Table of Contents

Image processing Unit - i.MX 6

- IPU **(Image processing Unit)** is present on most of i.MX products
- IPUv1 was 1st introduced on i.MX31 and upgraded on i.MX35
- IPUv3 is a family of IPs that are present on MX37, i.MX51, i.MX53, i.MX6 S/Q/D/DL

i.MX 6 Solo	1× IPUv3, 1× PXP
i.MX 6 DualLite	1× IPUv3, 1× PXP
i.MX 6 Dual	2× IPUv3
i.MX 6 Quad	2× IPUv3

enum ipuv3_type {		
IPUv3D,	/* i.MX37 */	0
IPUv3EX,	/* i.MX51 */	1
IPUv3M,	/* i.MX53 */	2
IPUv3H,	/* i.MX6Q/SDL */	3
};		

Introduction

The Freescale i.MX5/6 contains an **Image Processing Unit (IPU)**, which handles the flow of image frames to and from capture devices and display devices.

For image capture, the IPU contains the following internal subunits:

- Image DMA Controller (IDMAC)
- Camera Serial Interface (CSI)

- Image Converter (IC)
- Sensor Multi-FIFO Controller (SMFC)
- Image Rotator (IRT)
- Video De-Interlacing or Combining Block (VDIC)

The IDMAC is the DMA controller for transfer of image frames to and from memory. Various dedicated DMA channels exist for both video capture and display paths.

During transfer, the IDMAC is also capable of vertical image flip, 8x8 block transfer (see IRT description), pixel component re-ordering (for example UYVY to YUYV) within the same color space, and packed <–> planar conversion. The IDMAC can also perform a simple de-interlacing by interweaving even and odd lines during transfer (without motion compensation which requires the VDIC).

The CSI is the backend capture unit that interfaces directly with camera sensors over Parallel, BT.656/1120, and MIPI CSI-2 buses.

The IC handles color-space conversion, resizing (downscaling and upscaling), horizontal flip, and 90/270 degree rotation operations.

There are three independent “tasks” within the IC that can carry out conversions concurrently: pre-process encoding, pre-process viewfinder, and post-processing. Within each task, conversions are split into three sections: downsizing section, main section (upsizing, flip, colorspace conversion, and graphics plane combining), and rotation section.

The IPU time-shares the IC task operations. The time-slice granularity is one burst of eight pixels in the downsizing section, one image line in the main processing section, one image frame in the rotation section.

T**he SMFC is composed of four independent FIFOs** that each can

transfer captured frames from sensors directly to memory concurrently via four IDMAC channels.

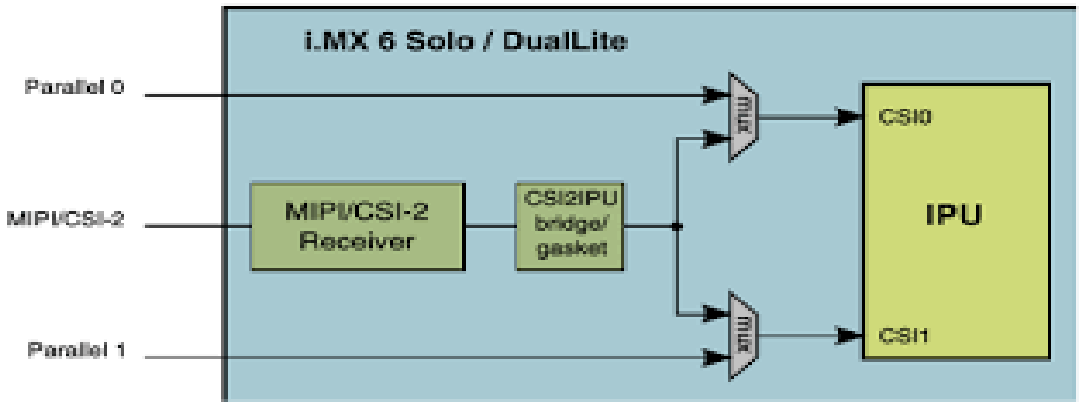
The IRT carries out 90 and 270 degree image rotation operations. The rotation operation is carried out on 8x8 pixel blocks at a time. This operation is supported by the IDMAC which handles the 8x8 block transfer along with block reordering, in coordination with vertical flip.

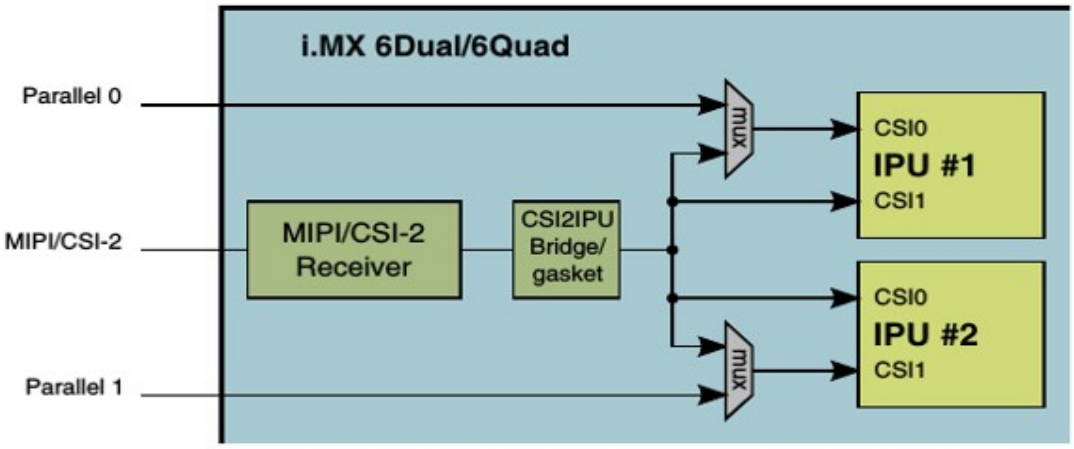
The VDIC handles the conversion of interlaced video to progressive, with support for different motion compensation modes (low, medium, and high motion). The deinterlaced output frames from the VDIC can be sent to the IC pre-process viewfinder task for further conversions. The VDIC also contains a Combiner that combines two image planes, with alpha blending and color keying.

