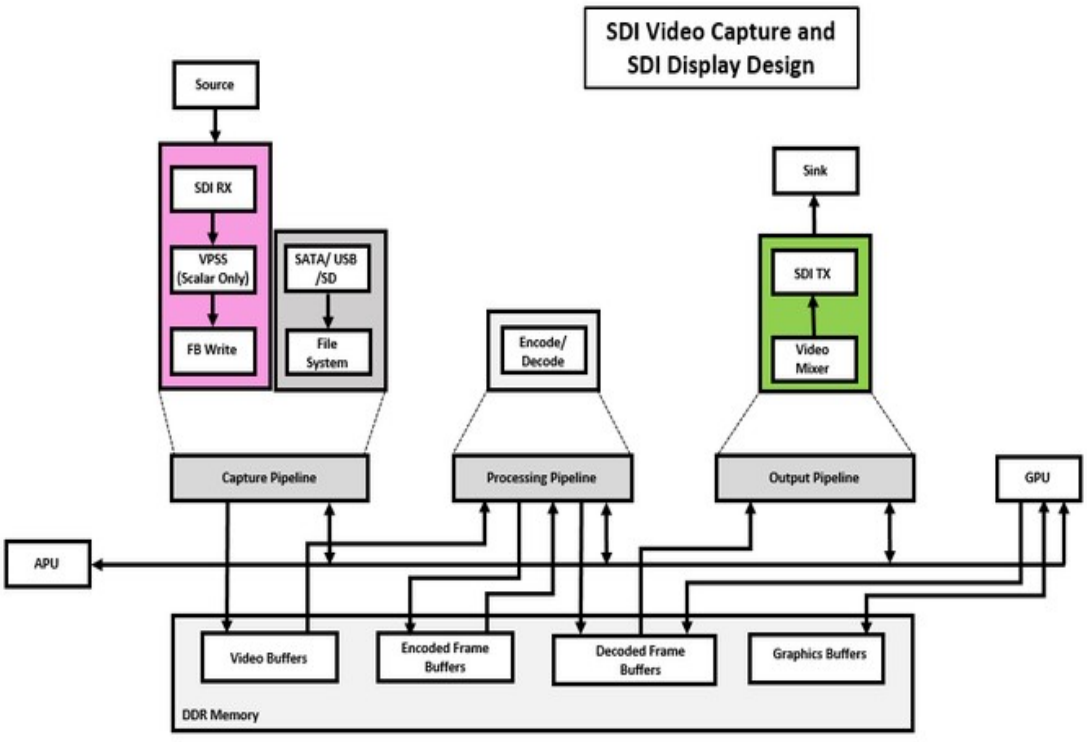
Table of Contents

Step 1: Device Tree

Step 1: Block Diagram



Step 2: Device Tree – Nodes

v_smpte_uhdsdi_rx_ss: v_smpte_uhdsdi_rx_ss@80000000 {
compatible = "xlnx,v-smpte-uhdsdi-rx-ss";
interrupt-parent = <&gic>;
interrupts = <0 89 4>;
reg = <0x0 0x80000000 0x0 0x10000>;
xlnx,include-axilite = "true";

```
                xlnx,include-edh = "true";
                xlnx,include-vid-over-axi = "true";
                xlnx,line-rate = "12G_SDI_8DS";
                clocks = <&clk_1>, <&si570_1>, <&clk_2>;
                clock-names = "s_axi_aclk", "sdi_rx_clk", "video_out_clk";

                ports {
                        #address-cells = <1>;
                        #size-cells = <0>;

                        port@0 {
                                reg = <0>;

                                xlnx,video-format = <XVIP_VF_YUV_422>;
                                xlnx,video-width = <10>;

                                sdirx_out: endpoint {
                                        remote-endpoint = <&vcap_sdirx_in>;
                                };
                        };
                };
        };

scaler_0:scaler@a0000000 {
                compatible = "xlnx,v-vpss-scaler-1.0";
                reg = <0x0 0xa0000000 0x0 0x40000>;
                clocks = <&vid_stream_clk>, <&misc_clk_2>;
                clock-names = "aclk_axis", "aclk_ctrl";
                xlnx,num-hori-taps = <8>;
                xlnx,num-vert-taps = <8>;
                xlnx,pix-per-clk = <2>;
                reset-gpios = <&gpio 87 1>;
                xlnx,max-width = <3840>;
                xlnx,max-height = <2160>;

                ports {
                        #address-cells = <1>;
                        #size-cells = <0>;

                        port@0 {
                                reg = <0>;

                                xlnx,video-format = <XVIP_VF_YUV_422>;
                                xlnx,video-width = <8>;

                                vcap_sdirx_in: endpoint {
                                        remote-endpoint = <&sdirx_out>;
                                };
```

```
                };
                port@1 {
                        reg = <1>;

                        xlnx,video-format = <XVIP_VF_YUV_422>;
                        xlnx,video-width = <8>;

                        scaler_out: endpoint {
                                remote-endpoint = <&vcap_in>;
                        };
                };
        };
};




video_cap {
        compatible = "xlnx,video";
        dmas = <&vdma_1 1>;
        dma-names = "port0";

        ports {
                #address-cells = <1>;
                #size-cells = <0>;

                port@0 {
                        reg = <0>;
                        direction = "input";
                        vcap_in: endpoint {
                                remote-endpoint = <&scaler_out>;
                        };
                };


};
```

