Clock Management in Zephyr RTOS

Daniel DeGrasse

NXP Semiconductors

EMBEDDED OPEN SOURCE SUMMIT

Agenda

- Overview of current Clock Control framework
- Discussion of framework requirements
- Proposed Implementation
  - SOC perspective
  - Driver API
- Framework Implementation Details

Overview of Clock Control Framework

Clock Control Framework

- Present in Zephyr since 1.0
- Based around hardware specific “subsystems”
- Supported operations
  - Get clock frequency as integer
  - Set rate via opaque driver specific parameter
  - Get clock status
  - Gate/enable a clock (optionally can do so asynchronously)
  - Configure a clock via opaque pointer
- Besides reading rates, all clock data is driver specific
- Drivers can read clock settings from devicetree to setup clocks at boot time
  - Configure API can also be used for runtime changes

Clock Control Usage Example- STM32

- SOC series specific clock drivers
- STM32 specific macro “STM32_DT_CLOCKS” used to setup configuration structure for clock data
- Clock control configure API permits devices to enable/configure their clocks
  - Device specific clock data encodes register offset and bits to set
- Multiplexer selections are made via devicetree node with “st,stm32-clock-mux” compatible, which applies selections via clock configure API at boot time
- Reads clock controller node devicetree properties to initialize root clocks
  - PLLs set up via this method
  - Validates that each resulting frequency will be within range supported by SOC at build time

Why Rework Clock Control?

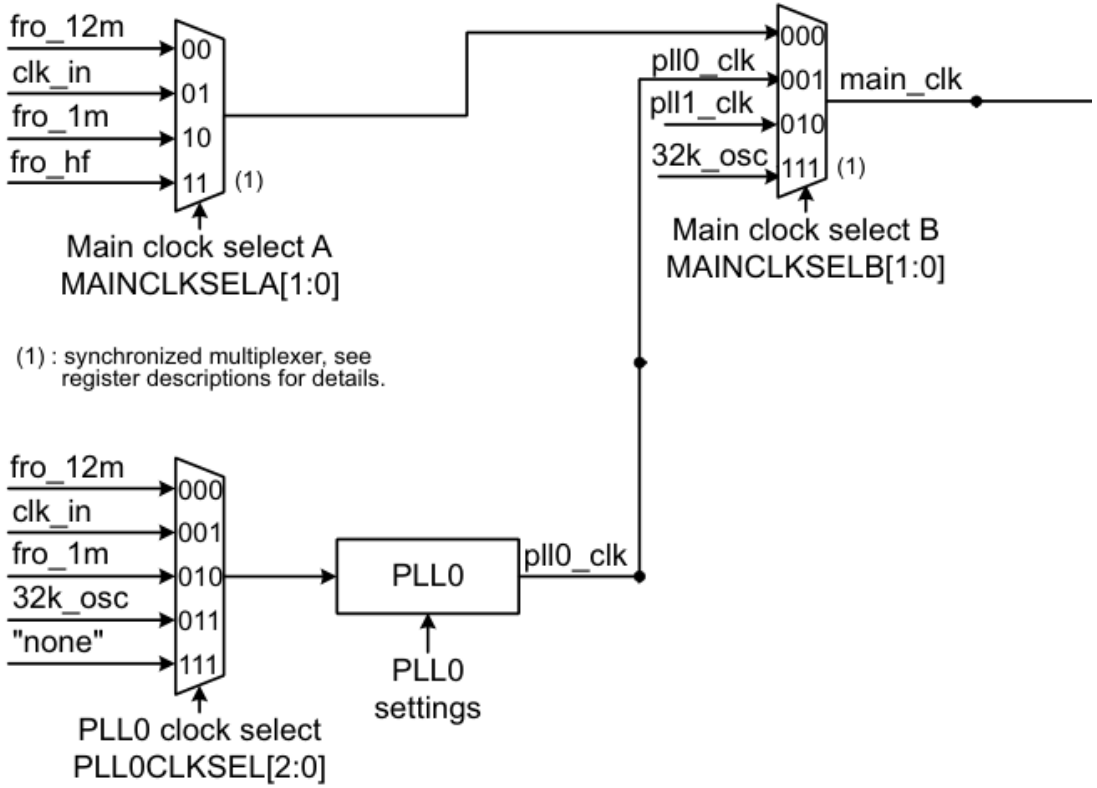
EMBEDDED OPEN SOURCE SUMMIT

- Better code reuse
  - Dividers and multiplexers are often reused across SOC lines, so we should reuse drivers
- Runtime clock configuration
  - Limited ability to configure complicated sources like PLLs at runtime
    - Usually use settings on clock controller node
  - No facility for clock callbacks exists in clock control framework
- Clock control framework leaks device specific data
  - Need vendor specific macros in driver to initialize clock data
  - Details about clock driver leak into device driver implementations
- Set rate API
  - Flexible, but high runtime/flash overhead to implement this feature

