Zephyr® Project

Developer Summit

Reworking the Zephyr Clock Control Subsystem

Moritz Fischer, Google
moritzf@google.com

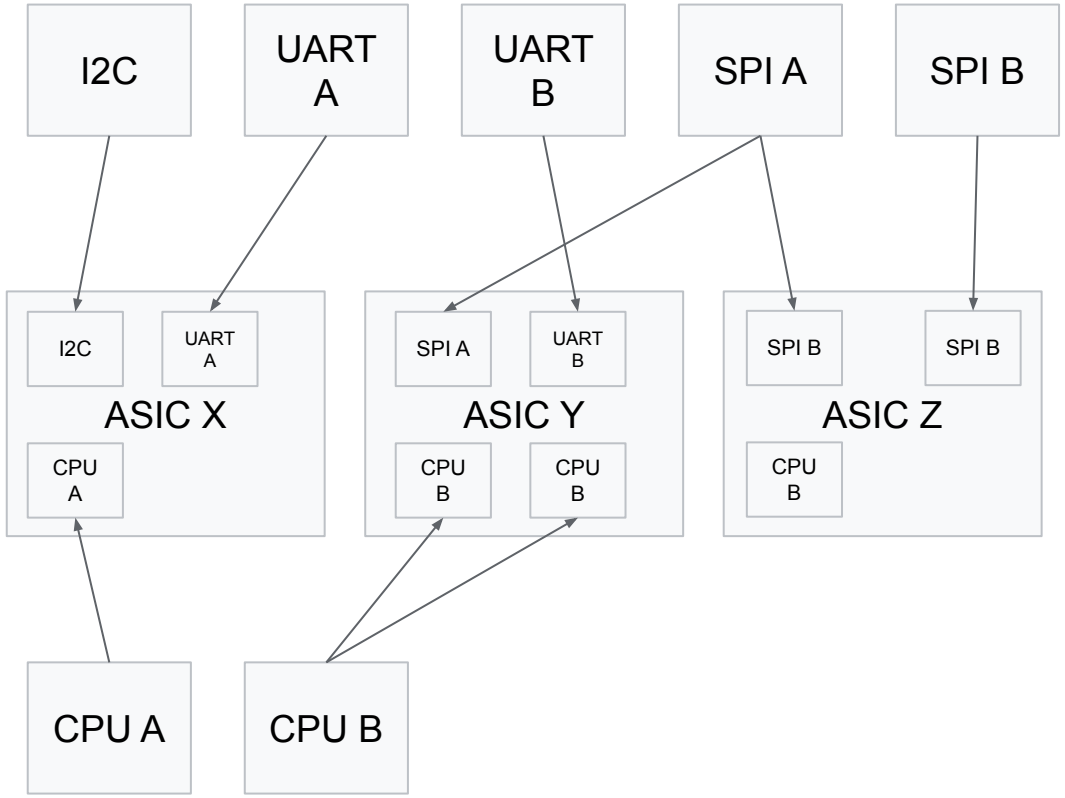
$whoami

Engineering Lead @ Google
Works on firmware for in-house ASICs
Open Source Hippie
I *hate* reinventing the wheel

We’re hiring!

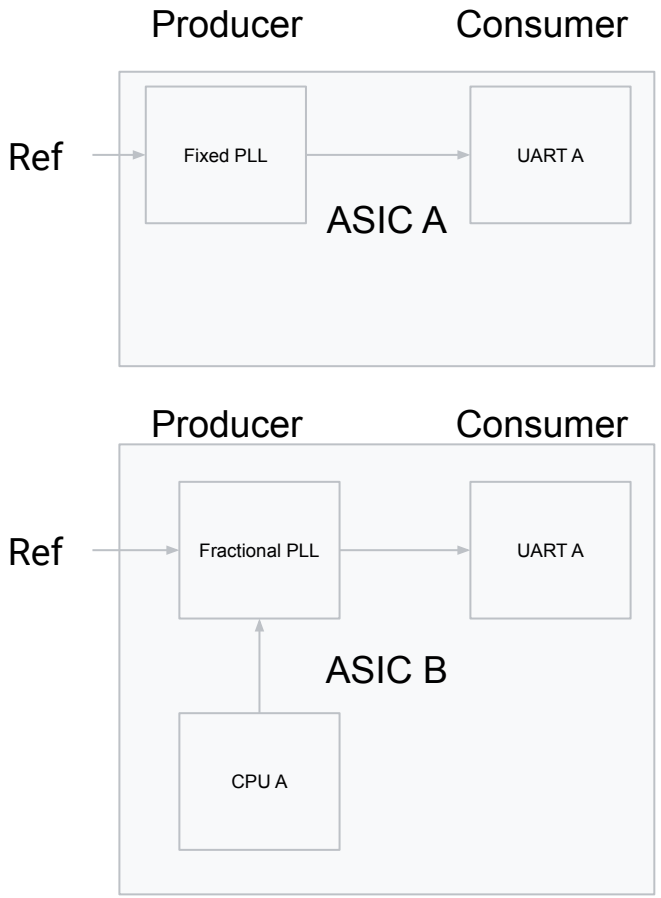
Why do I care?

