Simplify & reuse driver code

with regmaps

Ioan – Adrian Ratiu

adrian.ratiu@collabora.com

Overview

- By whom and for whom
- Problem statement
- Solution
- Two case studies
- Way forward

Who am I

- Consultant sr. sw. engineer
- Working on
  - HW bringup
  - Device drivers
  - Embedded Linux distros
  - Various other stuff

Who is this for

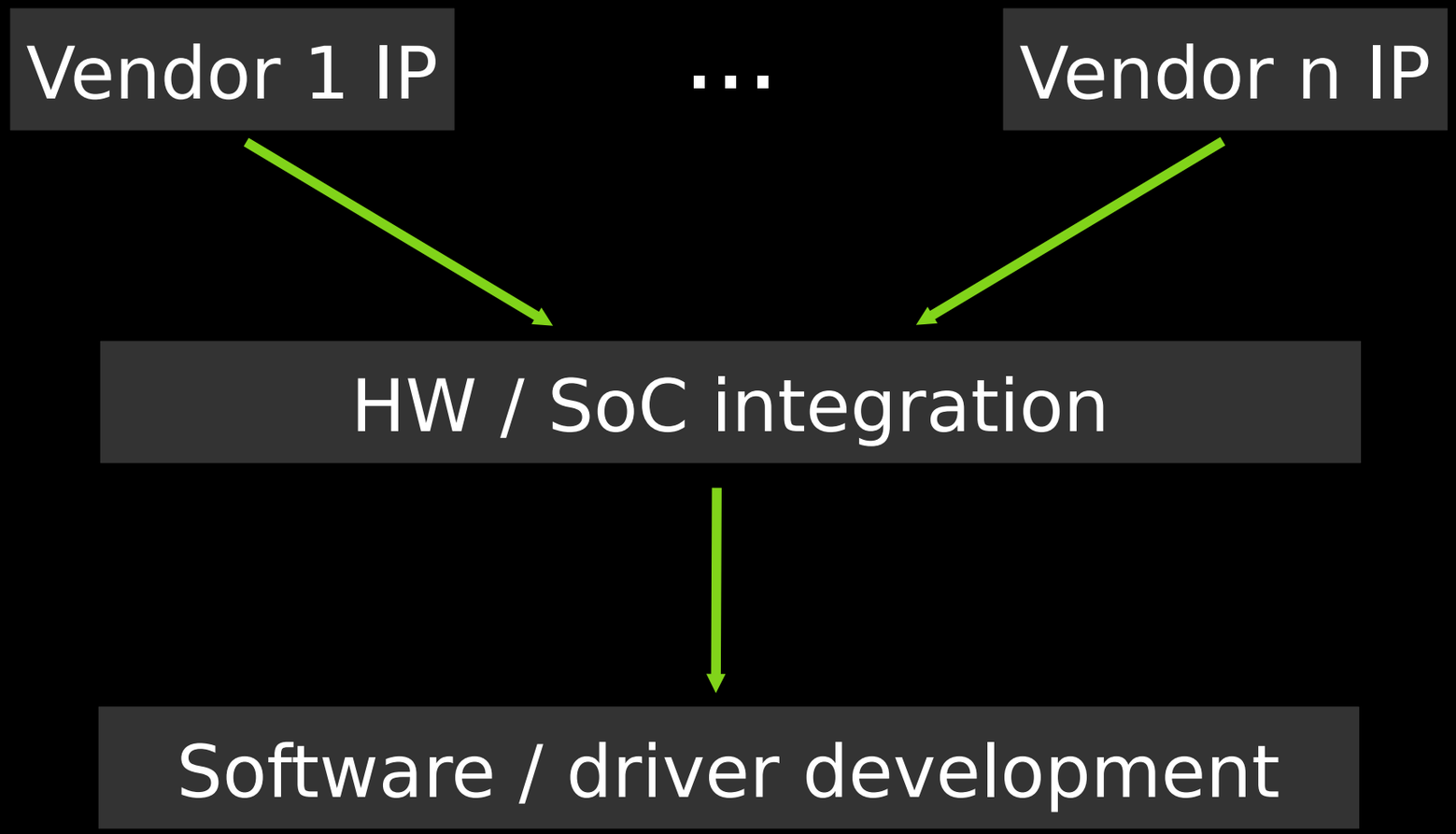
- Driver developers
- Linux developers
- Those wishing a decrease in driver proliferation
- Those wanting better drivers / less bugs

