# Linux Generic Clock Framework

Embedded Linux Conference – Europe 2009
Grenoble, Oct 16th 2009

***Francesco Virlinzi***
*Senior software engineer*
**STMicroelectronics**
francesco.virlinzi@st.com
www.stlinux.com

# Agenda:

- Current status
- The Linux Generic Clock Framework (GCF) and its features
- The data structures used in the GCF
- Overview of clock and device state machines, description of final transaction state machine
- How the transaction graph is built
- How the devices and the drivers are involved in the clk transaction
- Future works
- Conclusions

# Current status [1/2]

- Power management is becoming one of the most important issue in embedded systems
  - Dynamic power consumption is linear with respect to clock frequency
- Clocks are shared resources in the ST System On Chip
  - From 8 clks in the *stx7100* [2005] up to 18 clks in the *stx7111* [2007] (36 clks in the *stx7108* [2009])
- Linux does not have a sufficiently powerful clock framework
  - Linux has only a generic API (`<include/linux/clk.h>`)
- A lot of architectures create '*ad-hoc*' clock frameworks...

# Current status [2/2]

**Unfortunately**:

- Several '*arch*' clock frameworks fail to involve the devices during a clock operations

- None of the current '*arch*' clock framework are integrated into the Linux Device Model

- Any clock change may break a working device

# The Linux Generic Clock Framework features

- Written to be *arch* independant
- Integrated into the Linux Device Model
  - there is no `clk_register_device(..)`
- Provides sysfs interface; the user can
  - navigate `/sys/clocks/...` to analyse the status of clock tree;
  - check which devices are using any given clock
- Involves the platform_devices during the clock rate propagation
- Uses current Linux API

# Clocks, ... [1/3]

```
struct clk {
     struct kobject     kobj;
     struct kobject     *kdevices;
     int                id;
     const char         *name;
     struct module      *owner;
     struct clk         *parent;
     struct clk_ops     *ops;
     void               *private_data;
     unsigned long      rate;
     unsigned long      flags;
     unsigned long      nr_active_clocks;
     unsigned long      nr_active_devices;
     unsigned long      nr_clocks;

     void               *towner;
     struct klist       childs;
     struct klist       devices;
     struct klist_node  node;
     struct klist_node  child_node;
};
```

Each physical clock is managed through '**struct clk**' object which tracks:

- The clock relationship
- The clock-devices relationship
- How many child clocks are active
- How many child devices are active
- If the clock is undergoing transaction

Each clock is registered through `clk_register( )` (or early in the boot through `early_clk_register( )`)

Used during runtime

Used to manage the relationship

# ..., Clock Operations and … [2/3]

Each clock defines the operations it supports using a set of SOC specific callback collected in the ***struct clk_ops***.

Those are the entry points for any hardware access.

```
struct clk_ops {
    int (*init)(struct clk *);
    int (*enable)(struct clk *);
    int (*disable)(struct clk *);
    int (*set_rate)(struct clk *, unsigned long value);
    int (*set_parent)(struct clk *clk, struct clk *parent);
    void (*recalc)(struct clk *);
    unsigned long (*round)(struct clk *, unsigned long value);
    unsigned long (*eval)(struct clk *, unsigned long parent_rate);
    void (*observe)(struct clk *, void *);
    unsigned long (*measure)(struct clk *);
};
```

# … and Platform_objects [3/3]

```
struct platform_device {
     ...
#ifdef CONFIG_GENERIC_CLK_FM
     unsigned long            clk_flags;
     unsigned long            num_clks;
     struct pdev_clk_info    *clks;
#endif
 };


struct pdev_clk_info {
     struct platform_device *pdev;
     struct clk              *clk;
     struct klist_node       node;
};


struct platform_driver {
     ...
#ifdef CONFIG_GENERIC_CLK_FM
     int (*notify)(unsigned long code,
struct platform_device *, void *);
#endif
 };
```

Each platform_device can declare '*how many*' and '*which*' clocks it uses through the ***struct pdev_clk_info.***

The platform_driver has a new callback to notify:

- devices undergoing clock transaction to the driver;
- the clock environment the device will have

# A tipical usage mode

## All the devices are bound to the clock in setup-SOC.c file

```
static struct platform_device asc_device = {
    .name            = "stasc",
    ...
    .num_clks        = 1,
    .clks            = (struct pdev_clk_info []) {
                 {
                     .clk = &clk_IC_IF_100,
                 },
             },
};
```

## ... and in the driver....

```
static struct platform_driver asc_serial_driver = {
        .probe          = asc_serial_probe,
        .remove         = __devexit_p(asc_serial_remove),
        .driver = {
                .name   = DRIVER_NAME,
                .owner  = THIS_MODULE,
                 ...
        },
        .notify = asc_notify,
};
```

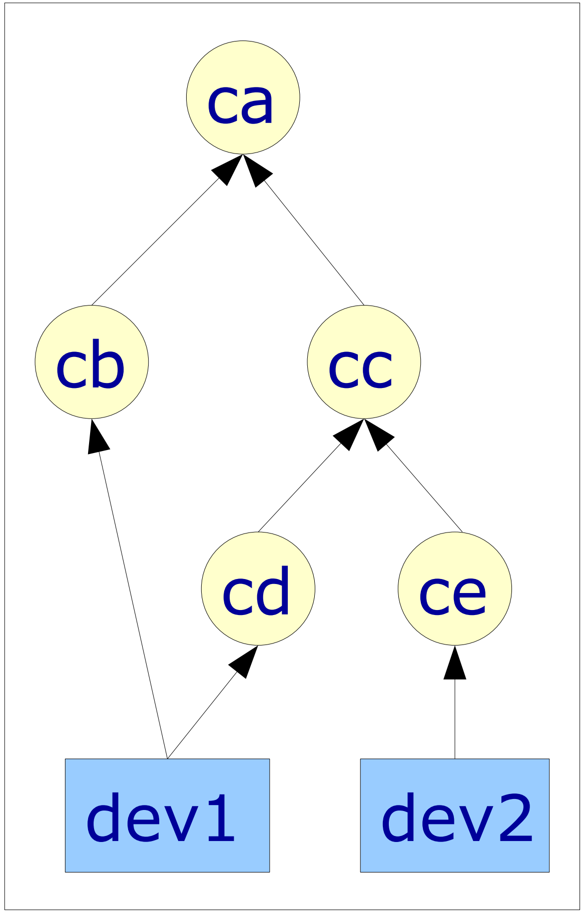
# During runtime...

The Generic Linux Clock framework has a sysfs interface to provide a lot of information about each clock

```
root@mb618:/sys/clocks/clkgena_clk_osc/clkgena_pll1_clk/ic_if_100# ls
clk_attribute  devices  module_clk

root@mb618:/sys/clocks/clkgena_clk_osc/clkgena_pll1_clk/ic_if_100# ls devices/
stasc.0  stasc.1  lirc_stm  i2c_st.0  i2c_st.1  spi_st.0  spi_st.1

root@mb618:/sys/clocks/clkgena_clk_osc/clkgena_pll1_clk/ic_if_100# ls clk_attribute
control  parent  rate  state

root@mb618:/sys/clocks/clkgena_clk_osc/clkgena_pll1_clk/ic_if_100# cat clk_attribute/rate
100000000

root@mb618:/sys/clocks/clkgena_clk_osc/clkgena_pll1_clk/ic_if_100# cat clk_attribute/state
clock name: ic_if_100
 + enabled
 + rate writable
 + allow_propagation
 + nr_clocks:  1
 + nr_active_clocks:  1
 + nr_active_devices:  4
 + rate: 100000000
```

