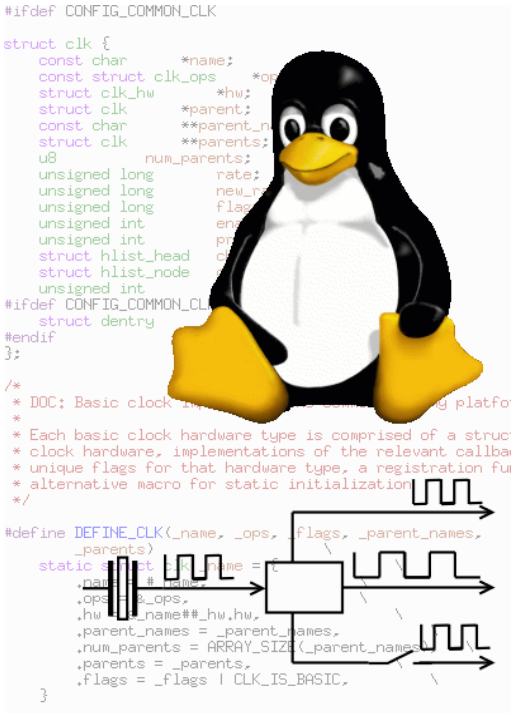
Common clock framework: how to use it



Gregory CLEMENT
Free Electrons
gregory.clement@free-electrons.com

Gregory CLEMENT

- Embedded Linux engineer and trainer at Free Electrons since 2010
  - 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`
- Contributing to **kernel support for the Armada 370 and Armada XP** ARM SoCs from Marvell.
- Co-maintainer of mvebu sub-architecture (SoCs from Marvell Embedded Business Unit)
- Living near **Lyon**, France

Overview

- What the common clock framework is
- Implementation of the common clock framework
- How to add your own clocks
- How to deal with the device tree
- Use of the clocks by device drivers

Clocks

- Most of the electronic chips are driven by **clocks**
- The clocks of the peripherals of an **SoC** (or even a **board**) are organized in a **tree**
- Controlling clocks is useful for:
  - **power management**: clock frequency is a parameter of the dynamic power consumption
  - **time reference**: to compute a baud-rate or a pixel clock for example