Step 3: media-ctrl output

```
# xmedia-ctl -d /dev/media0 -p
Media controller API version 4.14.0

Media device information
------------------------
driver     xilinx-video
model       Xilinx Video Composite Device
serial
bus info
hw revision     0x0
driver version  4.14.0

Device topology


- entity 1: vcap_sdi output 0 (1 pad, 1 link)
          type Node subtype V4L flags 0
          device node name /dev/video0
     pad0: Sink
               <- "a0080000.v_proc_ss":1 [ENABLED]

- entity 5: a0030000.v_smpte_uhdsdi_rx_ss (1 pad, 1 link)
          type V4L2 subdev subtype Unknown flags 0
          device node name /dev/v4l-subdev0
     pad0: Source
               -> "a0080000.v_proc_ss":0 [ENABLED]



- entity 7: a0080000.v_proc_ss (2 pads, 2 links)
          type V4L2 subdev subtype Unknown flags 0
          device node name /dev/v4l-subdev1
     pad0: Sink
               [fmt:RBG888_1X24/1280x720 field:none colorspace:srgb]
               <- "a0030000.v_smpte_uhdsdi_rx_ss":0 [ENABLED]
     pad1: Source
               [fmt:VYYUYY8_1X24/1920x1080 field:none colorspace:srgb]
               -> "vcap_sdi output 0":0 [ENABLED]
```

Note:

Make sure SDI-Rx media pipeline is configured for 4kp60 resolution and source/sink have the same colour format. Run below xmedia-ctl commands to set resolution and format of SDI scaler node

```
$ xmedia-ctl -d /dev/media0 -V ' "a0080000.v_proc_ss":0
[fmt:UYVY8_1X16/3840x2160 field:none]'
```

```
$ xmedia-ctl -d /dev/media0 -V ' "a0080000.v_proc_ss":1
[fmt:VYYUYY8_1X24/3840x2160 field:none]'
```

Step 2: Asynchronous Case Sub-device

Step 1: Asynchronous Sub-device

In the asynchronous case sub-device probing can be invoked independently of the bridge driver availability. The sub-device driver then has to verify whether all the requirements for a successful probing are satisfied. This can include a check for a master clock availability. If any of the conditions aren't satisfied the driver might decide to return -EPROBE_DEFER to request further re-probing attempts. Once all conditions are met the sub-device shall be registered using the **v4l2_async_register_subdev()** function. Unregistration is performed using the **v4l2_async_unregister_subdev()** call. Sub-devices registered this way are stored in a global list of sub-devices, ready to be picked up by bridge drivers.

Step 2: Bridge Drivers

Bridge drivers in turn have to register a notifier object with an array of sub-device descriptors that the bridge device needs for its operation. This is performed using the **v4l2_async_notifier_register()** call. To unregister the notifier the driver has to call v4l2_async_notifier_unregister(). The former of the two functions takes two arguments: a pointer to struct v4l2_device and

a pointer to **struct v4l2_async_notifier**. The latter contains a pointer to an array of pointers to sub-device descriptors of type **struct v4l2_async_subdev type**. The V4L2 core will then use these descriptors to match asynchronously registered sub-devices to them. If a match is detected the **.bound() notifier callback is called**. After all sub-devices have been located the **.complete() callback is called.** When a sub-device is removed from the system the **.unbind() method is called**. All three callbacks are optional.