Clock Control Requirements

Clock Hardware

- Gates
  - Often one per peripheral- enable or disable clock signal for peripheral device
  - Can also gate root clock sources
- Sources
  - Usually fixed output frequency, sometimes can be powered off at runtime
- Dividers/Multipliers
- Multiplexers
- PLLs
  - Often very vendor/SOC specific (analog IP)
  - Include multipliers and dividers, as well as vendor specific settings
  - Example: spread spectrum generator on LPC55sxx series
  - Significant power usage- need to be powered down in idle modes



Clock Diagram sourced from LPC55S6x series reference manual, "LPC55S6x Clock generation"

Framework Goals

- Feature parity with clock control
- Ability to configure clock settings at runtime for power savings
  - Support powering off clock sources
- Notify clock consumers when parent changes rate
- Hide clock implementation details from device drivers
- Describe clock hierarchy in devicetree
- Maximize code reuse across SOCs
  - Digital IP like multiplexers and dividers is often reused across SOC lines
- Minimize flash/ram footprint
  - Ideally similar footprint to existing driver with clock control framework
  - Runtime rate calculation should be optional- many devices won't need this feature

Framework Proposal

Framework Overview

EMBEDDED OPEN SOURCE SUMMIT

- Clock driver layer
  - Based on Linux common clock framework
  - Each clock is represented by a clock object
  - Clock objects hold references to parent (for rate calculation) and children (for callback notification)
- Clock management layer
  - Consumers only interact with this layer
  - Clock configurations are described as “states” in devicetree
    - Series of clock nodes with hardware specific configuration data as specifiers
  - Clock outputs can have their frequency queried directly
  - Setting rate can be accomplished via “output clock” nodes
  - Support for rate change callbacks

Clock Driver API

- Heavily inspired by Linux common clock framework
- Clock elements like multiplexers, dividers, and PLLs are considered “nodes”
  - Each clock node will have a clock structure describing it
  - Structures will have private hardware data, as well as clock API
- Clock nodes will implement clock API
  - notify- inform clock that parent is going to change rate
  - get_rate: reads clock rate in Hz
  - configure: configures clock with hardware specific data
  - set_rate: Sets clock rate in Hz
  - round_rate: Determine closest supportable clock rate
- Like device model, clock nodes can reference one another
  - Allows clock node to read parent rate and calculate its own rate
- External clock generators can also be supported
  - No distinction between on die and off die clock sources
- Set rate and round rate APIs are enabled via Kconfig- devices that don’t need this support can disable it

```
/* Notify clock that parent will be reconfigured */
int clock_notify(const struct clk *clk,
                 const struct clk *parent,
                 uint32_t parent_rate);
/* Read clock rate */
int clock_get_rate(const struct clk *clk);
/* Configure clock */
int clock_configure(const struct clk *clk, void *data);
/* Get closest rate clock can support */
int clock_round_rate(const struct clk *clk,
                     uint32_t req_rate);
/* Set clock to given rate */
int clock_set_rate(const struct clk *clk, uint32_t rate)
```

Clock Management API

EMBEDDED OPEN SOURCE SUMMIT

- Abstract clock settings to “states”- similar idea to pin control
  - Run and idle states defined by default
  - Drivers can define additional custom states
- Clock rate can be queried from clock outputs
- Devices can register for clock reconfiguration notifications
  - If a parent of a driver’s clock is reconfigured, the driver will be notified
- Driver clock API:
  - clock_mgmt_get_rate: Read the rate of a clock
  - clock_mgmt_apply_state: Apply a clock state
  - clock_mgmt_set_callback: Set a callback for a clock output
- No support for runtime rate setting
  - Drivers can define a state for a given frequency; devicetree implementer decides how to configure clocks

Clock Management SOC Devicetree

- SOC clock tree will be defined hierarchically
- Nodes reference a clock node in SOC devicetree to query rate
- Node references determine clock children and parent relationships in clock framework