Not a silver bullet

- This is a practical solution to a common problem
- Based on repeating patterns seen in drivers
- Linux kernel upstream friendly
- Pros / cons and trade-offs to be aware of
- Purpouse: avoid duplication and/or wheel-reinvention
- This will not fix all bugs by itself :)

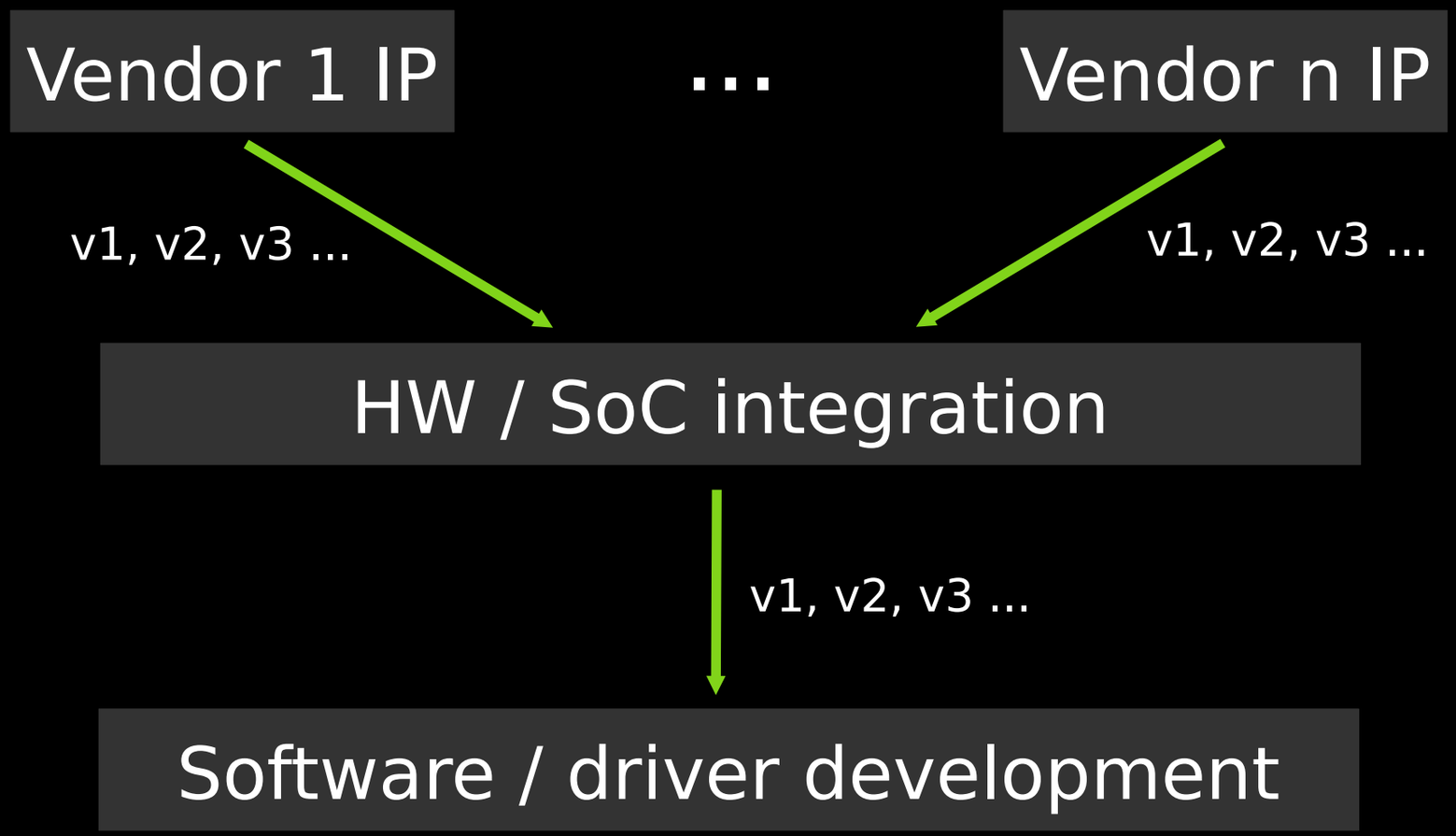
The problem

HW integration levels



The problem

HW integration levels



The problem

HW IF changes between revisions

- Vendors focused on optimizing HW design
  not on keeping HW programming IF compatibility