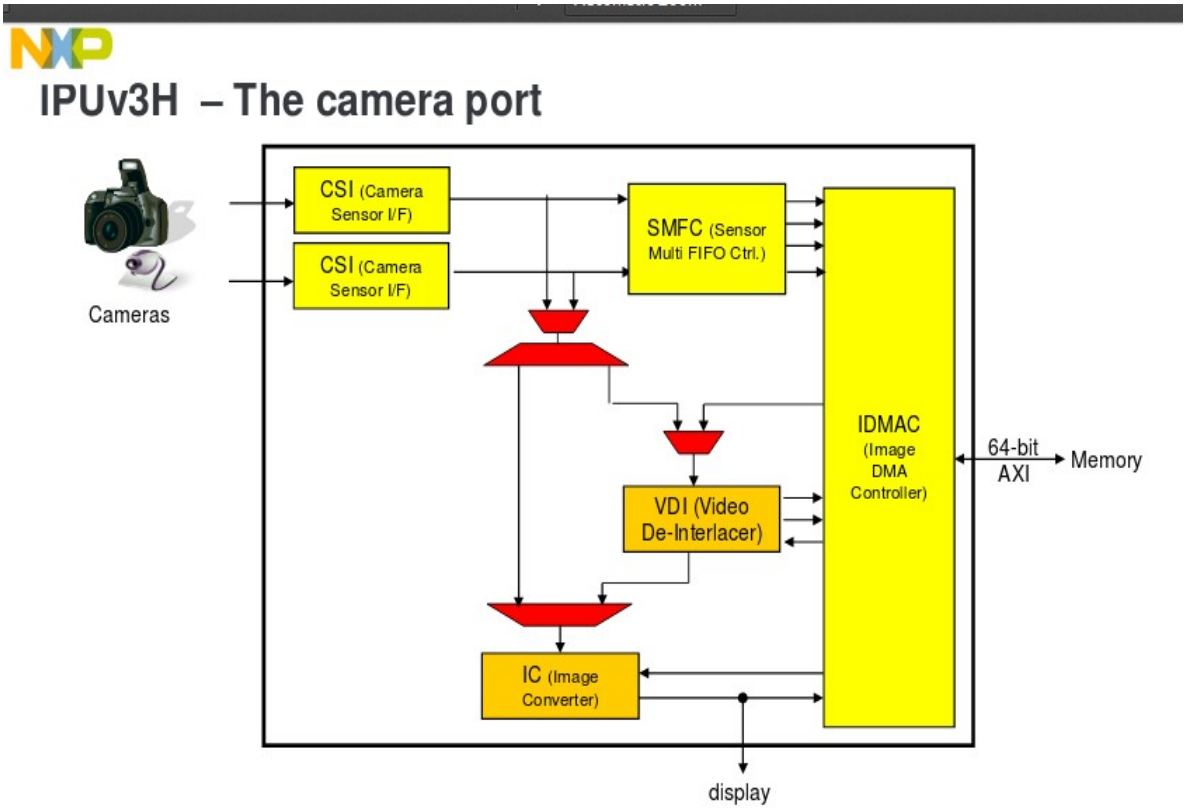
In addition to the IPU internal subunits, there are also two units outside the IPU that are also involved in video capture on i.MX:

- MIPI CSI-2 Receiver for camera sensors with the MIPI CSI-2 bus interface. This is a Synopsys DesignWare core.

- Two video multiplexers for selecting among multiple sensor inputs to send to a CSI.







CSI - Camera Sensor Interface : Controls a camera port; provides interface to an image sensor or a related device. IPUv3 includes 2 such blocks.

IC - Image Converter Performs resizing, color conversion/correction, combining with graphics, and horizontal inversion.

SMFC - Sensor Multi FIFO Controller Controls FIFO's for output from the CSI's to system memory.

IRT - Image Rotator Performs rotation (90 or 180 degrees) and inversion (vertical/horizontal)

VDIC - Video De-Interlacer and Combiner Performs de-interlacing - converting interlaced video to progressive - or combining.

IDMAC - Image DMA Controller Controls the memory port; transfers data to/from system memory.

Driver Source : Kernel Version : Download PATH

https://github.com/boundarydevices/linux-imx6/tree/boundary-imx_4.9.x_1.0.0_ga

linux-imx6-boundary-imx_4.9.x_1.0.0_ga/**drivers/media/platform/mxc/capture**

Step 1: mxc_v4l2_capture : init function

```
static __init int camera_init(void)
{
        u8 err = 0;
        pr_debug("%s\n", __func__);
        /* Register the device driver structure. */
        err = platform_driver_register(&mxc_v4l2_driver);
        if (err != 0) {
```

```
                pr_err("ERROR: v4l2 capture:camera_init: "
                        "platform_driver_register failed.\n");
                return err;
        }

        return err;
}


/*!
 * Exit and cleanup for the V4L2
 */
static void __exit camera_exit(void)
{
        pr_debug("%s\n", __func__);
        platform_driver_unregister(&mxc_v4l2_driver);
}

module_init(camera_init);
module_exit(camera_exit);

module_param(video_nr, int, 0444);
MODULE_AUTHOR("Freescale Semiconductor, Inc.");
MODULE_DESCRIPTION("V4L2 capture driver for Mxc based cameras");
MODULE_LICENSE("GPL");
MODULE_SUPPORTED_DEVICE("video");
```

Step 2: Platform driver

```
static const struct of_device_id mxc_v4l2_dt_ids[] = {
        {
                .compatible = "fsl,imx6q-v4l2-capture",
                .data = &imx_v4l2_devtype[IMX6_V4L2],
        }, {
                /* sentinel */
        }
};
MODULE_DEVICE_TABLE(of, mxc_v4l2_dt_ids);


static struct platform_driver mxc_v4l2_driver = {
        .driver = {
                  .name = "mxc_v4l2_capture",
```

```
                .owner = THIS_MODULE,
                .of_match_table = mxc_v4l2_dt_ids,
                },
        .id_table = imx_v4l2_devtype,
        .probe = mxc_v4l2_probe,
        .remove = mxc_v4l2_remove,
        .suspend = mxc_v4l2_suspend,
        .resume = mxc_v4l2_resume,
        .shutdown = NULL,
};
```

Step 3: Probe

```
static int mxc_v4l2_probe(struct platform_device *pdev)
{
        struct device_node *np = pdev->dev.of_node;
        int device_id = -1;
        int ret;

        /* Create cam and initialize it. */
        cam_data *cam = kmalloc(sizeof(cam_data), GFP_KERNEL);
        if (cam == NULL) {
                pr_err("ERROR: v4l2 capture: failed to register camera\n");
                return -1;
        }

        init_camera_struct(cam, pdev);
        pdev->dev.release = camera_platform_release;

        /* Set up the v4l2 device and register it*/
        cam->self->priv = cam;
        v4l2_int_device_register(cam->self);

        ret = of_property_read_u32(np, "device_id", &device_id);
        if (ret)
                device_id = -1;

        /* register v4l video device */
        if (video_register_device(cam->video_dev,
VFL_TYPE_GRABBER,
```

```
                (device_id >= 0) ? device_id : video_nr) < 0) {
            kfree(cam);
            cam = NULL;
            pr_err("ERROR: v4l2 capture: video_register_device failed\n");
            return -1;
        }
        pr_debug("   Video device registered: %s #%d\n",
                 cam->video_dev->name, cam->video_dev->minor);

        if (device_create_file(&cam->video_dev->dev,
                    &dev_attr_fsl_v4l2_capture_property))
            dev_err(&pdev->dev, "Error on creating sysfs file"
                    " for capture\n");

        if (device_create_file(&cam->video_dev->dev,
                    &dev_attr_fsl_v4l2_overlay_property))
            dev_err(&pdev->dev, "Error on creating sysfs file"
                    " for overlay\n");

        if (device_create_file(&cam->video_dev->dev,
                    &dev_attr_fsl_csi_property))
            dev_err(&pdev->dev, "Error on creating sysfs file"
                    " for csi number\n");

        return 0;
}
```