- Working on ASICs means re-using IP wherever possible
- You make something that works - suddenly everyone wants to integrate it
- Every ASIC integrates IP slightly differently
- **I can't rewrite all my firmware everytime**



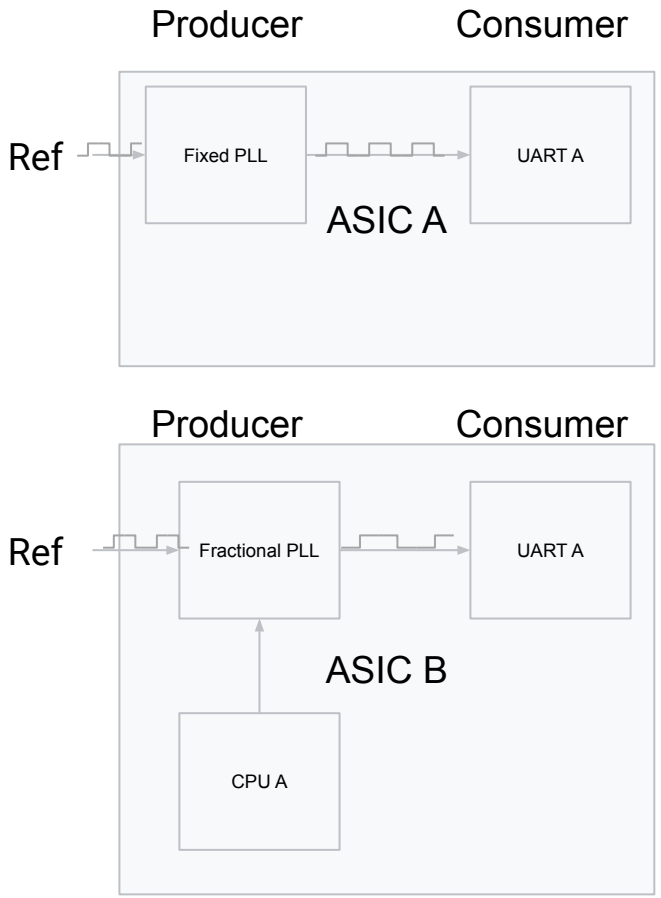
Typical integration differences (HW)