- HW programming protocols tend to follow standards
  so big breaking changes are rare

- Big breaks usually require new separate drivers

- Small incremental HW IF breakages are common
  and compensated for in drivers / software

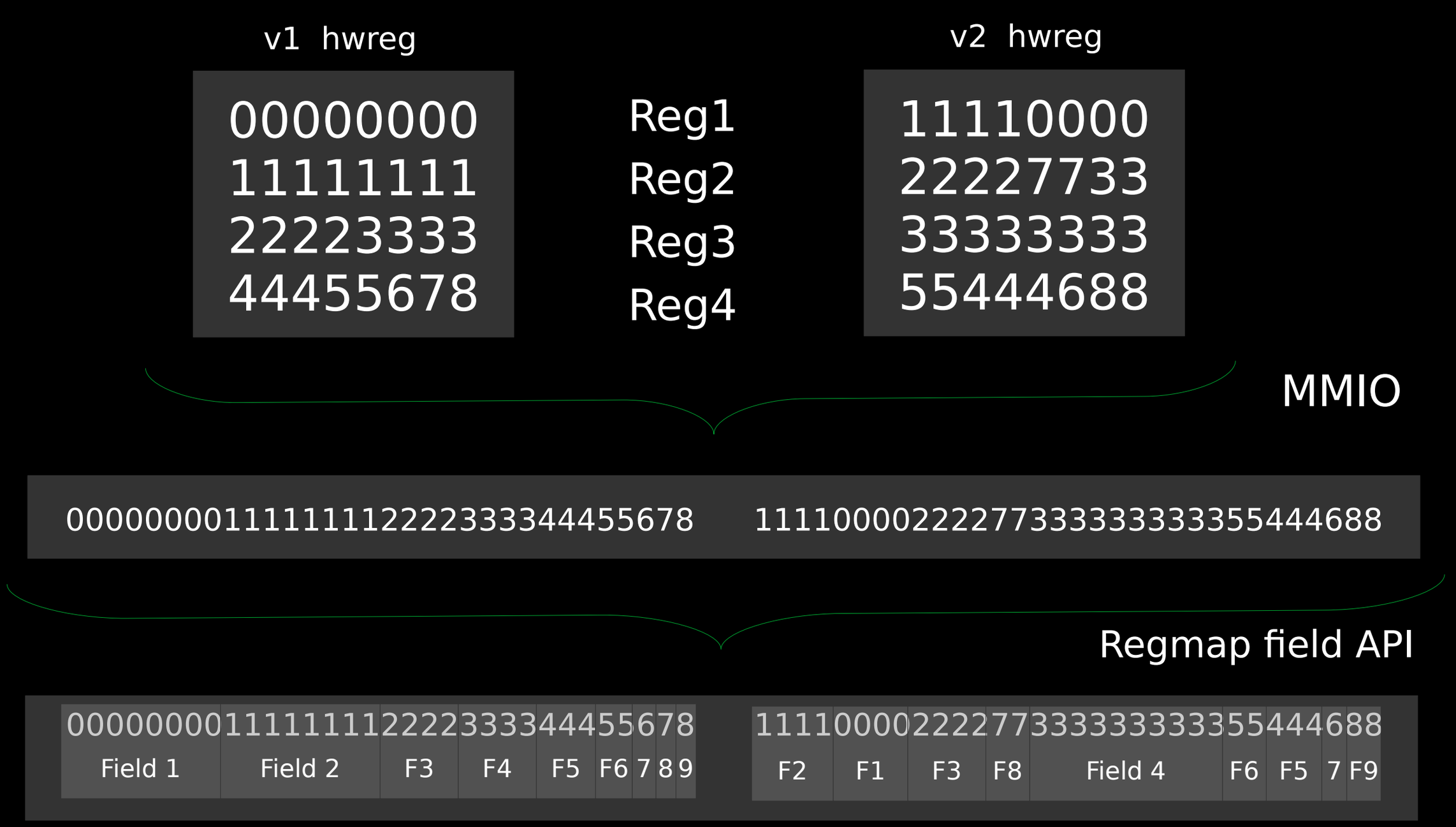
Most common annoyance

Register shuflling

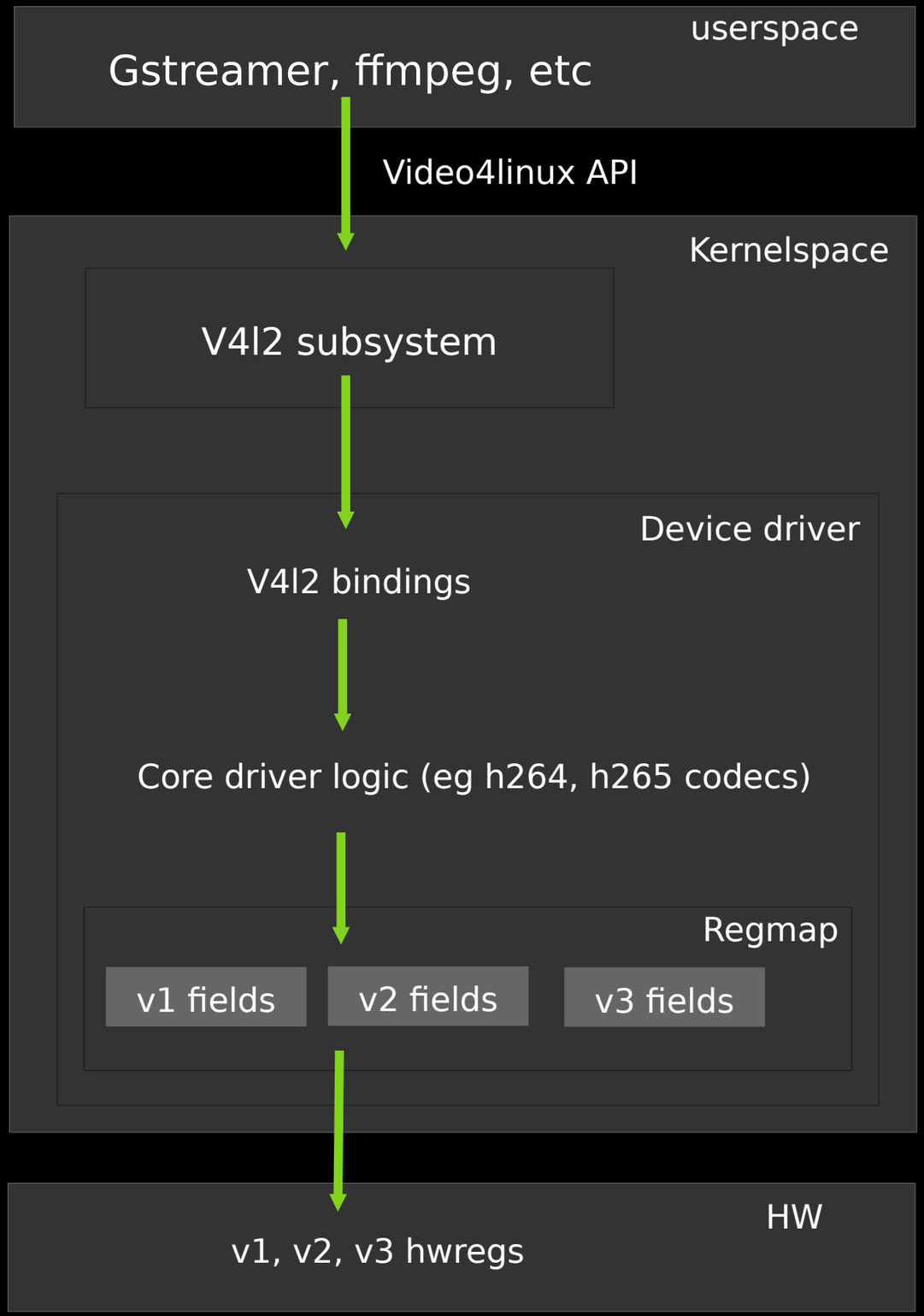
- Breakages may be necessary or unavoidable
  Eg new HW is capable of 8K video decoding requiring bigger resolution reg fields
- Breakages may also be due to non-technical reasons
- Can be big or small
  A total register shuffle may make HW hard to recognize
- Drivers can resort to bit manipulation tricks
  or add own abstractions on top of the bit magic