Step 4: Device tree

```
v4l2_cap_0 {
            compatible = "fsl,imx6q-v4l2-capture";
            ipu_id = <0>;
            csi_id = <0>;
            mclk_source = <0>;
            status = "okay";
        };

        v4l2_cap_1 {
            compatible = "fsl,imx6q-v4l2-capture";
            ipu_id = <0>;
            csi_id = <1>;
```

```
        mipi_camera = <1>;
        mclk_source = <0>;
        status = "okay";
    };

    v4l2_cap_2: v4l2_cap_2 {
        compatible = "fsl,imx6q-v4l2-capture";
        ipu_id = <0>;
        csi_id = <1>;
        mclk_source = <0>;
        status = "okay";
    };


    v4l2_cap_3 {
        compatible = "fsl,imx6q-v4l2-capture";
        ipu_id = <0>;
        csi_id = <0>;
        mipi_camera = <1>;
        mclk_source = <0>;
        status = "okay";
    };
```

Step 5: init_camera_struct

```
static int init_camera_struct(cam_data *cam, struct platform_device
*pdev)
{
    const struct of_device_id *of_id =
                of_match_device(mxc_v4l2_dt_ids, &pdev->dev);
    struct device_node *np = pdev->dev.of_node;
    int ipu_id, csi_id, mclk_source, mipi_camera;
    int ret = 0;
    struct v4l2_device *v4l2_dev;
    static int camera_id;

    pr_debug("%s\n", __func__);
```

```
	ret = of_property_read_u32(np, "ipu_id", &ipu_id);
	if (ret) {
		dev_err(&pdev->dev, "ipu_id missing or invalid\n");
		return ret;
	}

	ret = of_property_read_u32(np, "csi_id", &csi_id);
	if (ret) {
		dev_err(&pdev->dev, "csi_id missing or invalid\n");
		return ret;
	}

	ret = of_property_read_u32(np, "mclk_source", &mclk_source);
	if (ret) {
		dev_err(&pdev->dev, "sensor mclk missing or invalid\n");
		return ret;
	}

	ret = of_property_read_u32(np, "mipi_camera",
&mipi_camera);
	if (ret)
		mipi_camera = 0;

	/* Default everything to 0 */
	memset(cam, 0, sizeof(cam_data));

	/* get devtype to distinguish if the cpu is imx5 or imx6
	 * IMX5_V4L2 specify the cpu is imx5
	 * IMX6_V4L2 specify the cpu is imx6q or imx6sdl
	 */
	if (of_id)
		pdev->id_entry = of_id->data;
	cam->devtype = pdev->id_entry->driver_data;

	cam->ipu = ipu_get_soc(ipu_id);
	if (cam->ipu == NULL) {
		pr_err("ERROR: v4l2 capture: failed to get ipu\n");
		return -EINVAL;
	} else if (cam->ipu == ERR_PTR(-ENODEV)) {
		pr_err("ERROR: v4l2 capture: get invalid ipu\n");
		return -ENODEV;
	}

	init_MUTEX(&cam->param_lock);
```

```
        init_MUTEX(&cam->busy_lock);
        INIT_DELAYED_WORK(&cam->power_down_work,
power_down_callback);

        cam->video_dev = video_device_alloc();
        if (cam->video_dev == NULL)
                return -ENODEV;

        *(cam->video_dev) = mxc_v4l_template;

        video_set_drvdata(cam->video_dev, cam);
        dev_set_drvdata(&pdev->dev, (void *)cam);
        cam->video_dev->minor = -1;

        v4l2_dev = kzalloc(sizeof(*v4l2_dev), GFP_KERNEL);
        if (!v4l2_dev) {
                dev_err(&pdev->dev, "failed to allocate v4l2_dev structure\n");
                video_device_release(cam->video_dev);
                return -ENOMEM;
        }

        if (v4l2_device_register(&pdev->dev, v4l2_dev) < 0) {
                dev_err(&pdev->dev, "register v4l2 device failed\n");
                video_device_release(cam->video_dev);
                kfree(v4l2_dev);
                return -ENODEV;
        }
        cam->video_dev->v4l2_dev = v4l2_dev;

        init_waitqueue_head(&cam->enc_queue);
        init_waitqueue_head(&cam->still_queue);

        /* setup cropping */
        cam->crop_bounds.left = 0;
        cam->crop_bounds.width = 640;
        cam->crop_bounds.top = 0;
        cam->crop_bounds.height = 480;
        cam->crop_current = cam->crop_defrect = cam->crop_bounds;
        ipu_csi_set_window_size(cam->ipu, cam->crop_current.width,
                                cam->crop_current.height, cam->csi);
        ipu_csi_set_window_pos(cam->ipu, cam->crop_current.left,
                                cam->crop_current.top, cam->csi);
        cam->streamparm.parm.capture.capturemode = 0;
```

```
	cam->standard.index = 0;
	cam->standard.id = V4L2_STD_UNKNOWN;
	cam->standard.frameperiod.denominator = 30;
	cam->standard.frameperiod.numerator = 1;
	cam->standard.framelines = 480;
	cam->standard_autodetect = true;
	cam->streamparm.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
	cam->streamparm.parm.capture.timeperframe = cam-
>standard.frameperiod;
	cam->streamparm.parm.capture.capability =
V4L2_CAP_TIMEPERFRAME;
	cam->overlay_on = false;
	cam->capture_on = false;
	cam->v4l2_fb.flags = V4L2_FBUF_FLAG_OVERLAY;

	cam->v2f.fmt.pix.sizeimage = 352 * 288 * 3 / 2;
	cam->v2f.fmt.pix.bytesperline = 288 * 3 / 2;
	cam->v2f.fmt.pix.width = 288;
	cam->v2f.fmt.pix.height = 352;
	cam->v2f.fmt.pix.pixelformat = V4L2_PIX_FMT_YUV420;
	cam->win.w.width = 160;
	cam->win.w.height = 160;
	cam->win.w.left = 0;
	cam->win.w.top = 0;

	cam->ipu_id = ipu_id;
	cam->csi = csi_id;
	cam->mipi_camera = mipi_camera;
	cam->mclk_source = mclk_source;
	cam->mclk_on[cam->mclk_source] = false;

	cam->enc_callback = camera_callback;
	init_waitqueue_head(&cam->power_queue);
	spin_lock_init(&cam->queue_int_lock);
	spin_lock_init(&cam->dqueue_int_lock);

	cam->dummy_frame.vaddress = dma_alloc_coherent(0,
			SZ_8M, &cam->dummy_frame.paddress,
			GFP_DMA | GFP_KERNEL);
	if (cam->dummy_frame.vaddress == 0)
		pr_err("ERROR: v4l2 capture: Allocate dummy frame "
		    "failed.\n");
	cam->dummy_frame.buffer.length = SZ_8M;
```

```
        cam->self = kmalloc(sizeof(struct v4l2_int_device),
GFP_KERNEL);
        cam->self->module = THIS_MODULE;
        sprintf(cam->self->name, "mxc_v4l2_cap%d", camera_id++);
        cam->self->type = v4l2_int_type_master;
        cam->self->u.master = &mxc_v4l2_master;

        return 0;
}
```