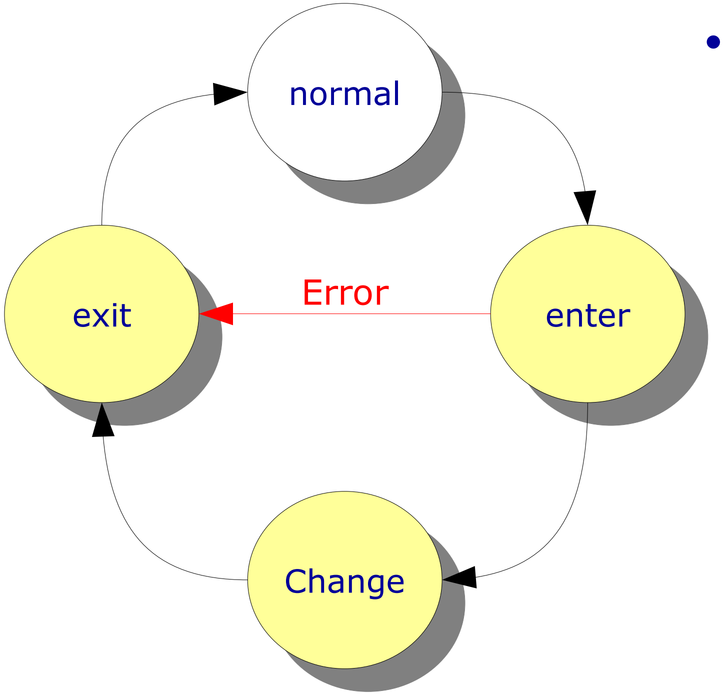
# Clock operation and clock transaction

A generic clock tree:



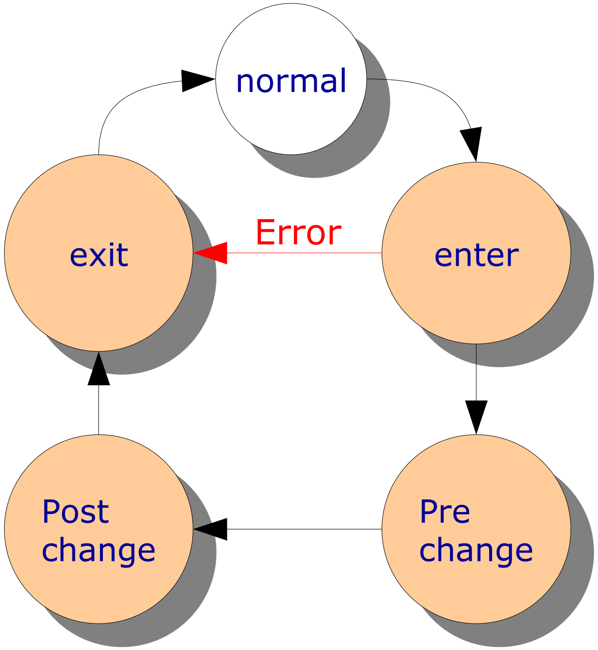
- Every clock operation is seen as a clock transaction
- The main actors during the transaction are:
  - clocks
  - devices
- The clock framework is able to:
  - Ensure the correct evolution for clocks
  - Ensure the correct evolution for devices
    - The devices can check the clk environment they will have at the end of a transaction
    - Ensure the correct device integrity

# Clk transaction: Clock state machine evolution



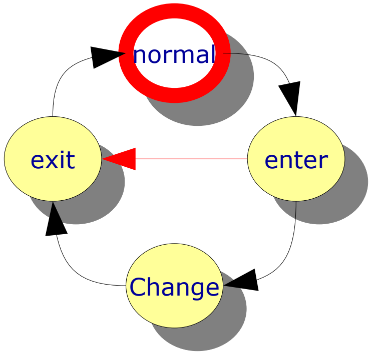
- Clock not undergoing transaction are in ***normal*** state;
- During an operation a clock can be either in:
  - ***enter*** state: where the clock is locked and the transaction graph is built
  - ***change*** state: where the clock is changed
  - ***exit*** state: where the transaction memory is freed and the clock is unlocked

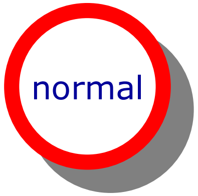
# Clk transaction: Device state machine evolution



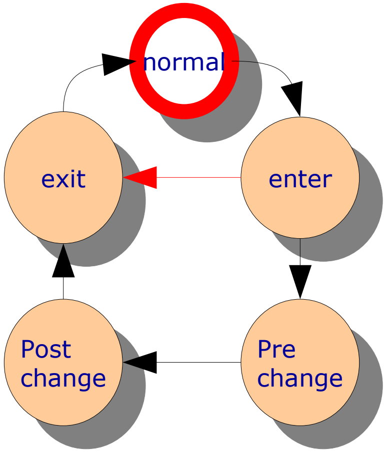
- Devices not undergoing transaction are in ***normal*** state;
- During an operation a devices can be either in:
  - ***enter_change*** state: where they can accept the clock change
  - ***pre_change*** state: where they could be suspended
  - ***post_change*** state: where they could be resumed
  - ***exit_change*** state: where they are aware the transaction is completed

# Clk transaction: Building the final fsm [0/7]

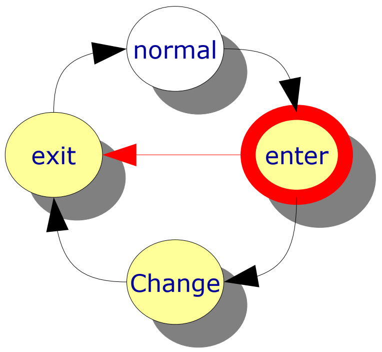


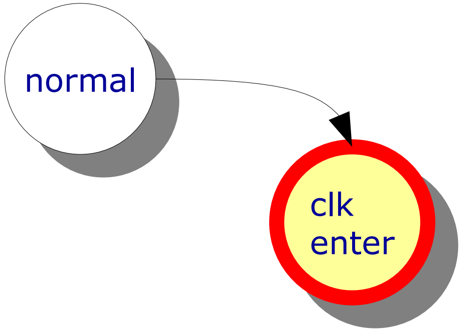


The system is running and
No transaction is ongoing



# Clk transaction: Building the final fsm [1/7]

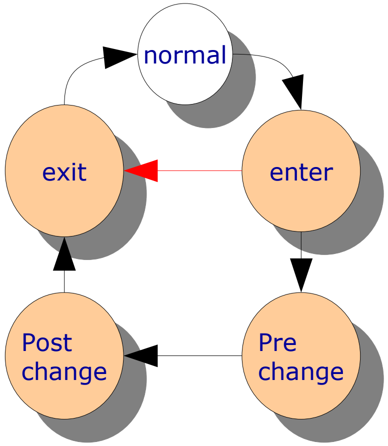




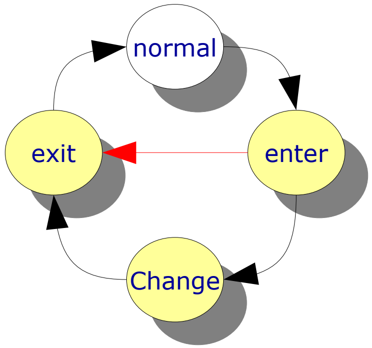
On a clock operation (i.e.: `clk_set_rate(...)`) the transaction begins;

