JETSON VIRTUAL CHANNEL WITH GMSL CAMERA FRAMEWORK GUIDE

DA_09421-001 | June 12, 2020
Advance Information | Subject to Change
NVIDIA

Application Note

DOCUMENT CHANGE HISTORY

Version	Date	Authors	Description of Change
v0.1	March 5, 2019	svyas sgollapudi	Initial release
v0.2	April 10, 2019	svyas	Update release
v1.0	June 21, 2019	sgollapudi	Update release

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

DESCRIPTION

This document provides the details on:

- The Gigabit Multimedia Serial Link (GMSL) protocol
- Hardware connectivity for the serializer/deserializer in the reference module (see Platforms)
- The software framework
- Configuration, including virtual channel programming

NVIDIA validates the reference module on NVIDIA® Jetson™ TX2 and NVIDIA® Jetson AGX Xavier™ platforms with Sony IMX390 sensors as source.

However, it can be used as reference guide for any other GMSL module bring up on Jetson TX2 and Jetson AGX Xavier.

In addition, the software framework described in this document can also work as reference for other SerDes (Serializer-Deserializer) links apart from GMSL.

Note:	**The reference GMSL module described here, uses CSI interface. No other interface is validated.**

PLATFORMS

GMSL and other aggregators are not supported on NVIDIA® Jetson Nano™ and NVIDIA® Jetson™ TX1 platforms, as they do not support virtual channels. This document applies to Jetson TX2 and Jetson AGX Xavier platforms only.

The reference GMSL setup is:

- Sensor: Sony's dual IMX390, RAW12/1080p/30fps, CSI port A, x2 lanes
- Serializer: Maxim's MAX9295
- Deserializer: Maxim's MAX9296

GMSL PROTOCOL

Maxim Integrated released GMSL as a communication link for video applications in the automotive industry. GMSL is based on SerDes (Serializer-Deserializer) technology, which means that it uses a serializer on the transmitting side and a deserializer on the receiving side. It is specifically designed for use in Advanced Driver Assistance Systems (ADAS) and Camera Monitoring Systems (CMS). It can provide video transfer speeds up to 6 GB/second. It uses STP or coaxial cables, which are both inexpensive and very robust for EMC disturbances.

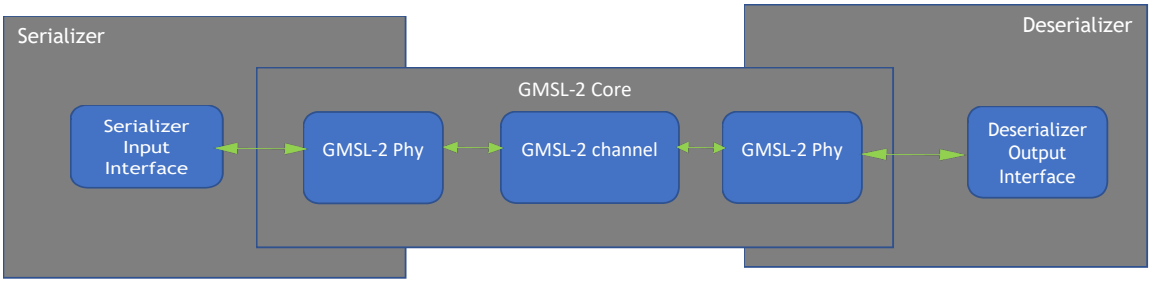

Figure 1: GMSL protocol

GMSL CAMERA

The diagram in the following Figure 2 shows the control, data, and clock connections for the reference GMSL module validated on Jetson TX2 and Jetson AGX Xavier.