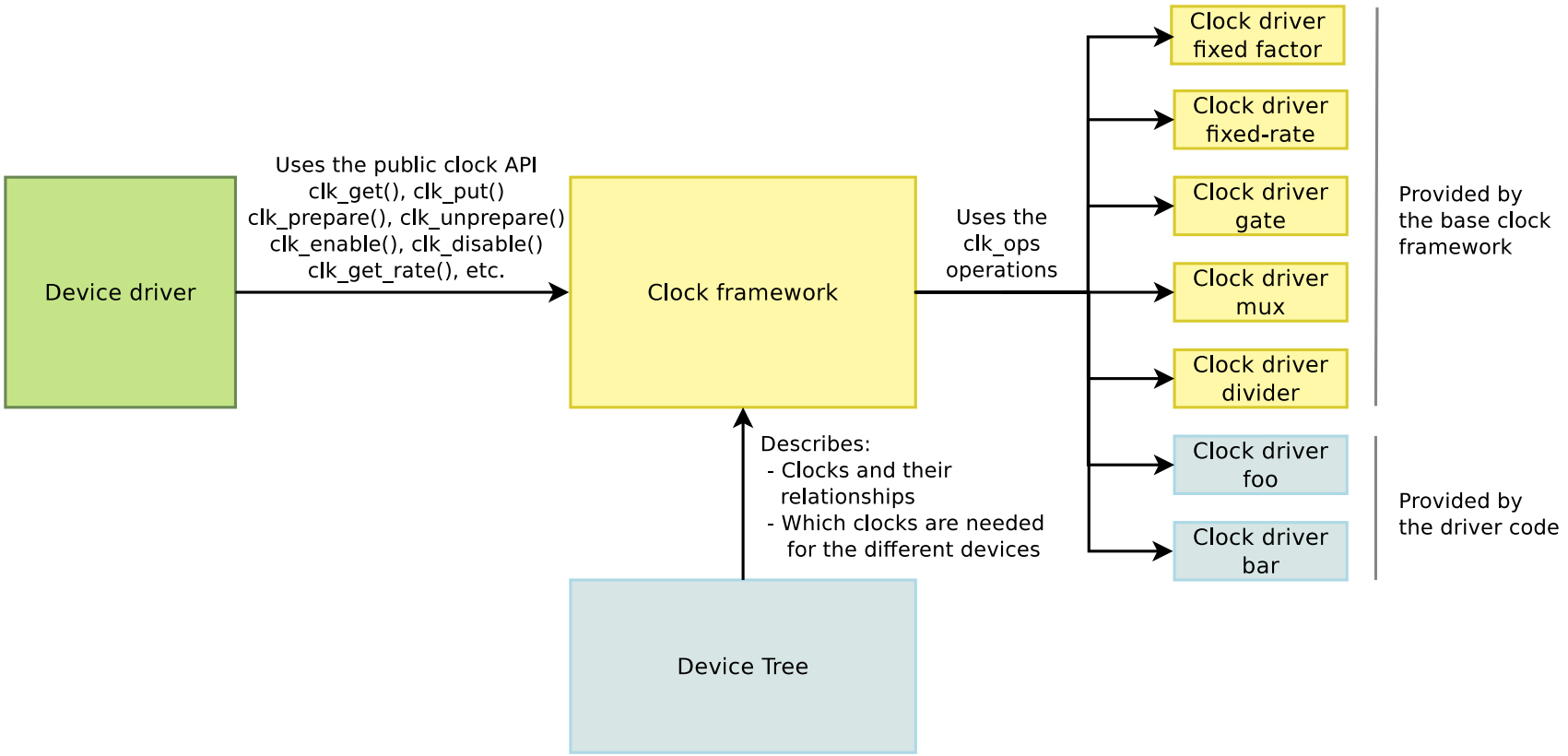
The clock framework

- A **clock framework** has been available for many years (it comes from the prehistory of git)
- Offers a a simple API: `clk_get`, `clk_enable`, `clk_get_rate`, `clk_set_rate`, `clk_disable`, `clk_put`,... that were used by device drivers.
- Nice but had several drawbacks and limitations:
  - Each machine class had its **own implementation** of this API.
  - Does not allow **code sharing**, and common mechanisms
  - Does not work for ARM **multiplatform** kernels.

The common clock framework

- Started by the introduction of a **common struct clk** in early 2010 by **Jeremy Kerr**
- Ended by the merge of the **common clock framework** in kernel 3.4 in May 2012, submitted by **Mike Turquette**
- Implements the **clock framework API**, **some basic clock drivers** and makes it possible to implement **custom clock drivers**
- 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`

Diagram overview of the common clock framework

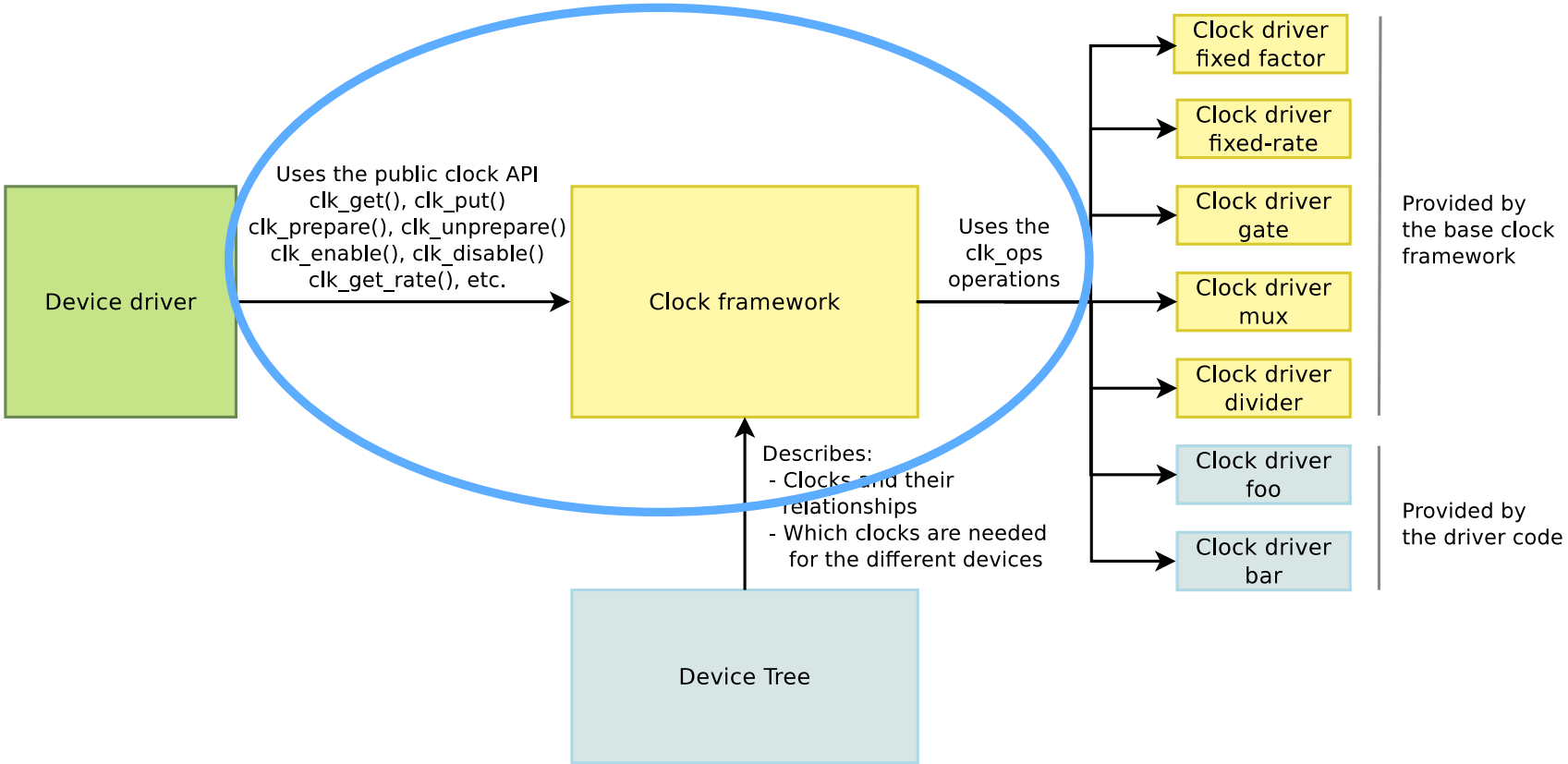


Interface of the CCF

Interface divided into two halves:

- Common Clock Framework core
  - Common definition of `struct clk`
  - Common implementation of the `clk.h` API (defined in `drivers/clk/clk.c`)
  - `struct clk_ops`: operations invoked by the clk API implementation
  - Not supposed to be modified when adding a new driver
- Hardware-specific
  - Callbacks registered with `struct clk_ops` and the corresponding hardware-specific structures (let's call it `struct clk_foo` for this talk)
  - Has to be written for each new hardware clock
- The two halves are tied together by `struct clk_hw`, which is defined in `struct clk_foo` and pointed to within `struct clk`.

Implementation of the CCF core



Implementation of the CCF core

Implementation defined in `drivers/clk/clk.c`. Takes care of:

- **Maintaining** the clock tree
- **Concurrency prevention** (using a global spinlock for `clk_enable()`/`clk_disable()` and a global mutex for all other operations)
- **Propagating** the operations through the clock tree
- **Notification** when rate change occurs on a given clock, the register callback is called.

Implementation of the CCF core

Common `struct clk` definition located in `include/linux/clk-private.h`:

```
struct clk {
        const char              *name;
        const struct clk_ops    *ops;
        struct clk_hw           *hw;
        char                    **parent_names;
        struct clk              **parents;
        struct clk              *parent;
        struct hlist_head       children;
        struct hlist_node       child_node;
...
};
```

Implementation of the CCF core

Implementation the API exposed to the drivers in two files:

- drivers/clk/clk.c:

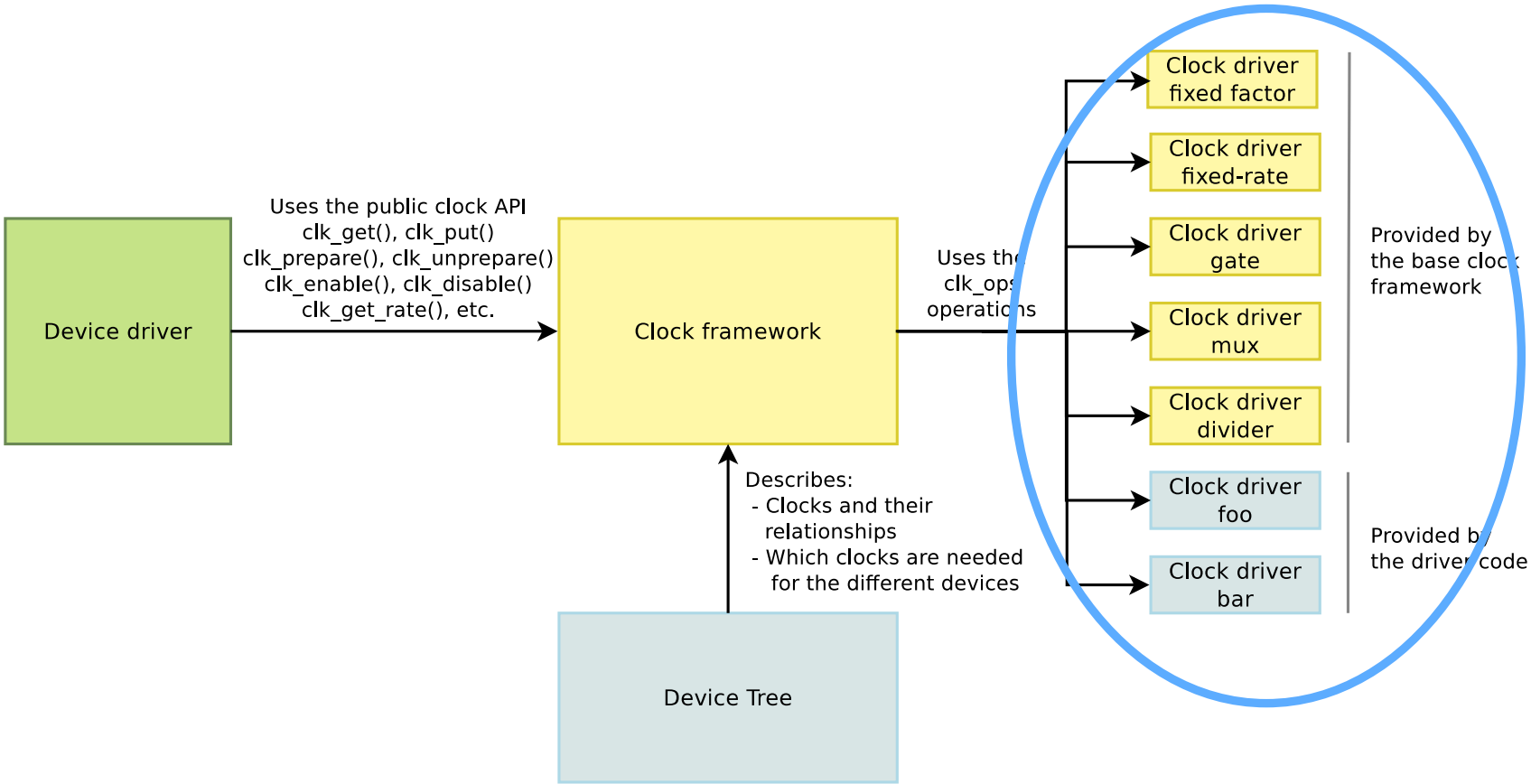
```
void clk_prepare(struct clk *clk);
void clk_unprepare(struct clk *clk);
int clk_enable(struct clk *clk);
void clk_disable(struct clk *clk);
unsigned long clk_get_rate(struct clk *clk);
long clk_round_rate(struct clk *clk, unsigned long rate);
int clk_set_rate(struct clk *clk, unsigned long rate);
int clk_set_parent(struct clk *clk, struct clk *parent);
struct clk *clk_get_parent(struct clk *clk);
...
```

