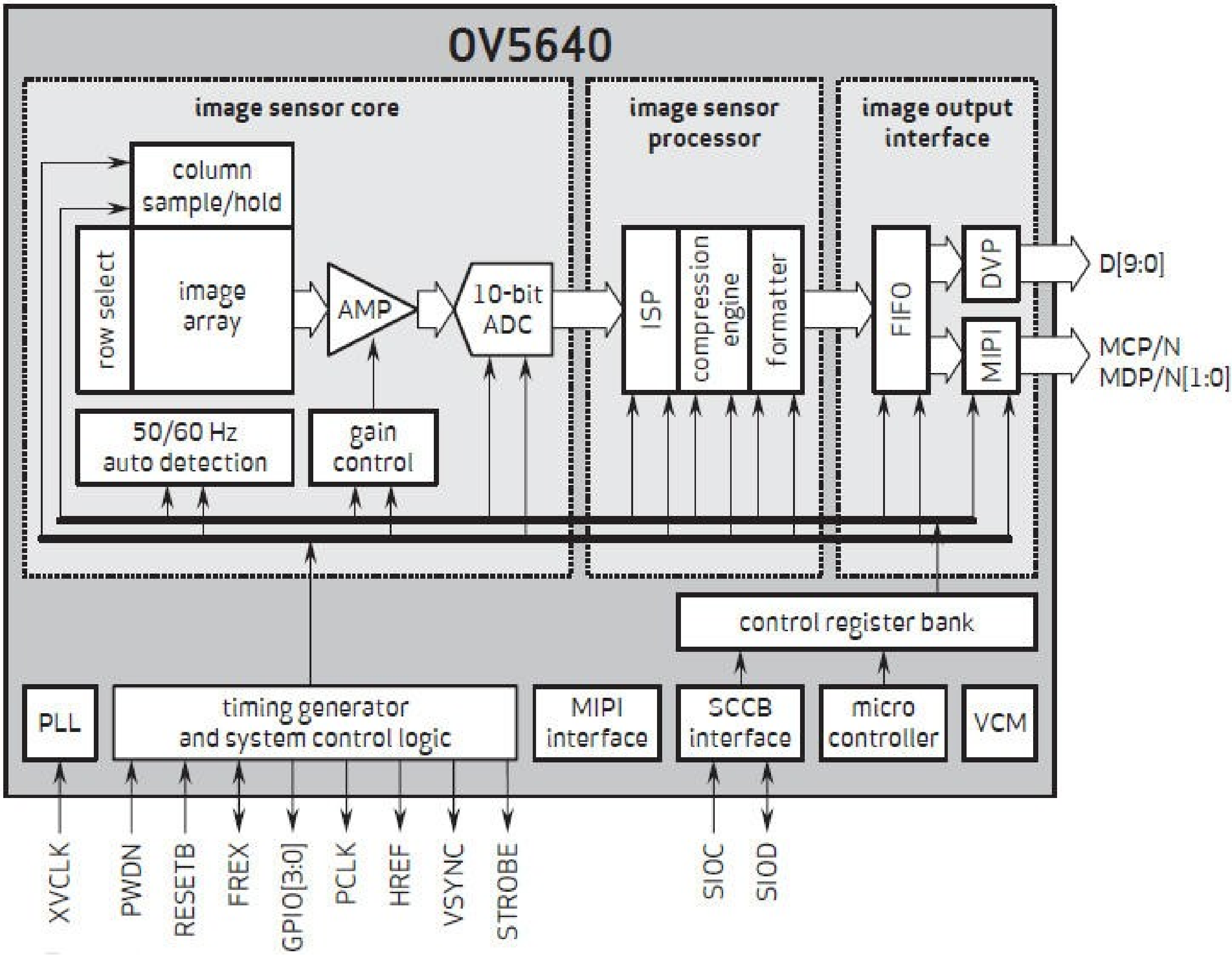
Table of Contents

OV5640 SENSOR INFORMATION

optical size of 1/4"
automatic image control functions: automatic exposure control (AEC), automatic white balance (AWB), automatic band filter (ABF), automatic 50/60 Hz luminance detection, and automatic black level calibration (ABLC)
programmable controls for frame rate, AEC/AGC 16-zone size/position/weight control, mirror and flip, cropping, windowing, and panning
image quality controls: color saturation, hue, gamma, sharpness (edge enhancement), lens correction, defective pixel canceling, and noise canceling
support for output formats: RAW RGB, RGB565/555/444, CCIR656, YUV422/420, YCbCr422, and compression
support for video or snapshot operations
support for horizontal and vertical sub-sampling, binning
digital video port (DVP) parallel output interface and **dual lane MIPI output interface**

support for images sizes: 5 mega pixel, and any arbitrary size scaling down from 5 mega pixel
support for auto focus control (AFC) with embedded AF VCM driver
active array size:2592 x 1944
power supply: core: 1.5V ± 5% (with embedded 1.5V regulator) analog: 2.6 ~ 3.0V (2.8V typical), I/O: 1.8V / 2.8V
Input clock frequency:6~27 MHz
maximum image transfer rate: QSXGA (2592x1944): 15 fps 1080p: 30 fps 1280x960: 45 fps 720p: 60 fps VGA (640x480): 90 fps QVGA (320x240): 120 fps
shutter: rolling shutter / frame exposure


OV5640
image sensor core
image sensor processor
image output interface
column sample/hold
row select
image array
AMP
10-bit ADC
50/60 Hz auto detection
gain control
ISP
compression engine
formatter
FIFO
DVP
MIPI
D[9:0]
MCP/N
MDP/N[1:0]
control register bank
PLL
timing generator and system control logic
MIPI interface
SCCB interface
micro controller
VCM
XVCLK
PWDN
RESETB
FREX
GPIO[3:0]
PCLK
HREF
VSYNC
STROBE
SIOC
SIOD

OV5640 Sensor Driver

Step 1: Kernel Version, Driver location

```
VERSION = 4
PATCHLEVEL = 19
SUBLEVEL = 0
EXTRAVERSION =
```

linux-xlnx-master/drivers/media/i2c/ov5640.c

Step 2: Driver Register / Init function

```
static const struct i2c_device_id ov5640_id[] = {
        {"ov5640", 0},
        {},
};
MODULE_DEVICE_TABLE(i2c, ov5640_id);

static const struct of_device_id ov5640_dt_ids[] = {
        { .compatible = "ovti,ov5640" },
        { /* sentinel */ }
};
MODULE_DEVICE_TABLE(of, ov5640_dt_ids);

static struct i2c_driver ov5640_i2c_driver = {
        .driver = {
                .name  = "ov5640",
                .of_match_table   = ov5640_dt_ids,
        },
        .id_table = ov5640_id,
        .probe    = ov5640_probe,
        .remove   = ov5640_remove,
};

module_i2c_driver(ov5640_i2c_driver);
```