- Bus widths (32 bit / 64 bit)
- Bus type
- IP revisions
- Resets
- **Clocking**



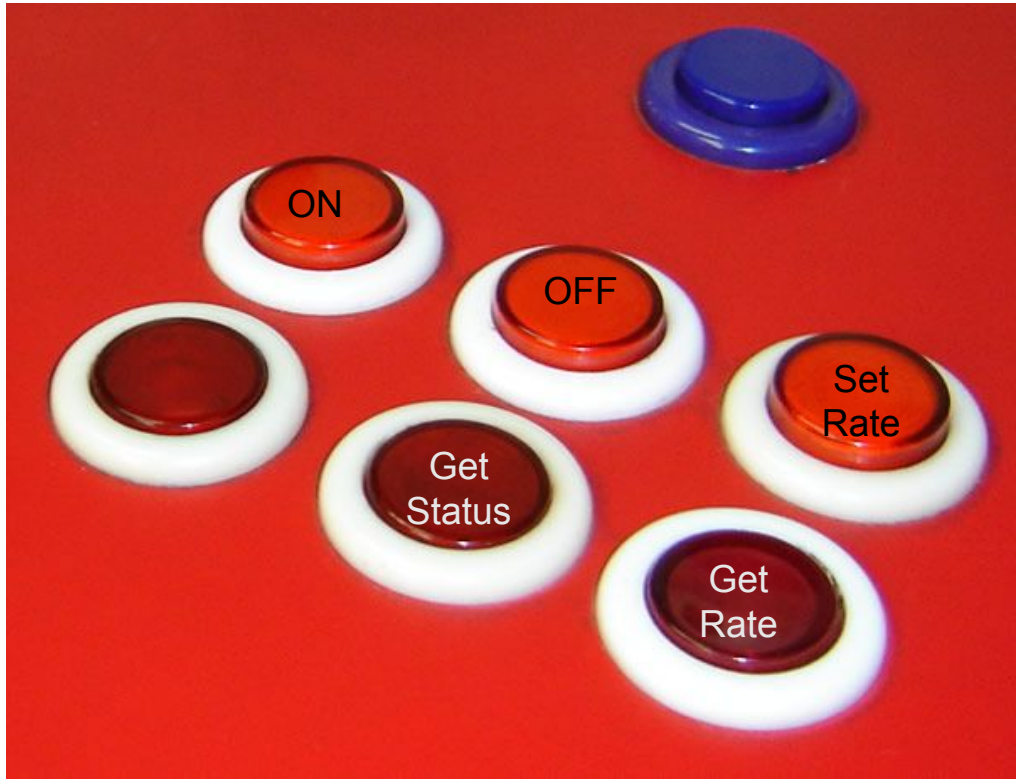
Clocks in ASICs

- *All* HW needs clocks to run
- Consumer might need info from /or control over producer
- Example:
  - UART output divider depends on input clock rate



What I’d expect from a clock API

- (ideally) Handles dependencies
- The operations are fairly simple
- **Consumers should not need to know about internals of the Producer**



Clocks in Zephyr today

- Fixed clocks
- Dynamic Clocks (Type A)
- Dynamic Clocks (Type B)

Clocks in Zephyr today - Fixed Clocks

- We **don't use the API**, we just grab the value using a macro
- All compile-time
- Different producers possible as long as they define clock-frequency in DT

DT

```
clk0: clk {
  clock-frequency = <1000000>;
  compatible = "fixed-clock;
  #clock-cells = <0>;
};

uart: uart@41000000 {
  compatible = "arm,pl011";
  [..]
  clocks = <&clk0>;
};
```

Driver

```
ret = pl011_set_baudrate(dev, config->sys_clk_freq,
                         data->baud_rate);
[..]
static struct pl011_config pl011_cfg_port_##n = {
       DEVICE_MMIO_ROM_INIT(DT_DRV_INST(n)),
       sys_clk_freq = DT_INST_PROP_BY_PHANDLE(n, clocks, clock_frequency),
       PINCTRL_INIT(n)
};
```

Clocks in Zephyr today - Dynamic Clocks (Type A)

- Use the API with hardcoded SoC specific data
- Consumer driver needs extra SoC specific knowledge about how to package opaque data
- No reusability without modifying consumer driver

DT

```
device_foo {
  compatible = "vendor,foo";
  [..]
  clocks = <&clk0>;
};
```

Driver

```
#include <zephyr/clock_control.h>

#define SOME_SOC_SPECIFIC_DATA 10

[..]
clock_control_on(cfg->clk_dev, (clock_subsys_t)SOME_SOC_SPECIFIC_DATA);

static struct foo_config foo_config_##n = {
       DEVICE_MMIO_ROM_INIT(DT_DRV_INST(n)),                    .
       clk_dev = DT_INST_CLOCK_CTLR_GET_BY_IDX(n, clocks, 0),
};
```

Clocks in Zephyr today - Dynamic Clocks (Type B)

- Use the API with DT encoded producer specific data
- Consumer driver needs extra producer specific knowledge about how to package opaque data
- No reusability without modifying consumer driver

DT

```
device_foo {
  compatible = "vendor,foo";
  [..]
  clocks = <&clk0 10>;
};
```

Driver

```
#include <zephyr/clock_control.h>


struct opaque_data {
       uint32_t some_cell;
};


[..]
clock_control_on(cfg->clk_dev, (clock_subsys_t)cfg->clk_data);

static const struct opaque_data foo_clock_data_##n = {
       .some_cell = DT_INST_CLOCKS_CELL_GET_BY_IDX(..., cell_name);
};
static struct foo_config foo_config_##n = {
       .base = DEVICE_MMIO_ROM_INIT(DT_DRV_INST(n)),
       .clk_dev = DT_INST_CLOCK_CTLR_GET_BY_IDX(n, clocks, 0),
       .clk_data = &foo_clock_data_##n
};
```

