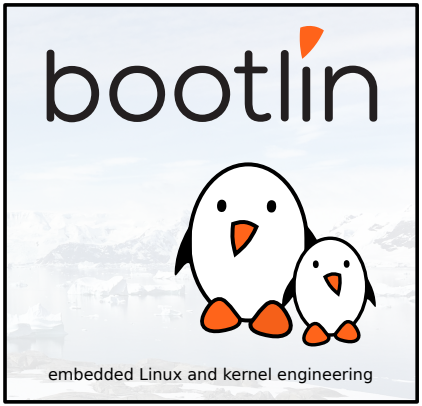
Your newer ARM64 SoC Linux check list!

Gregory CLEMENT
gregory@bootlin.com

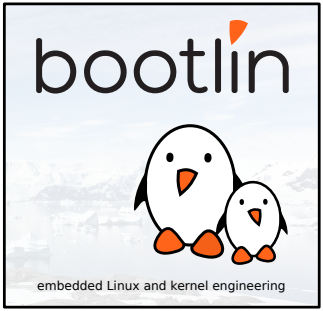


Your newer ARM64 SoC Linux check list!

What is new in the ARM linux kernel since our last check list ?

Gregory CLEMENT

- Embedded Linux engineer and trainer at **Bootlin**
  - Embedded Linux **expertise**
  - **Development**, consulting and training
  - Strong open-source focus
- Open-source contributor
  - Contributing to **kernel support** for the Armada 370, 375, 38x, 39x and Armada XP ARM SoCs and **Armada 3700 ARM64 SoCs** from Marvell.
  - Co-maintainer of mvebu sub-architecture (SoCs from Marvell Engineering Business Unit)
  - Living near **Lyon**, France



Background

- December 2012, initial support for ARM 64-bits merged in the kernel 3.7.
- No real hardware at this stage.
- Support for first `arm64` SoC in 3.11 (June 2013).
- Support for second `arm64` SoC in 3.18 (December 2014 through arm-soc).

```
commit 81f56e5375e84689b891e0e6c5a02ec12a1f18d9
Merge: 6c09931b3f98 27aa55c5e512
Author: Linus Torvalds <torvalds@linux-foundation.org>
Date:   Mon Oct 1 11:51:57 2012 -0700

    Pull arm64 support from Catalin Marinas:
     "Linux support for the 64-bit ARM architecture (AArch64)

      Features currently supported:
       - 39-bit address space for user and kernel (each)
       - 4KB and 64KB page configurations
       - Compat (32-bit) user applications (ARMv7, EABI only)
       - Flattened Device Tree (mandated for all AArch64 platforms)
       - ARM generic timers"
```

Why?

- New architecture usually comes with new requirements and a new development process.
- However `arm64` is still an ARM architecture.
- This talk is an **attempt** to **summarize some of the differences and similarities with ARM 32-bits**, and **provide guidelines** for developers willing to add support for new ARM 64-bits SoCs in the mainline Linux kernel.
  - but it might be useful for people porting Linux on new boards as well.
- Part of the talk will be based on Thomas Petazzoni talk at ELC 2013: "Your new ARM SoC Linux support check-list!" but focusing on `arm64` and updated for the common part.

ARM 32-bits vs ARM 64-bits

- ARM 32-bits in kernel: from ARMv4 to ARMv7.
- ARM 64-bits: one mode of the ARMv8.
- 64-bits mode of ARMv8 is called AARCH64 (gcc).
- ARM64 comes with virtualization instructions and other improvements.
- New Linux kernel architecture not merged with ARM 32-bits:
  - Assembly code is different.
  - System call interface: uses a new ABI.
  - Platform support already moved in the drivers.
  - Basic infrastructure should move too.

Know who are the maintainers

- **Catalin Marinas** and **Will Deacon** are the maintainers for the core ARM 64 support.
- **Arnd Bergmann** and **Olof Johansson** are the *arm-soc* maintainers. All the `arm64` (and still `arm`) SoC code must go through them. They ensure consistency between how the various SoC families handle similar problems.
- Also need to interact with the subsystem maintainers:
  - `drivers/clocksource`, **Daniel Lezcano**, **Thomas Gleixner**.
  - `drivers/irqchip`, **Thomas Gleixner**, **Jason Cooper**, **Marc Zyngier**.
  - `drivers/pinctrl`, **Linus Walleij**.
  - `drivers/gpio`, **Linus Walleij**, **Alexandre Courbot**.
  - `drivers/clk`, **Stephen Boyd**, **Mike Turquette**.
- Primary mailing list: `linux-arm-kernel@lists.infradead.org`

Where is the code, when to submit?