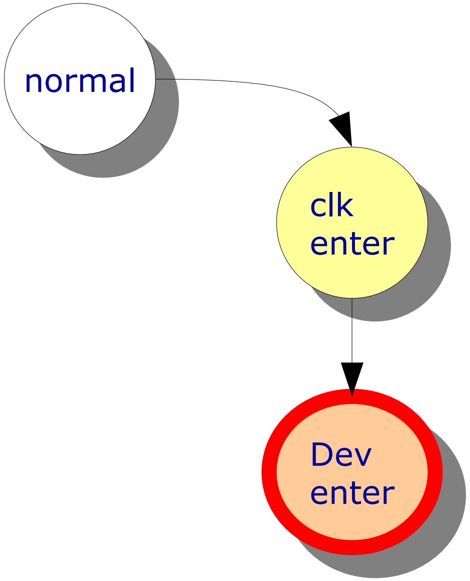
The GCF:

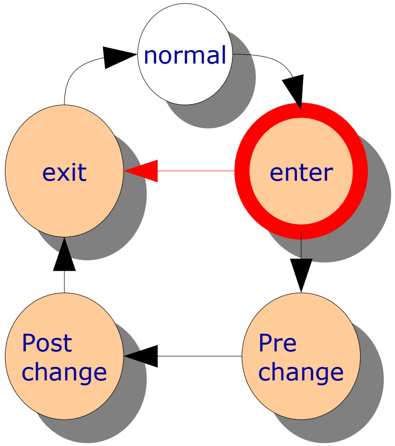
- acquires all the clocks it needs.
- creates the sub node transaction and evaluates all the clock rates.



# Clk transaction: Building the final fsm [2/7]

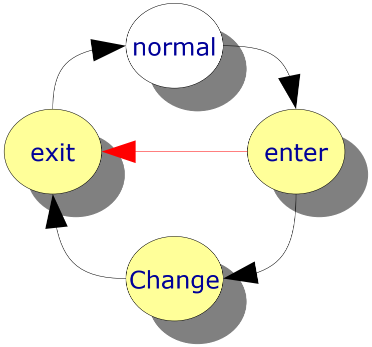


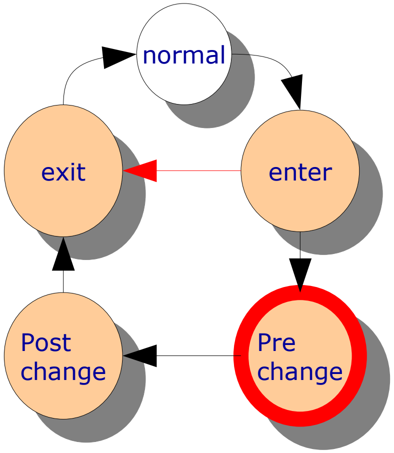


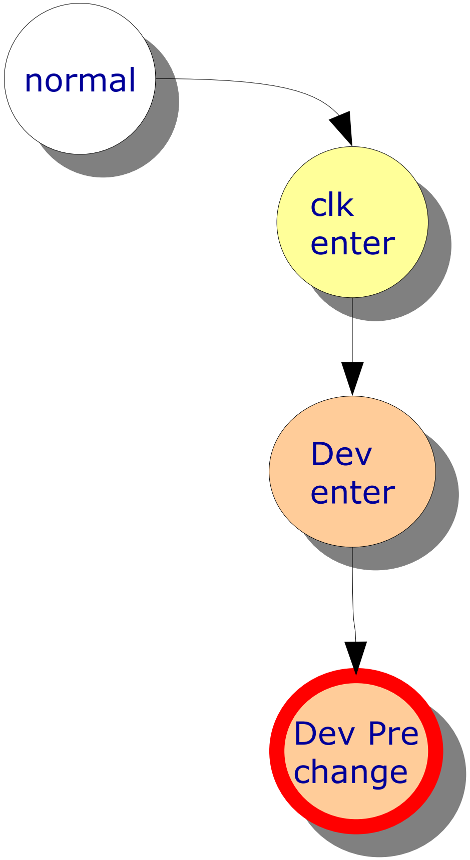


The GCF notifies all devices about the on going transaction and checks if they agree to the new clock settings

# Clk transaction: Building the final fsm [3/7]

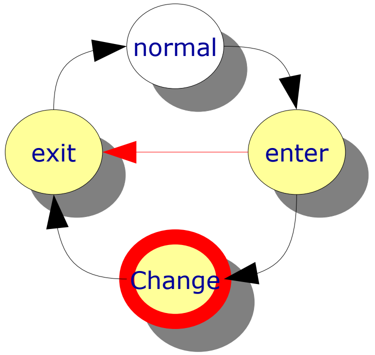


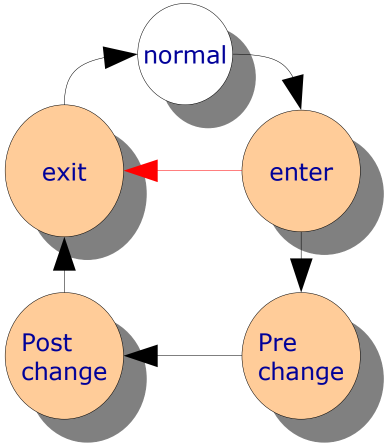


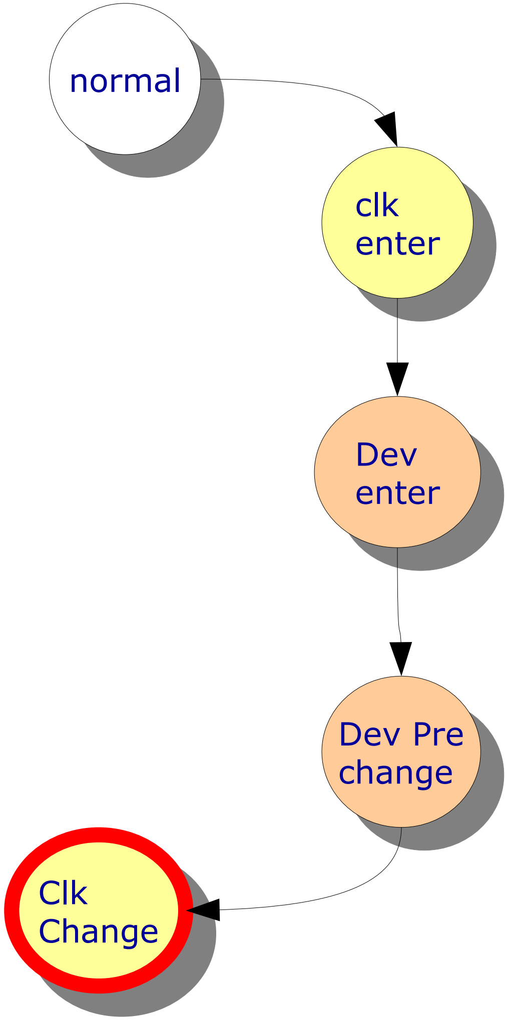


The GCF notifies the devices the framework is going to change the clocks then if required the GCF suspends the devices