Step 3: Probe function

```
static int ov5640_probe(struct i2c_client *client, const struct i2c_device_id *id)
{
        struct device *dev = &client->dev;
        struct fwnode_handle *endpoint;
        struct ov5640_dev *sensor;
        struct v4l2_mbus_framefmt *fmt;
        u32 rotation;
        int ret;

        sensor = devm_kzalloc(dev, sizeof(*sensor), GFP_KERNEL);
        if (!sensor)
                return -ENOMEM;

        sensor->i2c_client = client;

        /*
         * default init sequence initialize sensor to
         * YUV422 UYVY VGA@30fps
         */
        fmt = &sensor->fmt;
        fmt->code = MEDIA_BUS_FMT_UYVY8_2X8;
        fmt->colorspace = V4L2_COLORSPACE_SRGB;
        fmt->ycbcr_enc = V4L2_MAP_YCBCR_ENC_DEFAULT(fmt->colorspace);
        fmt->quantization = V4L2_QUANTIZATION_FULL_RANGE;
        fmt->xfer_func = V4L2_MAP_XFER_FUNC_DEFAULT(fmt->colorspace);
        fmt->width = 640;
        fmt->height = 480;
        fmt->field = V4L2_FIELD_NONE;
        sensor->frame_interval.numerator = 1;
        sensor->frame_interval.denominator =
        ov5640_framerates[OV5640_30_FPS];
        sensor->current_fr = OV5640_30_FPS;
        sensor->current_mode =
                &ov5640_mode_data[OV5640_30_FPS]
        [OV5640_MODE_VGA_640_480];
        sensor->last_mode = sensor->current_mode;

        sensor->ae_target = 52;

        /* optional indication of physical rotation of sensor */
        ret = fwnode_property_read_u32(dev_fwnode(&client->dev), "rotation",
                                      &rotation);
        if (!ret) {
                switch (rotation) {
                case 180:
```

```
            sensor->upside_down = true;
            /* fall through */
        case 0:
            break;
        default:
            dev_warn(dev, "%u degrees rotation is not supported,
ignoring...\n",
                 rotation);
        }
    }

    endpoint = fwnode_graph_get_next_endpoint(dev_fwnode(&client-
>dev),   NULL);
    if (!endpoint) {
        dev_err(dev, "endpoint node not found\n");
        return -EINVAL;
    }

    ret = v4l2_fwnode_endpoint_parse(endpoint, &sensor->ep);
    fwnode_handle_put(endpoint);
    if (ret) {
        dev_err(dev, "Could not parse endpoint\n");
        return ret;
    }

    /* get system clock (xclk) */
    sensor->xclk = devm_clk_get(dev, "xclk");
    if (IS_ERR(sensor->xclk)) {
        dev_err(dev, "failed to get xclk\n");
        return PTR_ERR(sensor->xclk);
    }

    sensor->xclk_freq = clk_get_rate(sensor->xclk);
    if (sensor->xclk_freq < OV5640_XCLK_MIN ||
        sensor->xclk_freq > OV5640_XCLK_MAX) {
        dev_err(dev, "xclk frequency out of range: %d Hz\n",
            sensor->xclk_freq);
        return -EINVAL;
    }

    /* request optional power down pin */
    sensor->pwdn_gpio = devm_gpiod_get_optional(dev, "powerdown",
                          GPIOD_OUT_HIGH);
    /* request optional reset pin */
    sensor->reset_gpio = devm_gpiod_get_optional(dev, "reset",
                          GPIOD_OUT_HIGH);

    v4l2_i2c_subdev_init(&sensor->sd, client, &ov5640_subdev_ops);
```

```
        sensor->sd.flags |= V4L2_SUBDEV_FL_HAS_DEVNODE;
        sensor->pad.flags = MEDIA_PAD_FL_SOURCE;
        sensor->sd.entity.function = MEDIA_ENT_F_CAM_SENSOR;
        ret = media_entity_pads_init(&sensor->sd.entity, 1, &sensor->pad);
        if (ret)
                return ret;

        ret = ov5640_get_regulators(sensor);
        if (ret)
                return ret;

        mutex_init(&sensor->lock);

        ret = ov5640_check_chip_id(sensor);
        if (ret)
                goto entity_cleanup;

        ret = ov5640_init_controls(sensor);
        if (ret)
                goto entity_cleanup;

        ret = v4l2_async_register_subdev(&sensor->sd);
        if (ret)
                goto free_ctrls;

        return 0;

free_ctrls:
        v4l2_ctrl_handler_free(&sensor->ctrls.handler);
entity_cleanup:
        mutex_destroy(&sensor->lock);
        media_entity_cleanup(&sensor->sd.entity);
        return ret;
}
```

Step 4: struct v4l2_subdev_ops

```
static const struct v4l2_subdev_core_ops ov5640_core_ops = {
        .s_power = ov5640_s_power,
};

static const struct v4l2_subdev_video_ops ov5640_video_ops = {
        .g_frame_interval = ov5640_g_frame_interval,
        .s_frame_interval = ov5640_s_frame_interval,
        .s_stream = ov5640_s_stream,
};

static const struct v4l2_subdev_pad_ops ov5640_pad_ops = {
        .enum_mbus_code = ov5640_enum_mbus_code,
        .get_fmt = ov5640_get_fmt,
        .set_fmt = ov5640_set_fmt,
        .enum_frame_size = ov5640_enum_frame_size,
        .enum_frame_interval = ov5640_enum_frame_interval,
};

static const struct v4l2_subdev_ops ov5640_subdev_ops = {
        .core = &ov5640_core_ops,
        .video = &ov5640_video_ops,
        .pad = &ov5640_pad_ops,
};
```

