How V4L2 Transformed To Support Embedded Cameras

ELCE 2025
Amsterdam, Netherlands

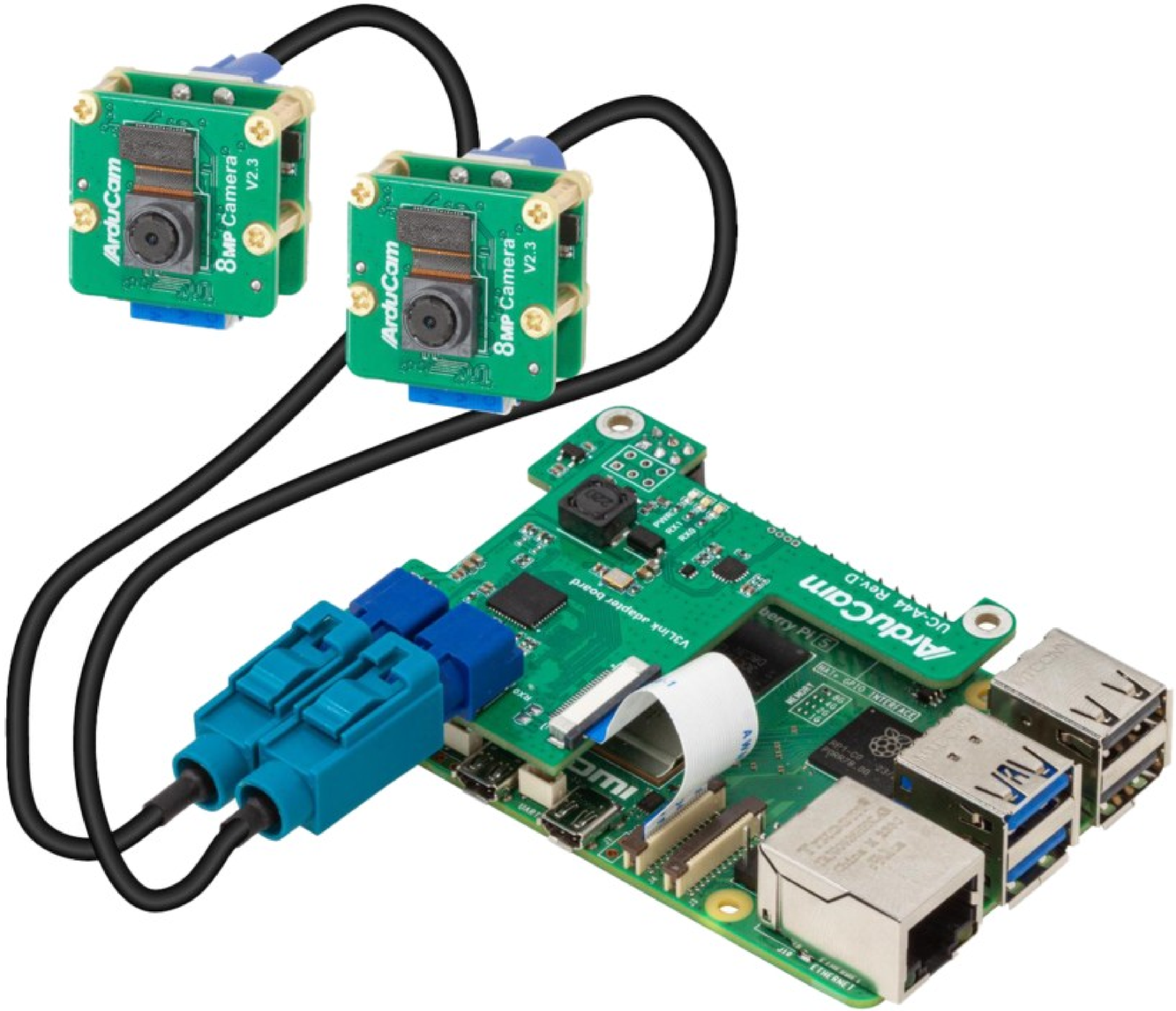
Laurent Pinchart
laurent.pinchart@ideasonboard.com