No change since last talk for `arm`.

- 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 **50% chances that you should throw it away completely**.
  - SoC support code written by SoC vendors used to not comply with the Linux coding rules, the Linux infrastructures, to have major design issues, to be ugly, etc.
  - `arm64` is still pretty new and SoC vendors didn't have time to be too creative.
  - For `arm` some SoC vendors have reduced the gap between their internal tree and the mainline kernel.
  - However some of them still rely on sources based on the old fashion kernel.
- Of course, existing code is useful as a reference to know how the hardware works. But the code to be submitted should still often be **written from scratch**.

Step 1: start minimal

Device Tree
(SoC and board)

`arch/arm64/boot/dts/<vendors>/`

Timer
driver

`drivers/clocksource/`

IRQ controller
driver

`drivers/irqchip/`

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.
- Usage of the Device Tree is **mandatory** for all new ARM SoCs. No way around it.
- For `arm64` it is the only part of the port which goes under the `arch/arm64` directory.

Writing your Device Tree

- Add one `<soc>.dtsi` file in `arch/arm64/boot/dts/<vendor>/` 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 an SoC family with multiple SoCs having common things, but also specific things.
  - The `.dtsi` files are also used to keep big amount of data like the pinctrl.
- Add one `<board>.dts` file in `arch/arm64/boot/dts/<vendor>/` for each of the boards you support. It should `/include/` the appropriate `.dtsi` file.
- Add a `dtb-$(CONFIG_ARCH_<yourarch>)` line in `arch/arm64/boot/dts/<vendor>/Makefile` for all your board `.dts` files so that all the `.dtb`s are automatically built.
- The main difference with `arch/arm`, is the use of a directory by vendor. For ARM 32-bits, all the device tree files are located in `arch/arm/boot/dts/`.

Example on Armada 3700 support

`armada-37xx.dtsi`

- `armada-371x.dtsi`
- `armada-372x.dtsi`
  - Board `armada-3720-db.dts`
  - Board `armada-3720-espressobin.dts`

vulcan.dtsi

```
#include <dt-bindings/interrupt-controller/arm-gic.h>

/ {
    model = "Broadcom Vulcan";
    compatible = "brcm,vulcan-soc";
    interrupt-parent = <&gic>;
    #address-cells = <2>;
    #size-cells = <2>;

    /* just 4 cpus now, 128 needed in full config */
    cpus {
        #address-cells = <0x2>;
        #size-cells = <0x0>;

        cpu@0 {
            device_type = "cpu";
            compatible = "brcm,vulcan", "arm,armv8";
            reg = <0x0 0x0>;
            enable-method = "psci";
        };
[...]
    };

    psci {
        compatible = "arm,psci-0.2";
        method = "smc";
    };
```

```
    gic: interrupt-controller@400080000 {
        compatible = "arm,gic-v3";
        #interrupt-cells = <3>;
        #address-cells = <2>;
        #size-cells = <2>;
[...]
    };

    timer {
        compatible = "arm,armv8-timer";
        interrupts = <GIC_PPI 13 IRQ_TYPE_LEVEL_HIGH>,
                     <GIC_PPI 14 IRQ_TYPE_LEVEL_HIGH>,
                     <GIC_PPI 11 IRQ_TYPE_LEVEL_HIGH>,
                     <GIC_PPI 10 IRQ_TYPE_LEVEL_HIGH>;
    };
[...]
    soc {
        compatible = "simple-bus";
        #address-cells = <2>;
        #size-cells = <2>;
        ranges;

        uart0: serial@402020000 {
            compatible = "arm,pl011", "arm,primecell";
            reg = <0x04 0x02020000 0x0 0x1000>;
            interrupt-parent = <&gic>;
            interrupts = <GIC_SPI 49 IRQ_TYPE_LEVEL_HIGH>;
[...]
        };
    };
};
```

vulcan-eval.dts

```
#include "vulcan.dtsi"

/ {
    model = "Broadcom Vulcan Eval Platform";
    compatible = "brcm,vulcan-eval", "brcm,vulcan-soc";

    memory {
        device_type = "memory";
        reg = <0x00000000 0x80000000 0x0 0x80000000>,  /* 2G @ 2G  */
              <0x00000008 0x80000000 0x0 0x80000000>;  /* 2G @ 34G */
    };

    aliases {
        serial0 = &uart0;
    };

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

No arch/arm64/mach-<yourarch> for arm64

- ARCH_<yourarch> option, as well as sub-options for each SoC is now directly in the arch/arm64/Kconfig.platforms file
- No more C files specific to an SoC family under arch/arm64.
  - The SoC features should be handled by the drivers.
  - If the feature of the SoC can't fit an existing class driver, then the code could go to drivers/soc/.

arch/arm64/Kconfig.platform