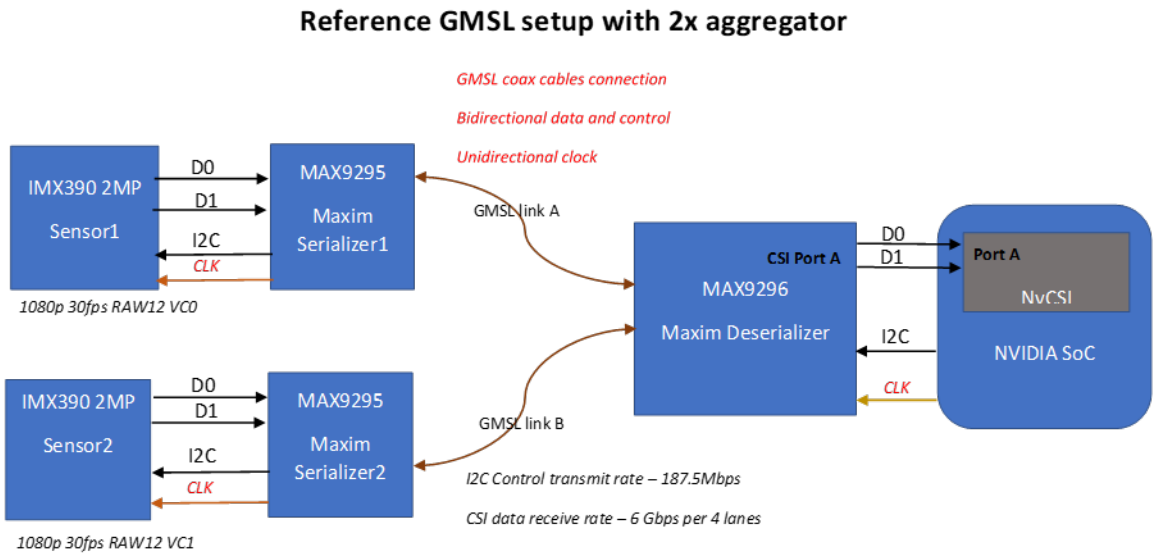

Figure 2: Reference GMSL setup with 2x aggregator

- In this GMSL setup, two sensors are paired with their respective serializers, each streaming 1080p/30fps RAW12 pixels over x2 CSI MIPI lanes.
- The serializers are connected to a GMSL deserializer device through different GMLS ports (ports A and B) and GMSL links using coaxial cables.
- On the output port side, the deserializer is connected to a Jetson TX2 or Jetson AGX Xavier on the desired CSI port (CSI port A in this example).
- To transmit two different pixel streams from two sensors to a receiver at a shared CSI port, the deserializer assigns a unique virtual channel ID to each stream. The

virtual channel ID is software configurable via the device tree. It must match the stream's virtual channel ID programmed at receiver side in the NVIDIA SoC.

This reference GMSL setup uses a 2x (`x2/x4/x1`) CSI deserializer. It can host up to two sensors via serializers.

To host four sensors over single shared CSI port, you must use a 4x aggregator, as shown in the following Figure 3:

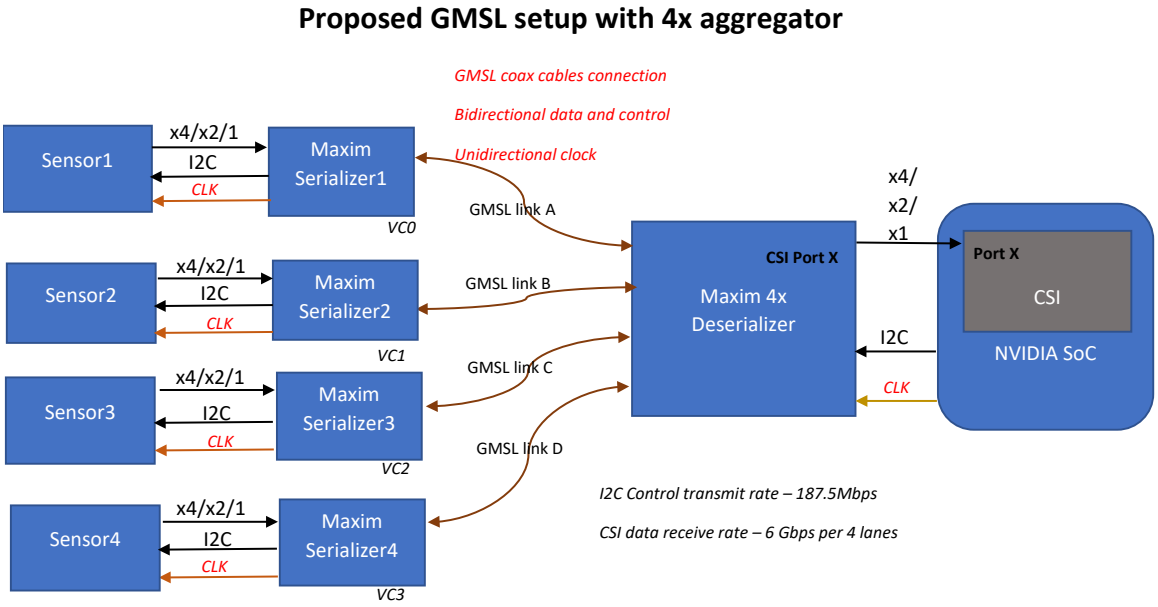

Figure 3: Proposed GMSL setup with 4x aggregator

CSI CONNECTIVITY

The table below shows the maximum sensor connections possible on each of the Jetson platforms.