Step 3: Host Device

Step 1: v4l2_async_subdev

struct v4l2_async_subdev : sub-device descriptor, as known to a bridge

```
struct v4l2_async_subdev {
  enum v4l2_async_match_type match_type;

  union {
    struct fwnode_handle *fwnode;
    const char *device_name;
        struct {
                int adapter_id;
                unsigned short address;
        } i2c;

        struct {
                bool (*match)(struct device *dev,  struct v4l2_async_subdev *sd);
                void *priv;
        } custom;
```

} **match;**

struct list_head list;

struct list_head asd_list;

};

Members	**Description**
`match_type`	type of match that will be used
`match`	union of per-bus type matching data sets
`match.fwnode`	pointer to `struct fwnode_handle` to be matched. Used if **match_type** is `V4L2_ASYNC_MATCH_FWNODE`.
`match.device_name`	string containing the device name to be matched. Used if **match_type** is `V4L2_ASYNC_MATCH_DEVNAME`.
`match.i2c`	embedded struct with I2C parameters to be matched. Both **match.i2c.adapter_id** and **match.i2c.address** should be matched. Used if **match_type** is `V4L2_ASYNC_MATCH_I2C`.
`match.i2c.adapter_id`	I2C adapter ID to be matched. Used if **match_type** is `V4L2_ASYNC_MATCH_I2C`.
`match.i2c.address`	I2C address to be matched. Used if **match_type** is `V4L2_ASYNC_MATCH_I2C`.
`match.custom`	Driver-specific match criteria. Used if **match_type** is `V4L2_ASYNC_MATCH_CUSTOM`.
`match.custom.match`	Driver-specific match function to be used if `V4L2_ASYNC_MATCH_CUSTOM`.
`match.custom.priv`	Driver-specific private struct with match parameters to be used if `V4L2_ASYNC_MATCH_CUSTOM`.

`list`	used to link struct v4l2_async_subdev objects, waiting to be probed, to a notifier->waiting list
`asd_list`	used to add struct v4l2_async_subdev objects to the master notifier **asd_list**

Step 2: `v4l2_async_match_type`

enum `v4l2_async_match_type` : type of asynchronous sub device logic to be used in order to identify a match.

This enum is used by the asynchronous sub-device logic to define the algorithm that will be used to match an asynchronous device.

Constants

`V4L2_ASYNC_MATCH_CUSTOM`	Match will use the logic provided by **`struct v4l2_async_subdev`.match** ops
`V4L2_ASYNC_MATCH_DEVNAME`	Match will use the device name
`V4L2_ASYNC_MATCH_I2C`	Match will check for I2C adapter ID and address
`V4L2_ASYNC_MATCH_FWNODE`	**Match will use firmware node**

Step 3: `v4l2_async_notifier_operations`

struct `v4l2_async_notifier_operations` : Asynchronous V4L2 notifier operations

struct v4l2_async_notifier_operations {

int (***bound**)(struct v4l2_async_notifier *notifier, struct v4l2_subdev

*subdev, struct v4l2_async_subdev *asd);

int (***complete**)(struct v4l2_async_notifier *notifier);

void (***unbind**)(struct v4l2_async_notifier *notifier, struct v4l2_subdev *subdev, struct v4l2_async_subdev *asd);

};

Members	Description
bound	a sub device driver has successfully probed one of the sub devices found
complete	All sub devices have been probed successfully. The complete callback is only executed for the root notifier.
unbind	a sub device is leaving

Step 4: v4l2_async_notifier

struct v4l2_async_notifier : v4l2_device notifier data

struct v4l2_async_notifier {

const struct v4l2_async_notifier_operations *ops;

struct v4l2_device *v4l2_dev;

struct v4l2_subdev *sd;

struct v4l2_async_notifier *parent;

struct list_head asd_list;

struct list_head waiting;

struct list_head done;

struct list_head list;

};

Members	Description
ops	notifier operations
v4l2_dev	v4l2_device of the root notifier, NULL otherwise
sd	sub-device that registered the notifier, NULL otherwise
parent	parent notifier
asd_list	master list of struct v4l2_async_subdev
waiting	list of struct v4l2_async_subdev, waiting for their drivers
done	list of struct v4l2_subdev, already probed
list	member in a global list of notifiers

Step 5: v4l2_async_notifier_init

void **v4l2_async_notifier_init**(struct v4l2_async_notifier **notifier*)

Initialize a notifier.