- drivers/clk/clkdev.c:

```
struct clk *clk_get(struct device *dev, const char *id);
void clk_put(struct clk *clk);
...
```

As well as the the managed interface version (devm_*) and the device tree related version (of_*).

Implementation of the hardware clock



Implementation of the hardware clock

- Relies on `.ops` and `.hw` pointers
- Abstracts the details of `struct clk` from the hardware-specific bits
- No need to implement all the operations, only a **few are mandatory** depending on the clock type
- The clock is created once the operation set is registered using `clk_register()`

Implementation of the hardware clock

Hardware operations defined in `include/linux/clk-provider.h`

```
struct clk_ops {
        int             (*prepare)(struct clk_hw *hw);
        void            (*unprepare)(struct clk_hw *hw);
        int             (*is_prepared)(struct clk_hw *hw);
        void            (*unprepare_unused)(struct clk_hw *hw);
        int             (*enable)(struct clk_hw *hw);
        void            (*disable)(struct clk_hw *hw);
        int             (*is_enabled)(struct clk_hw *hw);
        void            (*disable_unused)(struct clk_hw *hw);
        unsigned long   (*recalc_rate)(struct clk_hw *hw,
                                       unsigned long parent_rate);
        long            (*round_rate)(struct clk_hw *hw, unsigned long,
                                      unsigned long *);
        long            (*determine_rate)(struct clk_hw *hw,unsigned long rate,
                                      unsigned long *best_parent_rate,
                                      struct clk **best_parent_clk);
        int             (*set_parent)(struct clk_hw *hw, u8 index);
        u8              (*get_parent)(struct clk_hw *hw);
        int             (*set_rate)(struct clk_hw *hw, unsigned long);
        void            (*init)(struct clk_hw *hw);
};
```

Operations to implement depending on clk capabilities

	gate	change rate	single parent	multiplexer	root
.prepare .unprepare					
.enable .disable .is_enabled	y y y				
.recalc_rate .round_rate .determine_rate .set_rate		y y[1] y[1] y			
.set_parent .get_parent			n n	y y	n n

Legend: y = mandatory, n = invalid or otherwise unnecessary, [1]: at least one of the two operations

Hardware clock operations: making clocks available

The API is split in two pairs:

- `.prepare`(/`.unprepare`):
  - Called to **prepare** the clock **before** actually ungating it
  - Could be called in place of enable in some cases (accessed over I2C)
  - **May sleep**
  - **Must not** be called in **atomic context**
- `.enable`(/`.disable`):
  - Called to **ungate** the clock once it has been prepared
  - Could be called in place of prepare in some case (accessed over single registers in an SoC)
  - Must **not sleep**
  - Can be called in **atomic context**
  - `.is_enabled`: Instead of checking the enable count, **querying the hardware** to determine whether the clock is enabled.

Hardware clock operations: managing the rates

- `.round_rate`: Returns the **closest rate** actually **supported** by the clock. Called by `clk_round_rate()` or by `clk_set_rate()` during propagation.
- `.determine_rate`: Same as `.round_rate` but allow to **select the parent** to get the closet requested rate.
- `.set_rate`: **Changes the rate** of the clock. Called by `clk_set_rate()` or during propagation.
- `.recalc_rate`: **Recalculates the rate** of this clock, by querying hardware supported by the clock. Used internally to update the clock tree.