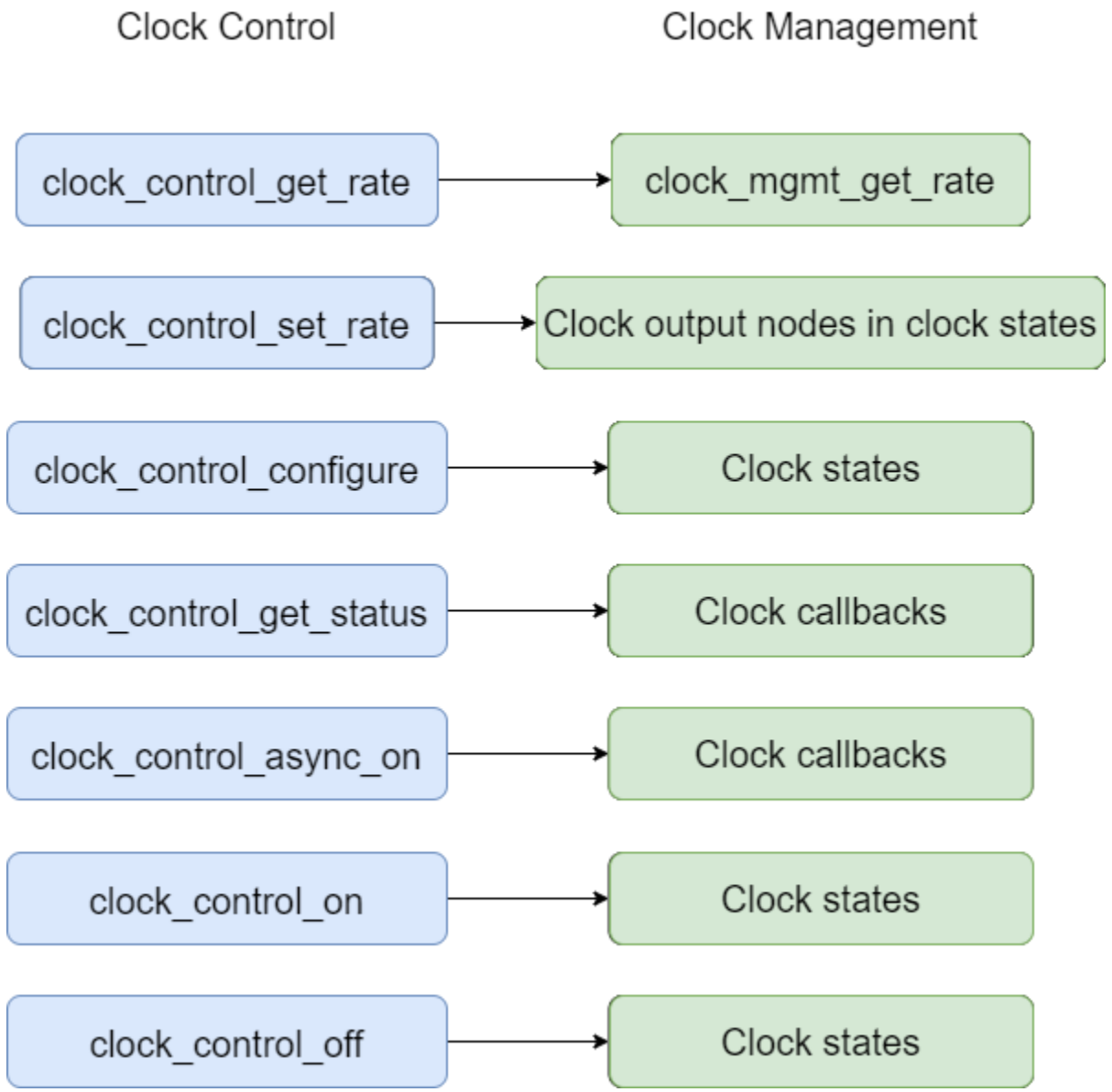
```
/* SOC devicetree */
fcclksel0: fcclksel0@400002b0 {
        compatible = "nxp,syscon-clock-mux";
        #clock-cells = <1>;
        /* SYSCON::FCCLKSEL0[SEL] */
        reg = <0x400002b0 0x3>;
        offset = <0x0>;
        input-sources = <&mainclkselb &pll0div
                &fro_12m &frohfdiv &fro_1m
                &mclk_in &rtcosc32ksel
                &no_clock>;
        #address-cells = <1>;
        #size-cells = <1>;

        frgctrl0_mul: frgctrl0-mul@40000320 {
                compatible = "nxp,syscon-flexfrg";
                #clock-cells = <1>;
                /* SYSCON::FLEXFRG0CTRL[MULT] */
                reg = <0x40000320 0x8>;
        };
};
```