Once upon a time, actually yesterday

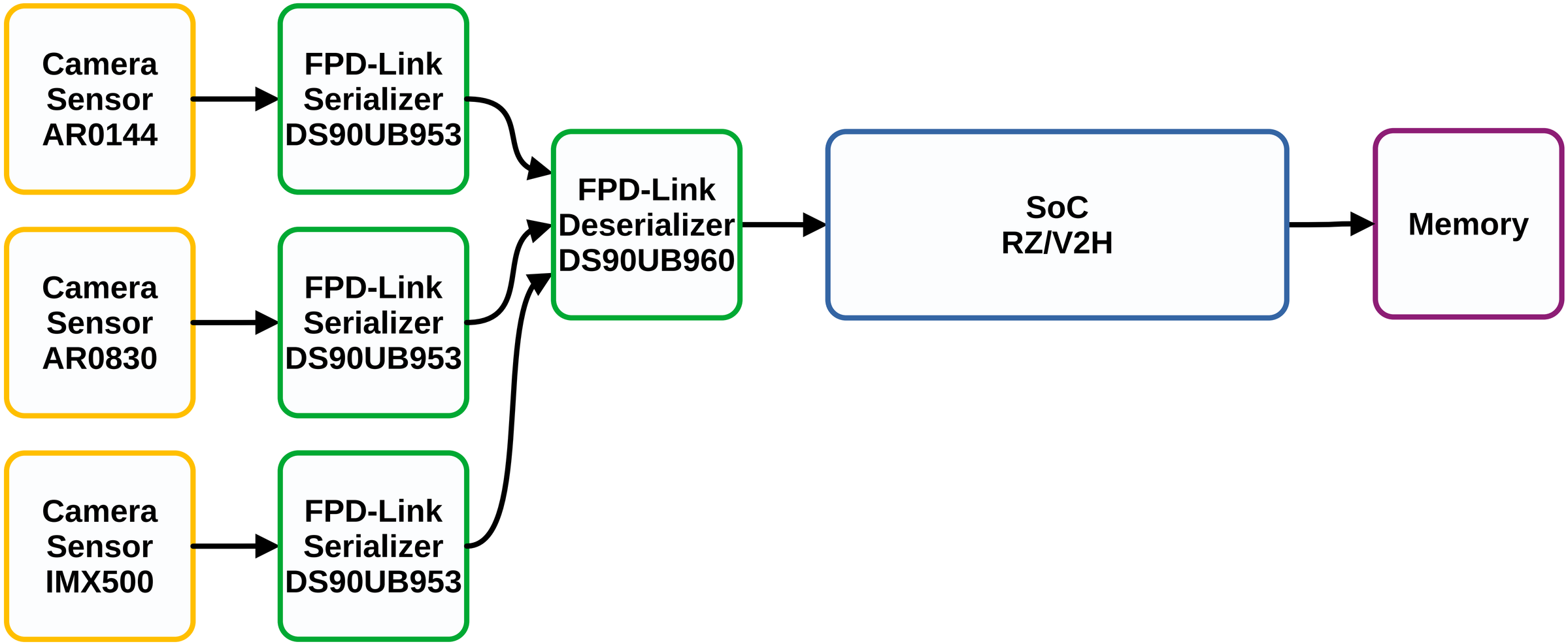
Misconception #1

V4L2 is a webcam API

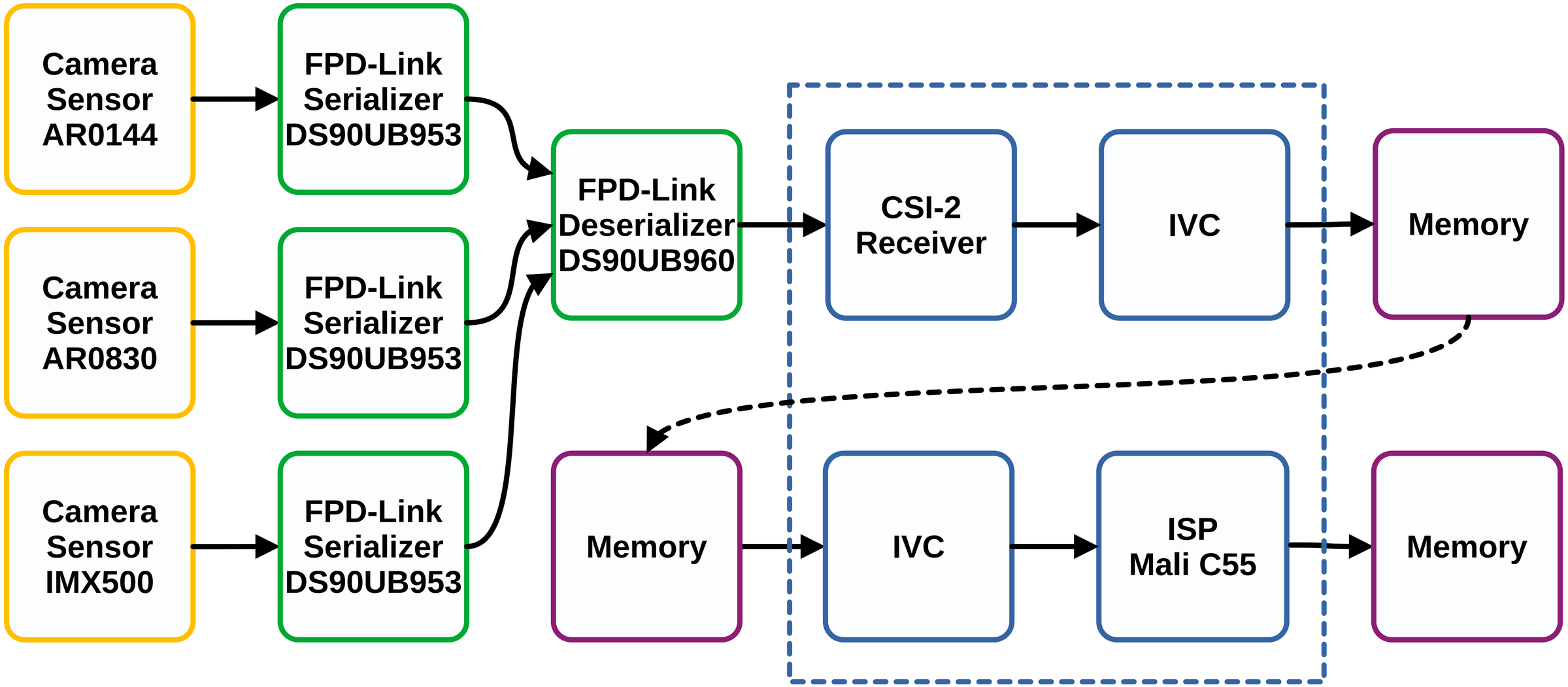
No it’s not.



The hardware



The hardware

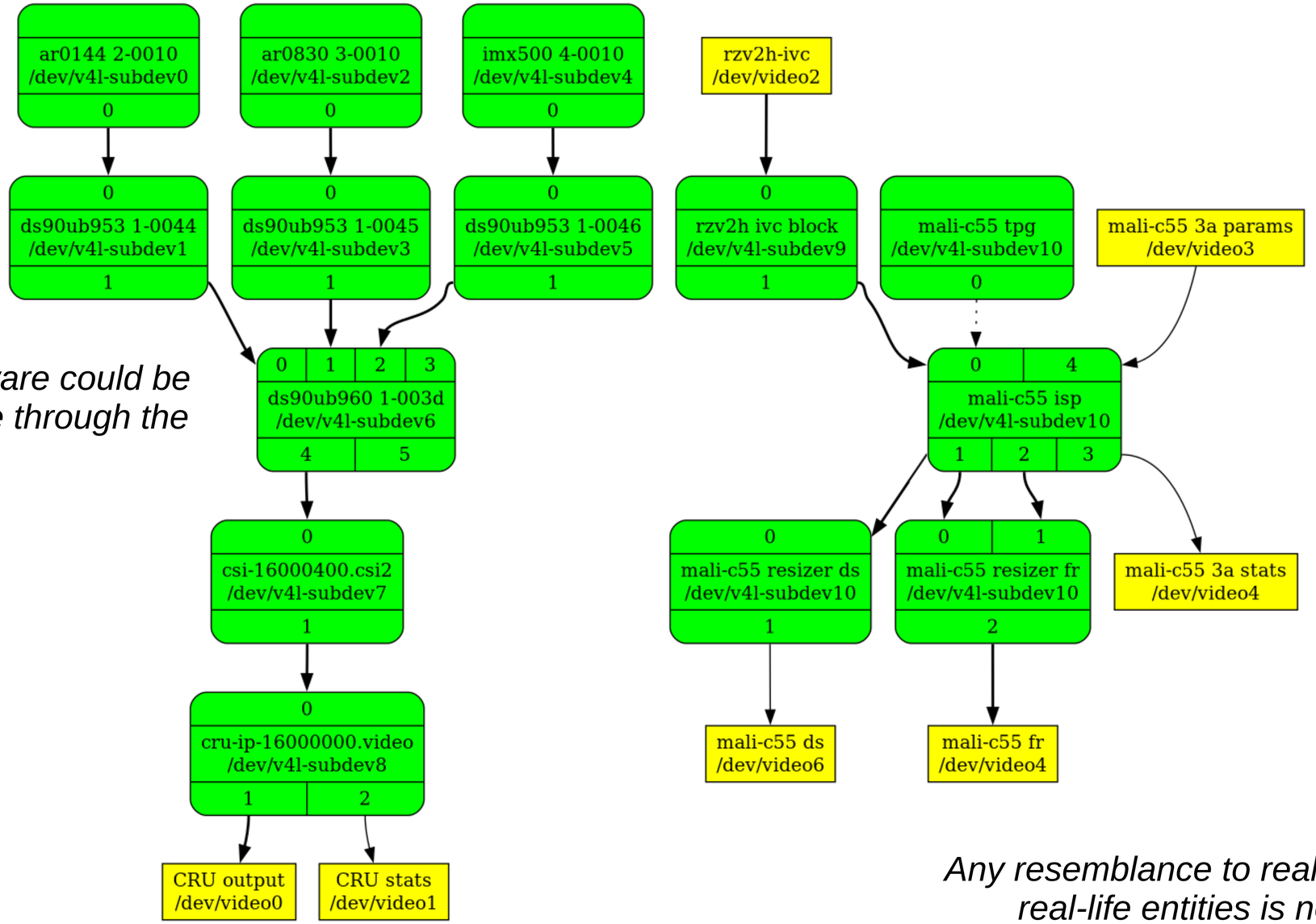


The hardware

Misconception #2

V4L2 is not made to handle that

This is how the hardware could be exposed to userspace through the Media Controller API.



Any resemblance to real media graphs or other real-life entities is not entirely coincidental.

Media controller

- The **Media Controller API** exposes the internal topology of the device to userspace as a graph of connected entities.
- Most entities are V4L2 video devices or subdevs.
- Complemented by the **V4L2 subdev userspace API** that allows direct configuration of subdevs from userspace.
- Shockingly old, introduced in v2.6.39.
- Widely documented, one of the best documents is *https://git.ideasonboard.org/doc/mc-v4l2.git/* (shameless advertising).

Media controller

Misconception #3

V4L2 can’t handle all of that

Misconception #3.1

V4L2 can’t handle CSI-2 VCs/DTs

First we need to talk about subdev states

- Stores **subdev state** information in a common structure, passed to (most) V4L2 subdev operations.
- Subdev state currently consist of formats, crop and compose rectangles, frame intervals and (more on that later) internal routing.
- The **ACTIVE** state stores the configuration applied to the device, the **TRY** state stores a per file handle configuration. Userspace selects for each IOCTL if it applies to the ACTIVE or TRY state.
- Two benefits: simplifies storage of data for drivers, but more importantly allows the V4L2 core to access the subdev configuration, paving the way for useful helper functions.

