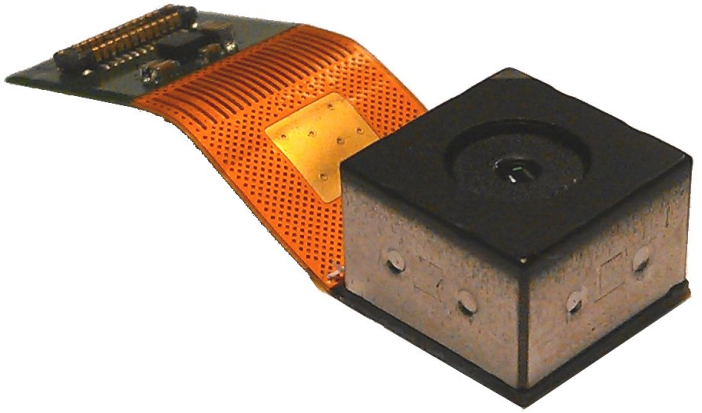
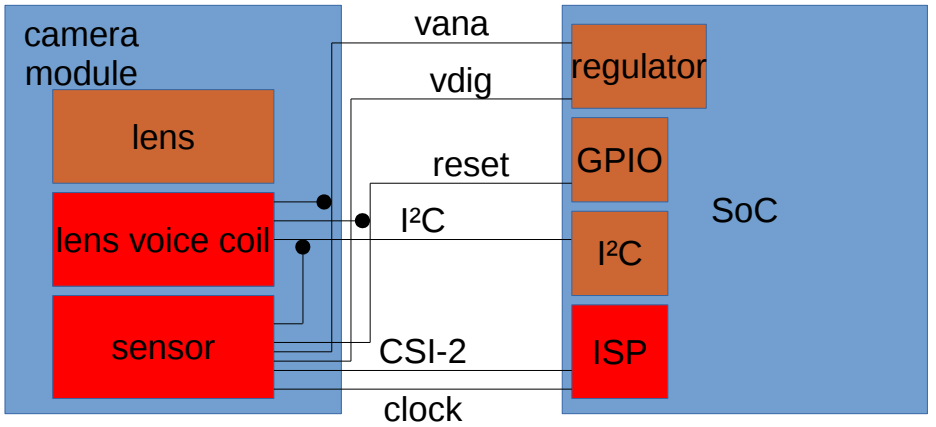
Cameras in embedded systems: Device tree and ACPI view

Sakari Ailus
<sakari.ailus@linux.intel.com>
2016-10-07

A typical embedded system with a camera

- Image Signal Processor
- Raw camera sensor
- Lens voice coil





Raw sensors

- Raw sensors have little processing logic in the sensor itself
  - Analogue and digital gain but not much more