Table 1: Maximum Possible Sensor Connections

	Jetson TX1	Jetson TX2	Jetson AGX Xavier
No aggregator	6	6	6
Aggregator + ISP	N/A	12	16
Aggregator w/o ISP	N/A	12	24

AGGREGATOR AND VIRTUAL CHANNELS

Jetson TX2

The maximum number of virtual channels supported on Jetson TX2 is 12 via three 4x aggregators connected to each of CSI bricks in `x4`, `x2`, or `x1` lane configuration (or any valid combination of them).

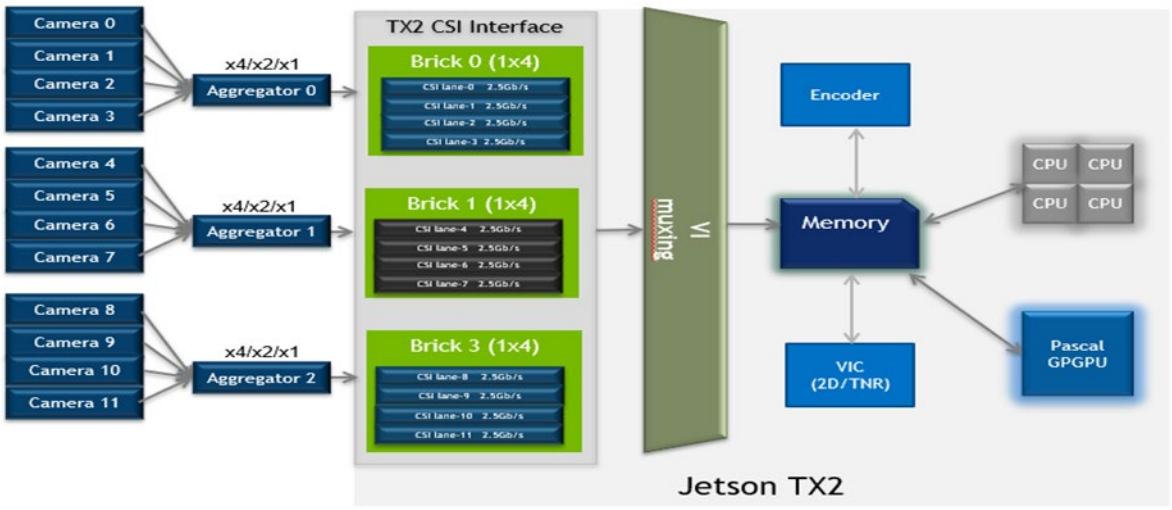

Figure 4: Jetson TX2: 12 cameras with virtual channels

Figure 4 shows Jetson TX2 connections for 12 cameras with virtual channels.

Jetson AGX Xavier

Jetson AGX Xavier supports a maximum of 16 virtual channels with ISP or 24 virtual channels without ISP.

