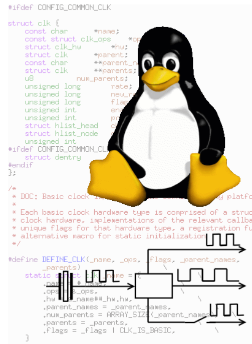
Common clock framework: how to use it



Gregory CLEMENT

Bootlin

gregory.clement@bootlin.com

Gregory CLEMENT

- Embedded Linux engineer and trainer at Bootlin 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://bootlin.com`
- Contributing the **kernel support for the new 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

Device driver

Uses the public clock API
clk_get(), clk_put()
clk_prepare(), clk_unprepare()
clk_enable(), clk_disable()
clk_get_rate(), etc.

Clock framework

Uses the
clk_ops
operations

Clock driver fixed factor

Clock driver fixed-rate

Clock driver gate

Clock driver mux

Clock driver divider

Provided by the base clock framework

Clock driver foo

Clock driver bar

Provided by the driver code

Describes:
- Clocks and their relationships
- Which clocks are needed for the different devices

Device Tree

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

Device driver

Uses the public clock API
clk_get(), clk_put()
clk_prepare(), clk_unprepare()
clk_enable(), clk_disable()
clk_get_rate(), etc.

Clock framework

Uses the
clk_ops
operations

Clock driver fixed factor

Clock driver fixed-rate

Clock driver gate

Clock driver mux

Clock driver divider

Provided by the base clock framework

Clock driver foo

Clock driver bar

Provided by the driver code

Describes:
- Clocks and their relationships
- Which clocks are needed for the different devices

Device Tree

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

The `clk_set_rate()` example:

```
int clk_set_rate(struct clk *clk, unsigned long rate)
{
        struct clk *top, *fail_clk;
        int ret = 0;
        /* prevent racing with updates to the clock topology */
        mutex_lock(&prepare_lock);
        /* bail early if nothing to do */
        if (rate == clk->rate)
                goto out;
        if ((clk->flags & CLK_SET_RATE_GATE) && clk->prepare_count) {
```

For this particular clock, setting its rate is possible only if the clock is ungated (not yet prepared)

```
                ret = -EBUSY;
                goto out;
        }
        /* calculate new rates and get the topmost changed clock */
        top = clk_calc_new_rates(clk, rate);
```

[...] *Exit with error if `clk_calc_new_rates()` failed*

Implementation of the CCF core

The `clk_set_rate()` example (continued):

```
        /* notify that we are about to change rates */
        fail_clk = clk_propagate_rate_change(top, PRE_RATE_CHANGE);
        if (fail_clk) {
                pr_warn("%s: failed to set %s rate\n", __func__,
                                fail_clk->name);
                clk_propagate_rate_change(top, ABORT_RATE_CHANGE);
                ret = -EBUSY;
                goto out;
        }
        /* change the rates */
        clk_change_rate(top);
```

Actually set the rate using the hardware operation

```
out:
        mutex_unlock(&prepare_lock);
        return ret;
}
```

Implementation of the hardware clock

Device driver

Uses the public clock API
clk_get(), clk_put()
clk_prepare(), clk_unprepare()
clk_enable(), clk_disable()
clk_get_rate(), etc.

Clock framework

Uses the clk_ops operations

Clock driver fixed factor

Clock driver fixed-rate

Clock driver gate

Clock driver mux

Clock driver divider

Provided by the base clock framework

Clock driver foo

Clock driver bar

Provided by the driver code

Describes:
- Clocks and their relationships
- Which clocks are needed for the different devices

Device Tree

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             (*enable)(struct clk_hw *hw);
        void            (*disable)(struct clk_hw *hw);
        int             (*is_enabled)(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 *);
        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	**y**				
.disable	**y**				
.is_enabled	**y**				
.recalc_rate		**y**			
.round_rate		**y**			
.set_rate		**y**			
.set_parent			**n**	**y**	**n**
.get_parent			**n**	**y**	**n**
.init					

Legend: **y** = mandatory, **n** = invalid or otherwise unnecessary

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 register 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 if 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.
- `.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: 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.
  - New clock type submitted recently: `clk-composite`. It will allow to aggregate the functionality of the basic clock types into one clock. Still under review.