Clock Management States

- States will be defined within devicetree, using specifiers on clock nodes
  - Specifiers will be specific to each clock node
  - Passed to clock configure API at runtime to setup clock
- Supporting set_rate with this API
  - Special “clock output” nodes will accept one specifier- the requested clock rate. This will allow drivers to configure clock rates directly if desired
- Nodes can define as many states as needed, and can configure any clock within a state

```
/* Board configuration */
&flexcomm0 {
        status = "okay";
        clock-outputs = <&frgctrl0_mul>;
        /* Setup multipler and divider selection */
        clock-state-0 = <&fcclksel0 0 &frgctrl0_mul 0>;
        /* State 1 uses lower power clock source */
        clock-state-1 = <&fcclksel0 4 &frgctrl0_mul 0>;
        clock-state-names = "default“, “sleep”;
};
```

Clock Control versus Clock Management



Framework Implementation

Clock Driver Implementation

EMBEDDED OPEN SOURCE SUMMIT

```
int syscon_clock_div_get_rate(const struct clk *clk) {
        const struct syscon_clock_div_config *config = clk->hw_config;
        int parent_rate = clock_get_rate(config->parent);
        uint8_t div_mask = GENMASK(0, (config->mask_width - 1));

        if (parent_rate <= 0) {
                return parent_rate;
        }

        /* Calculate divided clock */
        return parent_rate / ((*config->reg & div_mask) + 1);
}
```

Clock Driver Implementation

EMBEDDED OPEN SOURCE SUMMIT

```
int syscon_clock_div_configure(const struct clk *clk, const void *div) {
        const struct syscon_clock_div_config *config = clk->hw_data;
        uint8_t div_mask = GENMASK(0, (config->mask_width - 1));
        uint32_t div_val = (((uint32_t)div) - 1) & div_mask;
        int parent_rate = clock_get_rate(config->parent);
        uint32_t new_rate = (parent_rate / ((uint32_t)div));

        clock_notify_children(clk, new_rate);
        (*config->reg) = ((*config->reg) & ~div_mask) | div_val;
        return 0;
}
```

Clock Driver Implementation

EMBEDDED OPEN SOURCE SUMMIT

```
int syscon_clock_div_notify(const struct clk *clk,
                const struct clk *parent,
                uint32_t parent_rate) {
        const struct syscon_clock_div_config *config = clk->hw_data;
        uint8_t div_mask = GENMASK(0, (config->mask_width - 1));
        uint32_t new_rate = (parent_rate / ((*config->reg & div_mask) + 1));

        return clock_notify_children(clk, new_rate);
}
```

Clock Management Subsystem Implementation

- Need a method to couple clock management to clock driver API
  - Getting output rate is simple, we already have references to the clock node and an API to do so
- Configuring setpoints will be done via clock_configure API
  - Clock drivers will supply macros to define clock specific data based on node specifiers
- Output clocks and state data will be stored in struct defined by a macro in driver
  - Like pin control, this struct will be passed to clock management functions
- Clock callbacks are implemented by defining a minimal clock structure for the driver consuming a clock
  - This will be considered a child of the clock the driver references
  - Will have the “notify” API called on it when clock is reconfigured
  - Clock structure’s private data field will point to clock configuration structure

```
fcclksel0: fcclksel0@400002b0 {
                compatible = "nxp,syscon-clock-mux";
                #clock-cells = <1>;
};

&flexcomm0 {
        status = "okay";
        clock-state-0 = <&fcclksel0 3>
        clock-state-names = "default";
};


/* Defines any data structure needed for mux */
#define Z_CLOCK_MGMT_NXP_SYSCON_CLOCK_MUX_DATA_DEFINE(node_id, prop, idx)
/* Get value that will be passed to clock configure API */
#define Z_CLOCK_MGMT_NXP_SYSCON_CLOCK_MUX_DATA_GET(node_id, prop, idx) \
        DT_PHA_BY_IDX(node_id, prop, idx, multiplexer)
```