Parameters : `struct v4l2_async_notifier * notifier`

pointer to **`struct v4l2_async_notifier`**

Step 6: v4l2_async_notifier_register

int `v4l2_async_notifier_register`**(struct v4l2_device ****v4l2_dev*****, struct**

v4l2_async_notifier *_notifier_) : registers a sub device asynchronous notifier
Parameters

struct v4l2_device * v4l2_dev : pointer to `struct v4l2_device`
struct v4l2_async_notifier `* notifier` : pointer to `struct v4l2_async_notifier`

int v4l2_async_notifier_register(struct v4l2_device *v4l2_dev,
struct v4l2_async_notifier *notifier)

```
{
	int ret;

	if (WARN_ON(!v4l2_dev || notifier->sd))
		return -EINVAL;

	notifier->v4l2_dev = v4l2_dev;

	ret = __v4l2_async_notifier_register(notifier);
	if (ret)
		notifier->v4l2_dev = NULL;

	return ret;
}
```

EXPORT_SYMBOL(v4l2_async_notifier_register);

static int __v4l2_async_notifier_register(struct v4l2_async_notifier *notifier)

```
{
	struct device *dev =
		notifier->v4l2_dev ? notifier->v4l2_dev->dev : NULL;
	struct v4l2_async_subdev *asd;
	int ret;
	int i;

	if (notifier->num_subdevs > V4L2_MAX_SUBDEVS)
		return -EINVAL;

	INIT_LIST_HEAD(&notifier->waiting);
	INIT_LIST_HEAD(&notifier->done);
```

```
	mutex_lock(&list_lock);

	for (i = 0; i < notifier->num_subdevs; i++) {
		asd = notifier->subdevs[i];

		switch (asd->match_type) {
		case V4L2_ASYNC_MATCH_CUSTOM:
		case V4L2_ASYNC_MATCH_DEVNAME:
		case V4L2_ASYNC_MATCH_DEVNAME:
		case V4L2_ASYNC_MATCH_I2C:
			break;
		case V4L2_ASYNC_MATCH_FWNODE:
			if (v4l2_async_notifier_fwnode_has_async_subdev(
				    notifier, asd->match.fwnode, i)) {
				dev_err(dev,
					"fwnode has already been registered or in notifier's
subdev list\n");
				ret = -EEXIST;
				goto err_unlock;
			}
			break;
		default:
			dev_err(dev, "Invalid match type %u on %p\n",
				asd->match_type, asd);
			ret = -EINVAL;
			goto err_unlock;
		}
		list_add_tail(&asd->list, &notifier->waiting);
	}

	ret = v4l2_async_notifier_try_all_subdevs(notifier);
	if (ret < 0)
		goto err_unbind;

	ret = v4l2_async_notifier_try_complete(notifier);
	if (ret < 0)
		goto err_unbind;

	/* Keep also completed notifiers on the list */
	list_add(&notifier->list, &notifier_list);

	mutex_unlock(&list_lock);

	return 0;
}
```

Step 7: xvip_graph_init : xilinix -vipp.c

drivers/media/platform/xilinx/xilinx-vipp.c

```
/* Parse the graph to extract a list of subdevice DT nodes. */
  ret = xvip_graph_parse(xdev);
  if (ret < 0) {
        dev_err(xdev->dev, "graph parsing failed\n");
        goto done;
  }

  if (!xdev->num_subdevs) {
        dev_err(xdev->dev, "no subdev found in graph\n");
        goto done;
  }

  /* Register the subdevices notifier. */
  num_subdevs = xdev->num_subdevs;
  subdevs = devm_kcalloc(xdev->dev, num_subdevs, sizeof(*subdevs),
                         GFP_KERNEL);
  if (subdevs == NULL) {
        ret = -ENOMEM;
        goto done;
  }

  i = 0;
  list_for_each_entry(entity, &xdev->entities, list)
        subdevs[i++] = &entity->asd;
```

```
	xdev->notifier.subdevs = subdevs;
	xdev->notifier.num_subdevs = num_subdevs;
	xdev->notifier.ops = &xvip_graph_notify_ops;

	ret = v4l2_async_notifier_register(&xdev->v4l2_dev, &xdev->notifier);
	if (ret < 0) {
		dev_err(xdev->dev, "notifier registration failed\n");
		goto done;
	}
```