Step 5:   ov5640_s_power

```
static int ov5640_s_power(struct v4l2_subdev *sd, int on)
{
        struct ov5640_dev *sensor = to_ov5640_dev(sd);
        int ret = 0;

        mutex_lock(&sensor->lock);

        if (sensor->power_count == !on) {
                ret = ov5640_set_power(sensor, !!on);
                if (ret)
                        goto out;
        }

        /* Update the power count. */
        sensor->power_count += on ? 1 : -1;
        WARN_ON(sensor->power_count < 0);
out:
        mutex_unlock(&sensor->lock);
         return ret;
}


static int ov5640_set_power(struct ov5640_dev *sensor, bool on)
{
        int ret = 0;
```

```
if (on) {
        ret = ov5640_set_power_on(sensor);
        if (ret)
                return ret;

        ret = ov5640_restore_mode(sensor);
        if (ret)
                goto power_off;

           /*
         * Power up MIPI HS Tx and LS Rx; 2 data lanes mode
         *
         * 0x300e = 0x40
         * [7:5] = 010  : 2 data lanes mode (see FIXME note in
         *                "ov5640_set_stream_mipi()")
         * [4] = 0      : Power up MIPI HS Tx
         * [3] = 0      : Power up MIPI LS Rx
         * [2] = 0      : MIPI interface disabled
         */
        ret = ov5640_write_reg(sensor,
                               OV5640_REG_IO_MIPI_CTRL00, 0x40);
        if (ret)
                goto power_off;

        /*
         * Gate clock and set LP11 in 'no packets mode' (idle)
         *
         * 0x4800 = 0x24
         * [5] = 1      : Gate clock when 'no packets'
         * [2] = 1      : MIPI bus in LP11 when 'no packets'
         */
        ret = ov5640_write_reg(sensor,
                               OV5640_REG_MIPI_CTRL00, 0x24);
        if (ret)
                goto power_off;
        /*
         * Set data lanes and clock in LP11 when 'sleeping'
         *
         * 0x3019 = 0x70
         * [6] = 1      : MIPI data lane 2 in LP11 when 'sleeping'
         * [5] = 1      : MIPI data lane 1 in LP11 when 'sleeping'
         * [4] = 1      : MIPI clock lane in LP11 when 'sleeping'
         */
        ret = ov5640_write_reg(sensor,
                               OV5640_REG_PAD_OUTPUT00, 0x70);
        if (ret)
                goto power_off;
                /* Give lanes some time to coax into LP11 state. */
                usleep_range(500, 1000);
        }
}

                         -             -------------|
                                       |              |
Reset -------------------------|              | --------------

                ----------
                |          |
power dn |           |-----------------------------------



static void ov5640_power(struct ov5640_dev *sensor, bool enable)
{
        gpiod_set_value_cansleep(sensor->pwdn_gpio, enable ? 0 : 1);
}

static void ov5640_reset(struct ov5640_dev *sensor)
{
        if (!sensor->reset_gpio)
                return;

        gpiod_set_value_cansleep(sensor->reset_gpio, 0);

        /* camera power cycle */
        ov5640_power(sensor, false);
        usleep_range(5000, 10000);
        ov5640_power(sensor, true);
        usleep_range(5000, 10000);

        gpiod_set_value_cansleep(sensor->reset_gpio, 1);
        usleep_range(1000, 2000);

        gpiod_set_value_cansleep(sensor->reset_gpio, 0);
        usleep_range(5000, 10000);
}

static int ov5640_set_power_on(struct ov5640_dev *sensor)
{
        struct i2c_client *client = sensor->i2c_client;
        int ret;

        ret = clk_prepare_enable(sensor->xclk);
        if (ret) {
                dev_err(&client->dev, "%s: failed to enable clock\n",
                        __func__);
                return ret;
        }

        ret = regulator_bulk_enable(OV5640_NUM_SUPPLIES,
                                    sensor->supplies);
        if (ret) {
                dev_err(&client->dev, "%s: failed to enable regulators\n",
                        __func__);
                goto xclk_off;
        }

        ov5640_reset(sensor);
        ov5640_power(sensor, true);

        ret = ov5640_init_slave_id(sensor);
        if (ret)
                goto power_off;

        return 0;

power_off:
        ov5640_power(sensor, false);
        regulator_bulk_disable(OV5640_NUM_SUPPLIES, sensor->supplies);
xclk_off:
        clk_disable_unprepare(sensor->xclk);
        return ret;
}
```