The solution: regmap

- Upstream Linux kernel subsystem
  - Mature, stable, introduced cca 2010
- Initially for non-memory mapped HW bus accesses
  - MMIO support soon followed (cca 2012)
- Can be used to build abstractions on top of HW registers
  - Regmap field API (cca 2013) for bit-level reg access



Driver programs the HW using the field API
No need to worry about reg layout differences



Replace v4l2 with any other subsystem or userspace

Regmap fields abstraction layer between core driver logic and HW

Drivers focus on their logic without having to account for different HW layouts

Regmap field configuration (private register layout)

```
struct regmap_config hantro_regmap_dec = {
        .reg_bits = 32,
        .val_bits = 32,
        .reg_stride = 4,
        .max_register = 0x554;
        .disable_locking = true,
};
```

Configure how regs look and behave.

```
struct hantro_field_dec {
        struct reg_field cfg_dec_axi_rd_id;
        struct reg_field cfg_dec_axi_wr_id;
        ...
};
```

Field configuration
Struct naming is unfortunate:
reg_field (cfg) vs regmap_field (API)

```
static const struct hantro_field_dec g1_field = {
        .cfg_dec_axi_rd_id =            REG_FIELD(SWREG(16), 24, 31),
        .cfg_dec_axi_wr_id =            REG_FIELD(SWREG(30), 0, 7),
        ...
};

static const struct hantro_field_dec vc8000d_field = {
        .cfg_dec_axi_rd_id =            REG_FIELD(SWREG(77), 0, 15),
        .cfg_dec_axi_wr_id =            REG_FIELD(SWREG(77), 16, 31),
        ...
};
```

Fields,
Fields,
Fields,
...

For two HW revisions

Regmap field configuration (HW programming API for driver)

```
struct hantro_regmap_fields_dec {
        struct regmap_field *dec_axi_rd_id;
        struct regmap_field *dec_axi_wr_id;
        ...
};
```

Define unified API
Names can differ from those in reg cfgs

```
dec_axi_wr_id = devm_regmap_field_alloc(dev, regmap,
                                        v1_cfg_dec_axi_wr_id);
```

Associate API with cfg

(to do the association, the HW revison needs to be known at runtime)

```
regmap_field_write(fields->dec_axi_wr_id, 0);
regmap_field_write(fields->dec_axi_rd_id, 16);
```

Program HW via the API in driver(s)

For a more detailed introduction please visit my blog post :

https://bit.ly/3nf0IJt

Pros / Cons (you decide which is which)

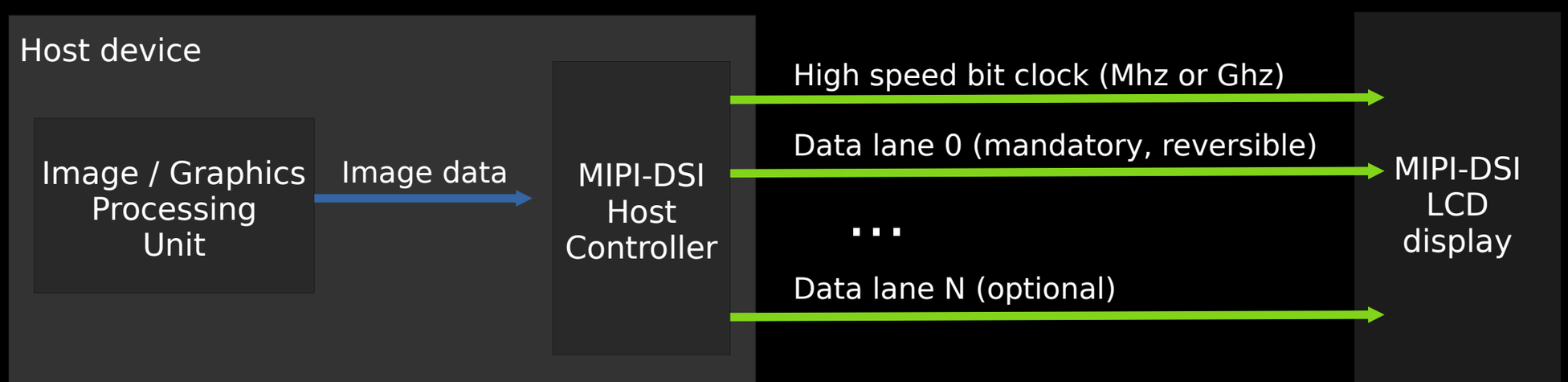
- Linux (only) kernel upstream mechanism
- Many optional features (bounds checks, caches, locks, callbacks, debugfs, etc)
- Unified reg layout abstraction implementation
- Removes boilerplate from driver code & make code reuse easier
- Easy to add new HW revisions to a driver
- Field config closely follows HW register datasheet info
- More verbose than direct bit-manipulation
- Low microsecond perf impact (depending on hwreg speed)