Step 8: xvip_graph_parse

```
static int xvip_graph_parse(struct xvip_composite_device *xdev)
{
	struct xvip_graph_entity *entity;
	int ret;

	/*
	 * Walk the links to parse the full graph. Start by parsing the
	 * composite node and then parse entities in turn. The list_for_each
	 * loop will handle entities added at the end of the list while walking
	 * the links.
	 */
	ret = xvip_graph_parse_one(xdev, xdev->dev->of_node);
	if (ret < 0)
		return 0;

	list_for_each_entry(entity, &xdev->entities, list) {
		ret = xvip_graph_parse_one(xdev, entity->node);
		if (ret < 0)
			break;
	}

	return ret;
```

```
}
```

Step 9: xvip_graph_parse_one

```
static int xvip_graph_parse_one(struct xvip_composite_device *xdev,
                                struct device_node *node)
{
        struct xvip_graph_entity *entity;
        struct device_node *remote;
        struct device_node *ep = NULL;
        int ret = 0;

        dev_dbg(xdev->dev, "parsing node %pOF\n", node);

        while (1) {
                ep = of_graph_get_next_endpoint(node, ep);
                if (ep == NULL)
                        break;

                dev_dbg(xdev->dev, "handling endpoint %pOF\n", ep);

                remote = of_graph_get_remote_port_parent(ep);
                if (remote == NULL) {
                        ret = -EINVAL;
                        break;
                }

                /* Skip entities that we have already processed. */
                if (remote == xdev->dev->of_node ||
                    xvip_graph_find_entity(xdev, remote)) {
                        of_node_put(remote);
                        continue;
                }

                entity = devm_kzalloc(xdev->dev, sizeof(*entity), GFP_KERNEL);
                if (entity == NULL) {
                        of_node_put(remote);
                        ret = -ENOMEM;
                        break;

                  /* Skip entities that we have already processed. */
                if (remote == xdev->dev->of_node ||
                    xvip_graph_find_entity(xdev, remote)) {
```

```
                of_node_put(remote);
                continue;
        }

        entity = devm_kzalloc(xdev->dev, sizeof(*entity), GFP_KERNEL);
        if (entity == NULL) {
                of_node_put(remote);
                ret = -ENOMEM;
                break;
        }

         entity->node = remote;
         entity->asd.match_type = V4L2_ASYNC_MATCH_FWNODE;
         entity->asd.match.fwnode = of_fwnode_handle(remote);
         list_add_tail(&entity->list, &xdev->entities);
         xdev->num_subdevs++;
    }

    of_node_put(ep);
    return ret;
}
```

Step 9: v4l2_async_notifier_operations xvip_graph_notify_ops

static const struct v4l2_async_notifier_operations xvip_graph_notify_ops = {

```
    .bound = xvip_graph_notify_bound,

    .complete = xvip_graph_notify_complete,

};
```