Step 6: ov5640_s_stream

```
static int ov5640_s_stream(struct v4l2_subdev *sd, int enable)
{
        struct ov5640_dev *sensor = to_ov5640_dev(sd);
        int ret = 0;

        mutex_lock(&sensor->lock);

        if (sensor->streaming == !enable) {
                if (enable && sensor->pending_mode_change) {
                        ret = ov5640_set_mode(sensor);
                        if (ret)
                                goto out;
                }
                if (enable && sensor->pending_fmt_change) {
                        ret = ov5640_set_framefmt(sensor, &sensor->fmt);
                        if (ret)
                                goto out;
                        sensor->pending_fmt_change = false;
                }

                if (sensor->ep.bus_type == V4L2_MBUS_CSI2)
                        ret = ov5640_set_stream_mipi(sensor, enable);
                else
                        ret = ov5640_set_stream_dvp(sensor, enable);

                if (!ret)
                        sensor->streaming = enable;
        }
out:
        mutex_unlock(&sensor->lock);
        return ret;
}

static int ov5640_set_mode(struct ov5640_dev *sensor)
{
        const struct ov5640_mode_info *mode = sensor->current_mode;
        const struct ov5640_mode_info *orig_mode = sensor->last_mode;
        enum ov5640_downsize_mode dn_mode, orig_dn_mode;
        bool auto_gain = sensor->ctrls.auto_gain->val == 1;
        bool auto_exp =  sensor->ctrls.auto_exp->val == V4L2_EXPOSURE_AUTO;
        int ret;

        dn_mode = mode->dn_mode;
        orig_dn_mode = orig_mode->dn_mode;

        /* auto gain and exposure must be turned off when changing modes */
        if (auto_gain) {
                ret = ov5640_set_autogain(sensor, false);
                if (ret)
                        return ret;
        }

        if (auto_exp) {
                ret = ov5640_set_autoexposure(sensor, false);
                if (ret)
                        goto restore_auto_gain;
        }

        if ((dn_mode == SUBSAMPLING && orig_dn_mode == SCALING) ||
            (dn_mode == SCALING && orig_dn_mode == SUBSAMPLING)) {
                /*
                 * change between subsampling and scaling
                * go through exposure calculation
               ret = ov5640_set_mode_exposure_calc(sensor, mode);
        } else {
                /*
                 * change inside subsampling or scaling
                 * download firmware directly
                 */
                ret = ov5640_set_mode_direct(sensor, mode);
        }
        if (ret < 0)
                goto restore_auto_exp_gain;

        /* restore auto gain and exposure */
        if (auto_gain)
                ov5640_set_autogain(sensor, true);
        if (auto_exp)
                ov5640_set_autoexposure(sensor, true);

        ret = ov5640_set_binning(sensor, dn_mode != SCALING);
        if (ret < 0)
                return ret;
        ret = ov5640_set_ae_target(sensor, sensor->ae_target);
        if (ret < 0)
                return ret;
        ret = ov5640_get_light_freq(sensor);
        if (ret < 0)
                return ret;
        ret = ov5640_set_bandingfilter(sensor);
        if (ret < 0)
                return ret;
        ret = ov5640_set_virtual_channel(sensor);
        if (ret < 0)
                return ret;

        sensor->pending_mode_change = false;
        sensor->last_mode = mode;

        return 0;

restore_auto_exp_gain:
        if (auto_exp)
                ov5640_set_autoexposure(sensor, true);
restore_auto_gain:
        if (auto_gain)
                ov5640_set_autogain(sensor, true);
        return ret;
}




static int ov5640_set_framefmt(struct ov5640_dev *sensor,
                               struct v4l2_mbus_framefmt *format)
{
        int ret = 0;
        bool is_rgb = false;
        bool is_jpeg = false;
        u8 val;

        switch (format->code) {
        case MEDIA_BUS_FMT_UYVY8_2X8:
        case MEDIA_BUS_FMT_UYVY8_1X16:
                /* YUV422, UYVY */
                val = 0x3f;
                break;
        case MEDIA_BUS_FMT_YUYV8_2X8:
        case MEDIA_BUS_FMT_YUYV8_1X16:
                /* YUV422, YUYV */
                val = 0x30;
                break;
        case MEDIA_BUS_FMT_RGB565_2X8_LE:
                /* RGB565 {g[2:0],b[4:0]},{r[4:0],g[5:3]} */
                val = 0x6F;
                is_rgb = true;
                break;
        case MEDIA_BUS_FMT_RGB565_2X8_BE:
                /* RGB565 {r[4:0],g[5:3]},{g[2:0],b[4:0]} */
                val = 0x61;
                is_rgb = true;
                break;
        case MEDIA_BUS_FMT_JPEG_1X8:
                  /* YUV422, YUYV */
                val = 0x30;
                is_jpeg = true;
                break;
        default:
                return -EINVAL;
        }

        /* FORMAT CONTROL00: YUV and RGB formatting */
        ret = ov5640_write_reg(sensor, OV5640_REG_FORMAT_CONTROL00, val);
        if (ret)
                return ret;
```

```
	/* FORMAT MUX CONTROL: ISP YUV or RGB */
	ret = ov5640_write_reg(sensor, OV5640_REG_ISP_FORMAT_MUX_CTRL,
			       is_rgb ? 0x01 : 0x00);
	if (ret)
		return ret;

	/*
	 * TIMING TC REG21:
	 * - [5]:	JPEG enable
	 */
	ret = ov5640_mod_reg(sensor, OV5640_REG_TIMING_TC_REG21,
			     BIT(5), is_jpeg ? BIT(5) : 0);
	if (ret)
		return ret;

	/*
	 * SYSTEM RESET02:
	 * - [4]:	Reset JFIFO
	 * - [3]:	Reset SFIFO
	 * - [2]:	Reset JPEG
	 */
	 ret = ov5640_mod_reg(sensor, OV5640_REG_SYS_RESET02,
			     BIT(4) | BIT(3) | BIT(2),
			     is_jpeg ? 0 : (BIT(4) | BIT(3) | BIT(2)));
	if (ret)
		return ret;

	/*
	 * CLOCK ENABLE02:
	 * - [5]:	Enable JPEG 2x clock
	 * - [3]:	Enable JPEG clock
	 */
	return ov5640_mod_reg(sensor, OV5640_REG_SYS_CLOCK_ENABLE02,
			      BIT(5) | BIT(3),
			      is_jpeg ? (BIT(5) | BIT(3)) : 0);
}

static int ov5640_set_stream_mipi(struct ov5640_dev *sensor, bool on)
{
	int ret;

	/*
	 * Enable/disable the MIPI interface
	 *
	 * 0x300e = on ? 0x45 : 0x40
	 *
	 * FIXME: the sensor manual (version 2.03) reports
```

```
         * [7:5] = 000  : 1 data lane mode
         * [7:5] = 001  : 2 data lanes mode
         * But this settings do not work, while the following ones
         * have been validated for 2 data lanes mode.
         *
         * [7:5] = 010  : 2 data lanes mode
         * [4] = 0      : Power up MIPI HS Tx
         * [3] = 0      : Power up MIPI LS Rx
         * [2] = 1/0    : MIPI interface enable/disable
         * [1:0] = 01/00: FIXME: 'debug'
         */
        ret = ov5640_write_reg(sensor, OV5640_REG_IO_MIPI_CTRL00,
                               on ? 0x45 : 0x40);
        if (ret)
                return ret;

        return ov5640_write_reg(sensor, OV5640_REG_FRAME_CTRL01,
                                on ? 0x00 : 0x0f);
}
```

Step 7: OV5640 SENSOR : [RESOLUTION, FPS, SCALE(OR)SUBSAMPLING]

Step 1: Available / Supporting Resolution

```
enum ov5640_mode_id {
         OV5640_MODE_QCIF_176_144 = 0,
         OV5640_MODE_QVGA_320_240,
         OV5640_MODE_VGA_640_480,
         OV5640_MODE_NTSC_720_480,
         OV5640_MODE_PAL_720_576,
         OV5640_MODE_XGA_1024_768,
         OV5640_MODE_720P_1280_720,
         OV5640_MODE_1080P_1920_1080,
         OV5640_MODE_QSXGA_2592_1944,
         OV5640_NUM_MODES,
};
```

Step 2: Available / Supporting FPS

```
enum ov5640_frame_rate {
         OV5640_15_FPS = 0,
         OV5640_30_FPS,
         OV5640_60_FPS,
         OV5640_NUM_FRAMERATES,
};

static const int ov5640_framerates [] = {
         [OV5640_15_FPS] = 15,
         [OV5640_30_FPS] = 30,
         [OV5640_60_FPS] = 60,
};
```

Step 3: Available / Supporting MEDIA BUS FORMAT

```
struct ov5640_pixfmt {
        u32 code;
        u32 colorspace;
};

static const struct ov5640_pixfmt ov5640_formats[] = {
        { MEDIA_BUS_FMT_JPEG_1X8, V4L2_COLORSPACE_JPEG, },
        { MEDIA_BUS_FMT_UYVY8_2X8, V4L2_COLORSPACE_SRGB, },
        { MEDIA_BUS_FMT_YUYV8_2X8, V4L2_COLORSPACE_SRGB, },
        { MEDIA_BUS_FMT_UYVY8_1X16, V4L2_COLORSPACE_SRGB, },
        { MEDIA_BUS_FMT_YUYV8_1X16, V4L2_COLORSPACE_SRGB, },
        { MEDIA_BUS_FMT_RGB565_2X8_LE, V4L2_COLORSPACE_SRGB, },
        { MEDIA_BUS_FMT_RGB565_2X8_BE, V4L2_COLORSPACE_SRGB, },
};


static int ov5640_enum_mbus_code(struct v4l2_subdev *sd,
                                struct v4l2_subdev_pad_config *cfg,
                                struct v4l2_subdev_mbus_code_enum *code)
{
        if (code->pad != 0)
                return -EINVAL;
        if (code->index >= ARRAY_SIZE(ov5640_formats))
                return -EINVAL;

        code->code = ov5640_formats[code->index].code;
        return 0;
}
```