Case study 1:

Synopsys MIPI-DSI host controllers

MIPI-DSI in a nutshell

- Simple HW implementation (small, cheap, few wires)
- Popular in mobile/gaming, automotive, IoT, maker etc
- Spec governed by MIPI alliance, not public (v1.0 – v1.31)
- Silicon IP vendors (like Synopsys) implement spec in DSI controllers
- SoC vendors (like NXP, STM, RK) integrate controller IP versions



MIPI-DSI – problem & solution

- HW IF layout breakages mostly due to MIPI-DSI spec changes
- HW functionality & programming mostly the same between revs
- Each SoC vendor provides own separate drivers
- Kernel upstream driver supports v1.30 & v1.31 with bit-manipulation
  (STM & RK SoCs)
- Wanted support for v1.01 in i.MX6
- Bigger 1.0 vs 1.3 layout divergence made bit-manipulation hard
- So a regmap field layer was introduced :)

Link to patch series v9: https://patchwork.kernel.org/cover/11596301/

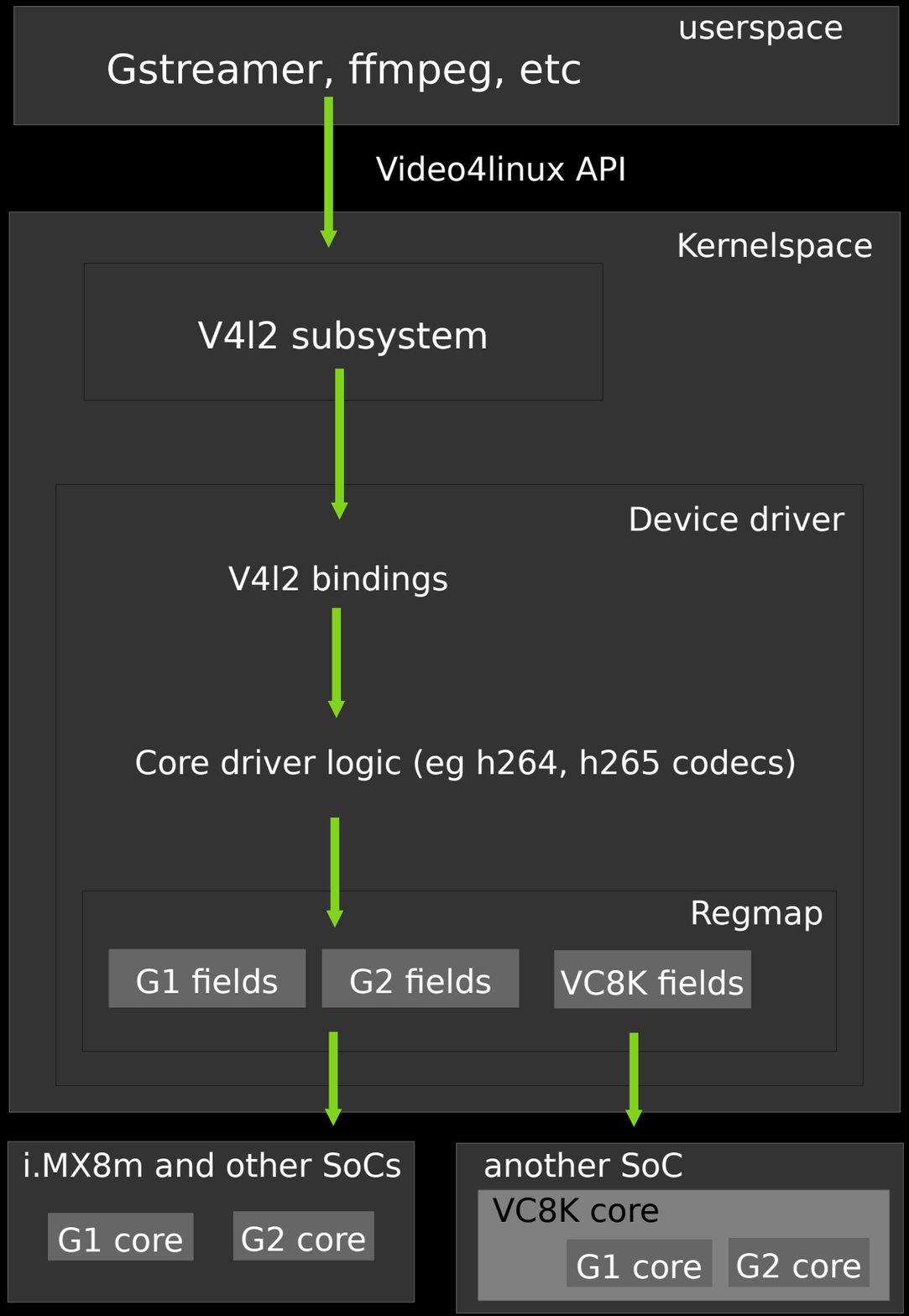
Blog post on the subject:
https://bit.ly/3nf0IJt