static int xvip_graph_notify_bound(struct v4l2_async_notifier *notifier,
struct v4l2_subdev *subdev,
struct v4l2_async_subdev *asd)

```
{
    struct xvip_composite_device *xdev =
            container_of(notifier, struct xvip_composite_device, notifier);
    struct xvip_graph_entity *entity;
```

```
	/* Locate the entity corresponding to the bound subdev and store the
	 * subdev pointer.
	 */
	list_for_each_entry(entity, &xdev->entities, list) {
		if (of_fwnode_handle(entity->node) != subdev->fwnode)
			continue;

		if (entity->subdev) {
			dev_err(xdev->dev, "duplicate subdev for node %pOF\n",
				entity->node);
			return -EINVAL;
		}

		dev_dbg(xdev->dev, "subdev %s bound\n", subdev->name);
		entity->entity = &subdev->entity;
		entity->subdev = subdev;
		return 0;
	}

	dev_err(xdev->dev, "no entity for subdev %s\n", subdev->name);
	return -EINVAL;
}



static int xvip_graph_notify_complete(struct v4l2_async_notifier *notifier)
{
	struct xvip_composite_device *xdev =
		container_of(notifier, struct xvip_composite_device, notifier);
	struct xvip_graph_entity *entity;
	int ret;

	dev_dbg(xdev->dev, "notify complete, all subdevs registered\n");

	/* Create links for every entity. */
	list_for_each_entry(entity, &xdev->entities, list) {
		ret = xvip_graph_build_one(xdev, entity);
		if (ret < 0)
			return ret;
	}

	/* Create links for DMA channels. */
	ret = xvip_graph_build_dma(xdev);
	if (ret < 0)
		return ret;

	ret = v4l2_device_register_subdev_nodes(&xdev->v4l2_dev);
	if (ret < 0)
```

```
            dev_err(xdev->dev, "failed to register subdev nodes\n");

    return media_device_register(&xdev->media_dev);
}
```

Step 10:xvip_graph_build_one

```
static int xvip_graph_build_one(struct xvip_composite_device *xdev,
                                 struct xvip_graph_entity *entity)
{
    u32 link_flags = MEDIA_LNK_FL_ENABLED;
    struct media_entity *local = entity->entity;
    struct media_entity *remote;
    struct media_pad *local_pad;
    struct media_pad *remote_pad;
    struct xvip_graph_entity *ent;
    struct v4l2_fwnode_link link;
    struct device_node *ep = NULL;
    int ret = 0;

    dev_dbg(xdev->dev, "creating links for entity %s\n", local->name);

    while (1) {
        /* Get the next endpoint and parse its link. */
        ep = of_graph_get_next_endpoint(entity->node, ep);
        if (ep == NULL)
            break;

        dev_dbg(xdev->dev, "processing endpoint %pOF\n", ep);

        ret = v4l2_fwnode_parse_link(of_fwnode_handle(ep), &link);
        if (ret < 0) {
            dev_err(xdev->dev, "failed to parse link for %pOF\n",
                ep);
            continue;
        }

        /* Skip sink ports, they will be processed from the other end of
         * the link.
         */
        if (link.local_port >= local->num_pads) {
            dev_err(xdev->dev, "invalid port number %u for %pOF\n",
                link.local_port,
                to_of_node(link.local_node));
            v4l2_fwnode_put_link(&link);
```

```
                ret = -EINVAL;
                break;
          local_pad = &local->pads[link.local_port];

        if (local_pad->flags & MEDIA_PAD_FL_SINK) {
                dev_dbg(xdev->dev, "skipping sink port %pOF:%u\n",
                        to_of_node(link.local_node),
                        link.local_port);
                v4l2_fwnode_put_link(&link);
                continue;
        }

        /* Skip DMA engines, they will be processed separately. */
        if (link.remote_node == of_fwnode_handle(xdev->dev->of_node)) {
                dev_dbg(xdev->dev, "skipping DMA port %pOF:%u\n",
                        to_of_node(link.local_node),
                        link.local_port);
                v4l2_fwnode_put_link(&link);
                continue;
        }

        /* Find the remote entity. */
        ent = xvip_graph_find_entity(xdev,
                                     to_of_node(link.remote_node));
        if (ent == NULL) {
                dev_err(xdev->dev, "no entity found for %pOF\n",
                        to_of_node(link.remote_node));
                v4l2_fwnode_put_link(&link);
                ret = -ENODEV;
                break;
        }

        remote = ent->entity;

        if (link.remote_port >= remote->num_pads) {
                dev_err(xdev->dev, "invalid port number %u on %pOF\n",
link.remote_port, to_of_node(link.remote_node));
                v4l2_fwnode_put_link(&link);
                ret = -EINVAL;
                break;
        }

        remote_pad = &remote->pads[link.remote_port];

        v4l2_fwnode_put_link(&link);

        /* Create the media link. */
        dev_dbg(xdev->dev, "creating %s:%u -> %s:%u link\n",
```

```
                local->name, local_pad->index,
                remote->name, remote_pad->index);

        ret = media_create_pad_link(local, local_pad->index,
                                    remote, remote_pad->index,
                                    link_flags);
        if (ret < 0) {
                dev_err(xdev->dev,
                        "failed to create %s:%u -> %s:%u link\n",
                        local->name, local_pad->index,
                        remote->name, remote_pad->index);
                break;
        }
    }

    return ret;
}
```