Step 5: mxc_v4l_fops

```
static struct v4l2_file_operations mxc_v4l_fops = {
        .owner = THIS_MODULE,
        .open = mxc_v4l_open,
        .release = mxc_v4l_close,
        .read = mxc_v4l_read,
        .unlocked_ioctl = mxc_v4l_ioctl,
        .mmap = mxc_mmap,
        .poll = mxc_poll,
};

static struct video_device mxc_v4l_template = {
        .name = "Mxc Camera",
        .fops = &mxc_v4l_fops,
        .release = video_device_release,
};
```

Step 6: cam_data

```
typedef struct _cam_data {
        struct video_device *video_dev;
        int device_type;

    /* Encoder */
        struct list_head ready_q;
        struct list_head done_q;
        struct list_head working_q;
```

```
        unsigned int ipu_id;
        unsigned int csi;
        unsigned mipi_camera;
        int csi_in_use;
        u8 mclk_source;

       struct v4l2_int_device *all_sensors[MXC_SENSOR_NUM];
        struct v4l2_int_device *sensor;
        struct v4l2_int_device *self;
        int sensor_index;
        void *ipu;
        void *csi_soc;
        enum imx_v4l2_devtype devtype;
};
```

Step 7: mxc_v4l2_master [v4l2_int_device]

```
/*! Information about this driver. */
static struct v4l2_int_master mxc_v4l2_master = {
        .attach = mxc_v4l2_master_attach,
        .detach = mxc_v4l2_master_detach,
};

static int mxc_v4l2_master_attach(struct v4l2_int_device *slave)
{
        cam_data *cam = slave->u.slave->master->priv;
        struct v4l2_format cam_fmt;
        int i;
        struct sensor_data *sdata = slave->priv;

        pr_debug("%s:slave.name = %s, master.name = %s\n", __func__,
                slave->name, slave->u.slave->master->name);

        if (slave == NULL) {
                pr_err("ERROR: v4l2 capture: slave parameter not valid.\n");
                return -1;
        }

        if ((sdata->ipu_id != cam->ipu_id) || (sdata->csi != cam->csi) ||
(sdata->mipi_camera != cam->mipi_camera)) {
                pr_info("%s: ipu(%d:%d)/csi(%d:%d)/mipi(%d:%d) doesn't
match\n", __func__,
                        sdata->ipu_id, cam->ipu_id, sdata->csi, cam->csi, sdata-
```

```
>mipi_camera, cam->mipi_camera);
                return -1;
        }

        cam->sensor = slave;

        if (cam->sensor_index < MXC_SENSOR_NUM) {
                cam->all_sensors[cam->sensor_index] = slave;
                cam->sensor_index++;
        } else {
                pr_err("ERROR: v4l2 capture: slave number exceeds "
                        "the maximum.\n");
                return -1;
        }
        for (i = 0; i < cam->sensor_index; i++) {
                pr_err("%s: %x\n", __func__, i);
                vidioc_int_dev_exit(cam->all_sensors[i]);
                vidioc_int_s_power(cam->all_sensors[i], 0);
        }

        cam_fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        vidioc_int_g_fmt_cap(cam->sensor, &cam_fmt);

        /* Used to detect TV in (type 1) vs. camera (type 0)*/
        cam->device_type = cam_fmt.fmt.pix.priv;

        /* Set the input size to the ipu for this device */
        cam->crop_bounds.top = cam->crop_bounds.left = 0;
        cam->crop_bounds.width = cam_fmt.fmt.pix.width;
        cam->crop_bounds.height = cam_fmt.fmt.pix.height;

        /* This also is the max crop size for this device. */
        cam->crop_defrect.top = cam->crop_defrect.left = 0;
        cam->crop_defrect.width = cam_fmt.fmt.pix.width;
        cam->crop_defrect.height = cam_fmt.fmt.pix.height;

        /* At this point, this is also the current image size. */
        cam->crop_current.top = cam->crop_current.left = 0;
        cam->crop_current.width = cam_fmt.fmt.pix.width;
        cam->crop_current.height = cam_fmt.fmt.pix.height;
}

static void mxc_v4l2_master_detach(struct v4l2_int_device *slave)
{
```

```
	unsigned int i;
	cam_data *cam = slave->u.slave->master->priv;

	pr_debug("%s\n", __func__);

	if (cam->sensor_index > 1) {
		for (i = 0; i < cam->sensor_index; i++) {
			if (cam->all_sensors[i] != slave)
				continue;
			/* Move all the sensors behind this
			 * sensor one step forward
			 */
			for (; i <= MXC_SENSOR_NUM - 2; i++)
				cam->all_sensors[i] = cam->all_sensors[i+1];
			break;
		}
		/* Point current sensor to the last one */
		cam->sensor = cam->all_sensors[cam->sensor_index - 2];
	} else
		cam->sensor = NULL;

	cam->sensor_index--;
	vidioc_int_dev_exit(slave);
}
```

Step 8 : V4L OPEN

```
static int mxc_v4l_open(struct file *file)
{
	struct video_device *dev = video_devdata(file);
	cam_data *cam = video_get_drvdata(dev);
	int err = 0;
	struct sensor_data *sensor;
	int csi_bit;

	if (!cam) {
		pr_err("%s: %s cam_data not found!\n", __func__, dev->name);
		return -EBADF;
	}
	if (!cam->sensor) {
		pr_err("%s: %s no sensor ipu%d/csi%d\n",
			__func__, dev->name, cam->ipu_id, cam->csi);
```

```
		return -EAGAIN;
	}
	if (cam->sensor->type != v4l2_int_type_slave) {
		pr_err("%s: %s wrong type ipu%d/csi%d, type=%d/%d\n",
			__func__, dev->name, cam->ipu_id, cam->csi,
			cam->sensor->type, v4l2_int_type_slave);
		return -EAGAIN;
	}

	sensor = cam->sensor->priv;
	if (!sensor) {
		pr_err("%s: Internal error, sensor_data is not found!\n",
			__func__);
		return -EBADF;
	}
	pr_debug("%s: %s ipu%d/csi%d\n", __func__, dev->name,
		cam->ipu_id, cam->csi);

	down(&cam->busy_lock);

	err = 0;
	if (signal_pending(current))
		goto oops;

	if (cam->open_count++ == 0) {
		struct regmap *gpr;

		csi_bit = (cam->ipu_id << 1) | cam->csi;
		if (test_and_set_bit(csi_bit, &csi_in_use)) {
			pr_err("%s: %s CSI already in use\n", __func__, dev->name);
			err = -EBUSY;
			cam->open_count = 0;
			goto oops;
		}
		cam->csi_in_use = 1;

		gpr = syscon_regmap_lookup_by_compatible("fsl,imx6q-iomuxc-
gpr");
		if (!IS_ERR(gpr)) {
			if (of_machine_is_compatible("fsl,imx6q")) {
				if (cam->ipu_id == cam->csi) {
					unsigned shift = 19 + cam->csi;
					unsigned mask = 1 << shift;
					unsigned val = (cam->mipi_camera ? 0 : 1) <<
```

```
shift;

                        regmap_update_bits(gpr, IOMUXC_GPR1, mask,
val);
 }
                } else if (of_machine_is_compatible("fsl,imx6dl")) {
                        unsigned shift = cam->csi * 3;
                        unsigned mask = 7 << shift;
                        unsigned val = (cam->mipi_camera ? csi_bit : 4) <<
shift;

                        regmap_update_bits(gpr, IOMUXC_GPR13, mask, val);
                }
        } else {
                pr_err("%s: failed to find fsl,imx6q-iomux-gpr regmap\n",
                        __func__);
        }

        wait_event_interruptible(cam->power_queue,
                                cam->low_power == false);

        err = mxc_cam_select_input(cam, cam->current_input);
        if (err)
                err = mxc_cam_select_input(cam, cam->current_input ^ 1);
        cam->enc_counter = 0;
        INIT_LIST_HEAD(&cam->ready_q);
        INIT_LIST_HEAD(&cam->working_q);
        INIT_LIST_HEAD(&cam->done_q);
        setup_ifparm(cam, 1);
        if (!IS_ERR(sensor->sensor_clk))
                clk_prepare_enable(sensor->sensor_clk);
        power_up_camera(cam);
    }

    file->private_data = dev;
oops:
    up(&cam->busy_lock);
    return err;
}
```

