# Writing camera sensor drivers

## CSI-2 and parallel (BT.601 and BT.656) busses

Please see :ref:`transmitter-receiver`.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]camera-sensor.rst, line 9); *backlink***
>
> Unknown interpreted text role "ref".

## Handling clocks

Camera sensors have an internal clock tree including a PLL and a number of divisors. The clock tree is generally configured by the driver based on a few input parameters that are specific to the hardware:: the external clock frequency and the link frequency. The two parameters generally are obtained from system firmware. **No other frequencies should be used in any circumstances.**

The reason why the clock frequencies are so important is that the clock signals come out of the SoC, and in many cases a specific frequency is designed to be used in the system. Using another frequency may cause harmful effects elsewhere. Therefore only the pre-determined frequencies are configurable by the user.

### ACPI

Read the `clock-frequency` _DSD property to denote the frequency. The driver can rely on this frequency being used.

### Devicetree

The currently preferred way to achieve this is using `assigned-clocks`, `assigned-clock-parents` and `assigned-clock-rates` properties. See `Documentation/devicetree/bindings/clock/clock-bindings.txt` for more information. The driver then gets the frequency using `clk_get_rate()`.

This approach has the drawback that there's no guarantee that the frequency hasn't been modified directly or indirectly by another driver, or supported by the board's clock tree to begin with. Changes to the Common Clock Framework API are required to ensure reliability.

## Frame size

There are two distinct ways to configure the frame size produced by camera sensors.

### Freely configurable camera sensor drivers

Freely configurable camera sensor drivers expose the device's internal processing pipeline as one or more sub-devices with different cropping and scaling configurations. The output size of the device is the result of a series of cropping and scaling operations from the device's pixel array's size.

An example of such a driver is the CCS driver (see `drivers/media/i2c/ccs`).

### Register list based drivers

Register list based drivers generally, instead of able to configure the device they control based on user requests, are limited to a number of preset configurations that combine a number of different parameters that on hardware level are independent. How a driver picks such configuration is based on the format set on a source pad at the end of the device's internal pipeline.

Most sensor drivers are implemented this way, see e.g. `drivers/media/i2c/imx319.c` for an example.

## Frame interval configuration

There are two different methods for obtaining possibilities for different frame intervals as well as configuring the frame interval. Which one to implement depends on the type of the device.

### Raw camera sensors

Instead of a high level parameter such as frame interval, the frame interval is a result of the configuration of a number of camera sensor implementation specific parameters. Luckily, these parameters tend to be the same for more or less all modern raw camera sensors.

The frame interval is calculated using the following equation:

```
frame interval = (analogue crop width + horizontal blanking) *
                 (analogue crop height + vertical blanking) / pixel rate
```

The formula is bus independent and is applicable for raw timing parameters on large variety of devices beyond camera sensors. Devices that have no analogue crop, use the full source image size, i.e. pixel array size.

Horizontal and vertical blanking are specified by `V4L2_CID_HBLANK` and `V4L2_CID_VBLANK`, respectively. The unit of the `V4L2_CID_HBLANK` control is pixels and the unit of the `V4L2_CID_VBLANK` is lines. The pixel rate in the sensor's **pixel array** is specified by `V4L2_CID_PIXEL_RATE` in the same sub-device. The unit of that control is pixels per second.

Register list based drivers need to implement read-only sub-device nodes for the purpose. Devices that are not register list based need these to configure the device's internal processing pipeline.

The first entity in the linear pipeline is the pixel array. The pixel array may be followed by other entities that are there to allow configuring binning, skipping, scaling or digital crop :ref:`v4l2-subdev-selections`.

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]camera-sensor.rst, line 107); *backlink***

Unknown interpreted text role "ref".

### USB cameras etc. devices

USB video class hardware, as well as many cameras offering a similar higher level interface natively, generally use the concept of frame interval (or frame rate) on device level in firmware or hardware. This means lower level controls implemented by raw cameras may not be used on uAPI (or even kAPI) to control the frame interval on these devices.

## Power management

Always use runtime PM to manage the power states of your device. Camera sensor drivers are in no way special in this respect: they are responsible for controlling the power state of the device they otherwise control as well. In general, the device must be powered on at least when its registers are being accessed and when it is streaming.

Existing camera sensor drivers may rely on the old struct v4l2_subdev_core_ops->s_power() callback for bridge or ISP drivers to manage their power state. This is however **deprecated**. If you feel you need to begin calling an s_power from an ISP or a bridge driver, instead please add runtime PM support to the sensor driver you are using. Likewise, new drivers should not use s_power.

Please see examples in e.g. `drivers/media/i2c/ov8856.c` and `drivers/media/i2c/ccs/ccs-core.c`. The two drivers work in both ACPI and DT based systems.

### Control framework

`v4l2_ctrl_handler_setup()` function may not be used in the device's runtime PM `runtime_resume` callback, as it has no way to figure out the power state of the device. This is because the power state of the device is only changed after the power state transition has taken place. The `s_ctrl` callback can be used to obtain device's power state after the power state transition:

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]camera-sensor.rst, line 149)**

Unknown directive type "c:function".

```
.. c:function:: int pm_runtime_get_if_in_use(struct device *dev);
```

The function returns a non-zero value if it succeeded getting the power count or runtime PM was disabled, in either of which cases the driver may proceed to access the device.