Step 4: Available / Supporting SCALING / SUBSAMPLING

```
/*
 * Image size under 1280 * 960 are SUBSAMPLING
 * Image size upper 1280 * 960 are SCALING
 */
```

```
enum ov5640_downsize_mode {
        SUBSAMPLING,
        SCALING,
};
```

Step 5: AVAILABLE MODES - STRUCTURE

```
struct ov5640_mode_info {
        enum ov5640_mode_id id;
        enum ov5640_downsize_mode dn_mode;
        bool scaler; /* Mode uses ISP scaler (reg 0x5001,BIT(5)=='1') */
        u32 hact;
        u32 htot;
        u32 vact;
        u32 vtot;
        const struct reg_value *reg_data;
        u32 reg_data_size;
};
```

Step 6: ov5640_dev - STRUCTURE

```
struct ov5640_dev {
        struct i2c_client *i2c_client;
        struct v4l2_subdev sd;
        struct media_pad pad;
        struct v4l2_fwnode_endpoint ep; /* the parsed DT endpoint info */
        struct clk *xclk; /* system clock to OV5640 */
        u32 xclk_freq;

        struct regulator_bulk_data supplies[OV5640_NUM_SUPPLIES];
        struct gpio_desc *reset_gpio;
        struct gpio_desc *pwdn_gpio;
        bool   upside_down;

        /* lock to protect all members below */
        struct mutex lock;

        int power_count;

        struct v4l2_mbus_framefmt fmt;
        bool pending_fmt_change;

        const struct ov5640_mode_info *current_mode;
        const struct ov5640_mode_info *last_mode;
        enum ov5640_frame_rate current_fr;
        struct v4l2_fract frame_interval;

        struct ov5640_ctrls ctrls;

        u32 prev_sysclk, prev_hts;
        u32 ae_low, ae_high, ae_target;

        bool pending_mode_change;
        bool streaming;
};

static int ov5640_get_fmt(struct v4l2_subdev *sd,
                          struct v4l2_subdev_pad_config *cfg,
                          struct v4l2_subdev_format *format)
{
        struct ov5640_dev *sensor = to_ov5640_dev(sd);
        struct v4l2_mbus_framefmt *fmt;

        if (format->pad != 0)
                return -EINVAL;

        mutex_lock(&sensor->lock);

        if (format->which == V4L2_SUBDEV_FORMAT_TRY)
                fmt = v4l2_subdev_get_try_format(&sensor->sd, cfg,
                                                 format->pad);
        else
                fmt = &sensor->fmt;

        format->format = *fmt;

        mutex_unlock(&sensor->lock);

        return 0;
}
```

```
struct v4l2_mbus_framefmt {
        __u32           width;
        __u32           height;
        __u32           code;
        __u32           field;
        __u32           colorspace;
        __u16           ycbcr_enc;
        __u16           quantization;
        __u16           xfer_func;
        __u16           reserved[11];
};
```

Step 8: v4l-ctl: sub-dev calls [v4l2-ctrl-subdev.c]

```
static const struct v4l2_subdev_pad_ops ov5640_pad_ops = {
        .enum_mbus_code = ov5640_enum_mbus_code,
        .get_fmt = ov5640_get_fmt,
        .set_fmt = ov5640_set_fmt,
        .enum_frame_size = ov5640_enum_frame_size,
        .enum_frame_interval = ov5640_enum_frame_interval,
};
```

https://linuxtv.org/downloads/v4l-utils/

utils/v4l2-ctrl/v4l2-ctrl-subdev.c

enum_mbus_code	callback for **VIDIOC_SUBDEV_ENUM_MBUS_CODE**() ioctl handler code.
enum_frame_size	callback for **VIDIOC_SUBDEV_ENUM_FRAME_SIZE(**) ioctl handler code.
enum_frame_interval	callback for **VIDIOC_SUBDEV_ENUM_FRAME_INTERVAL**() ioctl handler code.
get_fmt	callback for **VIDIOC_SUBDEV_G_FMT()** ioctl handler code.
set_fmt	callback for **VIDIOC_SUBDEV_S_FMT()** ioctl handler code.

```
void subdev_usage(void)
{
        printf("\nSub-Device options:\n"
               "  --list-subdev-mbus-codes <pad>\n"
               "                     display supported mediabus codes for this pad (0 is default)\n"
               "                     [VIDIOC_SUBDEV_ENUM_MBUS_CODE]\n"
               "  --list-subdev-framesizes pad=<pad>,code=<code>\n"
               "                     list supported framesizes for this pad and code\n"
               "                     [VIDIOC_SUBDEV_ENUM_FRAME_SIZE]\n"
               "                     <code> is the value of the mediabus code\n"
               "  --list-subdev-frameintervals
```

```
pad=<pad>,width=<w>,height=<h>,code=<code>\n"
        "                 list supported frame intervals for this pad and code and\n"
        "                 the given width and height
    [VIDIOC_SUBDEV_ENUM_FRAME_INTERVAL]\n"
        "                 <code> is the value of the mediabus code\n"
        "  --get-subdev-fmt [<pad>]\n"
        "                  query the frame format for the given pad
    [VIDIOC_SUBDEV_G_FMT]\n"
```

struct v4l2_subdev_video_ops

g_frame_interval	callback for VIDIOC_G_FRAMEINTERVAL ioctl handler code.
s_frame_interval	callback for VIDIOC_S_FRAMEINTERVAL ioctl handler code.

Step 9: V4L2 – CONTROLS

#include <media/v4l2-ctrls.h>

Step 1: define the controls

```
struct ov5640_ctrls {

    struct v4l2_ctrl_handler handler;
    struct {
        struct v4l2_ctrl *auto_exp;
        struct v4l2_ctrl *exposure;
    };
    struct {
        struct v4l2_ctrl *auto_wb;
        struct v4l2_ctrl *blue_balance;
        struct v4l2_ctrl *red_balance;
    };
    struct {
        struct v4l2_ctrl *auto_gain;
```

```
                struct v4l2_ctrl *gain;
        };
        struct v4l2_ctrl *brightness;
        struct v4l2_ctrl *light_freq;
        struct v4l2_ctrl *saturation;
        struct v4l2_ctrl *contrast;
        struct v4l2_ctrl *hue;
        struct v4l2_ctrl *test_pattern;
        struct v4l2_ctrl *hflip;
        struct v4l2_ctrl *vflip;
};
```