# Clk transaction: Building the final fsm [4/7]

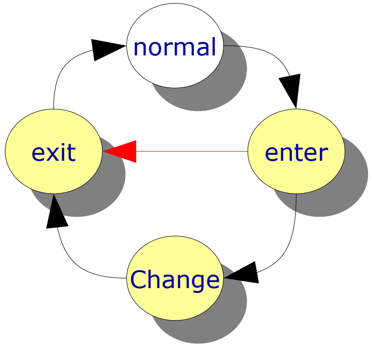


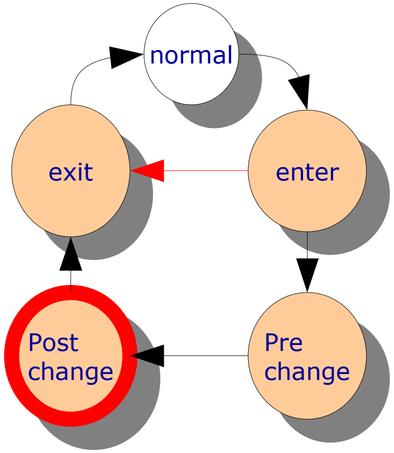


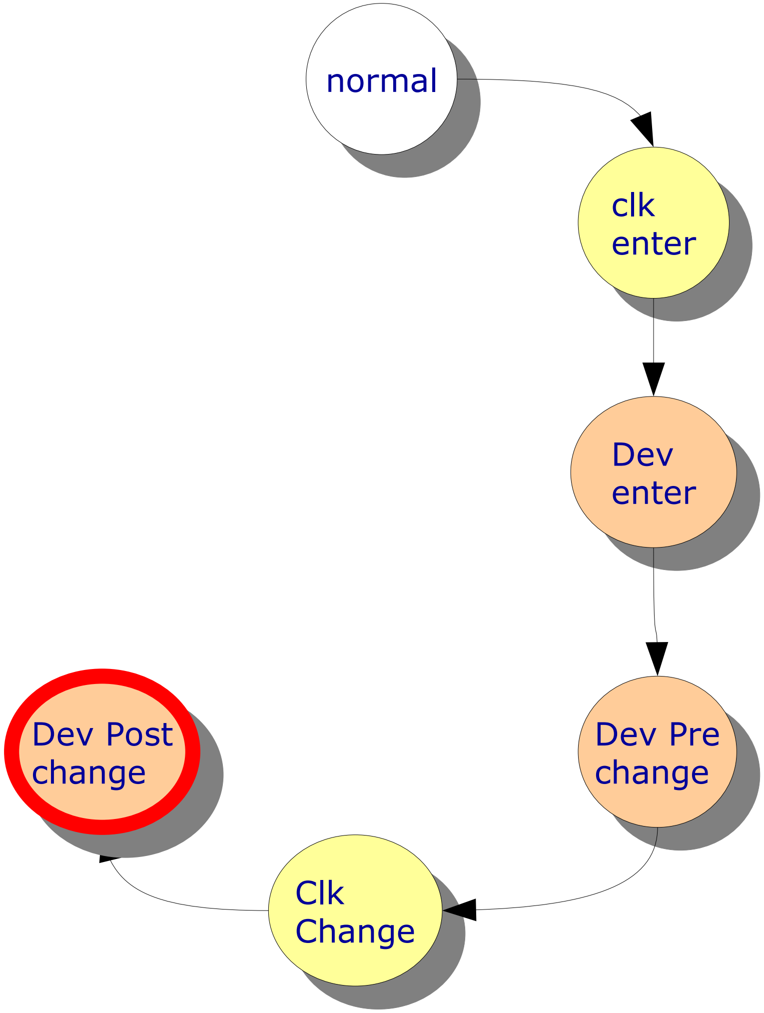


The GCF changes all the clocks

# Clk transaction: Building the final fsm [5/7]

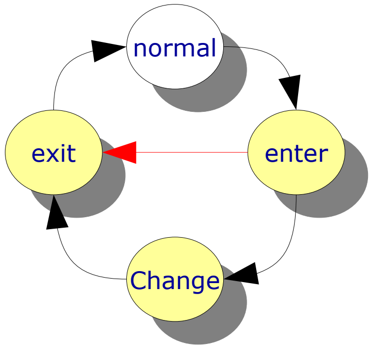


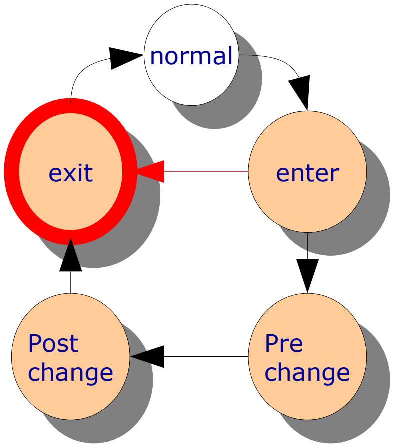


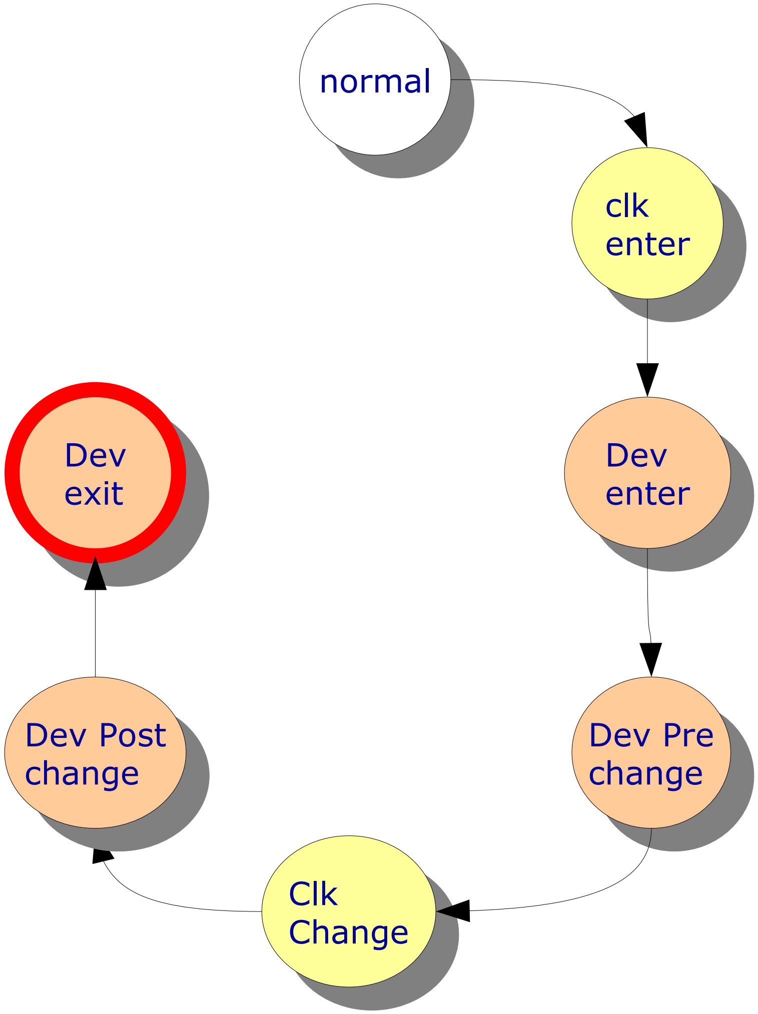


The GCF notifies the devices the clocks have changed, then if required the GCF resumes the devices