Hardware clock operations: static initialization

- Put in place to **ease migration** of the complex SoC to the common clock framework
  - Platforms used to use hundreds clocks statically defined
  - They had to include `include/linux/clk-private.h` and `__clk_init()` to reuse these definitions.
- Still **possible** (*but not recommended*) to do static initialization
  - **Absolutely no new platform** should include `clk-private.h`
  - Clocks **must** be initialized via a call to `clk_register()` using `clk_init_data` objects which get bundled with `clk_hw`

Hardware clock operations: device tree

Device driver

Uses the public clock API
clk_get(), clk_put()
clk_prepare(), clk_unprepare()
clk_enable(), clk_disable()
clk_get_rate(), etc.

Clock framework

Uses the
clk_ops
operations

Clock driver
fixed factor

Clock driver
fixed-rate

Clock driver
gate

Clock driver
mux

Clock driver
divider

Provided by
the base clock
framework

Clock driver
foo

Clock driver
bar

Provided by
the driver code

Describes:
- Clocks and their
relationships
- Which clocks are needed
for the different devices

Device Tree

Hardware clock operations: device tree

- The **device tree** is the **preferred 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: resources but also properties are retrieved.
  - Create an **array** of`struct of_device_id` to **match** the compatible clocks
  - Associate data and setup functions to each node

Declaration of clocks in DT: simple example (1)

From arch/arm/boot/dts/ecx-common.dtsi

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

        ddrpll: ddrpll {
                #clock-cells = <0>;
                compatible = "calxeda,hb-pll-clock";
                clocks = <&osc>;
                reg = <0x108>;
        };
[...]
```

Managing the device tree: simple example (1)

From drivers/clk/clk-highbank.c

```
static const __initconst struct of_device_id clk_match[] = {
        { .compatible = "fixed-clock", .data = of_fixed_clk_setup, },
[...]
};

void __init highbank_clocks_init(void)
{
        of_clk_init(clk_match);
}
```

From drivers/clk/clk.c

```
void __init of_clk_init(const struct of_device_id *matches)
{
        struct device_node *np;
        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);
        }
}
```

Managing the device tree: simple example (2)

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);
}
```

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/clk-core.c (some parts removed)

```
static const struct core_clocks armada_370_core_clocks = {
        .get_tclk_freq = armada_370_get_tclk_freq,
        .num_ratios = ARRAY_SIZE(armada_370_xp_core_ratios),
};
static const __initdata struct of_device_id clk_core_match[] = {
[...]
        {
                .compatible = "marvell,armada-xp-core-clock",
                .data = &armada_xp_core_clocks,
        },
[...]
};
void __init mvebu_core_clk_init(void)
{
        struct device_node *np;

        for_each_matching_node(np, clk_core_match) {
                const struct of_device_id *match =
                        of_match_node(clk_core_match, np);
                mvebu_clk_core_setup(np, (struct core_clocks *)match->data);
        }
}
```

Managing the device tree: advanced example (2)

From drivers/clk/mvebu/clk-core.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/clk-core.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

Device driver

Uses the public clock API
clk_get(), clk_put()
clk_prepare(), clk_unprepare()
clk_enable(), clk_disable()
clk_get_rate(), etc

Clock framework

Uses the
clk_ops
operations

Clock driver
fixed factor

Clock driver
fixed-rate

Clock driver
gate

Clock driver
mux

Clock driver
divider

Provided by
the base clock
framework

Clock driver
foo

Clock driver
bar

Provided by
the driver code

Describes:
- Clocks and their relationships
- Which clocks are needed for the different devices

Device Tree

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.
- Still **quite recent**:
  - Complex SoCs still need to finish their migration
- Upcoming features:
  - **DVFS** (Patch set from Mike Turquette adding new notifications and reentrancy)
  - **Composite clock** (Patch set from Prashant Gaikwad)
  - Improve **debugfs** output by adding **JSON** style (also from Prashant Gaikwad)

Questions?

Gregory CLEMENT

gregory.clement@bootlin.com

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

Slides under CC-BY-SA 3.0

http://bootlin.com/pub/conferences/2013/elc/common-clock-framework-how-to-use-it/