Step 11: v4l2_device_register_subdev_nodes

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;
#if defined(CONFIG_MEDIA_CONTROLLER)
            sd->entity.info.dev.major = VIDEO_MAJOR;
            sd->entity.info.dev.minor = vdev->minor;

            /* Interface is created by __video_register_device() */
            if (vdev->v4l2_dev->mdev) {
                  struct media_link *link;

                  link = media_create_intf_link(&sd->entity,
                                                &vdev->intf_devnode->intf,
                                                MEDIA_LNK_FL_ENABLED |
                                                MEDIA_LNK_FL_IMMUTABLE);
                  if (!link) {
                        err = -ENOMEM;
                        goto clean_up;
                  }
            }
#endif
      }
      return 0;

clean_up:
      list_for_each_entry(sd, &v4l2_dev->subdevs, list) {
            if (!sd->devnode)
                  break;
            video_unregister_device(sd->devnode);
      }

      return err;
}
EXPORT_SYMBOL_GPL(v4l2_device_register_subdev_nodes);
```

Step 12: media_device_register

Step 4: Sub Device

drivers/media/platform/xilinx/xilinx-sdirxss.c

Step 1: Device Tree Parse

static int xsdirxss_parse_of(struct xsdirxss_state *xsdirxss)

```
    for_each_child_of_node(ports, port) {
        const struct xvip_video_format *format;
        struct device_node *endpoint;

        if (!port->name || of_node_cmp(port->name, "port"))
            continue;

        format = xvip_of_get_format(port);
        if (IS_ERR(format)) {
            dev_err(core->dev, "invalid format in DT");
            return PTR_ERR(format);
        }

        dev_dbg(core->dev, "vf_code = %d bpc = %d bpp = %d\n",
            format->vf_code, format->width, format->bpp);

        if (format->vf_code != XVIP_VF_YUV_422 &&
           format->vf_code != XVIP_VF_YUV_420 && format->width != 10) {
            dev_err(core->dev,
                "Incorrect UG934 video format set.\n");
            return -EINVAL;
        }
        xsdirxss->vip_format = format;

         endpoint = of_get_next_child(port, NULL);
         if (!endpoint) {
             dev_err(core->dev, "No port at\n");
             return -EINVAL;
         }

        /* Count the number of ports. */
        nports++;
    }

    if (nports != 1) {
        dev_err(core->dev, "invalid number of ports %u\n", nports);
```

```
        return -EINVAL;
    }
     ret = v4l2_async_register_subdev(subdev);
```

Step 2: v4l2_async_register_subdev

1	**v4l2_async_notifier_find_v4l2_dev ==> (return) v4l2_device**
2	**v4l2_async_find_match ==>(return) vl42_async_subdev**
3	**v4l2_async_match_notify =>(do) media_device_register_entity**
4	v**4l2_async_notifier_call_bound** ==>(call) Host Notifier 'bound' function => v4l2_device_register_subdev_nodes (**media_create_intf_link**)
5	**v4l2_async_notifier_try_complete ==>v4l2_async_notifier_can_complete (check, if every thing ready)**
6	**v4l2_async_notifier_call_complete ==> Host Notifier 'complete' function**

```
int v4l2_async_register_subdev(struct v4l2_subdev *sd)
{
        struct v4l2_async_notifier *subdev_notifier;
        struct v4l2_async_notifier *notifier;
        int ret;

        /*
         * No reference taken. The reference is held by the device
         * (struct v4l2_subdev.dev), and async sub-device does not
         * exist independently of the device at any point of time.
         */
        if (!sd->fwnode && sd->dev)
                sd->fwnode = dev_fwnode(sd->dev);

        mutex_lock(&list_lock);

