So you want to write a Linux driver subsystem?

Michael Turquette <mturquette@baylibre.com>

Who am I? And why am I here?

CEO of BayLibre, Inc

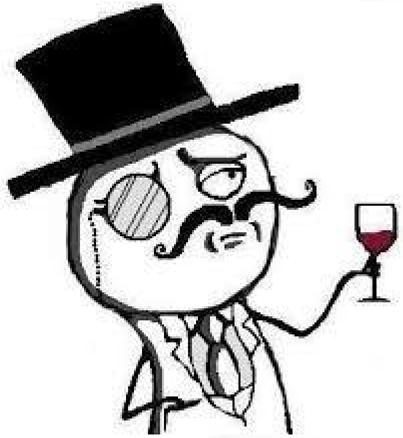
- Previously at Texas Instruments, Linaro, San Francisco start-up
- Contributor to various power management-related topics upstream

Author and co-maintainer of the common clk framework

- Merged in 3.4
- Maintenance for 3 years (and counting)

Lots of mistakes, rewrites and lessons learned.

Highly opinionated material!



Agenda

1. Overview: what makes a good subsystem?
2. Design considerations
3. (Very) Brief review of CCF design
4. Maintenance

1. What makes a good Linux driver subsystem?

What is a Linux driver subsystem?

- **Frameworks** and **libraries**; common code implementing a standard protocol, interface or behavior
- **Providers** are Linux kernel drivers that plug into the framework and **provide** access to hardware
- **Consumers** are Linux kernel drivers or subsystems that access the framework through a common API
- A Linux driver can be both a **provider** and a **consumer**

Some common subsystems...

- genirq
- clocksource
- clockevent
- pinctrl
- regulator
- clk
- cpufreq
- cpuidle
- pm runtime
- genpd
- alsa/asoc
- v4l2

What makes a good subsystem?

Solves for the most common cases

Maintainable

Concurrency / locking correctness

Respects the Driver Model

Architecture/platform independent

Module safe

Continuous testing

Write good code

Use best programming practices

Consolidate code

Provide helpers and accessors as needed

Use coccinelle to find bad patterns and fix them

Read an algorithm and data structures book

2. Design considerations

Patterns and pitfalls

1. Consider the sanity of your API
2. Consumer/provider API split
3. Consumers should not know about the hardware
4. Device versus Resource
5. Follow the Linux Driver Model
6. Locking and concurrent access
7. Protecting your internal data structures
8. Synchronous and async behavior

Rusty's API Levels

10. It's impossible to get wrong.
9. The compiler/linker won't let you get it wrong.
8. The compiler will warn if you get it wrong.
7. The obvious use is (probably) the correct one.
6. The name tells you how to use it.
5. Do it right or it will always break at runtime.
4. Follow common convention and you'll get it right.
3. Read the documentation and you'll get it right.
2. Read the implementation and you'll get it right.
1. Read the correct mailing list thread and you'll get it right.