MIPI-DSI – Challenges and results

- Regmap design & implementation was easy
- Testing on multiple SoCs with displays was hard (lack of HW)
- Big thank you to all those who helped testing & debugging
- Kernel upstream driver needs some unrelated improvements
- Unfortunately few people have time to invest (myself included)
- Hope the series gets picked up again and driven to inclusion

Case study 2:

Verisilicon “Hantro” video codecs



HW video codecs in a nutshell

Complex HW, many features and corner-cases, programmed via hundreds of registers

Only the video bitstream is standardized and is also complex (h264, h265, VP8/9, etc)

Verisilicon decided to merge its two separate G1 & G2 decoders into one chip, named VC8000

Register layouts took a heavy hit but HW functioning remained mostly the same

Hence the regmap layer :)

Hantro codec – problem & solution

- Upstream driver only supported a subset of G1 and G2 features
- Idea: Introduce regmap layer to also support newer VC8K chips
- Performance is critical
  - Needs to decode hi-res videos at high framerates in parallel
  - Regmap fields added a constant ~ 20 us of register IO overhead per frame
  - Acceptable considering VPU HW decoding takes up to 20 ms per frame

    (much depending on HW and frame resolution)
- Battery consumption must be minimized / CPU load optimized
  - This is why upstream driver did explicit relaxed & non-relaxed MMIO

Hantro codec – Challenges and results

- Regmap design & implementation was (again) easy
- Figuring out differences between new & old chips was difficult
- Could not measure relaxed vs non-relaxed MMIO impact
  - Extending regmap API to allow relaxed MMIO is easy
  - Hard to justify upstream API addition without good measurements
- Hantro driver still has own ‘struct hantro_reg’ abstraction
- Hantro still has a ‘driver within driver’ due to layout divergence
  (time & interest required to convert the rk3399 sub-driver to regmap)

Patch series should be posted publicly before this presentation

Way forward

Regmaps are widely used, but not to abstract hwreg layouts

More drivers can be converted / boilerplate removed

Helpers could be added to reduce init verbosity

Got upstream maintainer buy-in for the above two use-cases

Room for regmap field API standardization

For similar HW, abstraction layers / libs can be created

(eg VPU decoder lib with unified virtualized HW interface)

Thank you

```
Message {
   config {
      priority: "high"
      body: "Collabora is hiring"  // Many open
positions
      recipient: "you"             // Please
join us
      calltoaction: "http://col.la/join"
   }
}
```