Hardware clock operations: managing the parents

As seen on the matrix, only used for multiplexers

- `.get_parent`:
  - **Queries the hardware** to determine the parent of a clock.
  - Currently only used when clocks are statically initialized.
  - `clk_get_parent()` doesn't use it, simply returns the `clk->parent` internal struct
- `.set_parent`:
  - **Changes** the input **source** of this clock
  - Receives a index on in either the `.parent_names` or `.parents` arrays
  - `clk_set_parent()` translate `clk` in index

Hardware clock operations: more callbacks

Callbacks that have been recently added for more specific need:

- `.is_prepared`:
  - **Queries the hardware** instead of relying on the software counter to check if a clock was prepared
  - Can replace the `.is_enable` on some place
- `.disable_unused`:
  - Needed when a clock should be disable because it is unused but **can't use** `.disable`.
  - Introduced for OMAP needs
- `.unprepared_unused`:
  - Introduced for the same reason that `.disable_unused`

Hardware clock operations: base clocks

- The common clock framework provides **5 base clocks**:
  - **fixed-rate**: Is always running and provide always the same rate
  - **gate**: Have the same rate as its parent and can only be gated or ungated
  - **mux**: Allow to select a parent among several ones, get the rate from the selected parent, and can't gate or ungate
  - **fixed-factor**: Divide and multiply the parent rate by constants, can't gate or ungate
  - **divider**: Divide the parent rate, the divider can be selected among an array provided at registration, can't gate or ungate
- Most of the clocks can be registered using one of these base clocks.
- Complex hardware clocks have to be split in base clocks
  - For example a gate clock with a fixed rate will be composed of a fixed rate clock as a parent of a gate clock.
  - A special clock type `clk-composite` allows to aggregate the functionality of the basic clock types into one clock (since kernel 3.10).

Composite clocks

Composite clock allows to reuse existing base clock and to aggregate them into a single clock:

- 3 base clocks can be used: **mux**, **rate** and **gate**
- For each base clock aggregated, an **handle** and the **operation set** must be filled
- To register the composite clock, the following function is used:

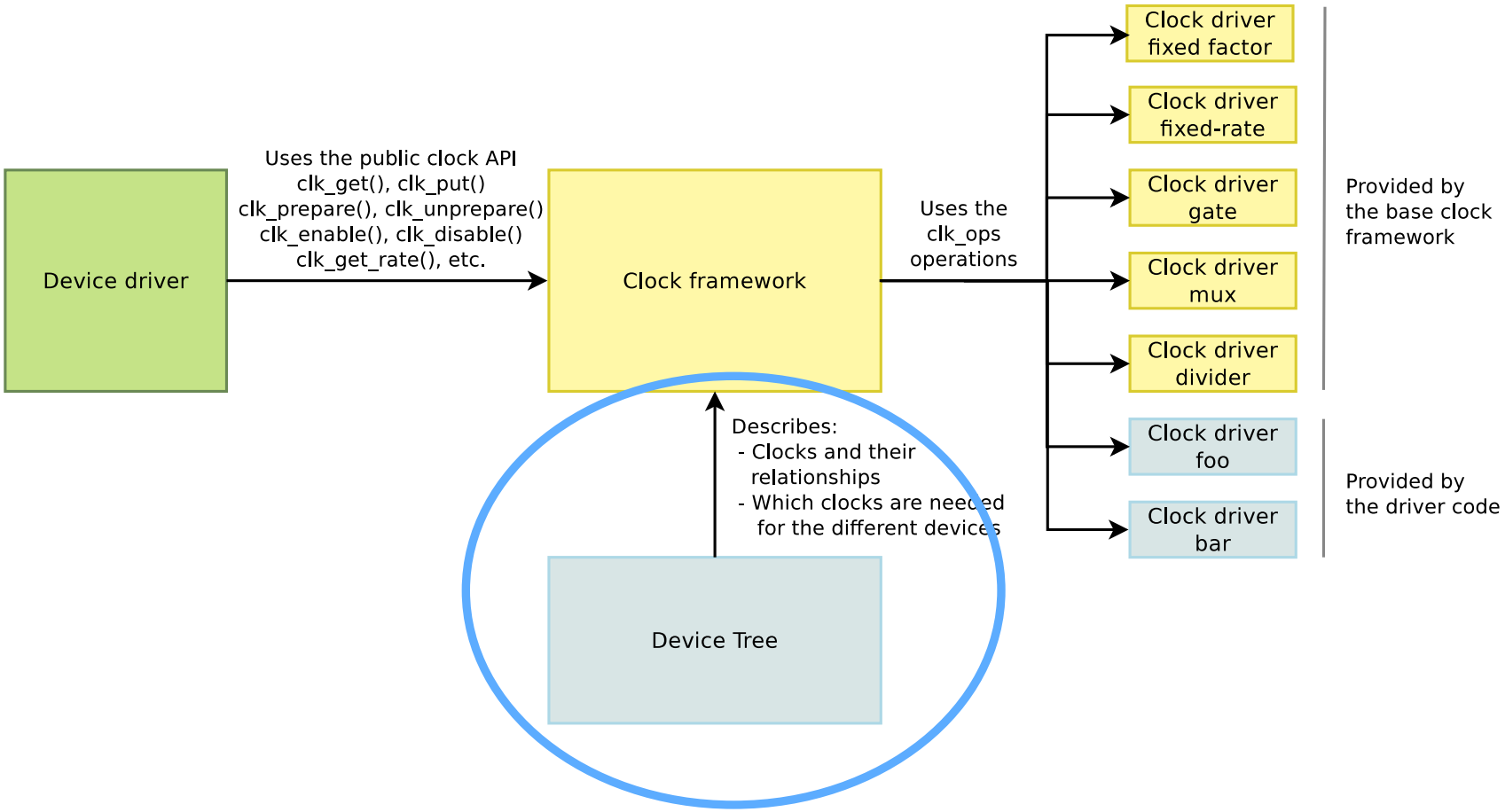
```
struct clk *clk_register_composite(struct device *dev, const char *name,
            const char **parent_names, int num_parents,
            struct clk_hw *mux_hw, const struct clk_ops *mux_ops,
            struct clk_hw *rate_hw, const struct clk_ops *rate_ops,
            struct clk_hw *gate_hw, const struct clk_ops *gate_ops,
            unsigned long flags);
```