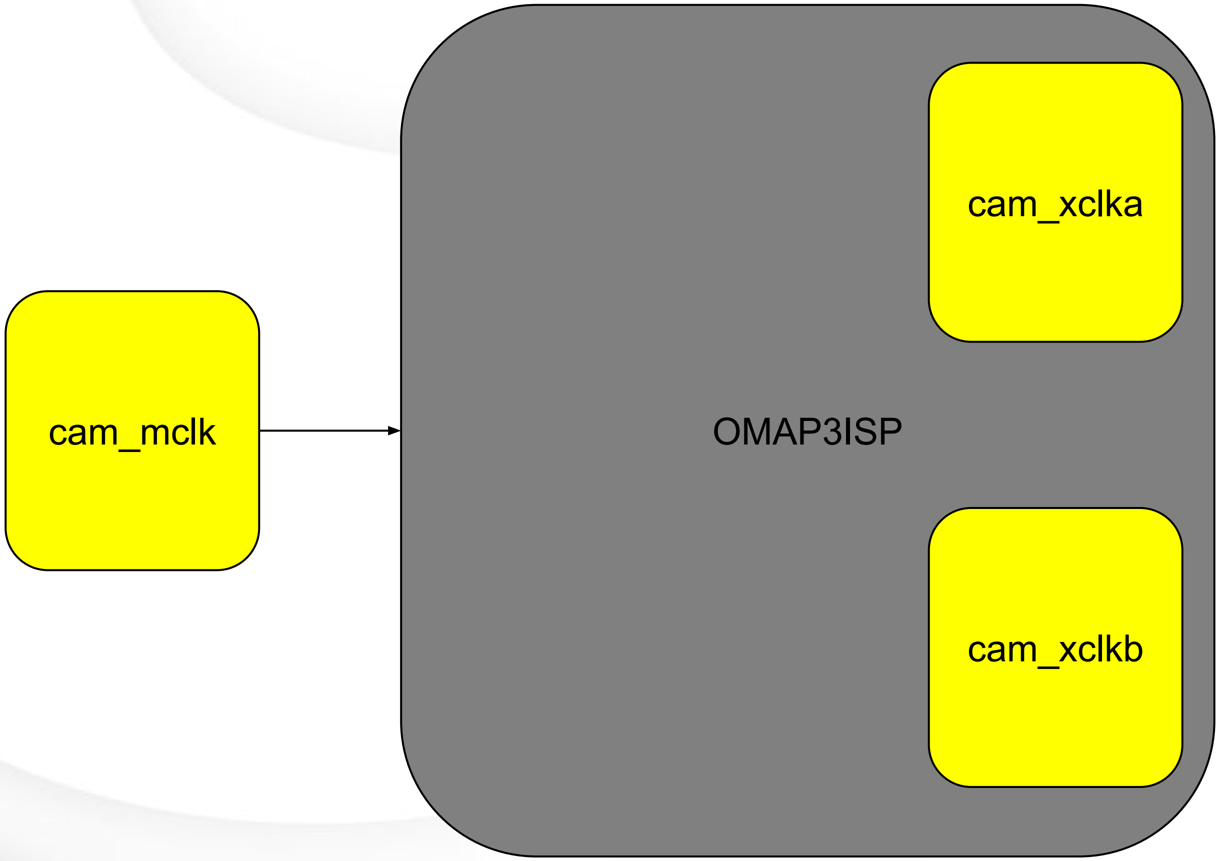
http://goo.gl/SmNqN8

http://goo.gl/yc6E4X

Consumer/provider API split

- Consumers want to get devices and resources and call functions on them
  - clk_get(), clk_set_rate(), clk_enable(), etc
- Providers register the devices and resources
  - clk_register(), clk_unregister(), struct clk_ops, etc
- Split them into separate headers
  - include/linux/clk-provider.h, include/linux/clk.h (consumer)