This is how white looks like! -->

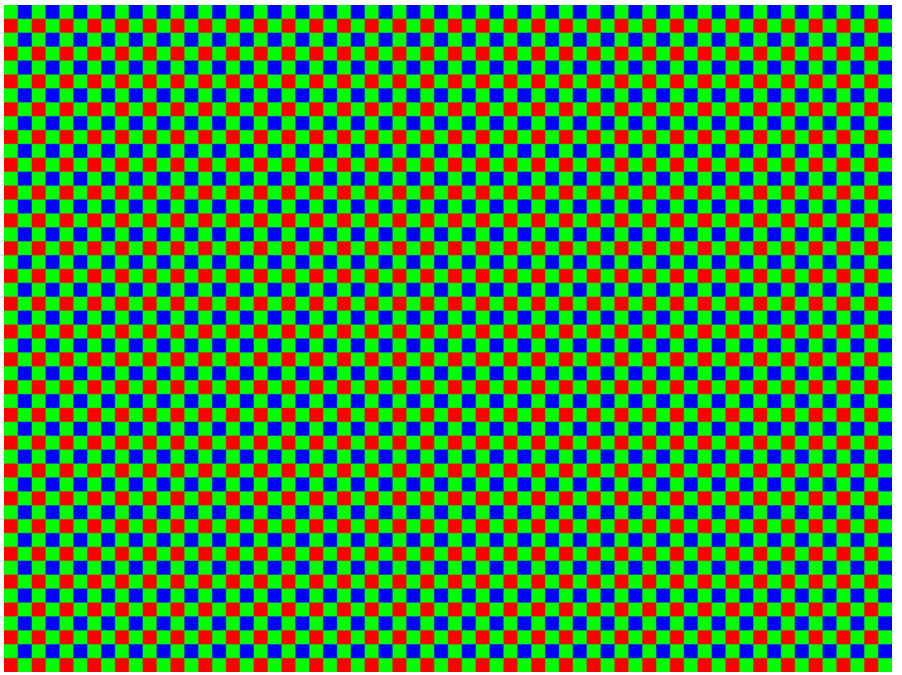


Image signal processors

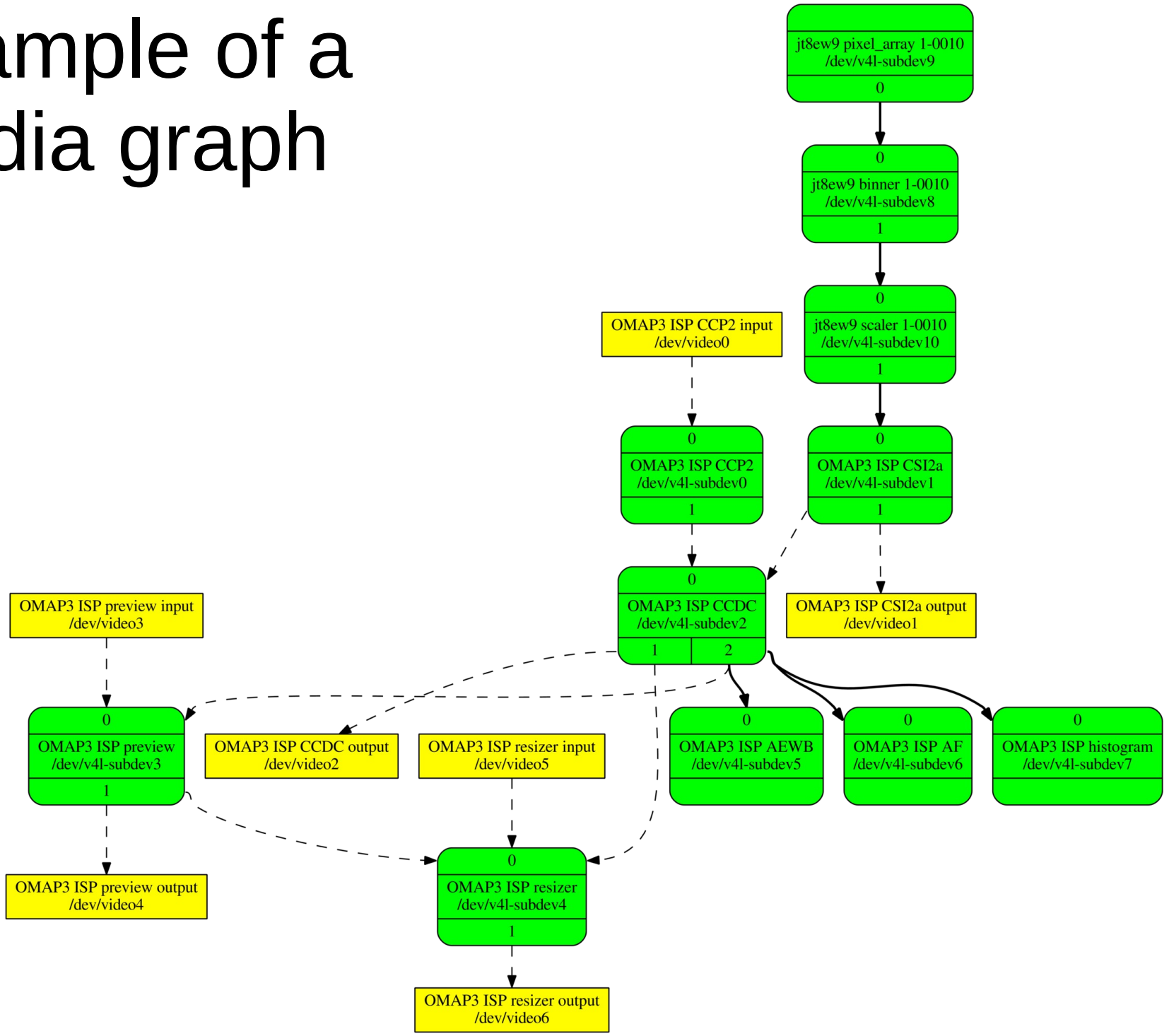
- Process the image for viewing

After ISP processing white looks like this --->

Video4Linux and Media controller

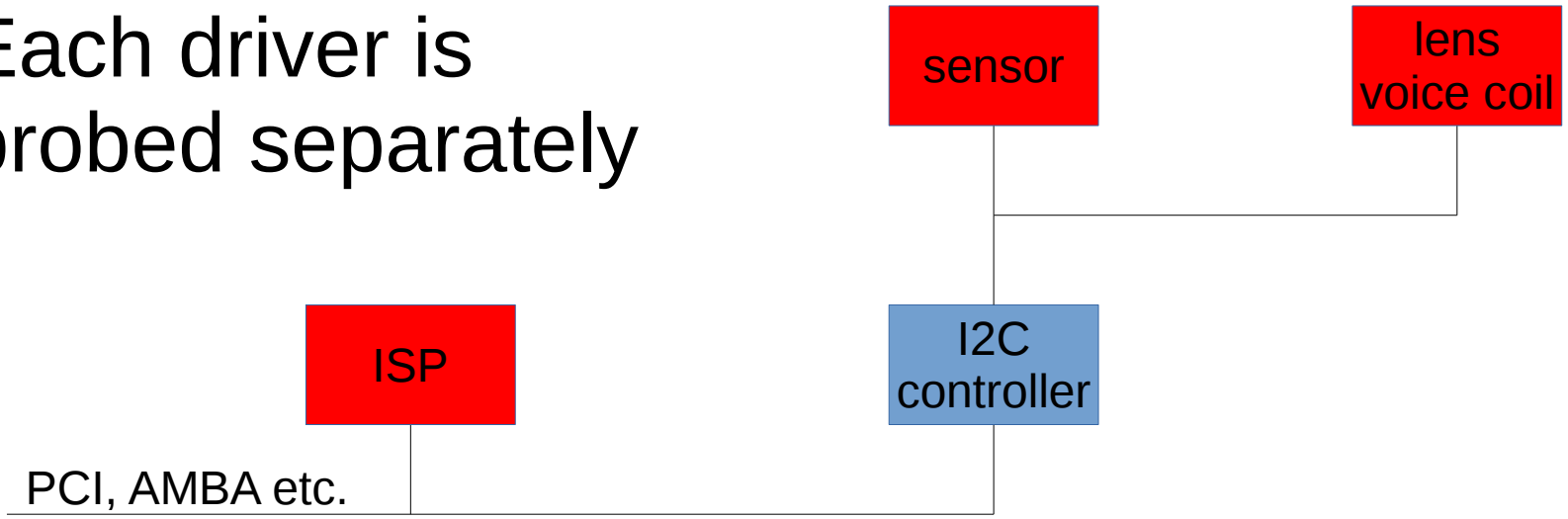
- Video4Linux (or V4L2) is the Linux API for capturing images
  - Video capture cards
  - USB webcams
  - Cameras in embedded devices
- Media controller is a control interface for complex media devices
  - Image pipeline discovery and configuration
  - Device discovery

Example of a media graph



Probing

- Each driver is probed separately



- How to tell drivers they all are part of the same media device?