Composite clocks: example

From `drivers/clk/sunxi/clk-sunxi.c` (some parts removed)

```
static void __init sun4i_osc_clk_setup(struct device_node *node)
{
        struct clk *clk; struct clk_fixed_rate *fixed;
        struct clk_gate *gate; const char *clk_name = node->name;
        u32 rate;
        /* allocate fixed-rate and gate clock structs */
        fixed = kzalloc(sizeof(struct clk_fixed_rate), GFP_KERNEL);
[...]
        of_property_read_u32(node, "clock-frequency", &rate);
        /* set up gate and fixed rate properties */
        gate->bit_idx = SUNXI_OSC24M_GATE;
[...]
        fixed->fixed_rate = rate;
        clk = clk_register_composite(NULL, clk_name, NULL, 0,
                        NULL, NULL, &fixed->hw, &clk_fixed_rate_ops,
                        &gate->hw, &clk_gate_ops, CLK_IS_ROOT);
        of_clk_add_provider(node, of_clk_src_simple_get, clk);
[...]
}
```

Hardware clock operations: device tree



Hardware clock operations: device tree

- The **device tree** is the **mandatory way** to declare a clock and to get its resources, as for any other driver using DT we have to:
  - **Parse** the device tree to **setup** the clock: the resources but also the properties are retrieved.
  - Declare the **compatible** clocks and associate it with an **initialization** function using `CLK_OF_DECLARE`

Declaration of clocks in DT: simple example (1)

From arch/arm/boot/dts/armada-370-xp.dtsi

```
[...]
        clocks {
                /* 2 GHz fixed main PLL */
                mainpll: mainpll {
                        compatible = "fixed-clock";
                        #clock-cells = <0>;
                        clock-frequency = <2000000000>;
                };
        };
[...]
        coredivclk: corediv-clock@18740 {
                compatible = "marvell,armada-370-corediv-clock";
                reg = <0x18740 0xc>;
                #clock-cells = <1>;
                clocks = <&mainpll>;
                clock-output-names = "nand";
        };
[...]
```

Managing the device tree: simple example (1)

From drivers/clk/clk-fixed-rate.c

```
void __init of_fixed_clk_setup(struct device_node *node)
{
        struct clk *clk;
        const char *clk_name = node->name;
        u32 rate;

        if (of_property_read_u32(node, "clock-frequency", &rate))
                return;

        of_property_read_string(node, "clock-output-names", &clk_name);

        clk = clk_register_fixed_rate(NULL, clk_name, NULL,
                                      CLK_IS_ROOT, rate);
        if (!IS_ERR(clk))
                of_clk_add_provider(node, of_clk_src_simple_get, clk);
}
CLK_OF_DECLARE(fixed_clk, "fixed-clock", of_fixed_clk_setup);
```

Managing the device tree: simple example (2)

From arch/arm/mach-mvebu/armada-370-xp.c

```
[...]
#include <linux/clk-provider.h>
[...]
static void armada_370_xp_timer_and_clk_init(void)
{
        of_clk_init(NULL);
[...]
}
```

From drivers/clk/clk.c

```
void __init of_clk_init(const struct of_device_id *matches)
{
        struct device_node *np;
        if (!matches)
                matches = __clk_of_table;

        for_each_matching_node(np, matches) {
                const struct of_device_id *match = of_match_node(matches, np);
                of_clk_init_cb_t clk_init_cb = match->data;
                clk_init_cb(np);
        }
}
```

Declaration of clocks in DT: advanced example (1)

From arch/arm/boot/dts/armada-xp.dtsi

```
[...]
coreclk: mvebu-sar@d0018230 {
        compatible = "marvell,armada-xp-core-clock";
        reg = <0xd0018230 0x08>;
        #clock-cells = <1>;
};

cpuclk: clock-complex@d0018700 {
        #clock-cells = <1>;
        compatible = "marvell,armada-xp-cpu-clock";
        reg = <0xd0018700 0xA0>;
        clocks = <&coreclk 1>;
};
[...]
```

Managing the device tree: advanced example (1)