V4L2 subdev state

- Enable in drivers: call `v4l2_subdev_init_finalize()` at probe time, and implement the `.init_state()` operation.
- Access state from the pointer passed to subdev operations.
- Call subdev operations with `v4l2_subdev_call_active_state()`. Do **not** use `v4l2_subdev_call_try_state()` in any new code.
- Bonus: simplify drivers with helpers.

```
static const struct v4l2_subdev_pad_ops my_pad_ops = {
        .get_fmt = v4l2_subdev_get_fmt,
        .get_frame_interval = v4l2_subdev_get_frame_interval,
};
```

V4L2 subdev state

- Access configuration from state:

```
struct v4l2_mbus_framefmt *
v4l2_subdev_state_get_format(struct v4l2_subdev_state *state,
                             unsigned int pad, u32 stream);
struct v4l2_rect *
v4l2_subdev_state_get_crop(struct v4l2_subdev_state *state, unsigned int pad,
                           u32 stream);
struct v4l2_rect *
v4l2_subdev_state_get_compose(struct v4l2_subdev_state *state,
                              unsigned int pad, u32 stream);
struct v4l2_fract *
v4l2_subdev_state_get_interval(struct v4l2_subdev_state *state,
                               unsigned int pad, u32 stream);
```

- The V4L2 core takes care of locking states, removing the need for manual locking in drivers.
- **Use the same lock for the active state and the control handler:**

```
dev->sd.state_lock = dev->ctrl_handler.lock;
ret = v4l2_subdev_init_finalize(&dev->sd);
```

- Subdev operations that receive a state are guaranteed to be called with the state locked. If you don’t have a pointer to the active state, you can get and lock it with `v4l2_subdev_lock_and_get_active_state()`. If the state is already locked, use `v4l2_subdev_get_locked_active_state()` (mostly in the `.s_ctrl()` handler, discouraged elsewhere). **Never** use `v4l2_subdev_get_unlocked_active_state()`.
- Locking is a bit messy, prone to AB-BA deadlocks in some rare cases. This will eventually be reworked to always pass locked states everywhere.

- Documented in *https://docs.kernel.org/driver-api/media/v4l2-subdev.html# centrally-managed-subdev-active-state.*
- Introduced in v6.2.
- (Unofficially) mandatory for all new drivers.
- **Volunteers needed**: missing support for V4L2 controls, and for subclassing state (to store driver-specific information).

V4L2 subdev state

Let’s go back to VCs and DTs

~~Let’s go back to VCs and DTs~~

Let’s talk about streams

“A media **stream** is a **stream** of pixels or metadata originating from one or more source devices (such as a sensors) and flowing through media entity pads towards the final sinks.” *(Documentation/driver-api/media/mc-core.rst*)

“A **stream** is a **stream** of content (e.g. pixel data or metadata) flowing through the media pipeline from a source (e.g. a sensor) towards the final sink (e.g. a receiver and demultiplexer in a SoC).” *(Documentation/userspace-api/media/v4l/dev-subdev.rst*)

Thanks...

- Streams model the data flows through a media graph. They serve as a base concept for APIs that control data **routing** within subdevs, or handle streams **multiplexing** over a single link.
- Subdevs now have an internal **routing table** that indicates how data streams flow inside a subdev from sink pads to source pads.
- Each pad can carry multiple streams, identified by a **stream ID**. Those IDs are local to a **link**: the same stream ID on the source and sink sides of a link between subdevs refers to the same stream.

- The table is exposed to userspace through the new `VIDIOC_SUBDEV_G_ROUTING` and `VIDIOC_SUBDEV_S_ROUTING` IOCTLs.

```
struct v4l2_subdev_route {
        __u32 sink_pad;
        __u32 sink_stream;
        __u32 source_pad;
        __u32 source_stream;
        __u32 flags;
};

struct v4l2_subdev_routing {
        __u32 which;
        __u32 len_routes;
        __u64 routes;         /* Pointer to a v4l2_subdev_route array */
        __u32 num_routes;
};
```

- Inside the kernel, the routing table is applied by a new subdev operation:

```
int (*set_routing)(struct v4l2_subdev *sd,
                   struct v4l2_subdev_state *state,
                   enum v4l2_subdev_format_whence which,
                   struct v4l2_subdev_krouting *route);
```

- Setting routes **creates streams**: each combination of a pad number and stream ID on the sink or source side of a route declares a stream to the kernel.
- All configuration stored in subdev states per pad is now stored per stream. A pad now has **one format per stream** it carries. Subdev state accessor helpers take a new optional stream number.
- Setting routes **resets all formats** on all subdev pads.

- In their implementation of `.(*set_routing)`, drivers **validate** routes, **store** them in the state and **reset formats**. Validation can be helped by

```
int v4l2_subdev_routing_validate(struct v4l2_subdev *sd,
                                 const struct v4l2_subdev_krouting *routing,
                                 enum v4l2_subdev_routing_restriction disallow);
```

- To store routes in the state, call one of

```
int v4l2_subdev_set_routing(struct v4l2_subdev *sd,
                            struct v4l2_subdev_state *state,
                            const struct v4l2_subdev_krouting *routing);
int v4l2_subdev_set_routing_with_fmt(struct v4l2_subdev *sd,
                                     struct v4l2_subdev_state *state,
                                     const struct v4l2_subdev_krouting *routing,
                                     const struct v4l2_mbus_framefmt *fmt);
```

The latter also resets all formats to the same value.

- Most userspace IOCTLs now take a stream number in addition to the pad number:

```
struct v4l2_subdev_format {
        __u32 which;
        __u32 pad;
        struct v4l2_mbus_framefmt format;
        __u32 stream;
};

struct v4l2_subdev_selection {
        __u32 which;
        __u32 pad;
        __u32 target;
        __u32 flags;
        struct v4l2_rect r;
        __u32 stream;
};
```