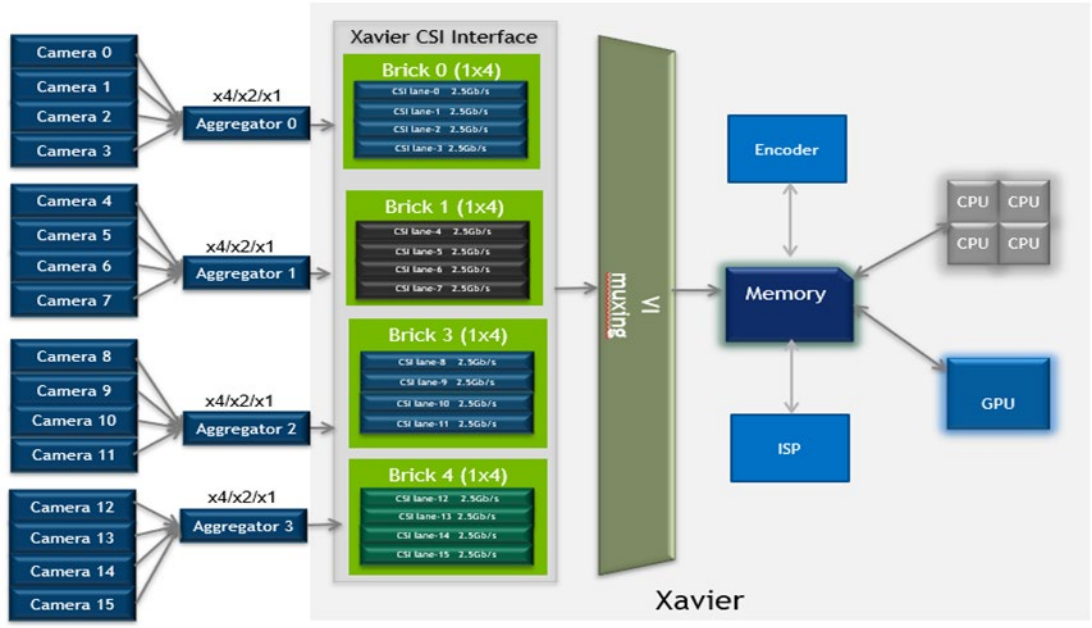

Figure 5: Jetson AGX Xavier 16 cameras with virtual channels

Figure 5 shows sensor connections to each of Xavier CSI bricks (total 4 CSI bricks in Xavier), in `x4` or /`x2` or /`x1` possible lane configuration.

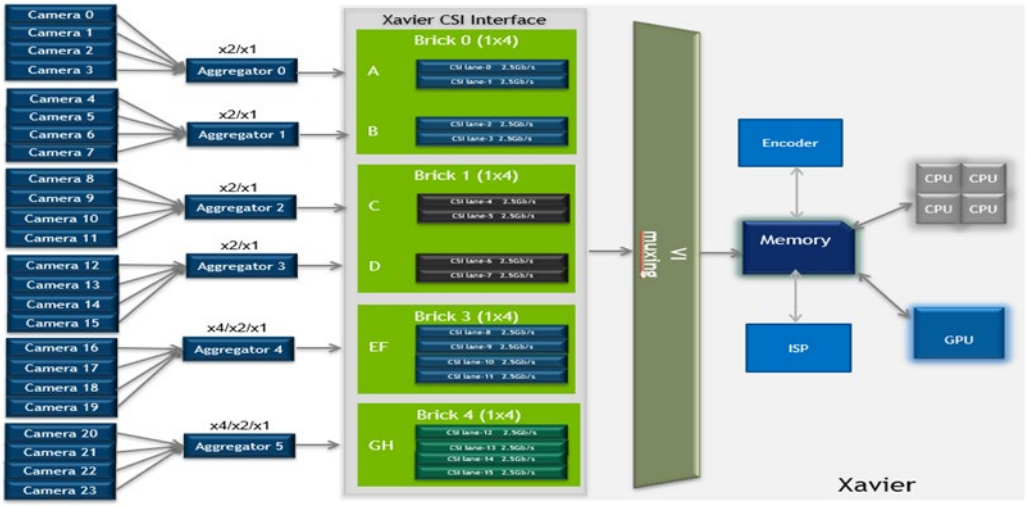

Figure 6: Jetson AGX Xavier: 24 cameras with virtual channels

Figure 6 shows 4 sensors connected to each port of CSI bricks AB and CD in `x2` or `x1` lane configuration and 4 sensors connected to each of remaining CSI bricks EF and GH, in `x4` or `x2` or `x1` lane configuration.

- The CSI aggregator uses Virtual Channels to connect to four cameras over one CSI connection.
- The Jetson TX2 or Jetson AGX Xavier VI sends each camera frame to a different location in memory.
- In software, each camera appears as a separate V4L2 device.

HARDWARE MODULE CONNECTIVITY

Figure 7 shows how two sensors are connected to Jetson AGX Xavier platform using a GMSL reference module and MAX9295/MAX9296 SerDes. A setup using Jetson TX2 is connected the same way.

The GMSL MAX9296 deserializer is connected to the Jetson platform via MIPI adaptor using MIPI white cables, which are plugged in to the platform's camera connector socket. This is one of several ways the aggregator hardware module can be connected to the Jetson platform.

