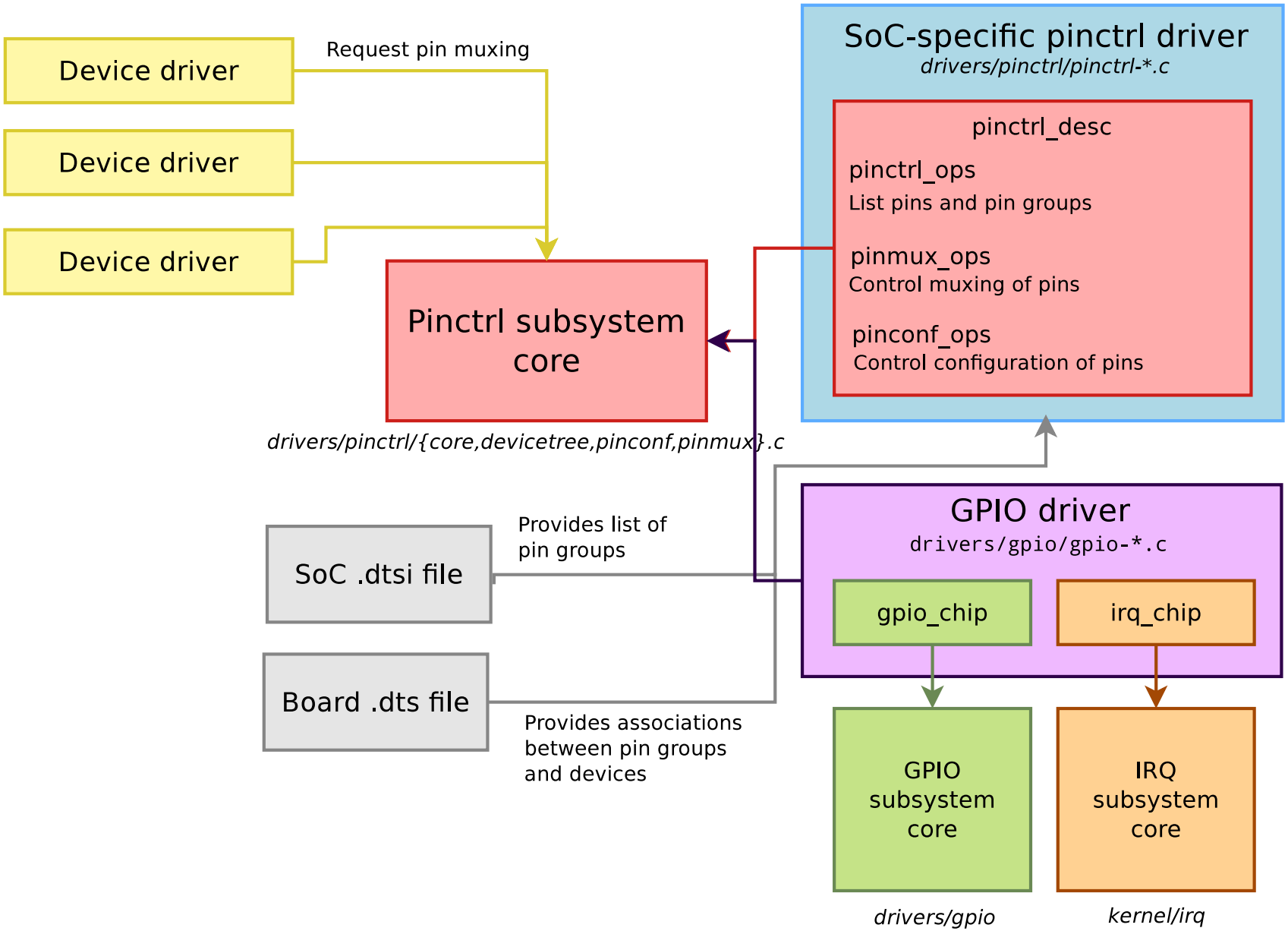
# The old pin-muxing code

- Each ARM sub-architecture had its own pin-muxing code
- The API was specific to each sub-architecture
- Lot of similar functionality implemented in different ways
- The pin-muxing had to be done at the SoC level, and couldn't be requested by device drivers