# Clk transaction: Building the final fsm [6/7]

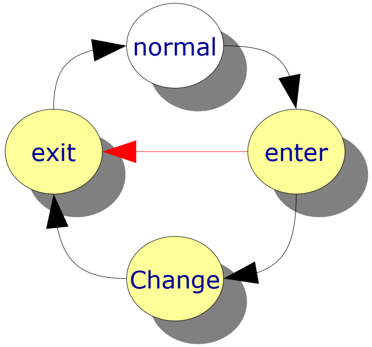


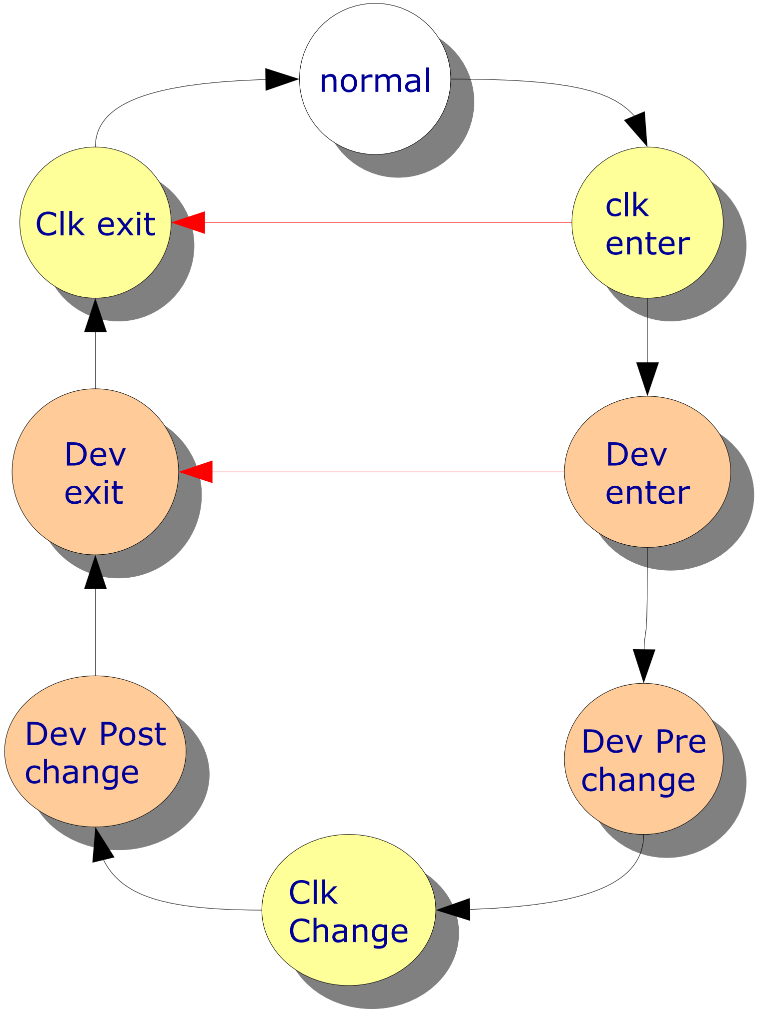




Each device is aware all the other devices were resumed and fully running

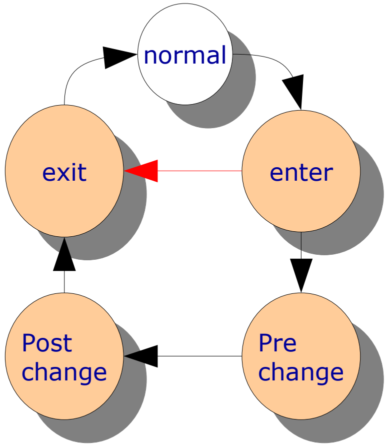
# Clk transaction: Building the final fsm [7/7]



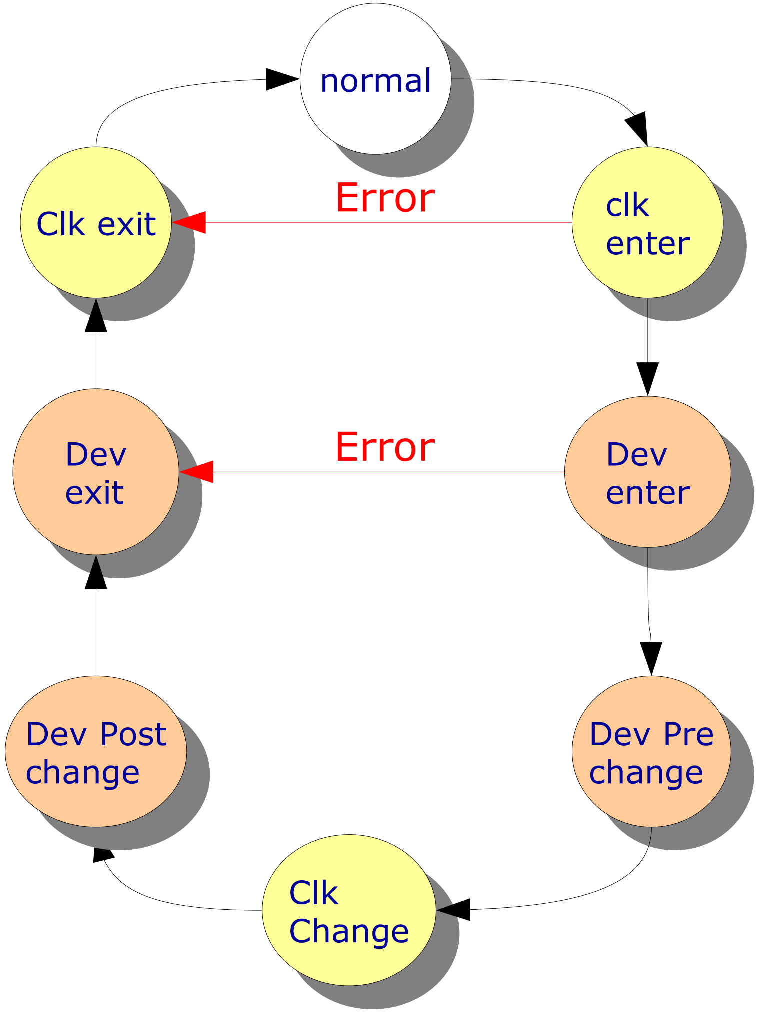


The transaction is complete.

The clocks and memory are released.



# Clk Framework Transaction Evolution

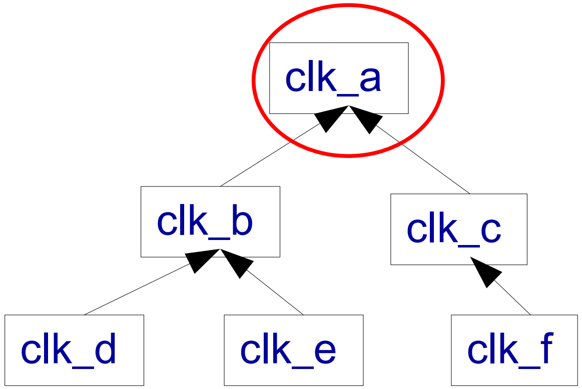


The transaction State machine provides seven states to cover both the clock and device requirement.

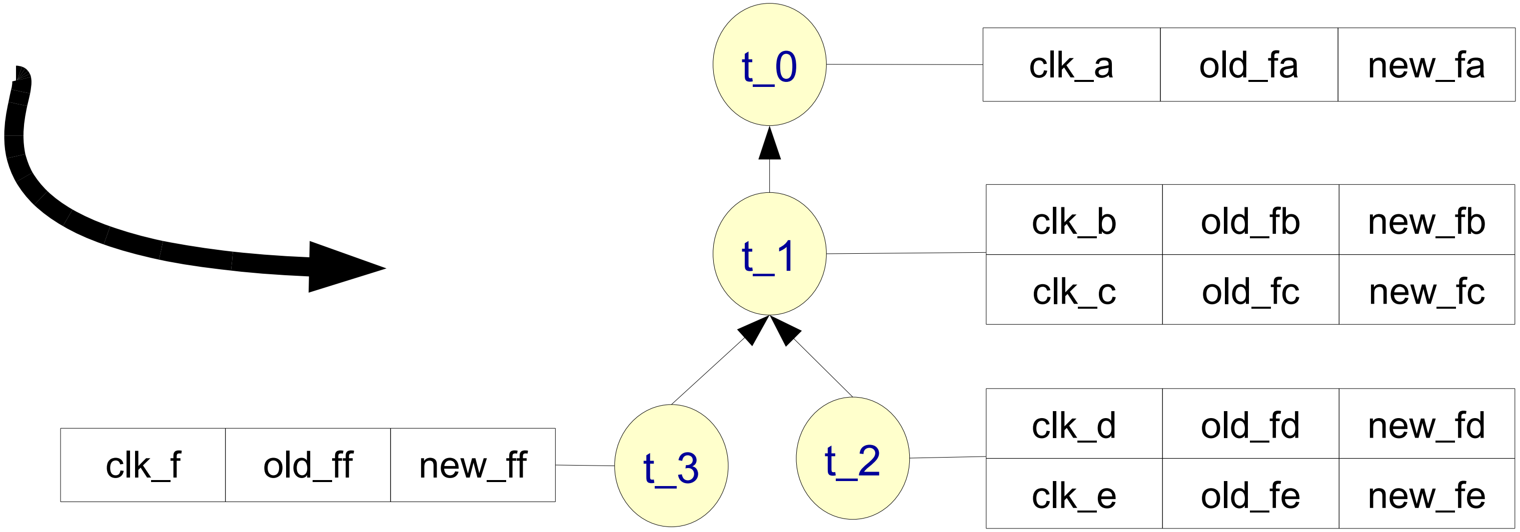
*The 'clk'* states are not visible to the devices and are managed internally to the framework.