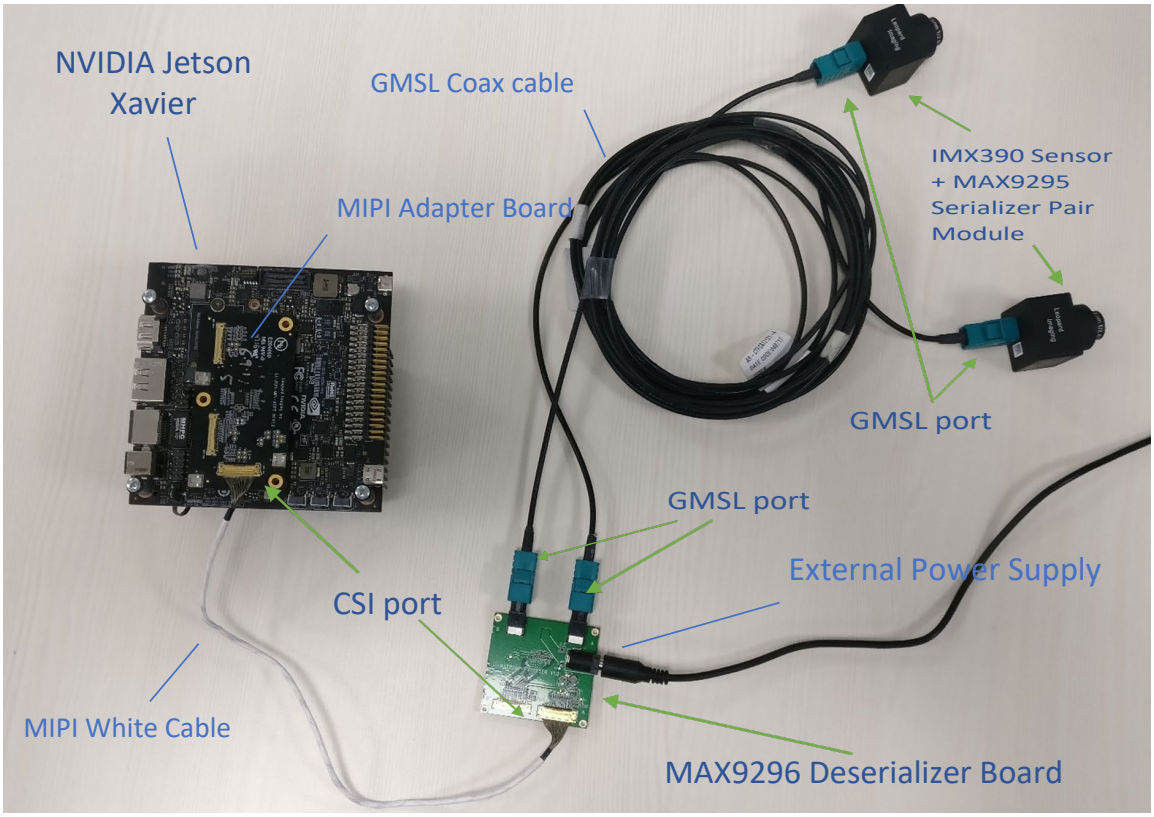

Figure 7: Reference GMSL hardware connectivity

SOFTWARE FRAMEWORK AND PROGRAMMING

This section describes the kernel drivers and device tree programming required for GMSL and virtual channel.

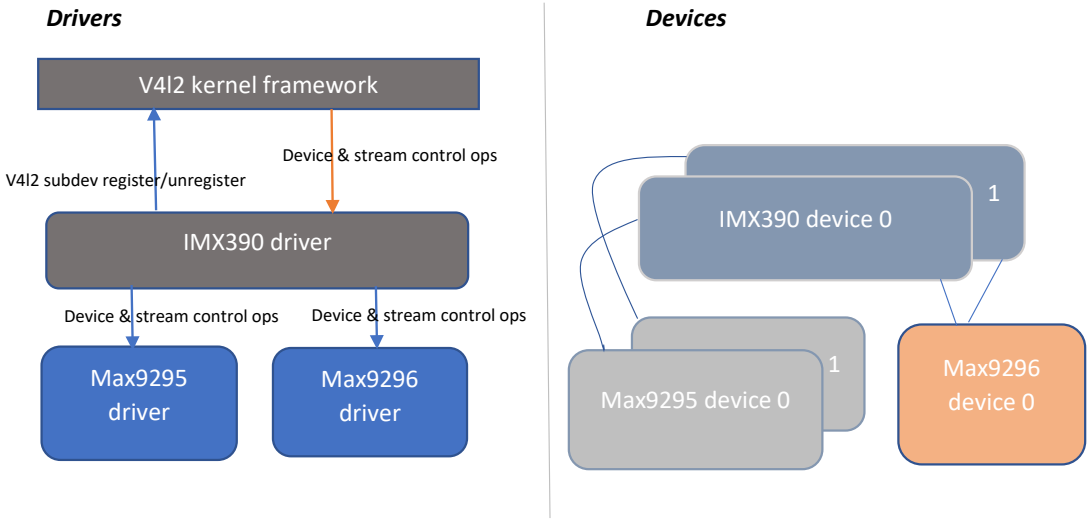

Figure 8: Top level view: kernel drivers and devices

DRIVER FRAMEWORK

As shown in Figure 8 above, there are separate kernel drivers for serializer and deserializer devices apart from the sensor driver. In the reference module, they are MAX9295 serializer and MAX9296 deserializer drivers.