```
config ARCH_MYSOC
        bool "Wonderful SoC"
        select CLKSRC_OF
        select GENERIC_IRQ_CHIP
        select GPIOLIB
        select MYSOC_CLK
        select PINCTRL
        select PINCTRL_MYSOC
```

- Less `CONFIG_` symbols that used to be needed for `ARM`.
- Many symbols are now set by default with `ARM64`.

Earlycon support

- The first thing to have is obviously a mean to get early messages from the kernel.
- For `arm`, *earlyprintk* used to be the way to achieve this.
  - Need to setup the UARTs address at SoCs level early during the boot.
  - Usage of *earlyprintk* was not compatible with multiarch kernel.
- For `arm64`, *earlycon* is mandatory.
- At serial driver level.
- Part of `console` support.
- Declared in the driver by using `EARLYCON_DECLARE` and `OF_EARLYCON_DECLARE`

IRQ controller support

- If your platform uses the GIC v2 or v3 interrupt controllers, there are already drivers in `drivers/irqchip`.
- Otherwise, implement a new one at the same location.
- 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 passing time. 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()`
- The driver must have a Device Tree binding, and the device be instantiated from your Device Tree `.dtsi` file.

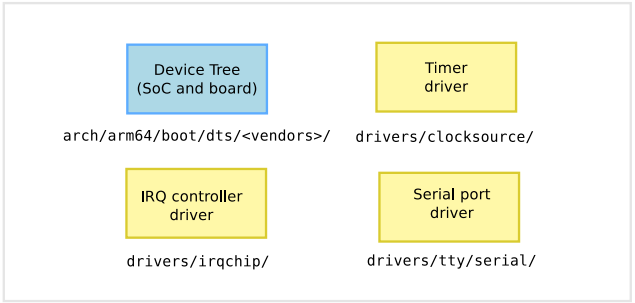
Serial port driver

- These days, many `arm` and `arm64` SoCs 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
  - Do not forget to include the *earlycon* support.
  - The maintainer is Greg Kroah-Hartmann.

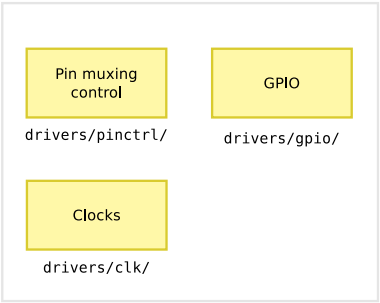
End of step 1

- 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*.
- For `arm64`, the kernel does not decompress itself, so if there is no support in the bootloader too, the kernel image can be pretty large: around 20MB.
- Time to submit your basic `arm64` 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



Step 1



Step 2

The clock framework

- A proper *clock framework* has been added in kernel 3.4, released in May 2012.
- 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.
  - Provides a *debugfs* representation of the clock tree.
  - Is implemented in `drivers/clk`.
  - See `Documentation/clk.txt`.
  - See also http://bootlin.com/pub/conferences/2013/elce/common-clock-framework-how-to-use-it/common-clock-framework-how-to-use-it.pdf
  - The new trend is to only have one node in the device tree which will expose all the clocks that can be consumed.

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);
        [...]
}
```

- The `remove()` part could be avoided by using the managed API: `devm_clk_get()`