Clocks in Zephyr today - Code Generation (Simplified)

DT Binding of Producer

```
#clock-cells
  -    const: 1


clock-cells
  -    some-cell
```

scripts/dts/gen_defines.py

DT for Consumer

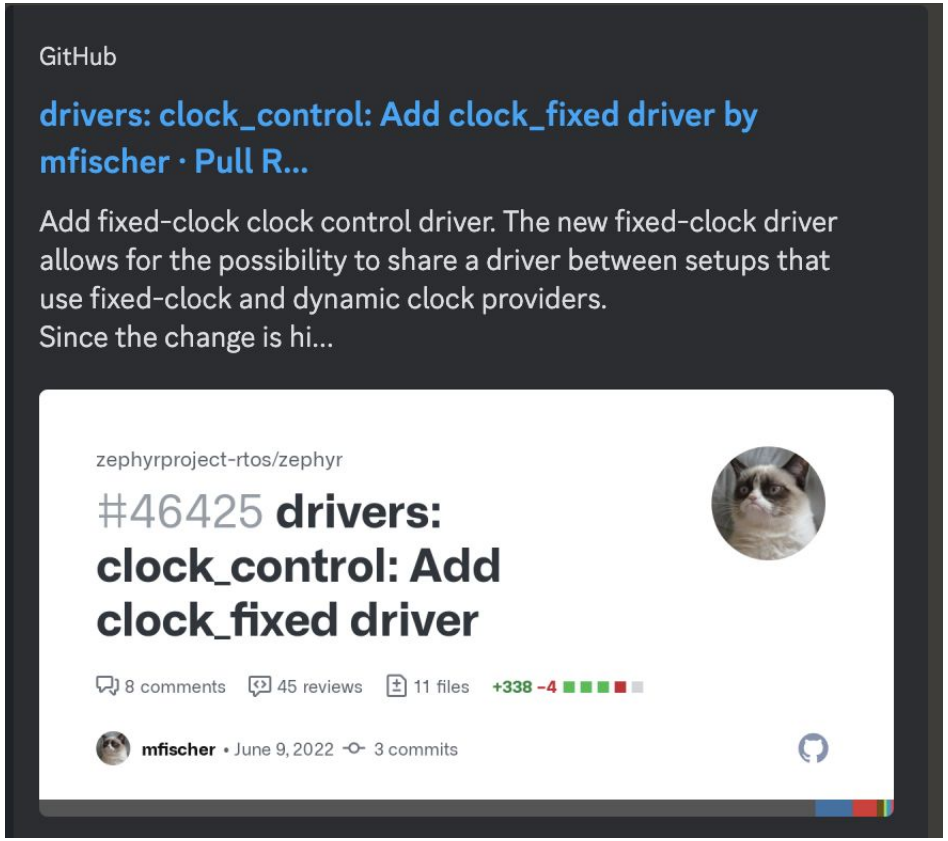
```
soc {
       clkc : clk@400000000 {
              clock-frequency = <1000000>;
              compatible = "vendor,some-clock-producer;
              #clock-cells = <1>;
       };

       uart: uart@41000000 {
              compatible = "vendor,uart-foo";
              [..]
              clocks = <&clkc 10>;
       };
};
```

Generated Header

```
/* devicetree-generated.h */

[..]

DT_N_S_soc_S_clkc_400000000

[..]

DT_N_S_soc_S_uart_41000000_P_clocks_IDX_0_EXISTS 1
DT_N_S_soc_S_uart_41000000_P_clocks_IDX_0_PH DT_N_S_soc_S_clkc_400000000
DT_N_S_soc_S_uart_41000000_P_clocks_VAL_some_cell 10
DT_N_S_soc_S_uart_41000000_P_clocks_VAL_some_cell_EXISTS 1
[..]
DT_N_S_soc_S_uart_41000000_P_clocks_LEN 1
DT_N_S_soc_S_uart_41000000_P_clocks_EXISTS 1
```

Change 1: fixed-clock needs a driver

- **Remove ‘clocks’ property**
- Add fixed-clock as a clock producer driver
- Need to get back to this