# The new pin-muxing subsystem

- The new **pinctrl** subsystem aims at solving those problems
- Mainly developed and maintained by Linus Walleij, from Linaro/ST-Ericsson
- Implemented in `drivers/pinctrl`
- Provides:
  - An API to register *pinctrl driver*, i.e entities knowing the list of pins, their functions, and how to configure them. Used by SoC-specific drivers to expose pin-muxing capabilities.
  - An API for *device drivers* to request the muxing of a certain set of pins
  - An interaction with the *GPIO* framework

# The new pin-muxing subsystem: diagram



# Declaring pin groups in the SoC dtsi

- From arch/arm/boot/dts/imx28.dtsi
- Declares the *pinctrl* device and various pin groups

```
pinctrl@80018000 {
    #address-cells = <1>;
    #size-cells = <0>;
    compatible = "fsl,imx28-pinctrl", "simple-bus";
    reg = <0x80018000 2000>;

    duart_pins_a: duart@0 {
            reg = <0>;
            fsl,pinmux-ids = <0x3102 0x3112>;
            fsl,drive-strength = <0>;
            fsl,voltage = <1>;
            fsl,pull-up = <0>;
    };
    duart_pins_b: duart@1 {
            reg = <1>;
            fsl,pinmux-ids = <0x3022 0x3032>;
            fsl,drive-strength = <0>;
            fsl,voltage = <1>;
            fsl,pull-up = <0>;
    };
    mmc0_8bit_pins_a: mmc0-8bit@0 {
            reg = <0>;
            fsl,pinmux-ids = <0x2000 0x2010 0x2020
                    0x2030 0x2040 0x2050 0x2060
                    0x2070 0x2080 0x2090 0x20a0>;
            fsl,drive-strength = <1>;
            fsl,voltage = <1>;
            fsl,pull-up = <1>;
    };
    mmc0_4bit_pins_a: mmc0-4bit@0 {
            reg = <0>;
            fsl,pinmux-ids = <0x2000 0x2010 0x2020
                    0x2030 0x2080 0x2090 0x20a0>;
            fsl,drive-strength = <1>;
            fsl,voltage = <1>;
            fsl,pull-up = <1>;
    };
    mmc0_cd_cfg: mmc0-cd-cfg {
            fsl,pinmux-ids = <0x2090>;
            fsl,pull-up = <0>;
    };
    mmc0_sck_cfg: mmc0-sck-cfg {
            fsl,pinmux-ids = <0x20a0>;
            fsl,drive-strength = <2>;
            fsl,pull-up = <0>;
    };
};
```

# Associating devices with pin groups, board dts

- From arch/arm/boot/dts/cfa10036.dts

```
apb@80000000 {
  apbh@80000000 {
     ssp0: ssp@80010000 {
         compatible = "fsl,imx28-mmc";
         pinctrl-names = "default";
         pinctrl-0 = <&mmc0_4bit_pins_a
                      &mmc0_cd_cfg &mmc0_sck_cfg>;
         bus-width = <4>;
         status = "okay";
     };
  };

  apbx@80040000 {
     duart: serial@80074000 {
         pinctrl-names = "default";
         pinctrl-0 = <&duart_pins_b>;
         status = "okay";
     };
  };
};
```