Example: combined provider/consumer



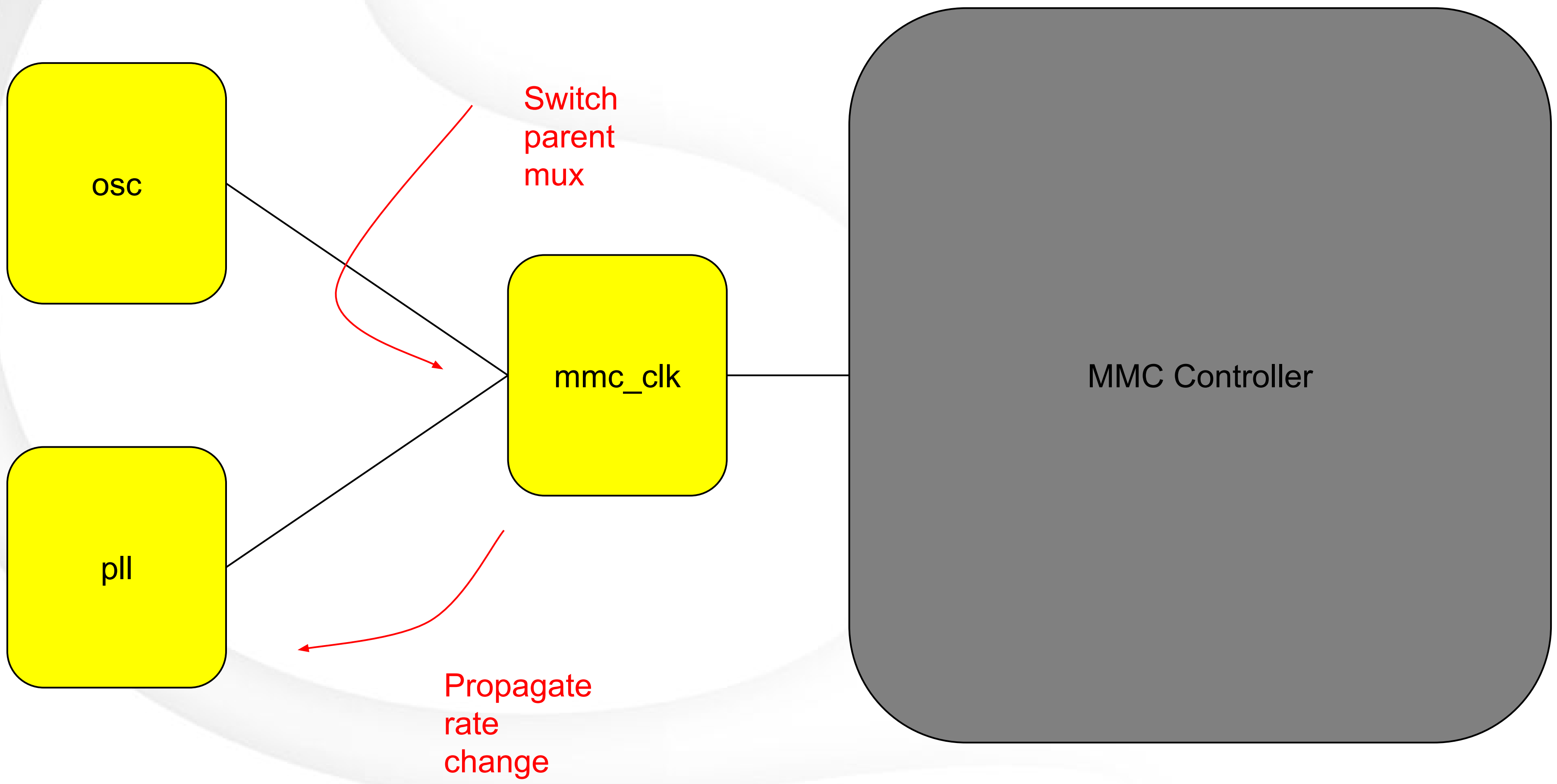
drivers/media/platform/omap3isp/isp.c

Abstract away any hardware details

The **subsystem** is incorrectly designed if **consumer drivers** need to know details about the underlying hardware

Write-only APIs are useful for this

Example: abstract away hardware details



Device versus Resource

CCF manages clock tree hierarchies

- Should clocks be Linux devices?
  - Hundreds of clocks…
- Does it match the data sheet?
  - Clock controller IP blocks expose hundreds of clock nodes
  - IP block roughly == Linux device

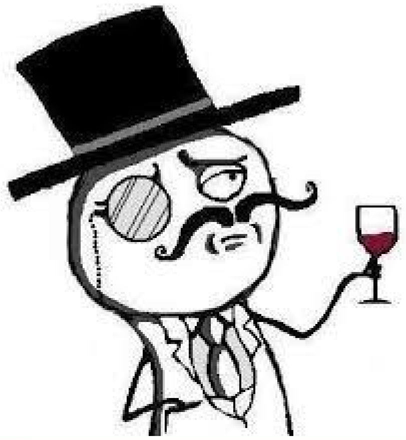
Bring Your Own Device

vs

Framework Provided Device

- struct regulator.dev versus struct clk
- CCF does not create a struct device

Purely a matter of taste



Reference counting

kobject

- creates sysfs object
- includes kref object for reference counting
- get this “for free” with struct device

kref

- lightweight alternative to kobject
- struct clk_core uses this to keep things sane around module unloading

Don’t forget the release() method!

Follow the Linux driver model

gross.

```
void __init nomadik_clk_init(void)
{
        struct clk *clk;

        clk = clk_register_fixed_rate(NULL,
                "apb_pclk", NULL,
                CLK_IS_ROOT, 0);
        ...
```

Still awake?