Step 9 : V4L CLOSE

```
static int mxc_v4l_close(struct file *file)
{
	struct video_device *dev = video_devdata(file);
	int err = 0;
	cam_data *cam = video_get_drvdata(dev);
	struct sensor_data *sensor;
	pr_debug("%s\n", __func__);

	if (!cam) {
		pr_err("%s: cam_data not found!\n", __func__);
		return -EBADF;
	}

	if (!cam->sensor) {
		pr_err("%s: Internal error, camera is not found!\n",
			__func__);
		return -EBADF;
	}

	sensor = cam->sensor->priv;
	if (!sensor) {
		pr_err("%s: Internal error, sensor_data is not found!\n",
			__func__);
		return -EBADF;
	}

	down(&cam->busy_lock);
	/* for the case somebody hit the ctrl C */
	if (cam->overlay_pid == current->pid && cam->overlay_on) {
		err = stop_preview(cam);
		cam->overlay_on = false;
	}

	if (--cam->open_count == 0) {
		err |= mxc_streamoff(cam);
		wake_up_interruptible(&cam->enc_queue);

		if (!IS_ERR(sensor->sensor_clk))
			clk_disable_unprepare(sensor->sensor_clk);
		wait_event_interruptible(cam->power_queue,
```

```
                                cam->low_power == false);
        pr_debug("mxc_v4l_close: release resource\n");

        if (strcmp(mxc_capture_inputs[cam->current_input].name,
                   "CSI MEM") == 0) {
#if defined(CONFIG_MXC_IPU_CSI_ENC) ||
defined(CONFIG_MXC_IPU_CSI_ENC_MODULE)
                err |= csi_enc_deselect(cam);
#endif
        } else if (strcmp(mxc_capture_inputs[cam->current_input].name,
                          "CSI IC MEM") == 0) {
#if defined(CONFIG_MXC_IPU_PRP_ENC) ||
defined(CONFIG_MXC_IPU_PRP_ENC_MODULE)
                err |= prp_enc_deselect(cam);
#endif
        }

        mxc_free_frame_buf(cam);
        file->private_data = NULL;

        /* capture off */
        wake_up_interruptible(&cam->enc_queue);
        mxc_free_frames(cam);
        cam->enc_counter++;
         power_off_camera(cam);

        if (cam->csi_in_use) {
                int csi_bit = (cam->ipu_id << 1) | cam->csi;

                clear_bit(csi_bit, &csi_in_use);
                cam->csi_in_use = 0;
        }
    }

    up(&cam->busy_lock);

    return err;
}
```

Step 10 : V4L MMAP

```
static int mxc_mmap(struct file *file, struct vm_area_struct *vma)
{
	struct video_device *dev = video_devdata(file);
	unsigned long size;
	int res = 0;
	cam_data *cam = video_get_drvdata(dev);

	pr_debug("%s:pgoff=0x%lx, start=0x%lx, end=0x%lx\n", __func__,
		 vma->vm_pgoff, vma->vm_start, vma->vm_end);

	/* make this _really_ smp-safe */
	if (down_interruptible(&cam->busy_lock))
		return -EINTR;

	size = vma->vm_end - vma->vm_start;
	vma->vm_page_prot = pgprot_writecombine(vma->vm_page_prot);

	if (remap_pfn_range(vma, vma->vm_start,
			    vma->vm_pgoff, size, vma->vm_page_prot)) {
		pr_err("ERROR: v4l2 capture: mxc_mmap: "
		       "remap_pfn_range failed\n");
		res = -ENOBUFS;
		goto mxc_mmap_exit;
	}

	vma->vm_flags &= ~VM_IO;	/* using shared anonymous pages */

mxc_mmap_exit:
	up(&cam->busy_lock);
	return res;
}
```