Clock framework, declaration of clocks in DT

From `arch/arm/boot/dts/sun6i-a31.dtsi`

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

    osc24M: osc24M {
        #clock-cells = <0>;
        compatible = "fixed-clock";
        clock-frequency = <24000000>;
    };

    osc32k: clk@0 {
        #clock-cells = <0>;
        compatible = "fixed-clock";
        clock-frequency = <32768>;
        clock-output-names = "osc32k";
    };
```

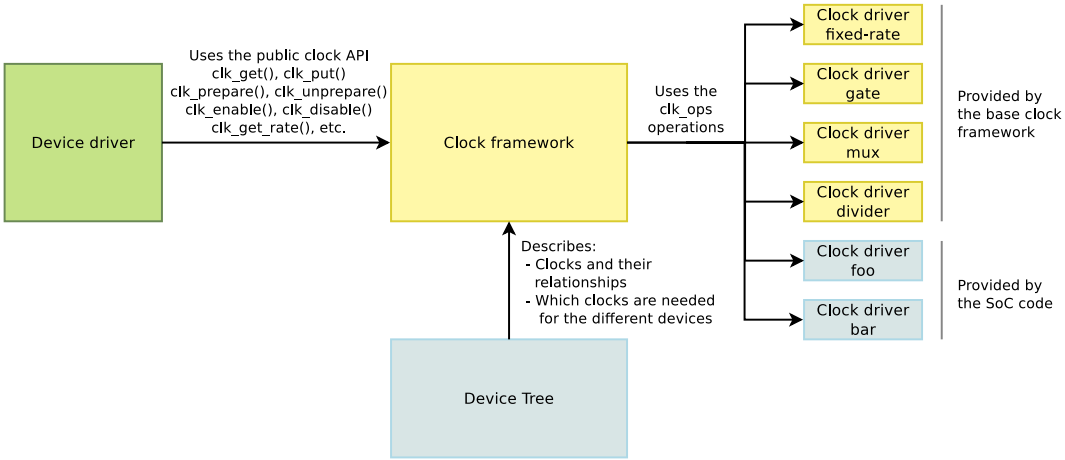
```
[...]
}
soc@01c00000 {
[...]
        ccu: clock@01c20000 {
        compatible = "allwinner,sun6i-a31-ccu";
        reg = <0x01c20000 0x400>;
        clocks = <&osc24M>, <&osc32k>;
        clock-names = "hosc", "losc";
        #clock-cells = <1>;
        #reset-cells = <1>;
    };
[...]
};
```

Clock framework, devices referencing their clocks

From arch/arm/boot/dts/sun6i-a31.dtsi

```
[...]
uart0: serial@01c28000 {
    compatible = "snps,dw-apb-uart";
    reg = <0x01c28000 0x400>;
    interrupts = <GIC_SPI 0 IRQ_TYPE_LEVEL_HIGH>;
    reg-shift = <2>;
    reg-io-width = <4>;
    clocks = <&ccu CLK_APB2_UART0>;
    resets = <&ccu RST_APB2_UART0>;
    dmas = <&dma 6>, <&dma 6>;
    dma-names = "rx", "tx";
    status = "disabled";
};
[...]
```

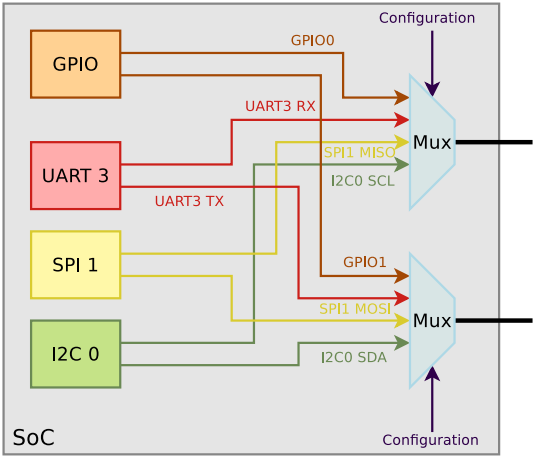
Clock framework: summary



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 buses
  - 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**.

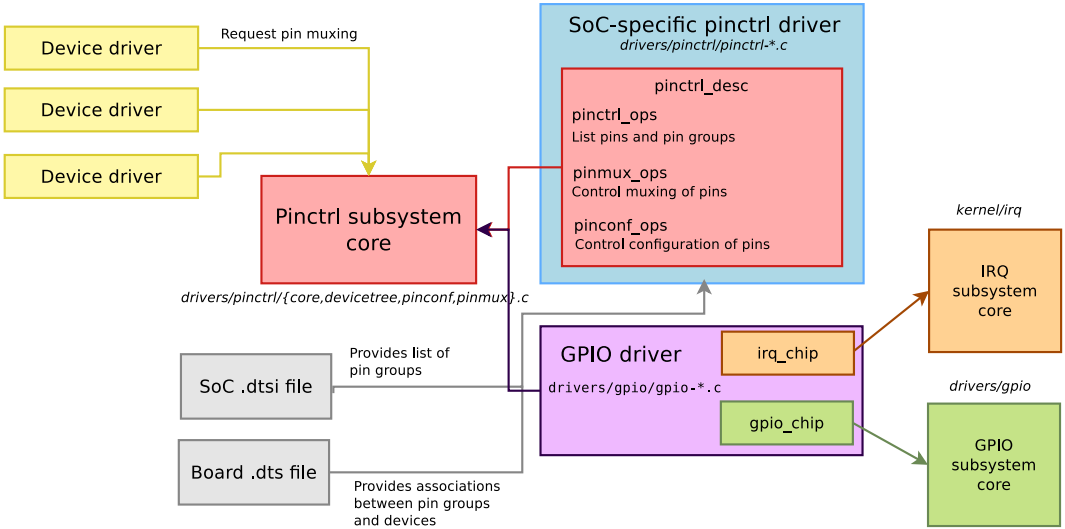
Pin muxing: principle



The 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 pin-muxing subsystem: diagram



Declaring pin groups in the SoC `dtsi`

- From `arch/arm64/boot/dts/amlogic/meson-gxbb.dtsi`.
- Declares the *pinctrl* device and various pin groups.