Media device setup

1.media_device_init()

2.v4l2_device_register()

3.video_register_device()

4.v4l2_device_register_subdev(sensor)

5.v4l2_device_register_subdev(isp)

6.v4l2_register_subdev_nodes()

7.media_device_register()

ISP driver

sensor driver

Device tree

Device tree

- System hardware description in an easily editable and human readable format
- Originates from Sparc / Open Firmware
- Primarily used on embedded systems
- Tree structure
  - Nodes
  - Properties
- Source code dompiled into binary before use

Sensor node

```
&i2c2 {
	smia_1: camera@10 {
		compatible = "nokia,smia";
		reg = <0x10>;
		/* No reset gpio */
		vana-supply = <&vaux3>;
		clocks = <&isp 0>;
		clock-frequency = <9600000>;
		nokia,nvm-size = <(16 * 64)>;
		port {
			smia_1_1: endpoint {
				link-frequencies = /bits/ 64 <199200000 210000000 499200000>;
				clock-lanes = <0>;
				data-lanes = <1 2>;
				remote-endpoint = <&csi2a_ep>;
			};
		};
	};
};
```

source: arch/arm/boot/dts/omap3-n9.dts

ISP node board specific part

```
&isp {
        vdd-csiphy1-supply = <&vaux2>;
        vdd-csiphy2-supply = <&vaux2>;
        ports {
                port@2 {
                        reg = <2>;
                        csi2a_ep: endpoint {
                                remote-endpoint = <&smia_1_1>;
                                clock-lanes = <2>;
                                data-lanes = <1 3>;
                                crc = <1>;
                                lane-polarities = <1 1 1>;
                        };
                };
        };
};
```

source: arch/arm/boot/dts/omap3-n9.dts

V4L2 async

V4L2 async

- The V4L2 async framework facilitates sub-device registration
- V4L2 sub-device device node creation and media device registration postponed after media device driver's probe function
- To do its job, the V4L2 async framework makes use of firmware provided information

V4L2 async example

- Two devices: foo and bar
- foo is a sensor driver
- bar is an ISP driver

foo: Registering an async sub-device

```
static int foo_probe(struct device *dev)
{
        int ret;

        ret = v4l2_async_register_subdev(subdev); /* struct v4l2_subdev */
        if (ret) {
            dev_err(dev, ”ouch\n”);
            return -ENODEV;
        }
        return 0;
}
```

struct bar_device

```
struct bar_device {
    struct media_device mdev;
    struct v4l2_device v4l2_dev;
    struct v4l2_async_notifier notifier;
    struct *subdevs[BAR_MAX_SUBDEVS];
};
```

bar_probe()

```
static int bar_probe(struct device *dev)
{
    struct bar_device *bar = kmalloc(sizeof(*bar));

    media_device_init(&bar->mdev);
    bar->dev = dev;
    bar->notifier.subdevs = kcalloc(BAR_MAX_SUBDEVS, sizeof(struct
        v4l2_async_subdev));
    bar_parse_nodes(bar);
    bar->notifier.bound = bar_bound;
    bar->notifier.complete = bar_complete;
    v4l2_async_notifier_register(&bar->v4l2_dev, &bar->notifier);
}
```

bar_parse_nodes()

```
struct bar_async {
    struct v4l2_async_subdev asd;
    struct v4l2_subdev *sd;
};
static void bar_parse_nodes(struct device *dev, struct v4l2_async_notifier *n)
{
        struct device_node *node = NULL;

        while ((node = of_graph_get_next_endpoint(dev->of_node, node))) {
            struct bar_async *ba = kmalloc(sizeof(*ba));
            n->subdevs[n->num_subdevs++] = &ba->asd;
            ba->asd.match.of.node = of_graph_get_remote_port_parent(node);
            ba->asd.match_type = V4L2_ASYNC_MATCH_OF;
        }
}
```

V4L2 async

- On registering a sub-device or a notifier registration with the V4L2 async framework
  - The existing information is matched against what was added
- When a match is found
  - the bound() callback of the notifier is called and
  - the sub-device which was bound is registered with the media device
- When all async sub-devices have been found
  - the complete() callback of the notifier is called

bar_bound() and bar_complete()

```
static int bar_bound(struct v4l2_async_notifier *n, struct v4l2_subdev *sd, struct v4l2_async_subdev
*asd)
{
    struct bar_async *ba = container_of(asd, struct bar_async, asd);
    ba->sd = sd;
}
static int bar_complete(struct v4l2_async_notifier *n)
{
    struct bar_device *bar = container_of(n, struct bar_async, notifier);
    struct v4l2_device *v4l2_dev = &isp->v4l2_dev;

    v4l2_device_register_subdev_nodes(&bar->v4l2_dev);
    media_device_register(&bar->mdev);
}
```