Step 11 : V4L IOCTL

```
static long mxc_v4l_do_ioctl(struct file *file,
			     unsigned int ioctlnr, void *arg)
{
	struct video_device *dev = video_devdata(file);
	cam_data *cam = video_get_drvdata(dev);
	int retval = 0;
	unsigned long lock_flags;
```

```
	pr_debug("%s: %x ipu%d/csi%d\n", __func__, ioctlnr, cam->ipu_id,
cam->csi);
	wait_event_interruptible(cam->power_queue, cam->low_power ==
false);
	/* make this _really_ smp-safe */
	if (ioctlnr != VIDIOC_DQBUF)
		if (down_interruptible(&cam->busy_lock))
			return -EBUSY;

	switch (ioctlnr) {
	/*!
	 * V4l2 VIDIOC_QUERYCAP ioctl
	 */
	case VIDIOC_QUERYCAP: {
		struct v4l2_capability *cap = arg;
		pr_debug("   case VIDIOC_QUERYCAP\n");
		strcpy(cap->driver, "mxc_v4l2");
		cap->version = KERNEL_VERSION(0, 1, 11);
		cap->device_caps = V4L2_CAP_VIDEO_CAPTURE |
V4L2_CAP_STREAMING;
		cap->capabilities = cap->device_caps | V4L2_CAP_DEVICE_CAPS |
				V4L2_CAP_VIDEO_OVERLAY |
				V4L2_CAP_READWRITE;

		if (cam && cam->sensor)
			strlcpy(cap->card, cam->sensor->name, sizeof(cap->card));
		else
			cap->card[0] = '\0';

		if (dev->v4l2_dev)
			strlcpy(cap->bus_info, dev->v4l2_dev->name, sizeof(cap-
>bus_info));
		else
		 cap->card[0] = '\0';

		if (dev->v4l2_dev)
			strlcpy(cap->bus_info, dev->v4l2_dev->name, sizeof(cap-
>bus_info));
		else
			cap->bus_info[0] = '\0';

		break;
	}
```

```
        /*!
         * V4l2 VIDIOC_G_FMT ioctl
         */
        case VIDIOC_G_FMT: {
                struct v4l2_format *gf = arg;
                pr_debug("   case VIDIOC_G_FMT\n");
                retval = mxc_v4l2_g_fmt(cam, gf);
                break;
        }

/*!
         * V4l2 VIDIOC_S_DEST_CROP ioctl
         */
    case VIDIOC_S_DEST_CROP: {
                struct v4l2_mxc_dest_crop  *of = arg;
                pr_debug("   case VIDIOC_S_DEST_CROP\n");
                cam->offset.u_offset = of->offset.u_offset;
                cam->offset.v_offset = of->offset.v_offset;
                break;
        }

        /*!
         * V4l2 VIDIOC_S_FMT ioctl
         */
    case VIDIOC_S_FMT: {
                struct v4l2_format *sf = arg;
                pr_debug("   case VIDIOC_S_FMT\n");
                retval = mxc_v4l2_s_fmt(cam, sf);
                break;
        }

   case VIDIOC_REQBUFS: {
                struct v4l2_requestbuffers *req = arg;
                pr_debug("   case VIDIOC_REQBUFS\n");

                if (req->count > FRAME_NUM) {
                        pr_err("ERROR: v4l2 capture: VIDIOC_REQBUFS: "
                                "not enough buffers\n");
                        req->count = FRAME_NUM;
                }

                if ((req->type != V4L2_BUF_TYPE_VIDEO_CAPTURE)) {
                        pr_err("ERROR: v4l2 capture: VIDIOC_REQBUFS: "
```

```
                "wrong buffer type\n");
            retval = -EINVAL;
            break;
        }

        mxc_streamoff(cam);
        if (req->memory & V4L2_MEMORY_MMAP) {
            mxc_free_frame_buf(cam);
            retval = mxc_allocate_frame_buf(cam, req->count);
        }
        break;
    }

case VIDIOC_QUERYBUF: {
        struct v4l2_buffer *buf = arg;
        int index = buf->index;
        pr_debug("   case VIDIOC_QUERYBUF\n");

        if (buf->type != V4L2_BUF_TYPE_VIDEO_CAPTURE) {
            pr_err("ERROR: v4l2 capture: "
                "VIDIOC_QUERYBUFS: "
                "wrong buffer type\n");
            retval = -EINVAL;
            break;
        }

        if (buf->memory & V4L2_MEMORY_MMAP) {
            memset(buf, 0, sizeof(buf));
            buf->index = index;
        }

        down(&cam->param_lock);
        if (buf->memory & V4L2_MEMORY_USERPTR) {
            mxc_v4l2_release_bufs(cam);
            retval = mxc_v4l2_prepare_bufs(cam, buf);
        }

        if (buf->memory & V4L2_MEMORY_MMAP)
            retval = mxc_v4l2_buffer_status(cam, buf);
        up(&cam->param_lock);
        break;
    }

case VIDIOC_QBUF: {
```

```
        struct v4l2_buffer *buf = arg;
        int index = buf->index;
        pr_debug("   case VIDIOC_QBUF, length=%d\n", buf->length);

        if (index < 0 || index >= FRAME_NUM) {
                retval = -EINVAL;
                break;
        }
        spin_lock_irqsave(&cam->queue_int_lock, lock_flags);
        if ((cam->frame[index].buffer.flags & 0x7) ==
           V4L2_BUF_FLAG_MAPPED) {
                cam->frame[index].buffer.flags |=
                   V4L2_BUF_FLAG_QUEUED;
                list_add_tail(&cam->frame[index].queue,
                              &cam->ready_q);
        } else if (cam->frame[index].buffer.
                  flags & V4L2_BUF_FLAG_QUEUED) {
                pr_err("ERROR: v4l2 capture: VIDIOC_QBUF: "
                       "buffer already queued\n");
                retval = -EINVAL;
        } else if (cam->frame[index].buffer.
                  flags & V4L2_BUF_FLAG_DONE) {
                pr_err("ERROR: v4l2 capture: VIDIOC_QBUF: "
                       "overwrite done buffer.\n");
                cam->frame[index].buffer.flags &=
                   ~V4L2_BUF_FLAG_DONE;
                cam->frame[index].buffer.flags |=
                   V4L2_BUF_FLAG_QUEUED;
                retval = -EINVAL;
        }

        buf->flags = cam->frame[index].buffer.flags;
        spin_unlock_irqrestore(&cam->queue_int_lock, lock_flags);
        break;
 }

case VIDIOC_DQBUF: {
        struct v4l2_buffer *buf = arg;
        pr_debug("   case VIDIOC_DQBUF\n");

        if ((cam->enc_counter == 0) &&
                (file->f_flags & O_NONBLOCK)) {
                retval = -EAGAIN;
                break;
```

```
            }

            retval = mxc_v4l_dqueue(cam, buf);
            break;
        }

      case VIDIOC_STREAMON: {
            pr_debug("   case VIDIOC_STREAMON\n");
            retval = mxc_streamon(cam);
            break;
        }

        /*!
         * V4l2 VIDIOC_STREAMOFF ioctl
         */
     case VIDIOC_STREAMOFF: {
            pr_debug("   case VIDIOC_STREAMOFF\n");
            retval = mxc_streamoff(cam);
            break;
        }

     case VIDIOC_G_CTRL: {
            pr_debug("   case VIDIOC_G_CTRL\n");
            retval = mxc_v4l2_g_ctrl(cam, arg);
            break;
        }

        /*!
         * V4l2 VIDIOC_S_CTRL ioctl
         */
     case VIDIOC_S_CTRL: {
            pr_debug("   case VIDIOC_S_CTRL\n");
            retval = mxc_v4l2_s_ctrl(cam, arg);
            break;
        }

    case VIDIOC_CROPCAP: {
            struct v4l2_cropcap *cap = arg;
            pr_debug("   case VIDIOC_CROPCAP\n");
            if (cap->type != V4L2_BUF_TYPE_VIDEO_CAPTURE &&
                cap->type != V4L2_BUF_TYPE_VIDEO_OVERLAY) {
                retval = -EINVAL;
                break;
            }
```

```
            cap->bounds = cam->crop_bounds;
            cap->defrect = cam->crop_defrect;
            cap->pixelaspect.numerator = 1;
            cap->pixelaspect.denominator = 1;
            break;
        }
    case VIDIOC_G_PARM: {
            struct v4l2_streamparm *parm = arg;
            pr_debug("   case VIDIOC_G_PARM\n");
            if (cam->sensor)
                retval = vidioc_int_g_parm(cam->sensor, parm);
            else {
                pr_err("ERROR: v4l2 capture: slave not found!\n");
                retval = -ENODEV;
            }
            break;
        }

        case VIDIOC_S_PARM:  {
            struct v4l2_streamparm *parm = arg;
            pr_debug("   case VIDIOC_S_PARM\n");
            if (cam->sensor)
                retval = mxc_v4l2_s_param(cam, parm);
            else {
                pr_err("ERROR: v4l2 capture: slave not found!\n");
                retval = -ENODEV;
            }
            break;
        }

        case VIDIOC_G_INPUT: {
            int *index = arg;
            pr_debug("   case VIDIOC_G_INPUT\n");
            *index = cam->current_input;
            break;
        }

       case VIDIOC_S_INPUT: {
            int index = *(int *)arg;
            pr_debug("   case VIDIOC_S_INPUT(%d)\n", index);
            if (index >= MXC_V4L2_CAPTURE_NUM_INPUTS) {
                retval = -EINVAL;
                break;
            }
```

```
        if (index == cam->current_input)
                break;

        if ((mxc_capture_inputs[cam->current_input].status &
           V4L2_IN_ST_NO_POWER) == 0) {
                retval = mxc_streamoff(cam);
                if (retval)
                        break;
                mxc_capture_inputs[cam->current_input].status |=
                                        V4L2_IN_ST_NO_POWER;
        }

        retval = mxc_cam_select_input(cam, index);
        break;
}
```