Step 2: fill the controls

```
static const struct v4l2_ctrl_ops ov5640_ctrl_ops = {
        .g_volatile_ctrl = ov5640_g_volatile_ctrl,
        .s_ctrl = ov5640_s_ctrl,
};

static int ov5640_init_controls(struct ov5640_dev *sensor)
{
        const struct v4l2_ctrl_ops *ops = &ov5640_ctrl_ops;
        struct ov5640_ctrls *ctrls = &sensor->ctrls;
        struct v4l2_ctrl_handler *hdl = &ctrls->handler;
        int ret;

        v4l2_ctrl_handler_init(hdl, 32);

        /* we can use our own mutex for the ctrl lock */
        hdl->lock = &sensor->lock;

        /* Auto/manual white balance */
        ctrls->auto_wb = v4l2_ctrl_new_std(hdl, ops,
                                          V4L2_CID_AUTO_WHITE_BALANCE,
                                          0, 1, 1, 1);
        ctrls->blue_balance = v4l2_ctrl_new_std(hdl, ops,
V4L2_CID_BLUE_BALANCE,
                                                0, 4095, 1, 0);
        ctrls->red_balance = v4l2_ctrl_new_std(hdl, ops,
V4L2_CID_RED_BALANCE,
                                              0, 4095, 1, 0);
        /* Auto/manual exposure */
        ctrls->auto_exp = v4l2_ctrl_new_std_menu(hdl, ops,
                                                 V4L2_CID_EXPOSURE_AUTO,
                                                 V4L2_EXPOSURE_MANUAL, 0,
                                                 V4L2_EXPOSURE_AUTO);
        ctrls->exposure = v4l2_ctrl_new_std(hdl, ops, V4L2_CID_EXPOSURE,
                                           0, 65535, 1, 0);
        /* Auto/manual gain */
        ctrls->auto_gain = v4l2_ctrl_new_std(hdl, ops, V4L2_CID_AUTOGAIN,
                                            0, 1, 1, 1);
        ctrls->gain = v4l2_ctrl_new_std(hdl, ops, V4L2_CID_GAIN,
                                        0, 1023, 1, 0);

        ctrls->saturation = v4l2_ctrl_new_std(hdl, ops,
V4L2_CID_SATURATION,
                                             0, 255, 1, 64);
        ctrls->hue = v4l2_ctrl_new_std(hdl, ops, V4L2_CID_HUE,
                                      0, 359, 1, 0);
        ctrls->contrast = v4l2_ctrl_new_std(hdl, ops, V4L2_CID_CONTRAST,
                                           0, 255, 1, 0);
        ctrls->test_pattern =
                v4l2_ctrl_new_std_menu_items(hdl, ops,
                                        V4L2_CID_TEST_PATTERN,
                                            ARRAY_SIZE(test_pattern_menu) - 1,
                                            0, 0, test_pattern_menu);
        ctrls->hflip = v4l2_ctrl_new_std(hdl, ops, V4L2_CID_HFLIP,
                                         0, 1, 1, 0);
        ctrls->vflip = v4l2_ctrl_new_std(hdl, ops, V4L2_CID_VFLIP,
                                         0, 1, 1, 0);

        ctrls->light_freq =
                v4l2_ctrl_new_std_menu(hdl, ops,
                                      V4L2_CID_POWER_LINE_FREQUENCY,
                                      V4L2_CID_POWER_LINE_FREQUENCY_AUTO, 0,
                                      V4L2_CID_POWER_LINE_FREQUENCY_50HZ);

        if (hdl->error) {
                ret = hdl->error;
                goto free_ctrls;
        }

        ctrls->gain->flags |= V4L2_CTRL_FLAG_VOLATILE;
        ctrls->exposure->flags |= V4L2_CTRL_FLAG_VOLATILE;

        v4l2_ctrl_auto_cluster(3, &ctrls->auto_wb, 0, false);
        v4l2_ctrl_auto_cluster(2, &ctrls->auto_gain, 0, true);
        v4l2_ctrl_auto_cluster(2, &ctrls->auto_exp, 1, true);
```

```
	sensor->sd.ctrl_handler = hdl;
	return 0;

free_ctrls:
	v4l2_ctrl_handler_free(hdl);
	return ret;
}
```

```
ret = v4l2_ctrl_handler_setup(&sensor->ctrls.handler);
```

```
static int ov5640_g_volatile_ctrl(struct v4l2_ctrl *ctrl)
{
	struct v4l2_subdev *sd = ctrl_to_sd(ctrl);
	struct ov5640_dev *sensor = to_ov5640_dev(sd);
	int val;

	/* v4l2_ctrl_lock() locks our own mutex */

	/*
	 * If the sensor is not powered up by the host driver, do
	 * not try to access it to update the volatile controls.
	 */
	if (sensor->power_count == 0)
		return 0;

	switch (ctrl->id) {
	case V4L2_CID_AUTOGAIN:
		val = ov5640_get_gain(sensor);
		if (val < 0)
			return val;
		sensor->ctrls.gain->val = val;
		break;
	case V4L2_CID_EXPOSURE_AUTO:
		val = ov5640_get_exposure(sensor);
		if (val < 0)
			return val;
		sensor->ctrls.exposure->val = val;
		break;
	}
```

```
        return 0;
}


static int ov5640_s_ctrl(struct v4l2_ctrl *ctrl)
{
        struct v4l2_subdev *sd = ctrl_to_sd(ctrl);
        struct ov5640_dev *sensor = to_ov5640_dev(sd);
        int ret;

        /* v4l2_ctrl_lock() locks our own mutex */

        /*
         * If the device is not powered up by the host driver do
         * not apply any controls to H/W at this time. Instead
         * the controls will be restored right after power-up.
         */
        if (sensor->power_count == 0)
                return 0;

        switch (ctrl->id) {
        case V4L2_CID_AUTOGAIN:
                ret = ov5640_set_ctrl_gain(sensor, ctrl->val);
                break;
        case V4L2_CID_EXPOSURE_AUTO:
                ret = ov5640_set_ctrl_exposure(sensor, ctrl->val);
                break;
        case V4L2_CID_AUTO_WHITE_BALANCE:
                ret = ov5640_set_ctrl_white_balance(sensor, ctrl->val);
                break;
        case V4L2_CID_HUE:
                ret = ov5640_set_ctrl_hue(sensor, ctrl->val);
                break;
        case V4L2_CID_CONTRAST:
                ret = ov5640_set_ctrl_contrast(sensor, ctrl->val);
                break;
        case V4L2_CID_SATURATION:
                ret = ov5640_set_ctrl_saturation(sensor, ctrl->val);
                break;
        case V4L2_CID_TEST_PATTERN:
                ret = ov5640_set_ctrl_test_pattern(sensor, ctrl->val);
                break;
        case V4L2_CID_POWER_LINE_FREQUENCY:
                ret = ov5640_set_ctrl_light_freq(sensor, ctrl->val);
                break;
        case V4L2_CID_HFLIP:
                ret = ov5640_set_ctrl_hflip(sensor, ctrl->val);
                break;
        case V4L2_CID_VFLIP:
                ret = ov5640_set_ctrl_vflip(sensor, ctrl->val);
                break;
        default:
                ret = -EINVAL;
                break;
        }

        return ret;
}
```