Locking and concurrent access

- Drivers will do crazy shit.

- Protect yourself!
    - Define strict entry points into the framework
    - Wrap all data structure accesses in a sane locking scheme

- Do you need to access the framework in interrupt context?
    - Provide irq-safe entry points using spinlocks
    - Otherwise use mutexes

Example: clk_prepare & clk_enable

CCF has competing needs:

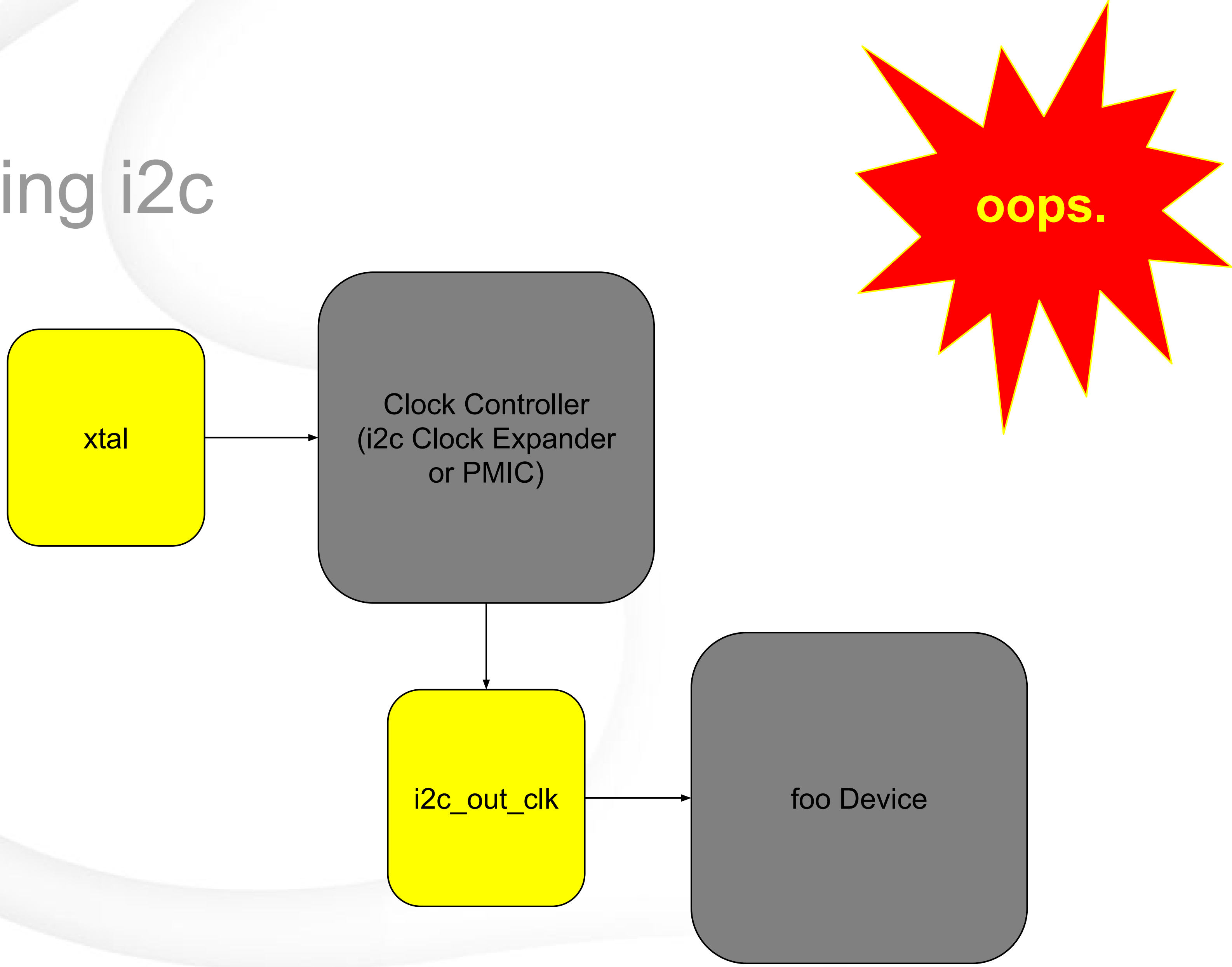
1. clk_enable/clk_disable can be called from interrupt context
2. Some enable ops may have delays/sleeps

clk_{un}prepare supplements clk_{en,dis}able

Mutex protects prepare ops, spinlock protects enable ops

Consumer drivers must always call both, in-order, and do not need to know the details of underlying hardware