Step 12 : V4L IOCTL : REQUEST BUF

```
static int mxc_allocate_frame_buf(cam_data *cam, int count)
{
        int i;

        pr_debug("%s: size=%d\n", __func__, cam->v2f.fmt.pix.sizeimage);

        for (i = 0; i < count; i++) {
                cam->frame[i].vaddress =
                   dma_alloc_coherent(0,
                                PAGE_ALIGN(cam->v2f.fmt.pix.sizeimage),
                                &cam->frame[i].paddress,
                                GFP_DMA | GFP_KERNEL);

                if (cam->frame[i].vaddress == 0) {
                        pr_err("%s: failed.\n", __func__);
                        mxc_free_frame_buf(cam);
                        return -ENOBUFS;
                }
                cam->frame[i].buffer.index = i;
                cam->frame[i].buffer.flags = V4L2_BUF_FLAG_MAPPED;
                cam->frame[i].buffer.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
                cam->frame[i].buffer.length =
```

```
                PAGE_ALIGN(cam->v2f.fmt.pix.sizeimage);
            cam->frame[i].buffer.memory = V4L2_MEMORY_MMAP;
            cam->frame[i].buffer.m.offset = cam->frame[i].paddress;
            cam->frame[i].index = i;
        }

        return 0;
}

static int mxc_free_frame_buf(cam_data *cam)
{
        int i;

        pr_debug("%s\n", __func__);

        for (i = 0; i < FRAME_NUM; i++) {
            if (cam->frame[i].vaddress != 0) {
                dma_free_coherent(0, cam->frame[i].buffer.length,
                                  cam->frame[i].vaddress,
                                  cam->frame[i].paddress);
                cam->frame[i].vaddress = 0;
            }
        }

        return 0;
}
```

Step 13 : V4L IOCTL : QUERY BUF

```
static int mxc_v4l2_buffer_status(cam_data *cam, struct v4l2_buffer *buf)
{
        pr_debug("%s\n", __func__);


        if (buf->index < 0 || buf->index >= FRAME_NUM) {
            pr_err("ERROR: v4l2 capture: mxc_v4l2_buffer_status buffers "
                   "not allocated\n");
            return -EINVAL;
        }
        memcpy(buf, &(cam->frame[buf->index].buffer), sizeof(*buf));
```

```
        return 0;
}
```

Step 14 : V4L IOCTL : QBUF

```
static void mxc_free_frames(cam_data *cam)
{
        int i;

        pr_debug("%s\n", __func__);

        for (i = 0; i < FRAME_NUM; i++)
                cam->frame[i].buffer.flags = V4L2_BUF_FLAG_MAPPED;

        cam->enc_counter = 0;
         INIT_LIST_HEAD(&cam->ready_q);
         INIT_LIST_HEAD(&cam->working_q);
         INIT_LIST_HEAD(&cam->done_q);
}
```

FLAG	VALUE
/* Flags for 'flags' field */ /* Buffer is mapped (flag) */ **#define** **V4L2_BUF_FLAG_MAPPED**	0x00000001
/* Buffer is queued for processing */ **#define** **V4L2_BUF_FLAG_QUEUED**	0x00000002
/* Buffer is ready */ **#define V4L2_BUF_FLAG_DONE**	0x00000004

```
        if ((cam->frame[index].buffer.flags & 0x7) ==
                V4L2_BUF_FLAG_MAPPED) {
                    cam->frame[index].buffer.flags |=
```

```
                V4L2_BUF_FLAG_QUEUED;
            list_add_tail(&cam->frame[index].queue,
                    &cam->ready_q);
        } else if (cam->frame[index].buffer.
                flags & V4L2_BUF_FLAG_QUEUED) {
            pr_err("ERROR: v4l2 capture: VIDIOC_QBUF: "
                "buffer already queued\n");
            retval = -EINVAL;
        } else if (cam->frame[index].buffer.
                flags & V4L2_BUF_FLAG_DONE) {
            pr_err("ERROR: v4l2 capture: VIDIOC_QBUF: "
                "overwrite done buffer.\n");
            cam->frame[index].buffer.flags &=
                ~V4L2_BUF_FLAG_DONE;
            cam->frame[index].buffer.flags |=
                V4L2_BUF_FLAG_QUEUED;
            retval = -EINVAL;
        }

        buf->flags = cam->frame[index].buffer.flags;
```

Step 15 : V4L IOCTL : DQBUF

```
static int mxc_v4l_dqueue(cam_data *cam, struct v4l2_buffer *buf)
{
    int retval = 0;
    struct mxc_v4l_frame *frame;
    unsigned long lock_flags;

    pr_debug("%s\n", __func__);

    if (!wait_event_interruptible_timeout(cam->enc_queue,
                                cam->enc_counter != 0,
                                10 * HZ)) {
        pr_err("ERROR: v4l2 capture: mxc_v4l_dqueue timeout "
            "enc_counter %x\n",
            cam->enc_counter);
        return -ETIME;
    } else if (signal_pending(current)) {
        pr_err("ERROR: v4l2 capture: mxc_v4l_dqueue() "
            "interrupt received\n");
        return -ERESTARTSYS;
    }

    if (down_interruptible(&cam->busy_lock))
        return -EBUSY;
```

```
        spin_lock_irqsave(&cam->dqueue_int_lock, lock_flags);
        cam->enc_counter--;

        frame = list_entry(cam->done_q.next, struct mxc_v4l_frame,
queue);
        list_del(cam->done_q.next);
        if (frame->buffer.flags & V4L2_BUF_FLAG_DONE) {
                frame->buffer.flags &= ~V4L2_BUF_FLAG_DONE;


        } else if (frame->buffer.flags & V4L2_BUF_FLAG_QUEUED) {
                pr_err("ERROR: v4l2 capture: VIDIOC_DQBUF: "
                        "Buffer not filled.\n");
                frame->buffer.flags &= ~V4L2_BUF_FLAG_QUEUED;
                retval = -EINVAL;
        } else if ((frame->buffer.flags & 0x7) == V4L2_BUF_FLAG_MAPPED) {
                pr_err("ERROR: v4l2 capture: VIDIOC_DQBUF: "
                        "Buffer not queued.\n");
                retval = -EINVAL;
        }


    cam->frame[frame->index].buffer.field = cam->device_type ?
                        V4L2_FIELD_INTERLACED : V4L2_FIELD_NONE;

        buf->length = buf->bytesused = cam->v2f.fmt.pix.sizeimage;
        buf->index = frame->index;
        buf->flags = frame->buffer.flags;
        buf->m = cam->frame[frame->index].buffer.m;
        buf->timestamp = cam->frame[frame->index].buffer.timestamp;
        buf->field = cam->frame[frame->index].buffer.field;
        spin_unlock_irqrestore(&cam->dqueue_int_lock, lock_flags);

        up(&cam->busy_lock);
        return ret;
}
```