Step 10: V4L2 – CONTROLS - TABLE

S.N0	**IMP STRUCTURES / FUNCTIONS**
1	**struct v4l2_ctrl_handler *ctrl_handler**; ==in==> struct v4l2_subdev

2	**struct v4l2_ctrl_ops {**
	int (*g_volatile_ctrl)(struct v4l2_ctrl *ctrl);
	int (*try_ctrl)(struct v4l2_ctrl *ctrl);
	int (*s_ctrl)(struct v4l2_ctrl *ctrl);
	};

3	**struct v4l2_ctrl** {
	const struct v4l2_ctrl_ops *ops;
	u32 id;
	const char *name;
	enum v4l2_ctrl_type type;
	s64 minimum, maximum, default_value;

4	**enum v4l2_ctrl_type** {
	V4L2_CTRL_TYPE_INTEGER = 1,
	V4L2_CTRL_TYPE_BOOLEAN = 2,
	V4L2_CTRL_TYPE_MENU = 3,
	V4L2_CTRL_TYPE_BUTTON = 4,
	V4L2_CTRL_TYPE_INTEGER64 = 5,
	V4L2_CTRL_TYPE_CTRL_CLASS = 6,

	V4L2_CTRL_TYPE_STRING = 7, V4L2_CTRL_TYPE_BITMASK = 8, V4L2_CTRL_TYPE_INTEGER_MENU = 9, /* Compound types are >= 0x0100 */ V4L2_CTRL_COMPOUND_TYPES = 0x0100, V4L2_CTRL_TYPE_U8 = 0x0100, V4L2_CTRL_TYPE_U16 = 0x0101, V4L2_CTRL_TYPE_U32 = 0x0102, };
5	struct v4l2_ctrl ***v4l2_ctrl_new_std**(struct v4l2_ctrl_handler *hdl, const struct v4l2_ctrl_ops *ops, u32 id, s64 min, s64 max, u64 step, s64 def)
6	struct v4l2_ctrl ***v4l2_ctrl_new_std_menu_items**(struct v4l2_ctrl_handler *hdl, const struct v4l2_ctrl_ops *ops, u32 id, u8 _max, u64 mask, u8 _def, const char * const *qmenu)
7	int v**4l2_ctrl_handler_setup**(struct v4l2_ctrl_handler *hdl)

Step 11: APP – V4L2 SUB DEV

Step 1:

ret = v4l2_async_register_subdev(&sensor->sd);

Step 2:

drivers/media/v4l2-core/v4l2-subdev.c

```
const struct v4l2_file_operations v4l2_subdev_fops = {

	.owner = THIS_MODULE,
	.open = subdev_open,
	.unlocked_ioctl = subdev_ioctl,
#ifdef CONFIG_COMPAT
        .compat_ioctl32 = subdev_compat_ioctl32,
#endif
        .release = subdev_close,
        .poll = subdev_poll,
};


static long subdev_ioctl(struct file *file, unsigned int cmd,
        unsigned long arg)
{
        return video_usercopy(file, cmd, arg, subdev_do_ioctl_lock);
}


static long subdev_do_ioctl_lock(struct file *file, unsigned int cmd,
void *arg)
{
        struct video_device *vdev = video_devdata(file);
        struct mutex *lock = vdev->lock;
        long ret = -ENODEV;

        if (lock && mutex_lock_interruptible(lock))
                return -ERESTARTSYS;
        if (video_is_registered(vdev))
                ret = subdev_do_ioctl(file, cmd, arg);
        if (lock)
                mutex_unlock(lock);
        return ret;
}



static long subdev_do_ioctl(struct file *file, unsigned int cmd, void
*arg)
{
        struct video_device *vdev = video_devdata(file);
        struct v4l2_subdev *sd = vdev_to_v4l2_subdev(vdev);
        struct v4l2_fh *vfh = file->private_data;
#if defined(CONFIG_VIDEO_V4L2_SUBDEV_API)
        struct v4l2_subdev_fh *subdev_fh = to_v4l2_subdev_fh(vfh);
        int rval;
#endif
```

```
	switch (cmd) {
	case VIDIOC_QUERYCTRL:
		/*
		 * TODO: this really should be folded into v4l2_queryctrl (this
		 * currently returns -EINVAL for NULL control handlers).
		 * However, v4l2_queryctrl() is still called directly by
		 * drivers as well and until that has been addressed I believe
		 * it is safer to do the check here. The same is true for the
		 * other control ioctls below.
		 */
		if (!vfh->ctrl_handler)
			return -ENOTTY;
		return v4l2_queryctrl(vfh->ctrl_handler, arg);

	case VIDIOC_QUERY_EXT_CTRL:
		if (!vfh->ctrl_handler)
			return -ENOTTY;
		   return v4l2_query_ext_ctrl(vfh->ctrl_handler, arg);

	case VIDIOC_QUERYMENU:
		if (!vfh->ctrl_handler)
			return -ENOTTY;
		return v4l2_querymenu(vfh->ctrl_handler, arg);

	case VIDIOC_G_CTRL:
		if (!vfh->ctrl_handler)
			return -ENOTTY;
		return v4l2_g_ctrl(vfh->ctrl_handler, arg);

	case VIDIOC_S_CTRL:
		if (!vfh->ctrl_handler)
			return -ENOTTY;
		return v4l2_s_ctrl(vfh, vfh->ctrl_handler, arg);

	case VIDIOC_LOG_STATUS: {
		int ret;

		pr_info("%s: =================  START STATUS
=================\n",
			sd->name);
		ret = v4l2_subdev_call(sd, core, log_status);
		pr_info("%s: ==================  END STATUS
==================\n",
			sd->name);
		return ret;
	}


#if defined(CONFIG_VIDEO_V4L2_SUBDEV_API)
	case VIDIOC_SUBDEV_G_FMT: {
		struct v4l2_subdev_format *format = arg;

		rval = check_format(sd, format);
		if (rval)
			return rval;

		memset(format->reserved, 0, sizeof(format->reserved));
		memset(format->format.reserved, 0, sizeof(format-
>format.reserved));
		return v4l2_subdev_call(sd, pad, get_fmt, subdev_fh->pad,
format);
	}

	case VIDIOC_SUBDEV_S_FMT: {
		struct v4l2_subdev_format *format = arg;

		rval = check_format(sd, format);
		if (rval)
			return rval;

		memset(format->reserved, 0, sizeof(format->reserved));
		memset(format->format.reserved, 0, sizeof(format-
>format.reserved));
		return v4l2_subdev_call(sd, pad, set_fmt, subdev_fh->pad,
format);
	}

case VIDIOC_SUBDEV_ENUM_MBUS_CODE: {
		struct v4l2_subdev_mbus_code_enum *code = arg;

		if (code->which != V4L2_SUBDEV_FORMAT_TRY &&
		    code->which != V4L2_SUBDEV_FORMAT_ACTIVE)
			return -EINVAL;

		if (code->pad >= sd->entity.num_pads)
			return -EINVAL;

		memset(code->reserved, 0, sizeof(code->reserved));
```

```
                return v4l2_subdev_call(sd, pad, enum_mbus_code, subdev_fh-
>pad,
                                        code);
        }

        case VIDIOC_SUBDEV_ENUM_FRAME_SIZE: {
                struct v4l2_subdev_frame_size_enum *fse = arg;

                if (fse->which != V4L2_SUBDEV_FORMAT_TRY &&
                    fse->which != V4L2_SUBDEV_FORMAT_ACTIVE)
                        return -EINVAL;

                if (fse->pad >= sd->entity.num_pads)
                        return -EINVAL;

                memset(fse->reserved, 0, sizeof(fse->reserved));
                return v4l2_subdev_call(sd, pad, enum_frame_size, subdev_fh-
>pad,
                                        fse);
        }

        case VIDIOC_SUBDEV_ENUM_FRAME_INTERVAL: {
                struct v4l2_subdev_frame_interval_enum *fie = arg;

                if (fie->which != V4L2_SUBDEV_FORMAT_TRY &&
                    fie->which != V4L2_SUBDEV_FORMAT_ACTIVE)
                        return -EINVAL;

                if (fie->pad >= sd->entity.num_pads)
                        return -EINVAL;

                memset(fie->reserved, 0, sizeof(fie->reserved));
                return v4l2_subdev_call(sd, pad, enum_frame_interval,
subdev_fh->pad,   fie);
        }
```