From `drivers/clk/mvebu/armada-xp.c` (some parts removed)

```
static const struct coreclk_soc_desc axp_coreclks = {
        .get_tclk_freq = axp_get_tclk_freq,
        .get_cpu_freq = axp_get_cpu_freq,
        .get_clk_ratio = axp_get_clk_ratio,
        .ratios = axp_coreclk_ratios,
        .num_ratios = ARRAY_SIZE(axp_coreclk_ratios),
};

static void __init axp_coreclk_init(struct device_node *np)
{
        mvebu_coreclk_setup(np, &axp_coreclks);
}
CLK_OF_DECLARE(axp_core_clk, "marvell,armada-xp-core-clock",
               axp_coreclk_init);
```

Managing the device tree: advanced example (2)

From drivers/clk/mvebu/common.c (some parts removed)

```
static void __init mvebu_clk_core_setup(struct device_node *np,
 struct core_clocks *coreclk)
{
        const char *tclk_name = "tclk";
        void __iomem *base;

        base = of_iomap(np, 0);
        /* Allocate struct for TCLK, cpu clk, and core ratio clocks */
        clk_data.clk_num = 2 + coreclk->num_ratios;
        clk_data.clks = kzalloc(clk_data.clk_num * sizeof(struct clk *),
                                GFP_KERNEL);
        /* Register TCLK */
        of_property_read_string_index(np, "clock-output-names", 0,
                                      &tclk_name);
        rate = coreclk->get_tclk_freq(base);
        clk_data.clks[0] = clk_register_fixed_rate(NULL, tclk_name, NULL,
                                                   CLK_IS_ROOT, rate);
[...]
}
```

Hardware clock operations: device tree

- **Expose** the clocks to other nodes of the device tree using `of_clk_add_provider()` which takes 3 parameters:
  - `struct device_node *np`: **Device node** pointer **associated** to **clock provider**. This one is usually received by the setup function, when there is a match, with the array previously defined.
  - `struct clk *(*clk_src_get)(struct of_phandle_args *args, void *data)`: Callback for **decoding clock**. For the devices, called through `clk_get()` to return the clock associated to the node.
  - `void *data`: context pointer for the callback, usually a **pointer to the clock(s)** to associate to the node.

Exposing the clocks on DT: Simple example

From drivers/clk/clk.c

```
struct clk *of_clk_src_simple_get(struct of_phandle_args *clkspec,
                                  void *data)
{
        return data;
}
```

From drivers/clk/clk-fixed-rate.c

```
void __init of_fixed_clk_setup(struct device_node *node)
{
        struct clk *clk;
[...]
        clk = clk_register_fixed_rate(NULL, clk_name, NULL,
                                      CLK_IS_ROOT, rate);
        if (!IS_ERR(clk))
                of_clk_add_provider(node, of_clk_src_simple_get, clk);
}
```

Exposing the clocks in DT: Advanced example (1)

From include/linux/clk-provider.h

```
struct clk_onecell_data {
        struct clk **clks;
        unsigned int clk_num;
};
```

From drivers/clk/clk.c

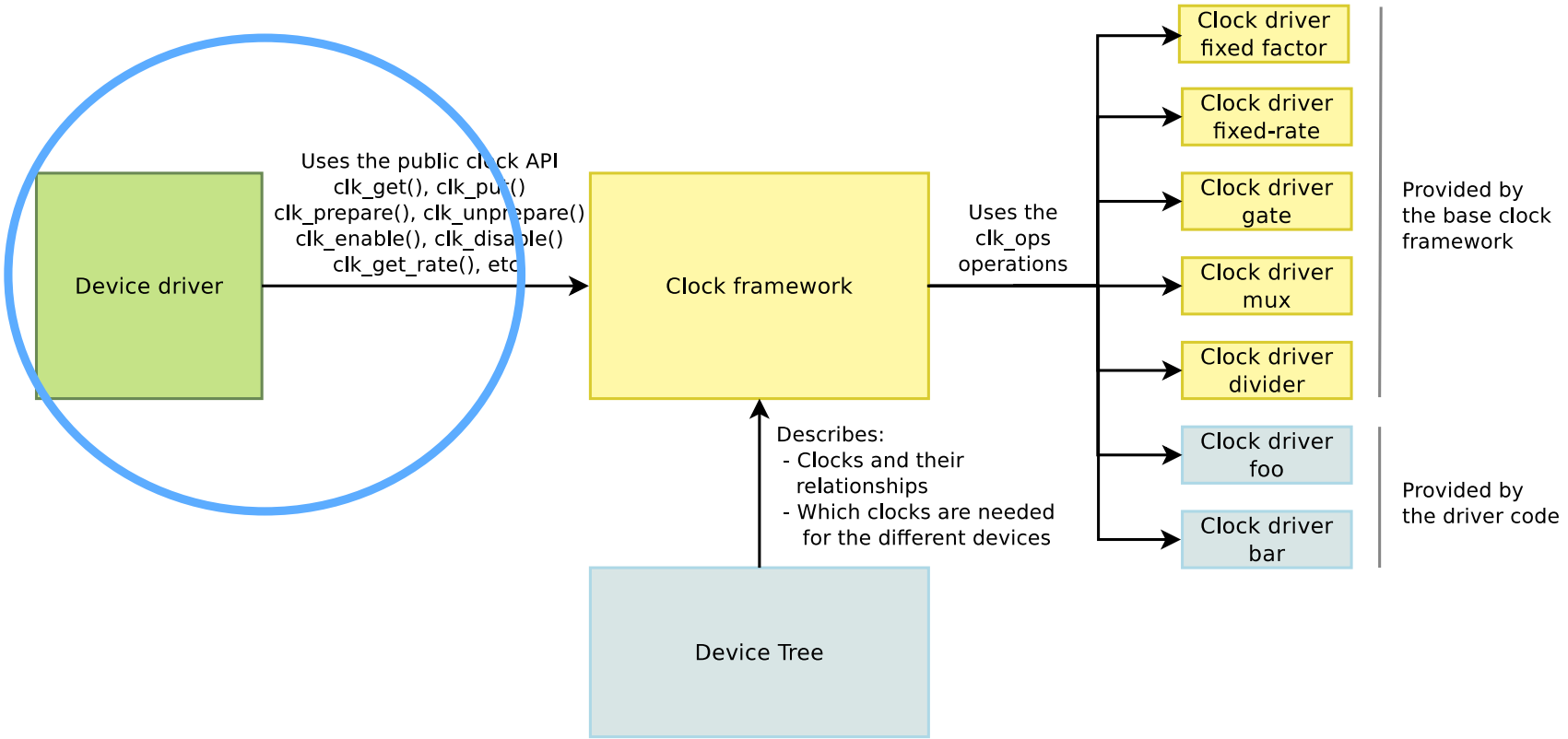
```
struct clk *of_clk_src_onecell_get(struct of_phandle_args *clkspec,
                                   void *data)
{
        struct clk_onecell_data *clk_data = data;
        unsigned int idx = clkspec->args[0];
        if (idx >= clk_data->clk_num) {
                return ERR_PTR(-EINVAL);
        }
        return clk_data->clks[idx];
}
```