        INIT_LIST_HEAD(&sd->async_list);
```

```
            list_for_each_entry(notifier, &notifier_list, list) {
            struct v4l2_device *v4l2_dev =
                    v4l2_async_notifier_find_v4l2_dev(notifier);
            struct v4l2_async_subdev *asd;

            if (!v4l2_dev)
                    continue;

            asd = v4l2_async_find_match(notifier, sd);
            if (!asd)
                    continue;

            ret = v4l2_async_match_notify(notifier, v4l2_dev, sd, asd);
            if (ret)
                    goto err_unbind;

            ret = v4l2_async_notifier_try_complete(notifier);
            if (ret)
                    goto err_unbind;

            goto out_unlock;
    }

    /* None matched, wait for hot-plugging */
    list_add(&sd->async_list, &subdev_list);

out_unlock:
    mutex_unlock(&list_lock);

    return 0;
}
```

Step 3: v4l2_async_notifier_find_v4l2_dev

```
/* Get v4l2_device related to the notifier if one can be found. */
static struct v4l2_device *v4l2_async_notifier_find_v4l2_dev(
        struct v4l2_async_notifier *notifier)
{
    while (notifier->parent)
            notifier = notifier->parent;

    return notifier->v4l2_dev;
}
```

Step 4: v4l2_async_find_match

```
static struct v4l2_async_subdev *v4l2_async_find_match(
        struct v4l2_async_notifier *notifier, struct v4l2_subdev *sd)
{
        bool (*match)(struct v4l2_subdev *, struct v4l2_async_subdev *);
        struct v4l2_async_subdev *asd;

        list_for_each_entry(asd, &notifier->waiting, list) {
                /* bus_type has been verified valid before */
                switch (asd->match_type) {
                case V4L2_ASYNC_MATCH_CUSTOM:
                        match = match_custom;
                        break;
                case V4L2_ASYNC_MATCH_DEVNAME:
                        match = match_devname;
                        break;
                case V4L2_ASYNC_MATCH_I2C:
                        match = match_i2c;
                        break;
                case V4L2_ASYNC_MATCH_FWNODE:
                          match = match_fwnode;
                          break;
                default:
                        /* Cannot happen, unless someone breaks us */
                        WARN_ON(true);
                        return NULL;
                }

                /* match cannot be NULL here */
                 if (match(sd, asd))
                        return asd;
        }

        return NULL;
}


static bool match_fwnode(struct v4l2_subdev *sd, struct v4l2_async_subdev *asd)

{

        return sd->fwnode == asd->match.fwnode;
```

```
}
```

Step 5: v4l2_async_match_notify

```
static int v4l2_async_match_notify(struct v4l2_async_notifier *notifier,
                                   struct v4l2_device *v4l2_dev,
                                   struct v4l2_subdev *sd,
                                   struct v4l2_async_subdev *asd)
{
        struct v4l2_async_notifier *subdev_notifier;
        int ret;

        ret = v4l2_device_register_subdev(v4l2_dev, sd);
        if (ret < 0)
                return ret;

        ret = v4l2_async_notifier_call_bound(notifier, sd, asd);
        if (ret < 0) {
                v4l2_device_unregister_subdev(sd);
                return ret;
        }

        /* Remove from the waiting list */
        list_del(&asd->list);
        sd->asd = asd;
        sd->notifier = notifier;

       /* Move from the global subdevice list to notifier's done */
        list_move(&sd->async_list, &notifier->done);

        /*
         * See if the sub-device has a notifier. If not, return here.
         */
        subdev_notifier = v4l2_async_find_subdev_notifier(sd);
        if (!subdev_notifier || subdev_notifier->parent)
                return 0;

        /*
         * Proceed with checking for the sub-device notifier's async
         * sub-devices, and return the result. The error will be handled by the
         * caller.
         */
        subdev_notifier->parent = notifier;
```

```
	return v4l2_async_notifier_try_all_subdevs(subdev_notifier);
}
```

Step 6: v4l2_async_notifier_try_complete

```
static int v4l2_async_notifier_try_complete(
	 struct v4l2_async_notifier *notifier)
{
	/* Quick check whether there are still more sub-devices here. */
	if (!list_empty(&notifier->waiting))
		return 0;

	/* Check the entire notifier tree; find the root notifier first. */
	while (notifier->parent)
		notifier = notifier->parent;

	/* This is root if it has v4l2_dev. */
	if (!notifier->v4l2_dev)
		return 0;

	/* Is everything ready? */
	 if (!v4l2_async_notifier_can_complete(notifier))
		return 0;

	return v4l2_async_notifier_call_complete(notifier);
}
```