Current Status

- Basic Implementation completed for LPC55S69
  - No support for set_rate or round_rate
  - Able to configure clock tree to run basic hello world application
- Flash usage
  - Currently about 1 KB more flash than clock control framework
  - Could make notification support optional
    - Save roughly 700KB by disabling clock notifications
- RFC
  - In progress, need documentation and completed implementation

Alternative Options

EMBEDDED OPEN SOURCE SUMMIT

- Setpoints could be implemented in SOC specific manner
  - Less code reuse, but potentially more optimization
  - Similar to how pin control handles states
  - SOCs would be responsible for implementing callback architecture
  - RFC for this approach: [RFC] Introduce Clock Management Subsystem (PR #70467)
- We could implement this using the current clock control subsystem, and build clock management API on to of it
  - Reusing device struct to describe clocks has negative utilization impacts when 20+ structures are defined for basic clock trees
  - Set_rate API is not optional, limiting optimization possibilities
  - Extra API fields in each clock add to flash usage
  - No facility for rate change callbacks

nxp.com

Backup Slides

Callback Handling

EMBEDDED OPEN SOURCE SUMMIT

- Clocks must have bidirectional references
  - Parent reference needed to query rate
  - child reference needed to issue callbacks
- Can’t reference children directly- they may not exist at link time
  - We also want to avoid referencing child structs unless they are needed to save flash
  - Solution- use DT ordinals and two stage link (like device dependency handles)
  - Python scripting replaces ordinal numbers with clock handles in second link stage
  - Result is that each clock has a pointer to its parent(s), and a reference to an array of children clock handles
- Generic callback helper supplied to forward callback to all children of a clock

Enabling Clocks

- Nodes are all set to “status=okay”
- Clock nodes will all be compiled in, and linker will discard unreferenced clocks
  - Clock structures must be referenced by their children
  - The clock structure created for a clock consumer by the clock management framework will hold references to all parents
- This avoids need to explicitly enable/disable clock nodes to optimize image
- Clocks can also explicitly be set to “status=disabled” if desired
  - Allows external clock generators to be enabled or disabled by overlays
- Reuse “DT_INST_FOREACH_STATUS_OKAY” macro to define clock structures

Setting Rates with Clock Management

- All clock trees will define special clock output nodes
  - SOC agnostic method to set clock rates
- Clock management subsystem calls clock_configure with clock node specific data
  - For clock outputs, this will be a frequency
- Clock outputs will simply call clock_set_rate on parent clock, to configure a given frequency
- Same concept can be used by vendor drivers to provide a node for requesting a specific output frequency with other parameters (such as resolution or accuracy needed)
- SOC clock driver implementation is responsible for supporting set_rate API
  - SOC will be best aware of quirks and features of its clock tree
  - Allows SOC to optimize implementation however needed

```
fxcom0_output: fccom0-output {
        compatible = "clock-output";
        #clock-cells = <1>;
};

&flexcomm0 {
        status = "okay";
        clock-state-0 = <&fxcom0_output 12000000>;
        clock-state-names = "default";
};


/* Defines any data structure needed for mux */
#define Z_CLOCK_MGMT_CLOCK_OUTPUT_DATA_DEFINE(node_id, prop, idx)
/* Get value that will be passed to clock configure API */
#define Z_CLOCK_MGMT_CLOCK_OUTPUT_DATA_GET(node_id, prop, idx) \
        DT_PHA_BY_IDX(node_id, prop, idx, frequency)
```

Flash Usage Considerations

EMBEDDED OPEN SOURCE SUMMIT

- Avoiding common data fields lets linker optimize better
  - Only having a clock hardware data field works best, as many clocks don't have runtime data
- Children registering for parent callback via linker sections required usage of "keep"
  - Large flash impact
  - Key takeaway: we can't have parents reference their clock children, since we need linker to discard clock children that aren't used to save flash space
- Flash usage is main reason I did not use device infrastructure
  - A simple hello world build for LPC55S69 includes *twenty-five* clock nodes
    - The "name" field alone would increase flash usage by 100 bytes

Setpoints RFC

EMBEDDED OPEN SOURCE SUMMIT

- Used preprocessor macros to generate optimized “clock setpoint” functions for each clock state
  - Heavy preprocessor optimization, errors could be hard to debug
  - Lots of macrobatics
  - No usage of device model- single instance
- Some good discussion there- a few key issues raised:
  - How to handle external clock producers (off chip)?
    - No good option in that framework
  - How to support set rate API?
    - Could theoretically provide nodes to do so at the SOC implementation, but it would be additional code overhead