Breaking i2c



Example: Reentrant locking

```
#define get_current() (current_thread_info()->task)

#define current get_current()


static void clk_prepare_lock(void)

{

        if (!mutex_trylock(&prepare_lock)) {

                if (prepare_owner == current) {

                        prepare_refcnt++;

                        return;

                }

                mutex_lock(&prepare_lock);

        }
```

Protect data structures & bookkeeping

Internal bookkeeping and private data structures should not be defined in headers

- Expose opaque handles to consumer drivers
- Think long and hard before giving provider drivers access to struct definitions and private pointers
- Reference count accesses to these resources

Drivers **will** muck with data structures and bookkeeping that they have no business touching

Per-user reference counting

```
struct clk {
	struct clk_core *core;
	const char *dev_id;
	const char *con_id;
	unsigned long min_rate;
	unsigned long max_rate;
	struct hlist_node clks_node;
	unsigned int prepare_count;
	unsigned int enable_count;
};
```

```
struct clk_core {
	const char *name;
	const struct clk_ops *ops;
	struct clk_hw *hw;
	struct module *owner;
	struct clk_core *parent;
	unsigned int enable_count;
	unsigned int prepare_count;
	…
};
```

Example: get/put bug

```
static struct clk *foo;
void probe()
{
    foo = clk_get(dev, “foo”);
    clk_prepare_enable(foo);
    clk_put(foo);
}
```

```
void module_exit()
{
    clk_unprepare_disable(foo);
}
```

Sync vs Async consumer API behavior

Sync

- Execution blocked until operation completes
- The right choice for some low-level operations where sequence is critical
- Examples: i2c and clk consumer APIs

Async

- Execution of calling thread proceeds after function is called
- Increases performance in many use cases
- Requires a model where waiting on a completion event makes sense
- Example: spi consumer APIs

Where does the data come from?

- Provide helper functions for the **primary** source of driver data
- In the embedded world this is often Device Tree

- Continuously scan provider drivers and consolidate common open-code solutions into helpers
- Design with a firmware interface in mind, but …
- … also do not design only for a single firmware interface

Misc tips & pitfalls

Test for memory leaks caused by module load/unload/reload

Pass pointers to structs instead of long argument lists

Merge tests upstream. Use CONFIG_COMPILE_TEST

Occasionally sort Makefiles lexicographically

3. (Very) Brief review of CCF design

Background on CCF

- clk.h API is pretty old
  - Consumer side of the API
  - pre-dates CCF
  - Multiple implementations
- Single implementation desirable
  - One definition of struct clk
  - Single zImage
  - Code consolidation
- Coincided with other developments

CCF design (in a single slide)

- It is a library
  - BYO(P)D
- Re-entrant for the same context
- Mixed use of mutex and spinlock
- Per-user, opaque handles
- Per-user reference counting kia kref
- Strict consumer/provider API split
- Internal data structures hidden
- Big global locks
- No async api
- Consumer API is shared with competing implementations

4. Maintenance

So now what?

- Merging a new Linux driver subsystem is the *beginning* of the work, not the end
- Set aside 50% of your time to maintain “popular” frameworks

Maintaining sanity

- Find a co-maintainer
- Participate in linux-next
- Setup subsystem-specific mailing list and irc channel
- Automate your life

Practical advice

Say “No” all the time

- This is your primary job now!
- A weak reason to not merge a patch is enough reason
- If your instincts tell you that a patch is bad, do not merge it

Wolfram Sang explained this well at LPC 2016

- https://youtu.be/BX3S8KFBSIE

You may not be the top contributor to your own code

- That is OK! It means the framework is being adopted

Questions?