Step 3:

```
int v4l2_device_register_subdev_nodes(struct v4l2_device *v4l2_dev)
```

```
        struct video_device *vdev;
         struct v4l2_subdev *sd;
        int err;

        /* Register a device node for every subdev marked with the
         * V4L2_SUBDEV_FL_HAS_DEVNODE flag.
         */
        list_for_each_entry(sd, &v4l2_dev->subdevs, list) {
                if (!(sd->flags & V4L2_SUBDEV_FL_HAS_DEVNODE))
                        continue;

                if (sd->devnode)
                         continue;



                 vdev = kzalloc(sizeof(*vdev), GFP_KERNEL);
                if (!vdev) {
                        err = -ENOMEM;
                        goto clean_up;
                }

                video_set_drvdata(vdev, sd);
                strlcpy(vdev->name, sd->name, sizeof(vdev->name));
                vdev->v4l2_dev = v4l2_dev;
                vdev->fops = &v4l2_subdev_fops;
                vdev->release = v4l2_device_release_subdev_node;
                vdev->ctrl_handler = sd->ctrl_handler;
                err = __video_register_device(vdev, VFL_TYPE_SUBDEV, -1,
1,sd->owner);
                if (err < 0) {
                        kfree(vdev);
                        goto clean_up;
                }
                sd->devnode = vdev;
```

Step 4:

drivers/media/v4l2-core/v4l2-dev.c

```
int __video_register_device(struct video_device *vdev,
                            enum vfl_devnode_type type,
                                int nr, int warn_if_nr_in_use,
                                struct module *owner)
{
        int i = 0;
        int ret;
        int minor_offset = 0;
        int minor_cnt = VIDEO_NUM_DEVICES;
        const char *name_base;

/* Part 1: check device type */
        switch (type) {
        case VFL_TYPE_GRABBER:
                name_base = "video";
                break;
        case VFL_TYPE_VBI:
                name_base = "vbi";
                break;
        case VFL_TYPE_RADIO:
                name_base = "radio";
                break;
        case VFL_TYPE_SUBDEV:
                name_base = "v4l-subdev";
                break;
        case VFL_TYPE_SDR:
                /* Use device name 'swradio' because 'sdr' was already taken. */
                name_base = "swradio";
                break;
        case VFL_TYPE_TOUCH:
                name_base = "v4l-touch";
                break;
        default:
                pr_err("%s called with unknown type: %d\n",
                        __func__, type);
                return -EINVAL;
        }

        vdev->vfl_type = type;
        vdev->cdev = NULL;

/* Part 2: find a free minor, device node number and device index.
*/
#ifdef CONFIG_VIDEO_FIXED_MINOR_RANGES
        /* Keep the ranges for the first four types for historical
         * reasons.
         * Newer devices (not yet in place) should use the range
         * of 128-191 and just pick the first free minor there
         * (new style). */
        switch (type) {
        case VFL_TYPE_GRABBER:
                minor_offset = 0;
                minor_cnt = 64;
                break;
        case VFL_TYPE_RADIO:
                minor_offset = 64;
                minor_cnt = 64;
                break;
        case VFL_TYPE_VBI:
                minor_offset = 224;
                minor_cnt = 32;
                break;
        default:
                minor_offset = 128;
                minor_cnt = 64;
                break;
        }

/* Part 3: Initialize the character device */
        vdev->cdev = cdev_alloc();
        if (vdev->cdev == NULL) {
                ret = -ENOMEM;
                goto cleanup;
        }
        vdev->cdev->ops = &v4l2_fops;
        vdev->cdev->owner = owner;
        ret = cdev_add(vdev->cdev, MKDEV(VIDEO_MAJOR, vdev->minor), 1);
        if (ret < 0) {
                pr_err("%s: cdev_add failed\n", __func__);
                kfree(vdev->cdev);
                vdev->cdev = NULL;
                goto cleanup;
        }

        /* Part 4: register the device with sysfs */
        vdev->dev.class = &video_class;
        vdev->dev.devt = MKDEV(VIDEO_MAJOR, vdev->minor);
        vdev->dev.parent = vdev->dev_parent;
        dev_set_name(&vdev->dev, "%s%d", name_base, vdev->num);
        ret = device_register(&vdev->dev);
        if (ret < 0) {
                pr_err("%s: device_register failed\n", __func__);
                goto cleanup;
        }
        /* Register the release callback that will be called when the last
           reference to the device goes away. */
        vdev->dev.release = v4l2_device_release;

        if (nr != -1 && nr != vdev->num && warn_if_nr_in_use)
                pr_warn("%s: requested %s%d, got %s\n", __func__,
                        name_base, nr, video_device_node_name(vdev));

        /* Increase v4l2_device refcount */
        v4l2_device_get(vdev->v4l2_dev);

        /* Part 5: Register the entity. */
        ret = video_register_media_controller(vdev);

        /* Part 6: Activate this minor. The char device can now be
used. */
        set_bit(V4L2_FL_REGISTERED, &vdev->flags);
```