In this framework, the sensor driver is exposed to the rest of the system the same as any other generic V4L2 sensor driver without external aggregator. All the SerDes programming happens “under the hood” through sensor driver.

The SerDes kernel drivers are not registered with as client programmable and isolated devices V4L2 or any other sensor framework.

The reason for this design is that a certain fixed sequence of operations must be performed in `SerDes`, and it does not quite fit the generic V4L2 framework sequence for sensors.

SerDes drivers are separate, though, and can be statically linked to any sensors in the device tree depending on hardware connectivity, but they are controlled by the sensor driver only. They do not expose any direct programmable functionality to user clients.

A sensor driver internally links to the SerDes drivers to perform device and stream control operations such as power on/off, control setup/release, stream setup/release, stream start/stop, and more, based on device tree configuration.

Driver Programming

The GMSL link structure below is the core entity which defines the entire link configuration from sensor to `serializer` to `deserializer` link for each sensor source.

```
struct gmsl_link_ctx {
    __u32 st_vc;           // default sensor virtual channel
    __u32 dst_vc;          // Destination virtual channel, user defined
    __u32 src_csi_port;    // Sensor to serializer CSI port connection
    __u32 dst_csi_port;    // Deserializer to Tegra CSI port connection
    __u32 serdes_csi_link; // GMSL link between Serializer and
Deserializer
                           //  device
    __u32 num_streams;     // Number of active streams from sensor to
be
                           //  mapped
    __u32 num_csi_lanes;   // Sensor’s CSI lane configuration
    __u32 csi_mode;        // Deserializer CSI mode
    __u32 ser_reg;         // Serializer slave address
    __u32 sdev_reg;        // Sensor proxy slave address
    __u32 sdev_def;        // Sensor default slave address
    struct gmsl_stream streams[GMSL_DEV_MAX_NUM_DATA_STREAMS];
                           // Array of active streams to be mapped
    struct device *s_dev;  // Sensor device handle
}
```

Most of the structure fields are populated by the sensor driver from sensor's device tree's gmsl-link node during device boot (see the next section for node details). Some of the fields are populated by the serializer and deserializer drivers.

The sensor driver populates this struct instance and passes it to the serializer and deserializer drivers, which make use of the configuration details found in the structure context during power-on, control pipeline setup, and data streaming pipeline setup calls.

As defined in this structure, the sensor and its corresponding serializer device have two different I2C slave addresses; one is the **physical** (default) assigned according to the device data sheet and the other, called the **proxy**, is user-defined.

The reason for this is that all the devices sharing the same hardware connection in the GMSL setup must be identified to the I2C bus with unique slave addresses. The physical slave address of each device is fixed and is the same for similar types of devices, such as all sensor devices of a single model and make that are assigned to same slave. This causes an address conflict. To resolve the conflict the driver needs proxy slave addresses for such devices. Proxy addresses are statically set in the device tree and are assigned during device boot.

Figure 9 shows the call flow between GMSL kernel drivers during device boot.