Exposing the clocks in DT: Advanced example (2)

From drivers/clk/mvebu/common.c (some parts removed)

```
static struct clk_onecell_data clk_data;
static void __init mvebu_clk_core_setup(struct device_node *np,
                                        struct core_clocks *coreclk)
{
    clk_data.clk_num = 2 + coreclk->num_ratios;
    clk_data.clks = kzalloc(clk_data.clk_num * sizeof(struct clk *),
                                       GFP_KERNEL);
[...]
    for (n = 0; n < coreclk->num_ratios; n++) {
[...]
        clk_data.clks[2+n] = clk_register_fixed_factor(NULL, rclk_name,
                                    cpuclk_name, 0, mult, div);
    };
[...]
    of_clk_add_provider(np, of_clk_src_onecell_get, &clk_data);
}
```

How device drivers use the CCF



How device drivers use the CCF

- Use `clk_get()` to **get the clock** of the device
- **Link** between **clock and device** done either by platform data (old method) or by **device tree** (preferred method)
- Managed version: `devm_get_clk()`
- **Activate** the clock by `clk_enable()` and/or `clk_prepare()` (depending of the context), **sufficient** for most drivers.
- Manipulate the clock using the clock API

Devices referencing their clock in the Device Tree

From arch/arm/boot/dts/armada-xp.dtsi

```
ethernet@d0030000 {
        compatible = "marvell,armada-370-neta";
        reg = <0xd0030000 0x2500>;
        interrupts = <12>;
        clocks = <&gateclk 2>;
        status = "disabled";
};
```

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

```
watchdog@fff10620 {
        compatible = "arm,cortex-a9-twd-wdt";
        reg = <0xfff10620 0x20>;
        interrupts = <1 14 0xf01>;
        clocks = <&a9periphclk>;
};
```

Example clock usage in a driver

From drivers/net/ethernet/marvell/mvneta.c

```
static void mvneta_rx_time_coal_set(struct mvneta_port *pp,
                    struct mvneta_rx_queue *rxq, u32 value)
{
[...]
        clk_rate = clk_get_rate(pp->clk);
        val = (clk_rate / 1000000) * value;
        mvreg_write(pp, MVNETA_RXQ_TIME_COAL_REG(rxq->id), val);
}

static int mvneta_probe(struct platform_device *pdev)
{
[...]
        pp->clk = devm_clk_get(&pdev->dev, NULL);
        clk_prepare_enable(pp->clk);
[...]
}
static int mvneta_remove(struct platform_device *pdev)
{
[...]
        clk_disable_unprepare(pp->clk);
[...]
}
```

Conclusion

- **Efficient** way to declare and use clocks: the amount of code to support new clocks is very reduced.
- More and more used:
  - Most of the complex ARM SoCs have now finished their migration
  - Other architectures start to use it: MIPS, x86.
- Recent added features:
  - Improve **debugfs** output by adding **JSON** style (since v3.9)
  - **Reentrancy** which is needed for **DVFS** (since 3.10)
  - **Composite clock** (since 3.10)

Questions?

Gregory CLEMENT

gregory.clement@free-electrons.com

Thanks to Thomas Petazzoni,(Free Electrons, working with me on Marvell mainlining), Mike Turquette (Linaro, CCF maintainer)

http://free-electrons.com/pub/conferences/2013/elce/common-clock-framework-how-to-use-it/