The Common Clk Framework

Author: Mike Turquette <mturquette@ti.com>

This document endeavours to explain the common clk framework details, and how to port a platform over to this framework. It is not yet a detailed explanation of the clock api in include/linux/clk.h, but perhaps someday it will include that information.

Introduction and interface split

The common clk framework is an interface to control the clock nodes available on various devices today. This may come in the form of clock gating, rate adjustment, muxing or other operations. This framework is enabled with the CONFIG_COMMON_CLK option.

The interface itself is divided into two halves, each shielded from the details of its counterpart. First is the common definition of struct clk which unifies the framework-level accounting and infrastructure that has traditionally been duplicated across a variety of platforms. Second is a common implementation of the clk.h api, defined in drivers/clk/clk.c. Finally there is struct clk_ops, whose operations are invoked by the clk api implementation.

The second half of the interface is comprised of the hardware-specific callbacks registered with struct clk_ops and the corresponding hardware-specific structures needed to model a particular clock. For the remainder of this document any reference to a callback in struct clk_ops, such as .enable or .set_rate, implies the hardware-specific implementation of that code. Likewise, references to struct clk_foo serve as a convenient shorthand for the implementation of the hardware-specific bits for the hypothetical "foo" hardware.

Tying the two halves of this interface together is struct clk_hw, which is defined in struct clk_foo and pointed to within struct clk_core. This allows for easy navigation between the two discrete halves of the common clock interface.

Common data structures and api

Below is the common struct clk_core definition from drivers/clk/clk.c, modified for brevity:

```
struct clk_core {
        const char              *name;
        const struct clk_ops    *ops;
        struct clk_hw           *hw;
        struct module           *owner;
        struct clk_core         *parent;
        const char              **parent_names;
        struct clk_core         **parents;
        u8                      num_parents;
        u8                      new_parent_index;
        ...
};
```

The members above make up the core of the clk tree topology. The clk api itself defines several driver-facing functions which operate on struct clk. That api is documented in include/linux/clk.h.

Platforms and devices utilizing the common struct clk_core use the struct clk_ops pointer in struct clk_core to perform the hardware-specific parts of the operations defined in 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 rate,
                                        unsigned long *parent_rate);
        int             (*determine_rate)(struct clk_hw *hw,
                                          struct clk_rate_request *req);
        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 rate,
                                    unsigned long parent_rate);
        int             (*set_rate_and_parent)(struct clk_hw *hw,
                                    unsigned long rate,
                                    unsigned long parent_rate,
                                    u8 index);
        unsigned long   (*recalc_accuracy)(struct clk_hw *hw,
                                           unsigned long parent_accuracy);
        int             (*get_phase)(struct clk_hw *hw);
        int             (*set_phase)(struct clk_hw *hw, int degrees);
        void            (*init)(struct clk_hw *hw);
        void            (*debug_init)(struct clk_hw *hw,
                                      struct dentry *dentry);
};
```

Hardware clk implementations

The strength of the common struct clk_core comes from its .ops and .hw pointers which abstract the details of struct clk from the hardware-specific bits, and vice versa. To illustrate consider the simple gateable clk implementation in drivers/clk/clk-gate.c:

```
struct clk_gate {
        struct clk_hw   hw;
        void __iomem    *reg;
        u8              bit_idx;
        ...
};
```

struct clk_gate contains struct clk_hw hw as well as hardware-specific knowledge about which register and bit controls this clk's gating. Nothing about clock topology or accounting, such as enable_count or notifier_count, is needed here. That is all handled by the common framework code and struct clk_core.

Let's walk through enabling this clk from driver code:

```
struct clk *clk;
clk = clk_get(NULL, "my_gateable_clk");

clk_prepare(clk);
clk_enable(clk);
```

The call graph for clk_enable is very simple:

```
clk_enable(clk);
        clk->ops->enable(clk->hw);
        [resolves to...]
                clk_gate_enable(hw);
                [resolves struct clk gate with to_clk_gate(hw)]
                        clk_gate_set_bit(gate);
```

And the definition of clk_gate_set_bit:

```
static void clk_gate_set_bit(struct clk_gate *gate)
{
        u32 reg;

        reg = __raw_readl(gate->reg);
        reg |= BIT(gate->bit_idx);
        writel(reg, gate->reg);
}
```

Note that to_clk_gate is defined as:

```
#define to_clk_gate(_hw) container_of(_hw, struct clk_gate, hw)
```

This pattern of abstraction is used for every clock hardware representation.

Supporting your own clk hardware

When implementing support for a new type of clock it is only necessary to include the following header:

```
#include <linux/clk-provider.h>
```

To construct a clk hardware structure for your platform you must define the following:

```
struct clk_foo {
        struct clk_hw hw;
        ... hardware specific data goes here ...
};
```

To take advantage of your data you'll need to support valid operations for your clk:

```
struct clk_ops clk_foo_ops = {
        .enable         = &clk_foo_enable,
        .disable        = &clk_foo_disable,
};
```

Implement the above functions using container_of:

```
#define to_clk_foo(_hw) container_of(_hw, struct clk_foo, hw)

int clk_foo_enable(struct clk_hw *hw)
{
        struct clk_foo *foo;

        foo = to_clk_foo(hw);

        ... perform magic on foo ...

        return 0;
};
```

Below is a matrix detailing which clk_ops are mandatory based upon the hardware capabilities of that clock. A cell marked as "y" means mandatory, a cell marked as "n" implies that either including that callback is invalid or otherwise unnecessary. Empty cells are either optional or must be evaluated on a case-by-case basis.

clock hardware characteristics

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

[1] (*1*,*2*) either one of round_rate or determine_rate is required.

Finally, register your clock at run-time with a hardware-specific registration function. This function simply populates struct clk_foo's data and then passes the common struct clk parameters to the framework with a call to:

```
clk_register(...)
```

See the basic clock types in `drivers/clk/clk-*.c` for examples.

Disabling clock gating of unused clocks

Sometimes during development it can be useful to be able to bypass the default disabling of unused clocks. For example, if drivers aren't enabling clocks properly but rely on them being on from the bootloader, bypassing the disabling means that the driver will remain functional while the issues are sorted out.

To bypass this disabling, include "clk_ignore_unused" in the bootargs to the kernel.

Locking

The common clock framework uses two global locks, the prepare lock and the enable lock.

The enable lock is a spinlock and is held across calls to the .enable, .disable operations. Those operations are thus not allowed to sleep, and calls to the clk_enable(), clk_disable() API functions are allowed in atomic context.

For clk_is_enabled() API, it is also designed to be allowed to be used in atomic context. However, it doesn't really make any sense to hold the enable lock in core, unless you want to do something else with the information of the enable state with that lock held. Otherwise, seeing if a clk is enabled is a one-shot read of the enabled state, which could just as easily change after the function returns because the lock is released. Thus the user of this API needs to handle synchronizing the read of the state with whatever they're using it for to make sure that the enable state doesn't change during that time.

The prepare lock is a mutex and is held across calls to all other operations. All those operations are allowed to sleep, and calls to the corresponding API functions are not allowed in atomic context.

This effectively divides operations in two groups from a locking perspective.

Drivers don't need to manually protect resources shared between the operations of one group, regardless of whether those resources are shared by multiple clocks or not. However, access to resources that are shared between operations of the two groups needs to be protected by the drivers. An example of such a resource would be a register that controls both the clock rate and the clock enable/disable state.

The clock framework is reentrant, in that a driver is allowed to call clock framework functions from within its implementation of clock operations. This can for instance cause a .set_rate operation of one clock being called from within the .set_rate operation of another clock. This case must be considered in the driver implementations, but the code flow is usually controlled by the driver in that case.

Note that locking must also be considered when code outside of the common clock framework needs to access resources used by the clock operations. This is considered out of scope of this document.