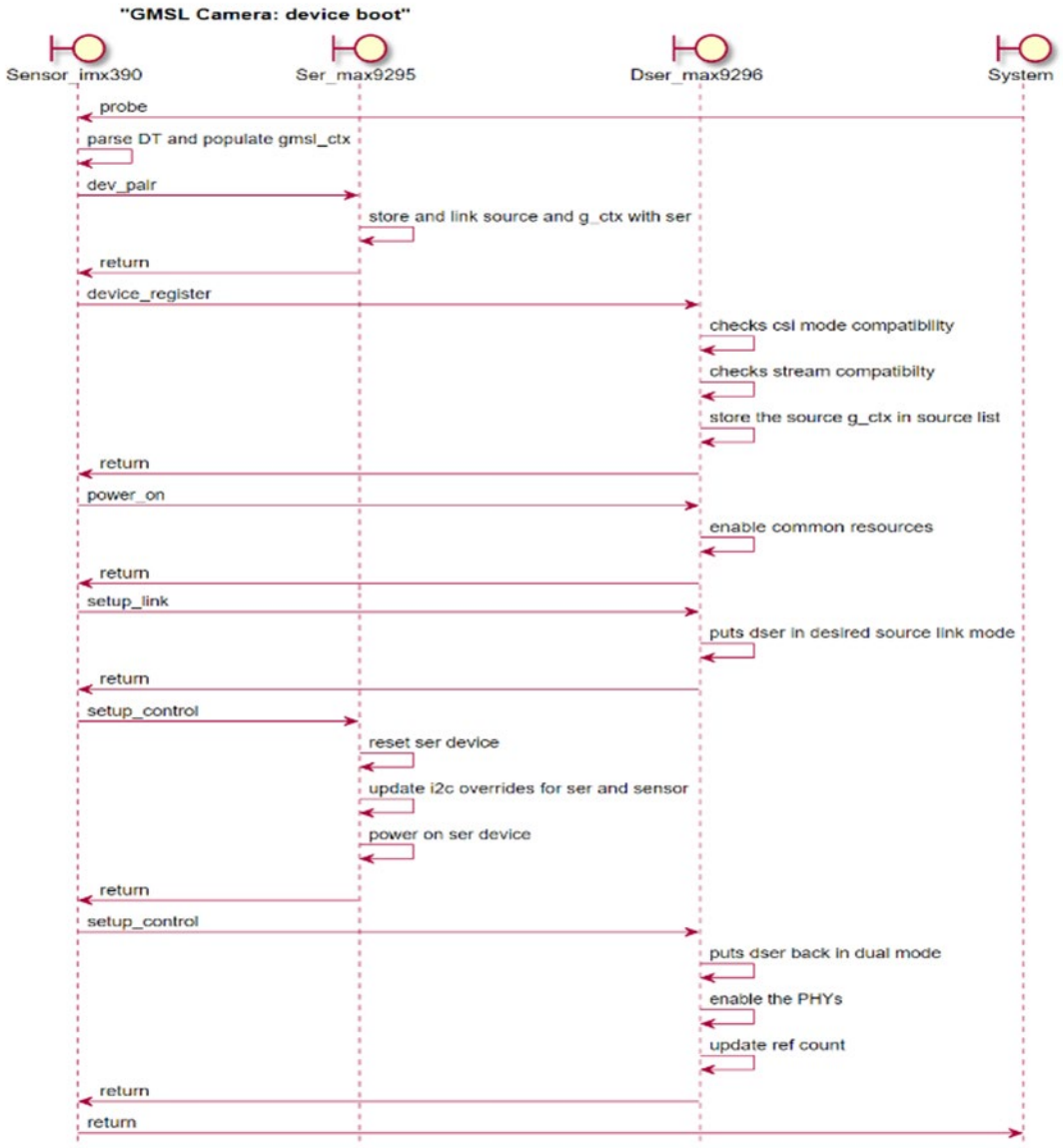

Figure 9: GMSL camera: device boot sequence

Power management is handled by the shared deserializer driver instead of the sensor driver because the deserializer device is common among all the sensors connected to it, and each sensor device is unaware of other sensor devices in the GMSL module setup.

The stream-on and stream-off calls are mapped directly to the serializer and deserializer drivers.

Stream setup is required for the SerDes drivers, whereas stream-on and stream-off are controlled by the deserializer driver only.

Figure 10 shows the stream-on and stream-off call flow.