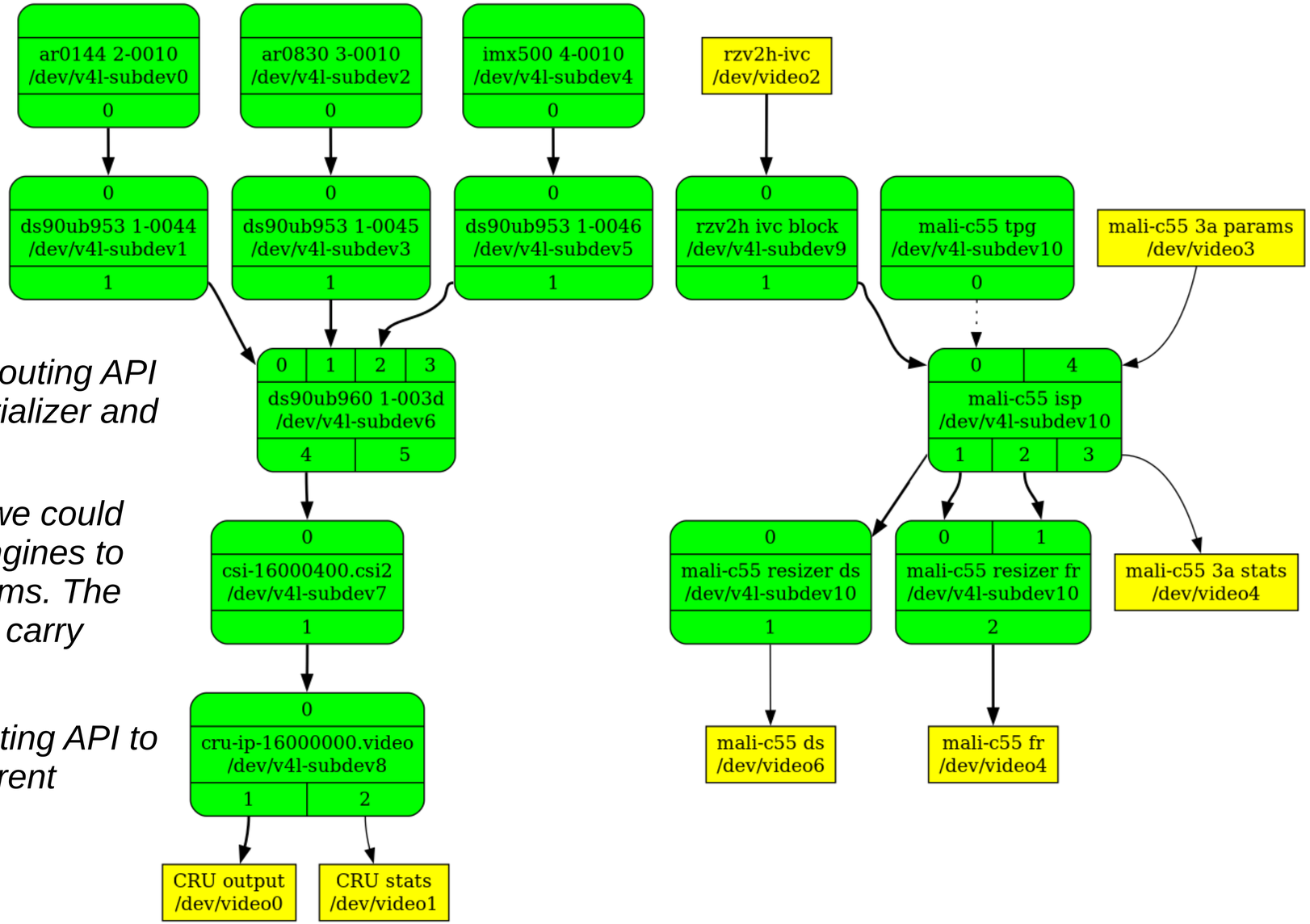
- New set of helpers to follow routes inside a subdev:

```
#define for_each_active_route(routing, route)

int v4l2_subdev_routing_find_opposite_end(
                                    const struct v4l2_subdev_krouting *routing,
                                    u32 pad, u32 stream, u32 *other_pad,
                                    u32 *other_stream);

struct v4l2_mbus_framefmt *
v4l2_subdev_state_get_opposite_stream_format(
                                            struct v4l2_subdev_state *state,
                                            u32 pad, u32 stream);

u64 v4l2_subdev_state_xlate_streams(const struct v4l2_subdev_state *state,
                                    u32 pad0, u32 pad1, u64 *streams);
```

- Opt-in feature for subdev drivers, controlled by the `V4L2_SUBDEV_FL_STREAMS` flag.
- `.s_stream()` is deprecated. Drivers must instead implement `.enable_streams()` and `.disable_streams()`. `.s_stream()` can be implemented for backward compatibility with `v4l2_subdev_s_stream_helper()`.
- Never call those subdev operations directly, use the `v4l2_subdev_enable_streams()` and `v4l2_subdev_disable_streams()` helpers.
- Drivers can use the new `v4l2_subdev_is_streaming()` helper instead of manually tracking the streaming state.

- Introduced in v6.3 but disabled by default, as still experimental:

```
commit 8a54644571fed484d55b3807f25f64cba8a9ca77
Author: Tomi Valkeinen <tomi.valkeinen@ideasonboard.com>
Date:   Sun Jan 15 13:40:08 2023 +0100

    media: subdev: Require code change to enable [GS]_ROUTING

    The Streams API is an experimental feature. To use the Streams API, the
    user needs to change a variable in v4l2-subdev.c and recompile the
    kernel.

    This commit should be reverted when the Streams API is deemed ready for
    production use.

    Signed-off-by: Tomi Valkeinen <tomi.valkeinen@ideasonboard.com>
```



Now we can use the routing API to configure the deserializer and select one input.

In a different device, we could have multiple DMA engines to capture multiple streams. The CSI-2 RX pads would carry multiple streams.

We would use the routing API to direct streams to different memory outputs.

Streams in the media pipeline

Let’s go back to VCs and DTs

- Virtual Channels and Data Types are modelled by streams. A CSI-2 link often carries multiple streams.
- VC and DT values are managed within the kernel only. They are **not set in the device tree**, and are **not exposed to userspace**. Stream IDs values do not map to specific VC and DT values.
- VC and DT values are communicated between drivers through **frame descriptors**. Source subdevs implement the `.get_frame_desc()` operation to expose the structure of a frame transmitted over a pad.

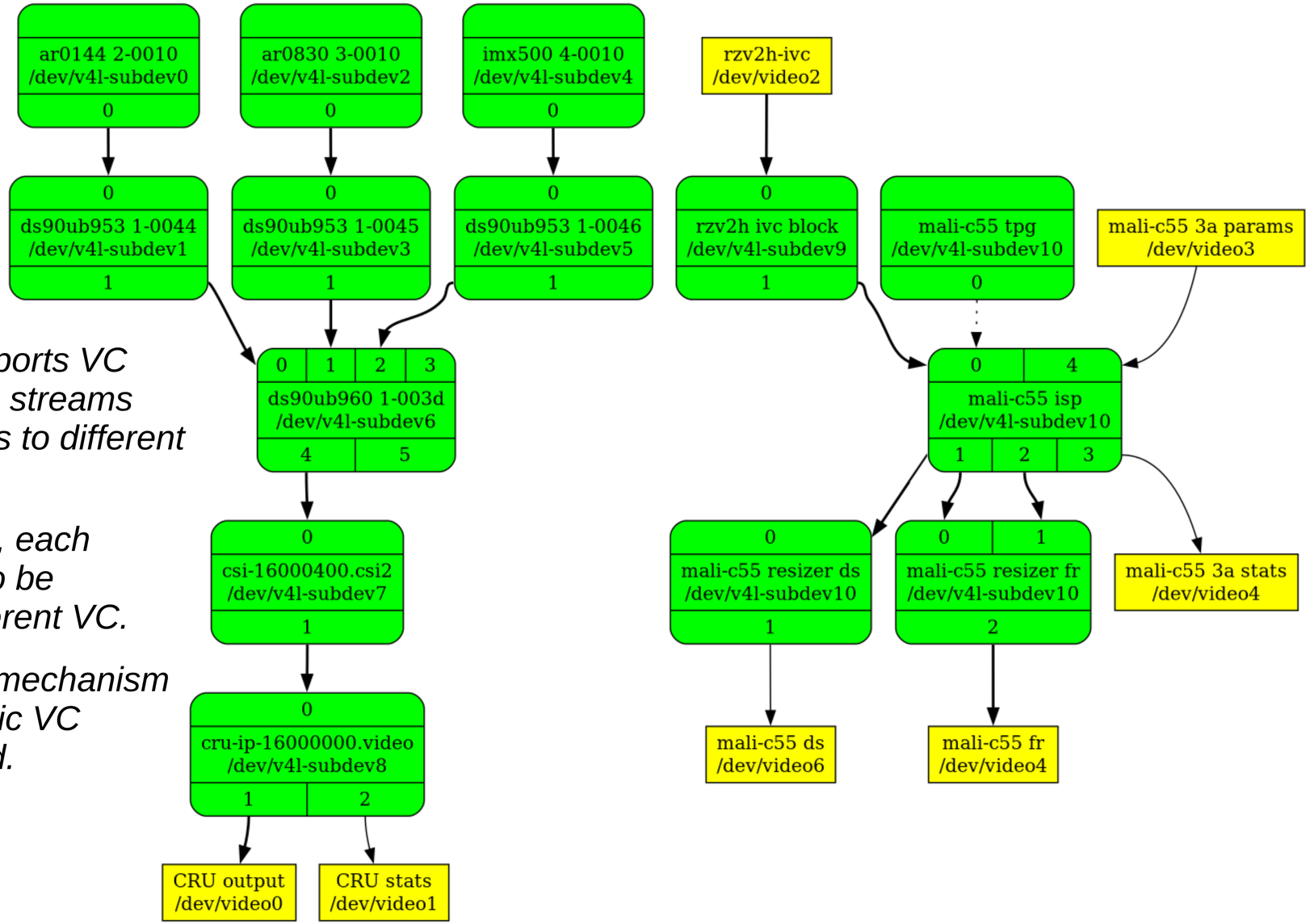
VCs and DTs

```
struct v4l2_mbus_frame_desc_entry_csi2 {
        u8 vc;
        u8 dt;
};

struct v4l2_mbus_frame_desc_entry {
        enum v4l2_mbus_frame_desc_flags flags;
        u32 stream;
        u32 pixelcode;
        u32 length;
        union {
                struct v4l2_mbus_frame_desc_entry_csi2 csi2;
        } bus;
};

struct v4l2_mbus_frame_desc {
        enum v4l2_mbus_frame_desc_type type;
        struct v4l2_mbus_frame_desc_entry entry[V4L2_FRAME_DESC_ENTRY_MAX];
        unsigned short num_entries;
};
```