# From clock graph to a transaction graph [1/2]

```
clk_set_rate(clk_a, xxx);
```



- The Transaction graph (usually) follows the clock hierarchy.
- It's built during the *clk_enter* state
- Each clock is marked by the node owner
- Each node can manage more than one clock
- Only the root node is on the process stack all the children are built dynamically



t_0:

| | | |
|---|---|---|
| clk_a | old_fa | new_fa |

t_1:

| | | |
|---|---|---|
| clk_b | old_fb | new_fb |
| clk_c | old_fc | new_fc |

t_2:

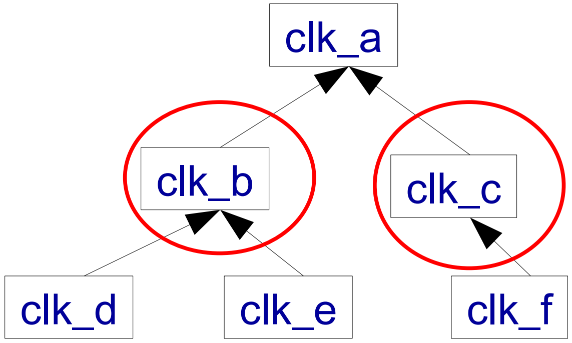
| | | |
|---|---|---|
| clk_d | old_fd | new_fd |
| clk_e | old_fe | new_fe |

t_3:

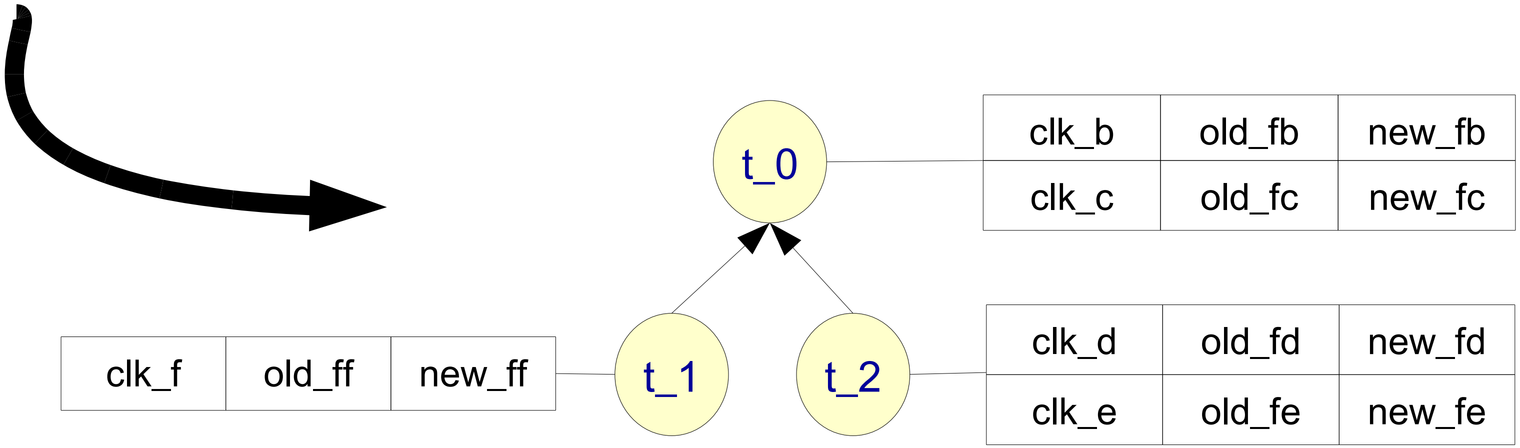
| | | |
|---|---|---|
| clk_f | old_ff | new_ff |

# From clock graph to a transaction graph [2/2]

```
clk_set_rates(**clks, *rates);
```



- The new function `clk_set_rates` can change more that one clock in a single transaction.
- The transaction graph is built according to the involved clocks



| | | |
|---|---|---|
| clk_b | old_fb | new_fb |
| clk_c | old_fc | new_fc |

| | | |
|---|---|---|
| clk_f | old_ff | new_ff |

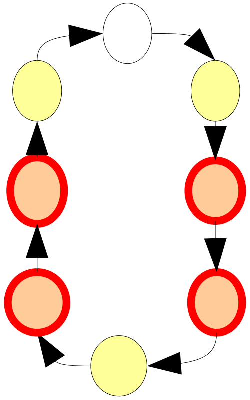
| | | |
|---|---|---|
| clk_d | old_fd | new_fd |
| clk_e | old_fe | new_fe |

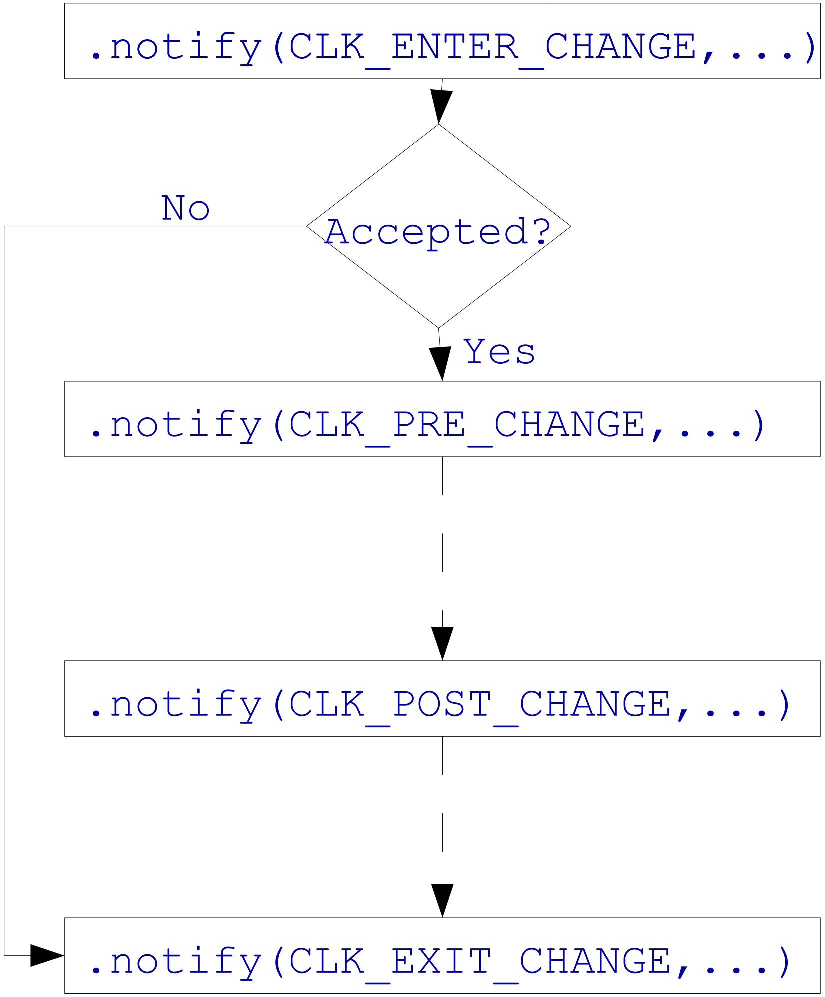
# Device driver point of view [1/3]

Only 4 clock transaction states are visible to the device drivers.