# Device drivers requesting pin muxing

- From `drivers/mmc/host/mxs-mmc.c`

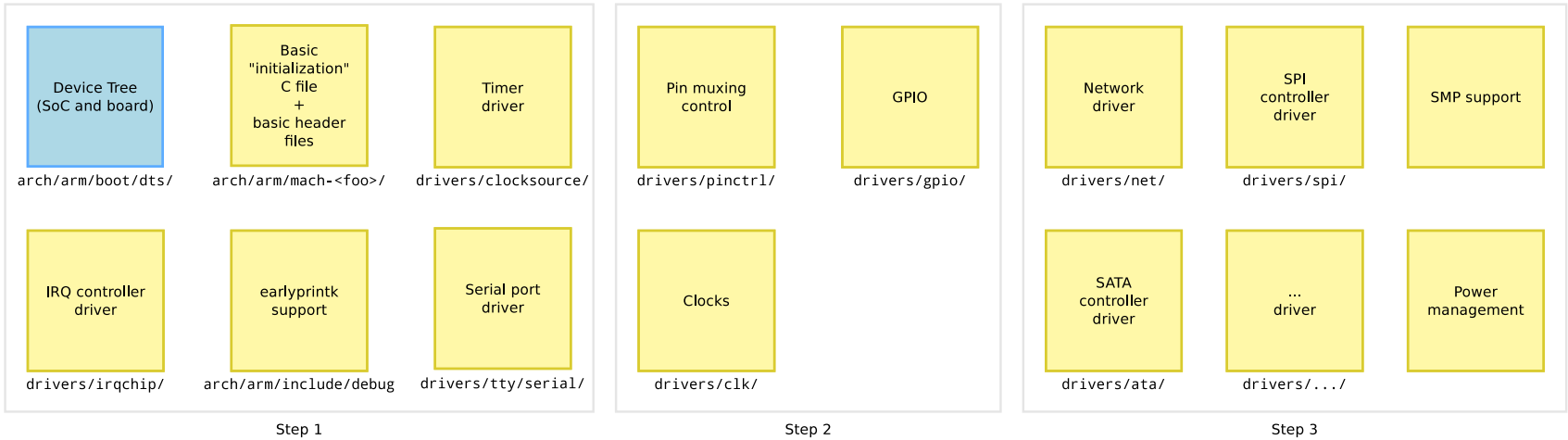
```
static int mxs_mmc_probe(struct platform_device *pdev)
{
    [...]
    pinctrl = devm_pinctrl_get_select_default(&pdev->dev);
    if (IS_ERR(pinctrl)) {
        ret = PTR_ERR(pinctrl);
        goto out_mmc_free;
    }
    [...]
}
```

# GPIO

The GPIO subsystem has been around for quite some time now, only a few things have evolved recently:

- All GPIO drivers, including drivers for GPIO controllers internal to the SoC must be in `drivers/gpio`
- If the GPIO pins are muxed, the driver must interact with the *pinctrl* subsystem to get the proper muxing: `pinctrl_request_gpio()` and `pinctrl_free_gpio()`.

# Step 3: more drivers, advanced features



# Device drivers

On the device driver side, not much has changed, except:

- Each device driver must have a **device tree binding**
  - A *binding* describes the *compatible* string and the properties that a DT node instantiating the device must carry.
  - The binding must be documented in `Documentation/devicetree/bindings`
- The drivers are no longer allows to include `<mach/something.h>`, due to multiplatform kernel support.

# Finding good examples

As things are moving quickly, not all ARM sub-architecture comply with the current best practices. The following recent ones are good starting points:

- `arch/arm/mach-highbank`
- `arch/arm/mach-socfpga`
- `arch/arm/mach-bcm2835`
- `arch/arm/mach-mxs`
- `arch/arm/mach-mvebu`

# Conclusion

- The code in the `arch/arm` tree has changed significantly over the last two years
- These changes allow the usage of more generic and common infrastructures and less SoC-specific code, which can only be a good thing
- However, the best practices are quickly evolving: requires a constant reading of the Linux ARM Kernel mailing-list discussion to stay aware of most recent changes.

# Questions?

Thomas Petazzoni

thomas.petazzoni@free-electrons.com

Thanks to Grégory Clement (Free Electrons, working with me on Marvell mainlining), Lior Amsalem and Maen Suleiman (Marvell)