VCs and DTs

- Each frame descriptor entry indicates which stream it relates to, and what VC and DT is carries.
- Sink drivers use the information to retrieve the VC and DT for each stream.
- The frame descriptors assign a line-based frame structure. They can support embedded data lines or dark pixel lines before the image, but don’t support left and right dark pixel columns.
- Introduced in v5.19.

VCs and DTs



The DS90UB960 supports VC remapping, outputting streams from different cameras to different VCs.

If this wasn't the case, each camera would need to be configured with a different VC.

There is no standard mechanism to do so yet. A dynamic VC allocator is envisioned.

VCs and DTs

Misconception #3.2

V4L2 can’t handle all sensor data

Raw camera sensor with embedded data

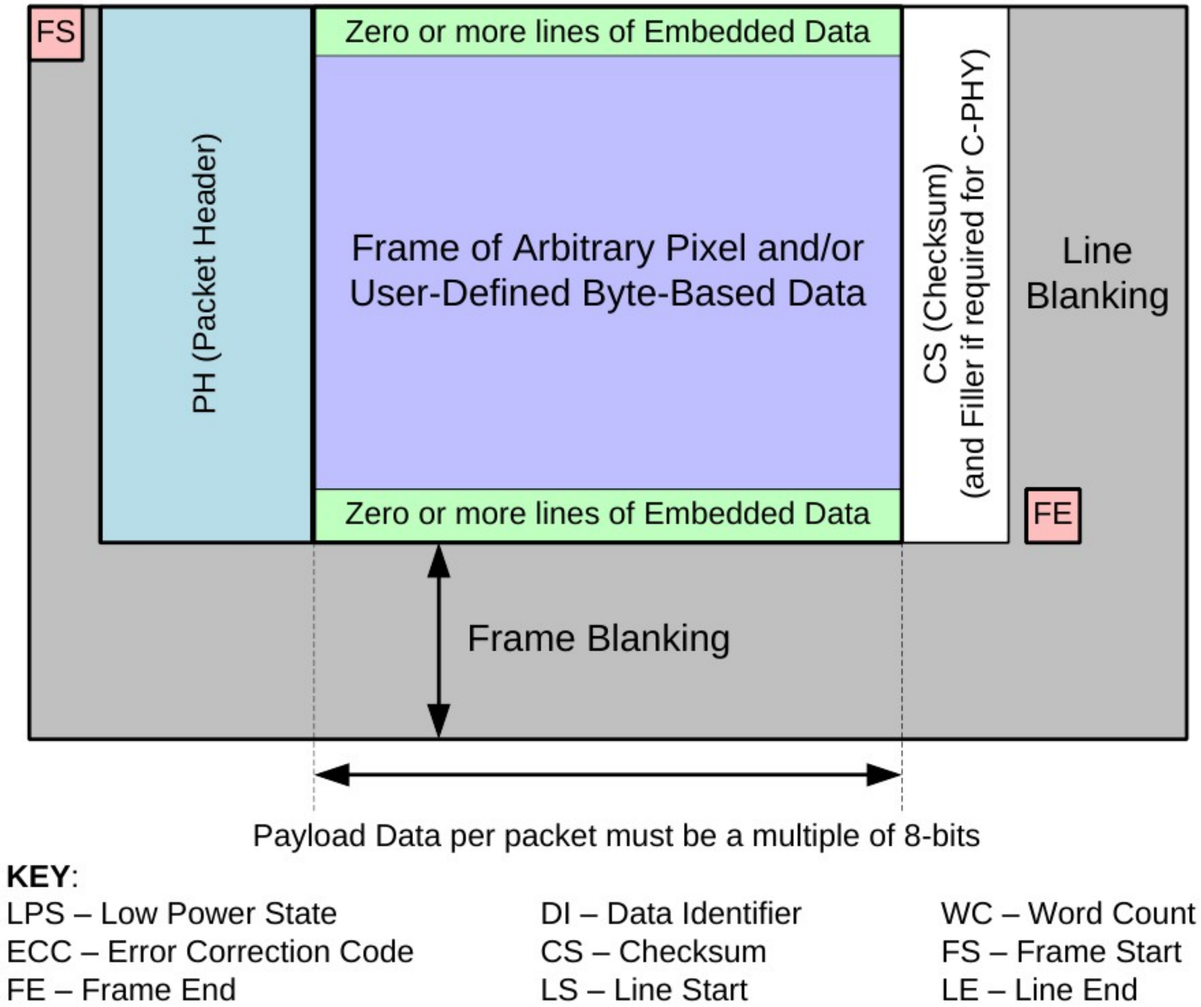
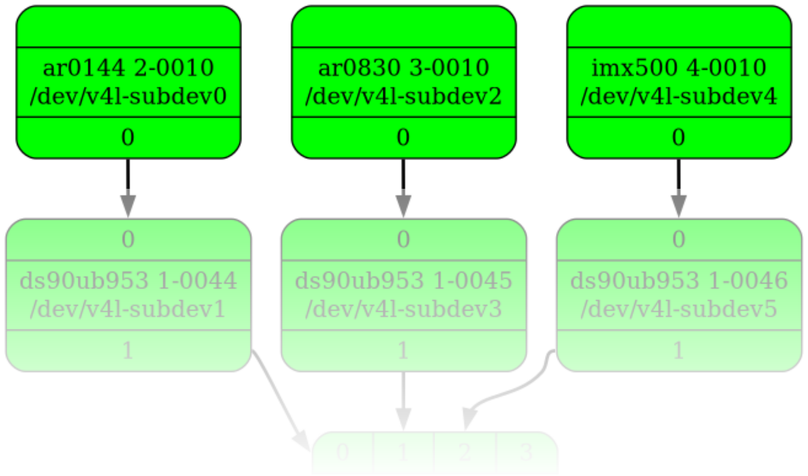


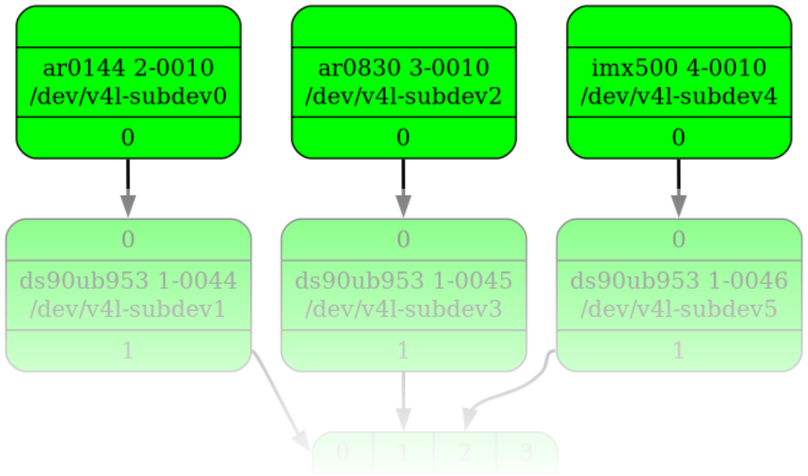
Figure 110 Frame Structure with Embedded Data at the Beginning and End of the Frame

- Modelled through streams: 1x embedded data stream, 1x image data stream. Streams are created by the routing table.
- DT values exposed through the frame descriptor.
- Formats configured on the sensor subdev source pad, separately on the two streams.