The information in the transaction graph is used to build an *ad-hoc* clk_event array for each device.

The *.notify* callback (in the platform_driver) is used to notify the driver of state machine evolution.





# Device driver point of view [2/3]

The GCF uses `.notify` return value to:

- check whether the device accepts or not the clock operation
- suspend and/or resume the device as requested

| | | Notified clk fsm code | | | |
|---|---|---|---|---|---|
| | | CLK_ENTER_CHANGE | CLK_PRE_CHANGE | CLK_POST_CHANGE | CLK_EXIT_CHANGE |
| .notify return value | NOTIFY_EVENT_HANDLED | Accept | Suspend the device | Resume the device | No action |
| | NOTIFY_EVENT_NOTHANDLED | Refuse | No action | No action | No action |

# Device driver point of view [3/3]

```
int asc_notify(unsigned long code, struct platform_device *pdev, void *data)
{
    struct clk_event *event = (struct clk_event *)data;

    switch (code) {
    case NOTIFY_CLK_ENTERCHANGE:
        return NOTIFY_EVENT_HANDLED;                    /* to accept */

    case NOTIFY_CLK_PRECHANGE:
        if (!event->old_rate && event->new_rate)   /* clk enable*/
            return NOTIFY_EVENT_NOTHANDLED;
        return NOTIFY_EVENT_HANDLED;                    /* to suspend */

    case NOTIFY_CLK_POSTCHANGE:
        if (event->old_rate && !event->new_rate)   /* clk disable */
            return NOTIFY_EVENT_NOTHANDLED;
        return NOTIFY_EVENT_HANDLED;                    /* to resume */

    case NOTIFY_CLK_EXITCHANGE:
        return NOTIFY_EVENT_HANDLED;
    }
    return NOTIFY_EVENT_HANDLED;
}
```

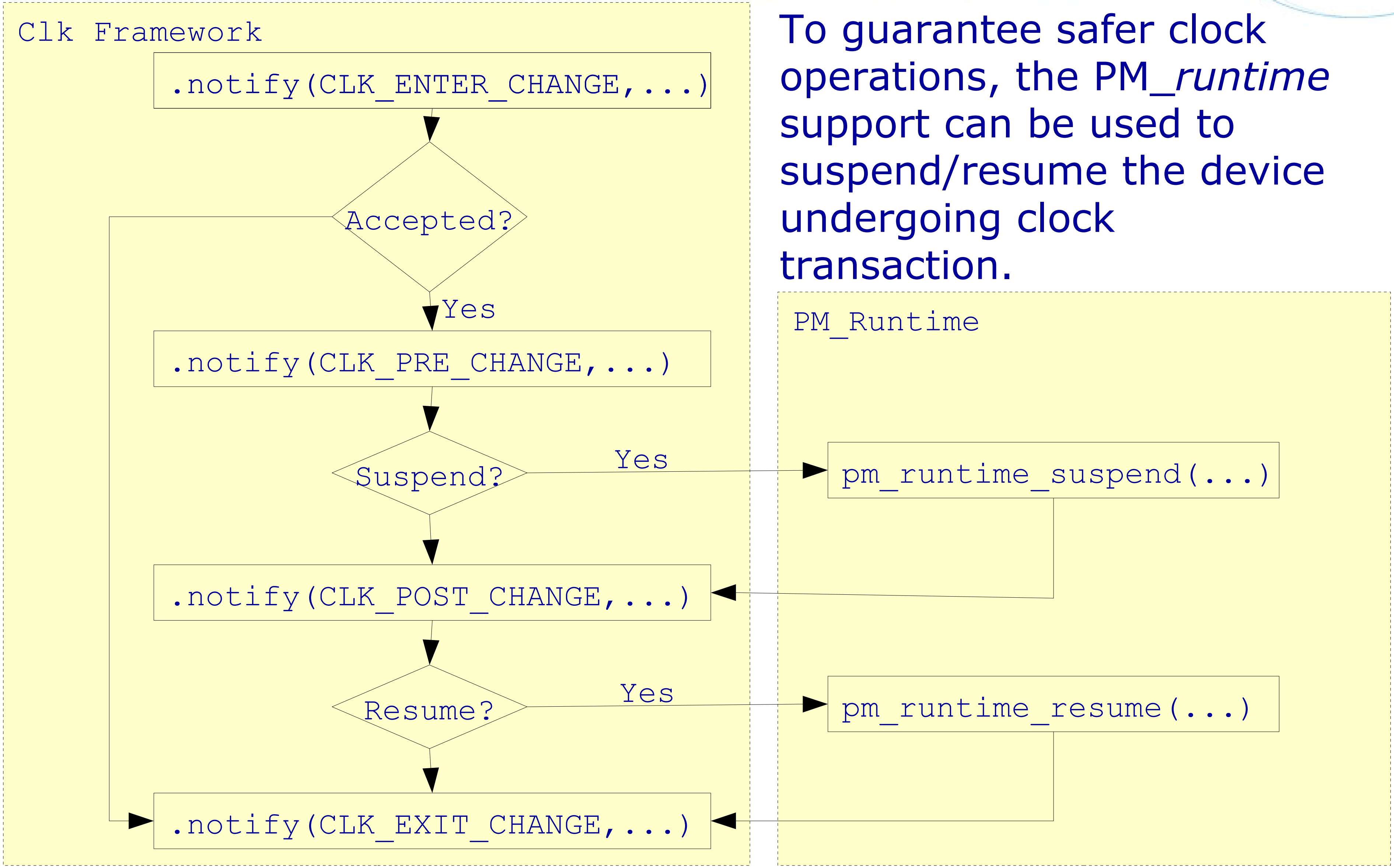
# Future works [1/5]

Several areas can be investigated:

- Integration with *PM_runtime* kernel subsystem:
  - Inside the clk transaction for safer clk propagation;
  - Outside the clk transaction to manage clocks on the fly;
- Add device constraints:
  - To fine-tune clock rates

# Future works [2/5]

To guarantee safer clock operations, the PM_*runtime* support can be used to suspend/resume the device undergoing clock transaction.



Clk Framework

.notify(CLK_ENTER_CHANGE,...)

Accepted?

Yes

.notify(CLK_PRE_CHANGE,...)

Suspend?

Yes

.notify(CLK_POST_CHANGE,...)

Resume?

Yes

.notify(CLK_EXIT_CHANGE,...)

PM_Runtime

pm_runtime_suspend(...)