Change 2: Rethink the API

- Current API on the right

```
/* include/zephyr/drivers/clock_control.h */

int clock_control_on(const struct device *dev, clock_control_subsys_t sys);

int clock_control_off(const struct device *dev, clock_control_subsys_t sys);

int clock_control_async_on(const struct device *dev,
                           clock_control_subsys_t sys,
                           clock_control_cb_t cb,
                           void *user_data);

enum clock_control_status clock_control_get_status(const struct device *dev,
                                                   clock_control_subsys_t sys);

int clock_control_get_rate(const struct device *dev,
                           clock_control_subsys_t sys, uint32_t *rate);

int clock_control_set_rate(const struct device *dev,
                           clock_control_subsys_t sys, uint32_t rate);

int clock_control_configure(const struct device *dev,
                           clock_control_subsys_t sys, void *data);
```

Change 2: Rethink the API

- What we really want I think is a **struct clock** to operate on
- This struct clock encapsulates all info for a given clock

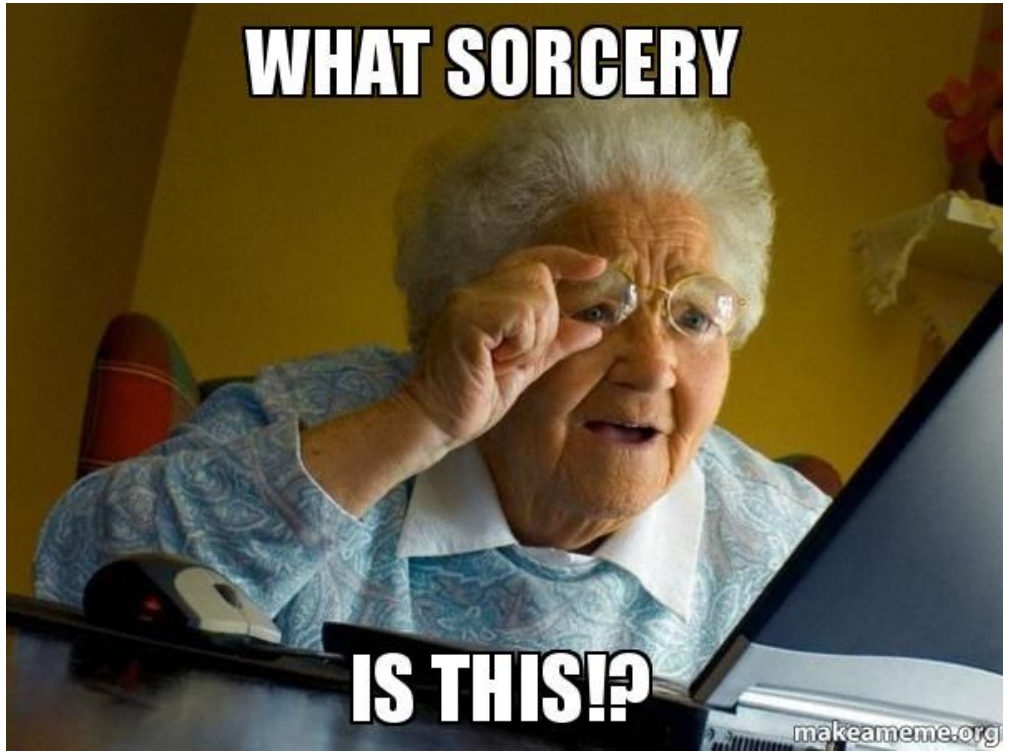
```
/* include/zephyr/drivers/clock_control.h */

+ struct clock_dt_spec;

+ struct clock {
+       const struct device *dev;
+       const struct clock_dt_spec dt_spec;
+};

- int clock_control_on(const struct device *dev, clock_control_subsys_t sys);
+ int clock_control_on(const struct clock *clk);

- int clock_control_off(const struct device *dev, clock_control_subsys_t sys);
+ int clock_control_off(const struct clock *clk);

- int clock_control_get_rate(const struct device *dev,
-                            clock_control_subsys_t sys, uint32_t *rate);
+ int clock_control_get_rate(const struct clock *clk, uint32_t *rate);

- int clock_control_set_rate(const struct device *dev,
-                            clock_control_subsys_t sys, uint32_t rate);
+ int clock_control_set_rate(const struct clock *clk, uint32_t rate);
```

Change 3: How do consumers work?