CSI-2 embedded data

- Modelled through streams: 1x embedded data stream, 1x image data stream. Streams are **created by the routing table**.
- DT values exposed through the frame descriptor.
- Formats configured on the sensor subdev source pad, separately on the two streams.



What routing table?

CSI-2 embedded data

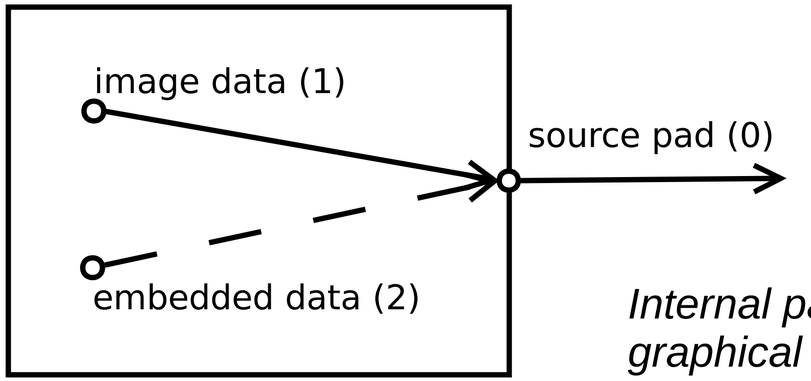
- We introduce the concept of **internal pads**.
- An internal pad is just like a pad, but internal to the entity. It can’t be connected to other entities through links, but can be part of a subdev internal routing table and have a configuration.
- Internal pads typically model the **original source of streams** in a pipeline.
- Identified by the `MEDIA_PAD_FL_INTERNAL` flag.
- Internal pads open the door to lots of creative (ab)use.

Internal pads

- An uncontrolled epidemic of creativity would hinder userspace development by making the API device-specific. Usage of internal pads need to be clearly documented. They however may benefit different categories of devices in different ways.
- Standardize the internal pads usage (and more) based on a newly introduced subdev **configuration model**. The configuration model is a read-only property exposed through the `V4L2_CID_CONFIG_MODEL` control. Values are standardized by V4L2, with the documentation clearly describing how V4L2 **API elements** such as internal pads or selection rectangles **map to device features**.
- The configuration models create classes of devices defined by a common API behaviour, to enable implementation of **generic userspace code**.

Subdev configuration model

- The first configuration model to be introduced is the **common raw camera sensor configuration model** (`V4L2_CONFIG_MODEL_COMMON_RAW_SENSOR`).
- The model defines a source pad (#0), an internal image pad (#1) and an internal embedded data pad (#2). It documents how formats and selection rectangles represent on all those pads and how they behave.



Internal pads have no standard graphical representation in a media graph yet. Volunteer graphic designers can apply.

Subdev configuration model

- The raw camera sensor configuration model mandates usage of **raw media bus codes**. Those new media bus codes, named `MEDIA_BUS_FMT_RAW_{8,10,12,14}`, represent raw pixel data transported over a serial bus.
- Unlike existing Bayer media bus codes, they do not encode the Colour Filter Array (CFA) pattern. The CFA pattern is conveyed out-of-band through the new `V4L2_CID_COLOR_PATTERN` control.

```
#define V4L2_CID_COLOR_PATTERN      (V4L2_CID_IMAGE_SOURCE_CLASS_BASE + 10)
#define V4L2_COLOR_PATTERN_GRBG     0U
#define V4L2_COLOR_PATTERN_RGGB     1U
#define V4L2_COLOR_PATTERN_BGGR     2U
#define V4L2_COLOR_PATTERN_GBRG     3U
```

- New pixel formats for raw image data and metadata (embedded data) are also defined by the same patch series.

Raw formats

- Work in progress from Sakari & co, "[PATCH v11 00/66] Generic line based metadata support, internal pads" (*https://lore.kernel.org/r/20250825095107.1332313-1-sakari.ailus@linux.intel.com*) posted to the linux-media mailing list.
- Reviews are needed and appreciated.

Metadata & internal pads

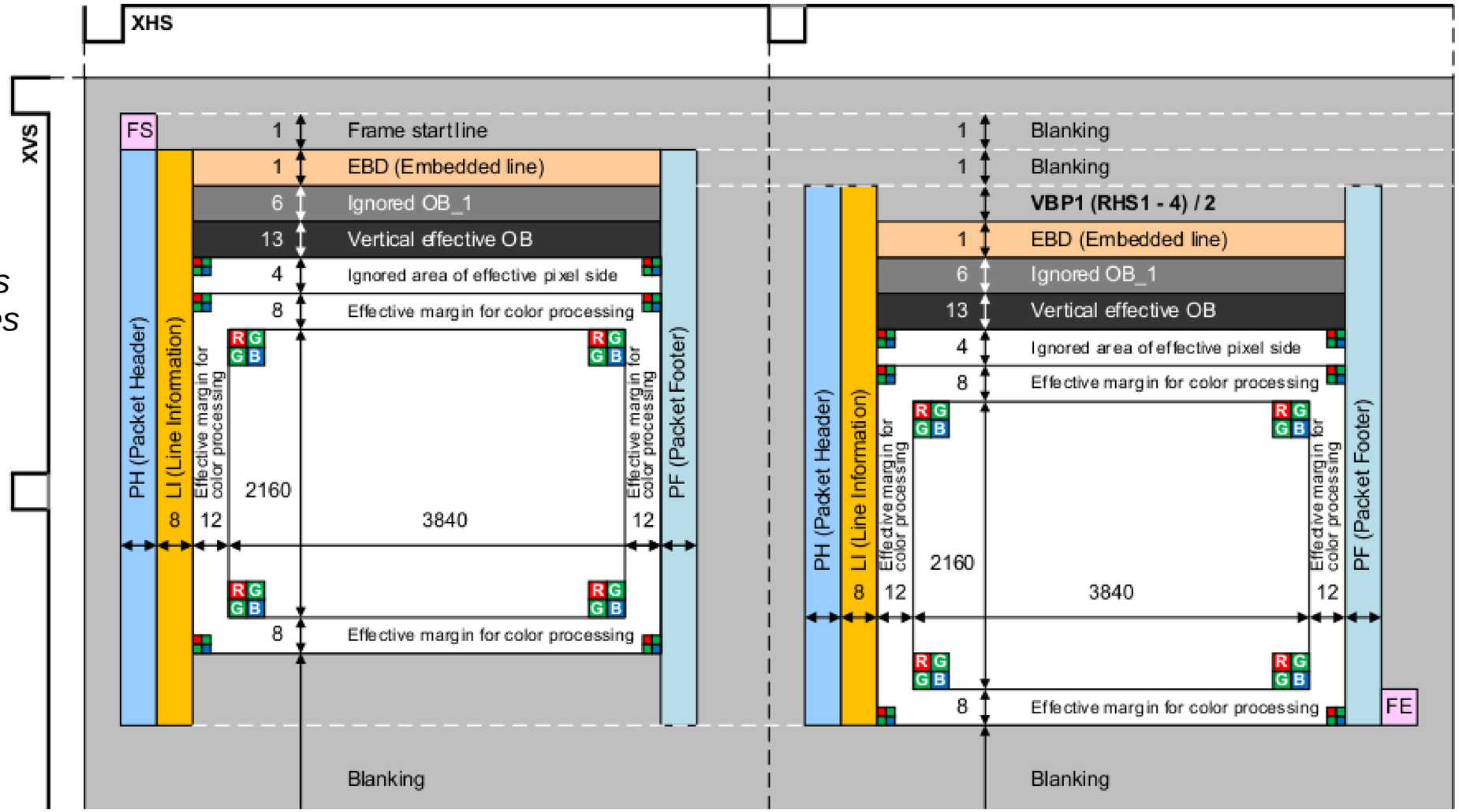
How about HDR sensors?