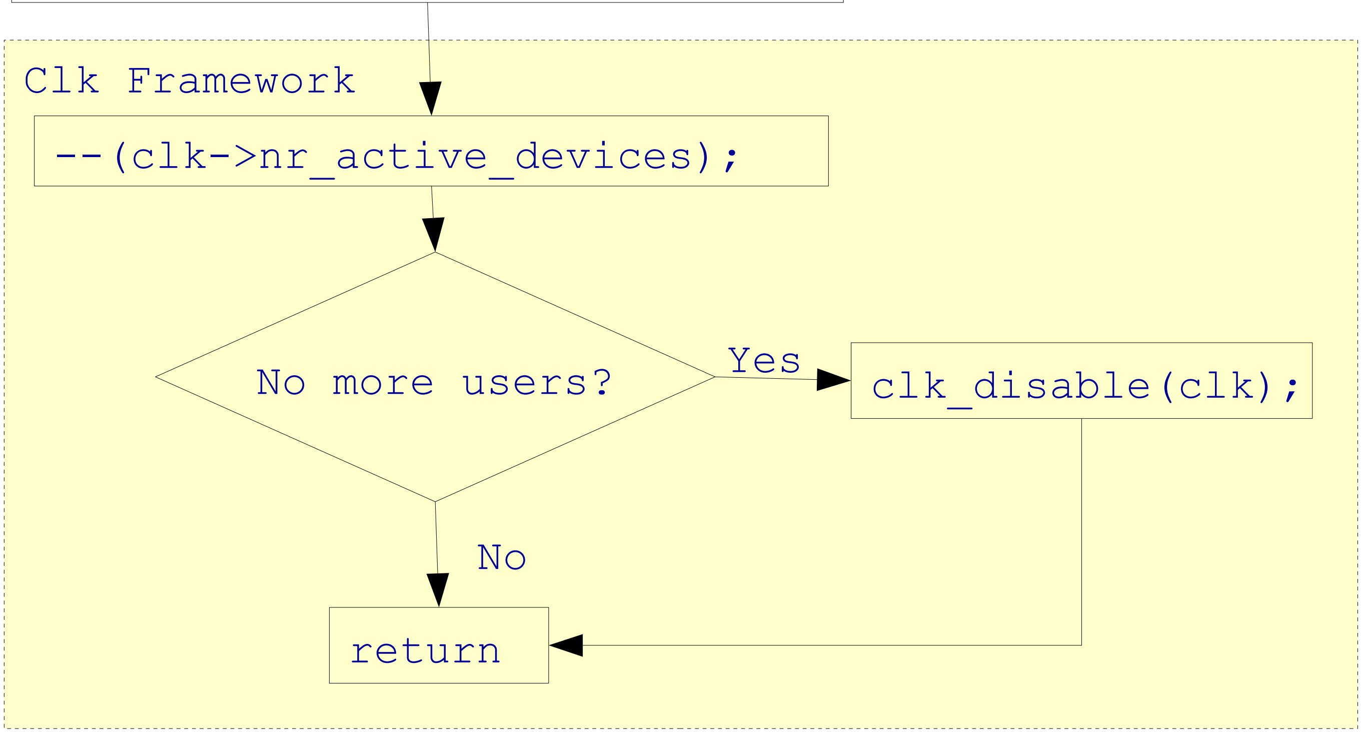
pm_runtime_resume(...)

# Future works

[3/5]

```
pm_runtime_suspend(...){
       ...
dev->pm->runtime_suspend(...);
       ...
clk_pm_runtime_notify(...);
}
```

When a device is suspended, GCF turns-off the clock (if possible)



Clk Framework

`--(clk->nr_active_devices);`

No more users?

Yes

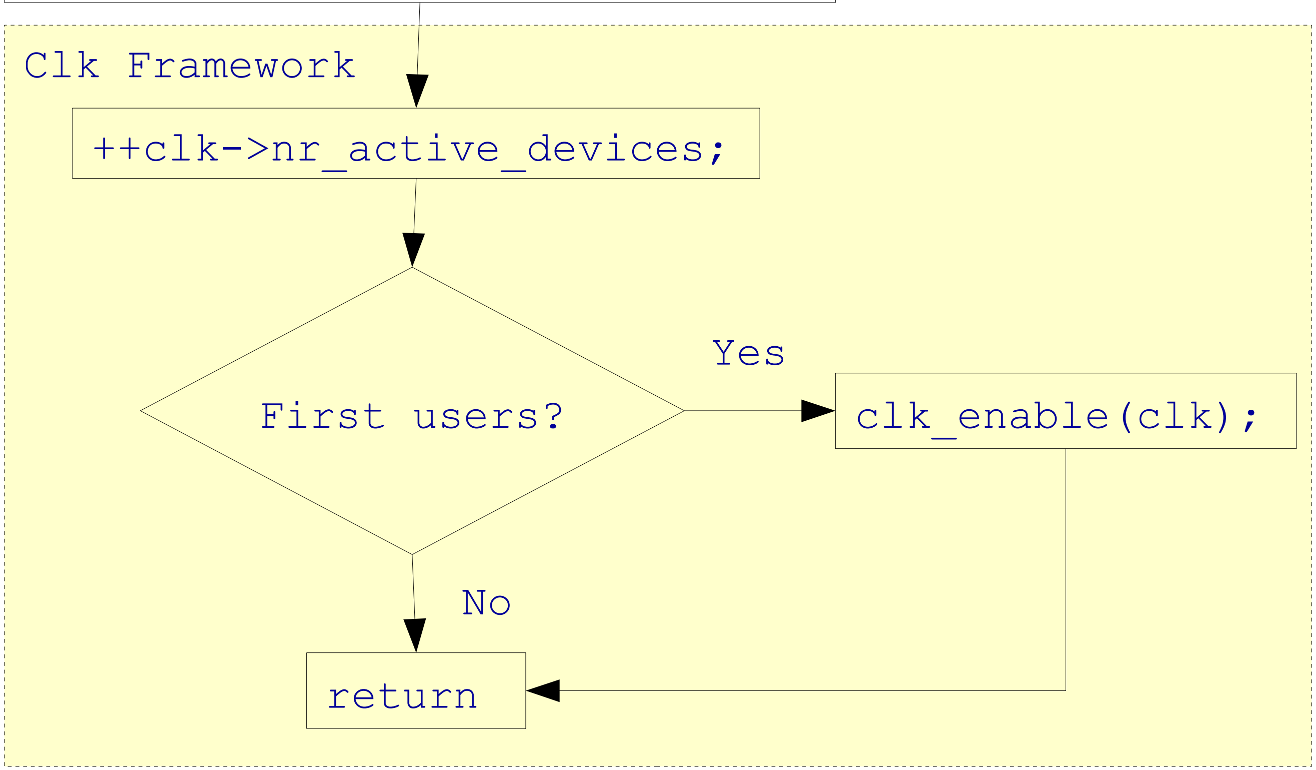
`clk_disable(clk);`

No

`return`

# Future works [4/5]

```
pm_runtime_resume(...){
          ...
clk_pm_runtime_notify(...);
          ...
dev->pm->runtime_resume(...);
}
```



Clk Framework

`++clk->nr_active_devices;`

First users?

Yes

`clk_enable(clk);`

No

`return`

When a device is resumed, the GCF turn-on the clock (if required)

# Future works [5/5]

Device constraints:

- A new `dev_clk_constraint` object could be added to each device to define the operating
    - frequency range and/or
    - fixed frequency

- To reduce power consumption, for each clock, the GCF can evaluate and set the lowest frequency based on the currently active devices.

# Clk API: mainly from <linux/clk.h>

```
int clk_register(struct clk *clk);

int clk_unregister(struct clk *clk);

int clk_enable(struct clk *clk);

int clk_disable(struct clk *clk);

int clk_get_rate(struct clk *clk);

int clk_set_rate(struct clk *clk, unsigned long rate);

int clk_set_parent(struct clk *clk, struct clk *pclk);

int clk_set_rates(struct clk **clk, unsigned long *rate);

int clk_for_each(int (*fn)(struct clk *, void *), void *);
```

# Conclusions:

- The GCF runs on both 2.6.23 and 2.6.30 kernel;
- Uses the Linux API;
- No code in the GCF uses *arch* specific features;
- Involves the devices and the drivers in the clk propagation;
- New `.notify` function easy to implement;

# The Linux Generic Clock Framework

***Thanks !***

***Q & A***