- Introduce a new set of helpers to populate a struct clock

```
/* include/zephyr/drivers/clock_control.h */

+ struct clock_dt_spec {
+       uint32_t cell_0;
+       uint32_t cell_1;
+       uint32_t cell_2;
+ };

+ struct clock {
+       const struct device *dev;
+       const struct clock_dt_spec dt_spec;
+ };

/* include/zephyr/devicetree/clocks.h */
+ #define DT_CLOCKS_GET_CLOCK_BY_IDX(node_id, idx) \
+  { \
+       .dev = DEVICE_DT_GET(DT_CLOCKS_GET_CTLR_BY_IDX(node_id, idx)), \
+       .dt_spec {
+              .cell_0 = DT_PHA_BY_IDX_OR(node_id, clocks, idx, generic_clock_cell_0, 0) \
+              .cell_1 = DT_PHA_BY_IDX_OR(node_id, clocks, idx, generic_clock_cell_1, 0) \
+              .cell_2 = DT_PHA_BY_IDX_OR(node_id, clocks, idx, generic_clock_cell_2, 0) \
+              [..]   \
+ }
```

Change 3: How do consumers work?




```
/* include/zephyr/drivers/clock_control.h */

+ struct clock_dt_spec {
+       uint32_t cell_0;
+       uint32_t cell_1;
+       uint32_t cell_2;
+ };

+ struct clock {
+       const struct device *dev;
+       const struct clock_dt_spec dt_spec;
+ };

/* include/zephyr/devicetree/clocks.h */
+ #define DT_CLOCKS_GET_CLOCK_BY_IDX(node_id, idx) \
+  { \
+       .dev = DEVICE_DT_GET(DT_CLOCKS_GET_CTLR_BY_IDX(node_id, idx)), \
+       .dt_spec {
+              .cell_0 = DT_PHA_BY_IDX_OR(node_id, clocks, idx, generic_clock_cell_0, 0) \
+              .cell_1 = DT_PHA_BY_IDX_OR(node_id, clocks, idx, generic_clock_cell_1, 0) \
+              .cell_2 = DT_PHA_BY_IDX_OR(node_id, clocks, idx, generic_clock_cell_2, 0) \
+              [..]   \
+ }
```

Change 3: How do consumers work?

- Modification to scripts/dts/gen_defines.py
- In addition to named clock cells we generated generic aliases / extra entries
- Room for improvement, on struct sizes
  - Look for largest #clock-cells in DT
  - Kconfig

```
/* include/zephyr/devicetree/clocks.h */

#define DT_CLOCKS_GET_CLOCK_BY_IDX(node_id, idx) \
  { \
       .dev = DEVICE_DT_GET(DT_CLOCKS_GET_CTLR_BY_IDX(node_id, idx)), \
       .dt_spec {
              .cell_0 = DT_PHA_BY_IDX_OR(node_id, clocks, idx, generic_clock_cell_0, 0) \
              .cell_1 = DT_PHA_BY_IDX_OR(node_id, clocks, idx, generic_clock_cell_1, 0) \
              .cell_2 = DT_PHA_BY_IDX_OR(node_id, clocks, idx, generic_clock_cell_2, 0) \
       [..]   \
  }
```

Generated Header

```
/* devicetree-generated.h */
DT_N_S_soc_S_uart_41000000_P_clocks_IDX_0_EXISTS 1
DT_N_S_soc_S_uart_41000000_P_clocks_IDX_0_PH DT_N_S_soc_S_clkc_400000000
DT_N_S_soc_S_uart_41000000_P_clocks_VAL_some_cell 10
DT_N_S_soc_S_uart_41000000_P_clocks_VAL_some_cell_EXISTS 1
DT_N_S_soc_S_uart_41000000_P_clocks_VAL_generic_clock_cell_0 10
DT_N_S_soc_S_uart_41000000_P_clocks_VAL_generic_clock_cell_0_EXISTS 1
[..]
DT_N_S_soc_S_uart_41000000_P_clocks_LEN 1
DT_N_S_soc_S_uart_41000000_P_clocks_EXISTS 1
```

Change 3: How do consumers work?

Decoupled producer and consumer?

```
struct foo_config {
        struct clock clk;
        [..]
};

int foo_init(const struct device *dev)
{
        const struct foo_config *config = dev->config;
        int err;

        err = clock_control_on(&config->clk);
        if (err) {
                return err;
        }
        [..]
}

[..]

#define FOO_INIT(n)                                                    \
        struct foo_config foo_config_#n = {                            \
                .clk = DT_CLOCKS_INST_GET_CLOCK_BY_IDX(n, 0);          \
        };
```

Opens & Discussion

- Large-ish change
- Some SoCs do not encode clock relations in DT
- What's the overhead for folks that don't care?
- How do we deal with clock dependencies?

Let’s discuss!