HDR camera sensors

There are many types of HDR:

- Multi-exposure or multi-gains
- Line-interleaved or alternating frames
- 2, 3 or 4 exposures
- Stitching in the sensor or on the receiver side
- …

Typical DOL (multi-exposure) data transmission. Sensors transmit the exposures with different DTs or VCs.



HDR camera sensors

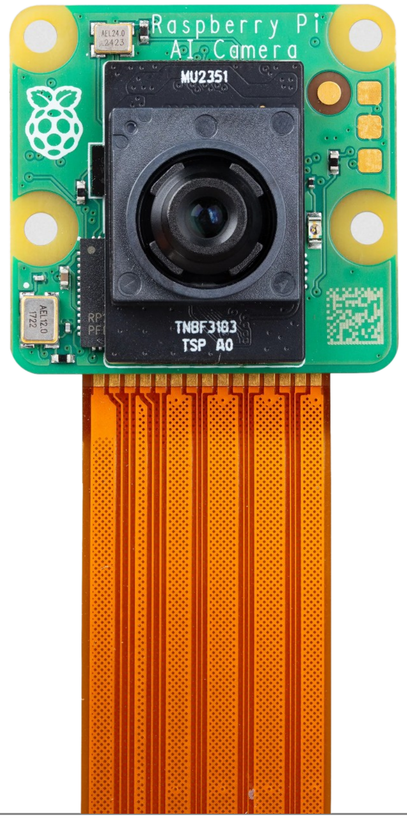
- Focussing first on DOL (multi-exposure) and DCG (multi-gains) with interleaved lines and stitching on the host side. Other developers may have different priorities.
- Use **multiple streams** originating from the same internal image pad.
- VC and DT values currently hardcoded in the source.
- Will require standardization in the raw camera sensor configuration model.
- `V4L2_CID_HDR_SENSOR_MODE` already merged, but its values are not standardized.
- Still WIP, now is a good time to influence the design.

HDR camera sensors

- Will require new controls to set the multiple exposures or gains.
- RFC by Mirela Rabulea from NXP: “[RFC v2 0/5] Add standard exposure and gain controls for multiple captures” (https://lore.kernel.org/r/20250818155809.469479-1-mirela.rabulea@nxp.com).
- Defines new `V4L2_CID_EXPOSURE_MULTI`, `V4L2_CID_AGAIN_MULTI`, and `V4L2_CID_DGAIN_MULTI` controls.
- Standardize on multiple exposure times, emulate if the sensor only supports exposure ratios.

HDR camera sensors

And sensors with integrated NPUs?



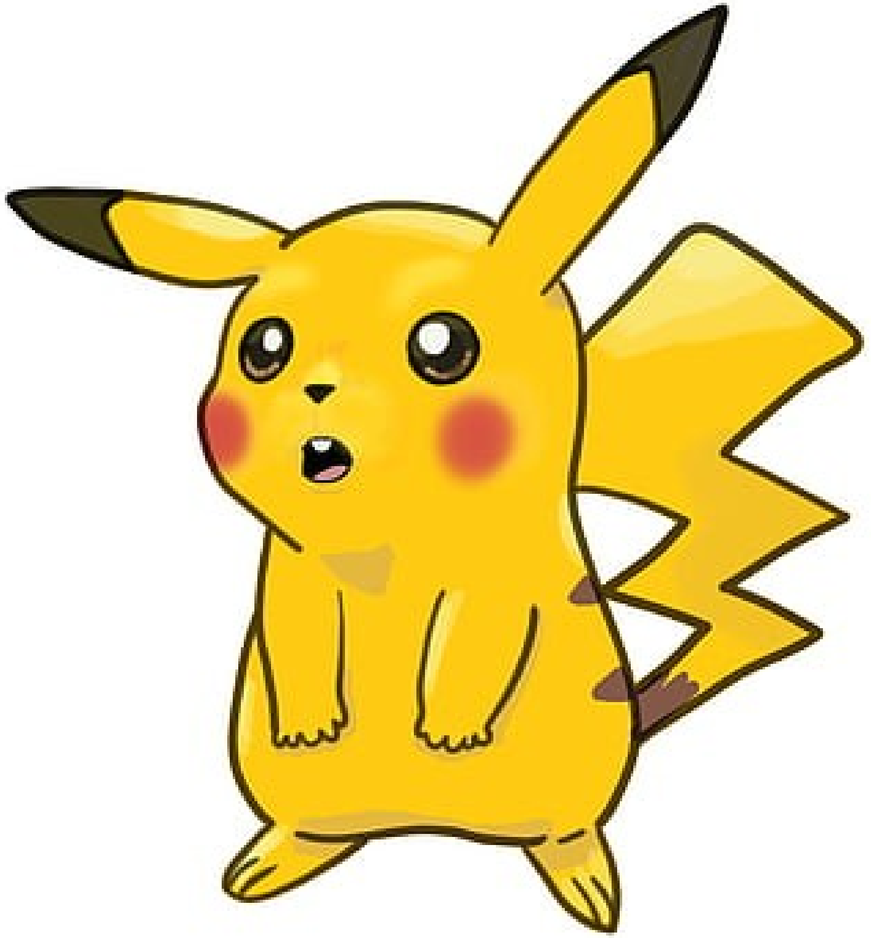
Sony IMX500

- Tensors sent over a separate DT or VC.
- Use streams and routing, as for embedded data or HDR.
- Work of course required to define APIs to control the NPU.

“Streamlining AI Solutions Development: Seamless Development from Device to Cloud Harnessing WebAssembly on the Edge”, Shinsuke Tashiro and Munehiro Shimomura, Tue 16:20 CEST

Misconception #3.3

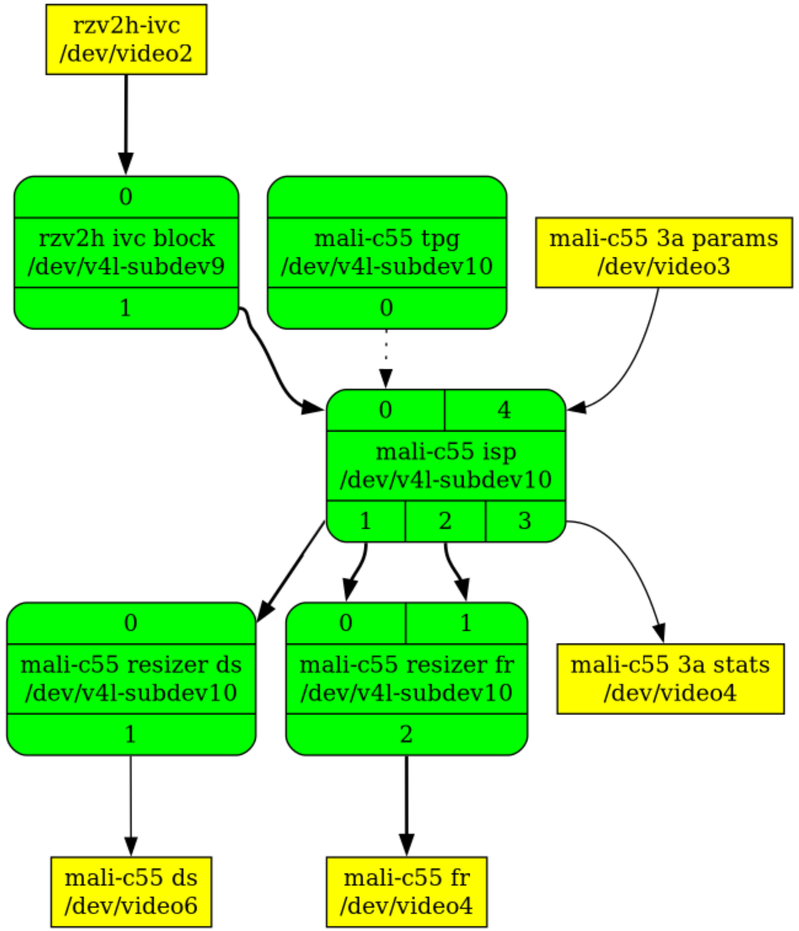
V4L2 can’t properly handle ISPs



IDEAS ON BOARD

Proven design with multiple drivers.