Sensor Driver

Step 1: I2C Client driver

```
static __init int ov5640_init(void)
{
	u8 err;

	err = i2c_add_driver(&ov5640_i2c_driver);
	if (err != 0)
		pr_err("%s:driver registration failed, error=%d\n",
			__func__, err);

	return err;
}

/*!
 * OV5640 cleanup function
 * Called on rmmod ov5640_camera.ko
 *
 * @return  Error code indicating success or failure
 */
static void __exit ov5640_clean(void)
{
	i2c_del_driver(&ov5640_i2c_driver);
}

module_init(ov5640_init);
module_exit(ov5640_clean);

MODULE_AUTHOR("Freescale Semiconductor, Inc.");
MODULE_DESCRIPTION("OV5640 MIPI Camera Driver");
MODULE_LICENSE("GPL");
MODULE_VERSION("1.0");
MODULE_ALIAS("CSI");
```

Step 2: Probe

```
enum v4l2_int_type {
	v4l2_int_type_master = 1,
	v4l2_int_type_slave
};

struct v4l2_int_device {
	/* Don't touch head. */
	struct list_head head;
```

```
        struct module *module;

        char name[V4L2NAMESIZE];

        enum v4l2_int_type type;
        union {
                struct v4l2_int_master *master;
                struct v4l2_int_slave *slave;
        } u;

        void *priv;
};
```

struct v4l2_int_slave {

```
        /* Don't touch master. */
        struct v4l2_int_device *master;

        char attach_to[V4L2NAMESIZE];

        int num_ioctls;
        struct v4l2_int_ioctl_desc *ioctls;
};
```

```
static struct v4l2_int_slave ov5640_slave = {
        .ioctls = ov5640_ioctl_desc,
        .num_ioctls = ARRAY_SIZE(ov5640_ioctl_desc),
};

static struct v4l2_int_device ov5640_int_device = {
        .module = THIS_MODULE,
        .name = "ov5640_mipi",
        .type = v4l2_int_type_slave,
        .u = {
                .slave = &ov5640_slave,
        },
};
```

retval = v4l2_int_device_register(&ov5640_int_device);

Step 3: v4l2 int device register

```
int v4l2_int_device_register(struct v4l2_int_device *d)
{
	if (d->type == v4l2_int_type_slave)
		sort(d->u.slave->ioctls, d->u.slave->num_ioctls,
		    sizeof(struct v4l2_int_ioctl_desc),
		    &ioctl_sort_cmp, NULL);
	mutex_lock(&mutex);
	list_add(&d->head, &int_list);
	v4l2_int_device_try_attach_all();
	mutex_unlock(&mutex);

	return 0;
}
EXPORT_SYMBOL_GPL(v4l2_int_device_register);


void v4l2_int_device_try_attach_all(void)
{
	struct v4l2_int_device *m, *s;

	list_for_each_entry(m, &int_list, head) {
		if (m->type != v4l2_int_type_master)
			continue;

		list_for_each_entry(s, &int_list, head) {
			if (s->type != v4l2_int_type_slave)
				continue;

			/* Slave is connected? */
			if (s->u.slave->master)
				continue;

			/* Slave wants to attach to master? */
			if (s->u.slave->attach_to[0] != 0
			    && strncmp(m->name, s->u.slave->attach_to,
					V4L2NAMESIZE))
				continue;

			if (!try_module_get(m->module))
				continue;

			  s->u.slave->master = m;
```

```
                    if (m->u.master->attach(s)) {
                        s->u.slave->master = NULL;
                        module_put(m->module);
                        continue;
                    }
                }
            }
        }
}
EXPORT_SYMBOL_GPL(v4l2_int_device_try_attach_all);
```

Step 4: ioctl

```
struct v4l2_int_ioctl_desc {
        int num;
        v4l2_int_ioctl_func *func;
};

/*!
 * This structure defines all the ioctls for this module and links them to the
 * enumeration.
 */
static struct v4l2_int_ioctl_desc ov5640_ioctl_desc[] = {
        {vidioc_int_dev_init_num, (v4l2_int_ioctl_func *) ioctl_dev_init},
        {vidioc_int_dev_exit_num, ioctl_dev_exit},
        {vidioc_int_s_power_num, (v4l2_int_ioctl_func *) ioctl_s_power},
        {vidioc_int_g_ifparm_num, (v4l2_int_ioctl_func *) ioctl_g_ifparm},
/*      {vidioc_int_g_needs_reset_num,
                        (v4l2_int_ioctl_func *)ioctl_g_needs_reset}, */
/*      {vidioc_int_reset_num, (v4l2_int_ioctl_func *)ioctl_reset}, */
        {vidioc_int_init_num, (v4l2_int_ioctl_func *) ioctl_init},
        {vidioc_int_enum_fmt_cap_num,
                        (v4l2_int_ioctl_func *) ioctl_enum_fmt_cap},
/*      {vidioc_int_try_fmt_cap_num,
                        (v4l2_int_ioctl_func *)ioctl_try_fmt_cap}, */
        {vidioc_int_g_fmt_cap_num, (v4l2_int_ioctl_func *) ioctl_g_fmt_cap},
/*      {vidioc_int_s_fmt_cap_num, (v4l2_int_ioctl_func *) ioctl_s_fmt_cap}, */
        {vidioc_int_g_parm_num, (v4l2_int_ioctl_func *) ioctl_g_parm},
        {vidioc_int_s_parm_num, (v4l2_int_ioctl_func *) ioctl_s_parm},
/*      {vidioc_int_queryctrl_num, (v4l2_int_ioctl_func *)ioctl_queryctrl}, */
        {vidioc_int_g_ctrl_num, (v4l2_int_ioctl_func *) ioctl_g_ctrl},
        {vidioc_int_s_ctrl_num, (v4l2_int_ioctl_func *) ioctl_s_ctrl},
```

```
	{vidioc_int_enum_framesizes_num,
			(v4l2_int_ioctl_func *) ioctl_enum_framesizes},
	{vidioc_int_enum_frameintervals_num,
		(v4l2_int_ioctl_func *) ioctl_enum_frameintervals},
	{vidioc_int_g_chip_ident_num,
			(v4l2_int_ioctl_func *) ioctl_g_chip_ident},
	{vidioc_int_send_command_num,
			(v4l2_int_ioctl_func *) ioctl_send_command},
};
```