```
pinctrl_aobus: pinctrl@14 {
    compatible = "amlogic,meson-gxbb-aobus-pinctrl";
    #address-cells = <2>;
    #size-cells = <2>;
    ranges;

    gpio_ao: bank@14 {
        reg = <0x0 0x00014 0x0 0x8>,
              <0x0 0x0002c 0x0 0x4>,
              <0x0 0x00024 0x0 0x8>;
        reg-names = "mux", "pull", "gpio";
        gpio-controller;
        #gpio-cells = <2>;
    };
```

```
    uart_ao_a_pins: uart_ao_a {
        mux {
            groups = "uart_tx_ao_a", "uart_rx_ao_a";
            function = "uart_ao";
        };
    };
[...]
    i2c_ao_pins: i2c_ao {
    mux {
        groups = "i2c_sck_ao", "i2c_sda_ao";
        function = "i2c_ao";
    };
};
```

Associating devices with pin groups, board dts

- From arch/arm64/boot/dts/amlogic/meson-gxbb-odroidc2.dts.

```
[...]
&uart_AO {
    status = "okay";
    pinctrl-0 = <&uart_ao_a_pins>;
    pinctrl-names = "default";
};

[...]
&i2c_A {
    status = "okay";
    pinctrl-0 = <&i2c_a_pins>;
    pinctrl-names = "default";
};
```

Device drivers requesting pin muxing

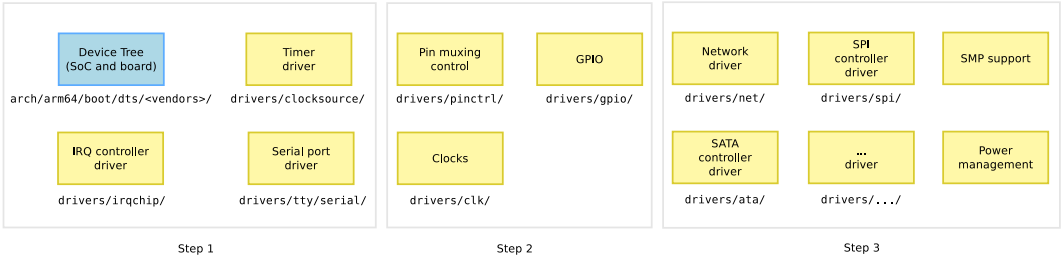
- From `drivers/mmc/host/mxs-mmc.c`

```
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

- 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

- 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`.
- Pay attention of the 64-bits support
  - `arm64` SoCs reused driver developed for ARM 32-bits.
  - Driver supposed to be portable,
  - However being used only on 32-bits architecture some bugs could have been missed.
  - Use `uintptr_t` to cast pointer.
  - Pay attention to the peripheral bus which can have a smaller size than the CPU bus.

Conclusion

- The code in the `arch/arm64` tree is cleaner than the one in `arm/soc`.
- It benefits on all the consolidation done in `arch/arm` without having to deal with legacy code.
- Adding an initial support for a new ARM64 SoCs needs very few line of code now.
- However some of the complexity is hidden in a firmware (SCPI and PSCI) and power management is still challenging.

Questions?

Gregory CLEMENT

gregory.clement@bootlin.com

http://bootlin.com/pub/conferences/2016/elce/clement-arm64-soc-checklist