Supports both inline and offline (memory-to-memory) ISPs.



ISP parameters passed through buffers queued on a metadata video device.

ISP statistics captured from a metadata video device.

Enable/disable inputs and outputs through link setup.

Video devices used to capture process images, and feed raw images to the ISP.

ISPs in V4L2

Misconception #4

DRM is better suited for offline ISPs

- That’s a loaded question with a bit of an agenda.
- DRM does not have support for camera sensors or complex pipelines of serializers and deserializers, and nobody is proposing adding it. ISP support would be limited to offline ISPs.
- ISP cores can typically be integrated as inline or offline in an SoC, and sometimes the integration supports switching between those two modes dynamically. Using DRM for offline mode would either preclude inline mode support, or require two separate drivers (DRM and V4L2).
- There are claims that V4L2 requires documenting 100% of the hardware features while DRM is more “pragmatic”. This is not true, discussions since LPC 2025 made it clear that the two subsystems do not differ significantly in that area.

ISPs – DRM or V4L2?

- On the technical side, V4L2 was until recently lacking support for multi-context operation of offline ISPs (time-multiplexing between multiple cameras).
- Work in progress from Jacopo Mondi, presented during the Linux Media Summit 2025 (https://ideasonboard.org/talks/20250513-media-summit-multi-context.pdf)
- “[PATCH v2 00/27] media: Add support for multi-context operations” (https://lore.kernel.org/r/20250724-multicontext-mainline-2025-v2-0-c9b316773486@ideasonboard.com) posted to the linux-media mailing list.

Multi-context support in V4L2

- Contexts provide multiple virtual instances of a physical memory-to-memory device.
- The top-level context object is associated with the media device. Each video device and subdev also has a local context that is bound to the media device context.
- A context stores information about the device. It complement subdev states: each subdev context stores a subdev ACTIVE state. Video device contexts store a vb2 queue. All together, the context and video device states provide information about the whole device.
- Create a media device context by opening media device node (automatic).
- Create and bind a video device or subdev context to the media device context with `VIDIOC_BIND_CONTEXT` and `VIDIOC_SUBDEV_BIND_CONTEXT`.
- Will help codecs that need to manage reconstruction buffers from userspace.

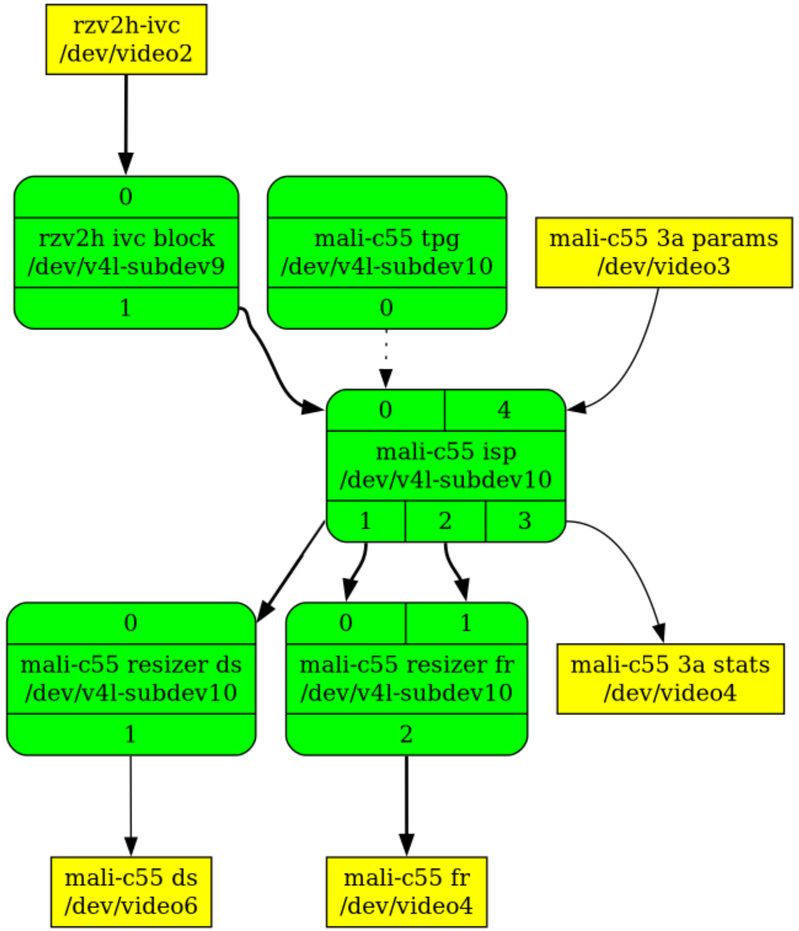
Multi-context support in V4L2

- Similar to subdev states but for video devices.
- Initial RFC “[PATCH RFC 0/3] Add support for video device state for capture devices” (https://lore.kernel.org/r/20250703-vdev-state-v1-0-d647a5e4986d@ideasonboard.com) being rewritten to be much more like subdev state.
- The video device context will be passed to all IOCTL handlers. The short term goal is simplifications of drivers, the longer term goal is to support other larger improvements.
- Preparatory work “[PATCH v3 00/76] media: Rationalise usage of v4l2_fh” (https://lore.kernel.org/r/20250810013100.29776-1-laurent.pinchart+renesas@ideasonboard.com) was merged for v6.18.

Video device state

The IVC is an IP core specific to the SoC, designed to provide frames to the ISP. The rest of the ISP is a generic IP core licenses by Arm.

V4L2 assumes that all video nodes are managed by a single driver that can coordinate their operation. This is not the case here.



Media jobs

- The media jobs framework is designed to tie together multiple drivers that provide video devices.
- The framework lets drivers declare a set of steps for a job, with ordering and dependencies. It runs all steps in the job once all dependencies are met.
- “[PATCH v2 0/3] Add media jobs framework” (https://lore.kernel.org/r/ 20250624-media-jobs-v2-0-8e649b069a96@ideasonboard.com) posted on the linux-media mailing list. Some design questions remain unanswered.
- DRM does not provide any standard framework to solve this issue, it would need to be addressed with driver-specific solutions.

Media jobs

- The media jobs framework wasn’t designed to provide a fully-fledged scheduler, but it may need to do so in the future. There is overlap with the scheduling needs of media context and V4L2 M2M devices, and a desire to cover all those needs with a single implementation.
- Share code with the DRM scheduler seems impractical, but it may provide inspiration.
- A future version will likely make interoperability with DRM easier, probably by using the same dma fences mechanism on the kernel side.

Media jobs

What about userspace?

- v4l2-compliance tests for all new APIs.
- All APIs also need to be tested in a real userspace framework, not just in test applications.
- libcamera is the de facto userspace framework for cameras on Linux.

Userspace

Is that all?

Lots of future (and yet unscheduled) work:

- Full scheduler for media jobs, shared with codecs and multi-context devices.
- New request API on media device, based on states.
- Fences (synchronisation objects) support.
- Per-plane data offset for multi-planar formats.

What else would **you** like to see in V4L2?

Future

laurent.pinchart@ideasonboard.com

Veel bedankt