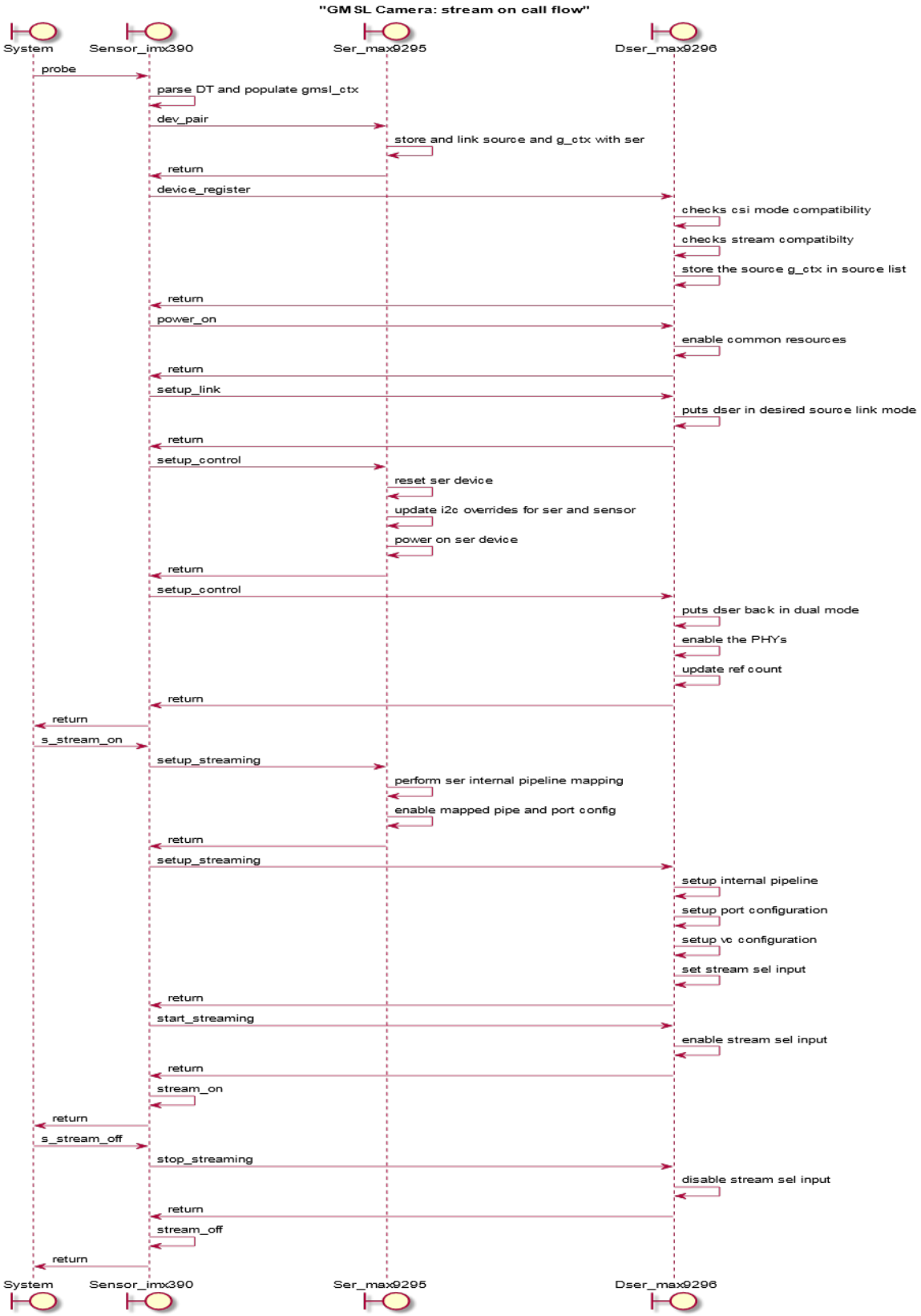

Figure 10: GMSL camera: stream_on and stream_off sequence

SERDES DRIVER API

The SerDes Driver API is described below.

MAX9296 Deserializer Driver

void max9296_power_off (struct device * dev)

Powers off the max9296 deserializer module.

De-asserts the shared reset GPIO and powers off the regulator based on ref count.

Parameters

[in] dev Reference to deserializer device handle

Returns

None

int max9296_power_on (struct device * dev)

Powers on the max9296 deserializer module.

Asserts shared reset GPIO and powers on the regulator, maintains the ref count internally for source devices.

Parameters

[in] dev Reference to deserializer device handle

Returns

0 for success, -1 otherwise.

int max9296_reset_control (struct device * dev, struct device * s_dev)

Resets the link control pipeline, deserializer driver internally decrements the ref count and resets the deserializer device if all the source sensor devices are powered off, which means all the control and streaming configuration is wiped off.

Parameters

[in] dev Reference to deserializer device handle

Returns

0 for success, -1 otherwise.

int max9296_sdev_register (struct device * dev, struct gmsl_link_ctx * g_ctx)

Registers the source sensor device with the deserializer device.

Internally deserializer driver checks all the perquisites and compatibility and finally if valid, stores the source's gmsl_link_ctx context handle is stored in source list maintained by deserializer driver instance.

Parameters

[in]	dev	Reference to deserializer device handle
[in]	g_ctx	Reference to gmsl_link_ctx structure handle

Returns

0 for success, -1 otherwise.

int max9296_sdev_unregister (struct device * dev, struct device * s_dev)

Unregisters the source sensor device with the deserializer device.

Parameters

[in]	dev	Reference to deserializer device handle
[in]	s_dev	Reference to sensor device handle

Returns

0 for success, -1 otherwise.

int max9296_setup_control (struct device * dev)

Sets up the deserializer link's control pipeline.

This gets called during device boot to put deserializer in dual splitter mode. This must be called after max9296_setup_link().

Parameters

[in]	dev	Reference to deserializer device handle

Returns

0 for success, -1 otherwise.

int max9296_setup_link (struct device * dev, struct device * s_dev)

Puts deserializer device in single exclusive link mode, so link specific i2c overrides can be performed for sensor and serializer devices.

Parameters

[in]	dev	Reference to deserializer device handle
[in]	s_dev	Reference to sensor device handle

Returns

0 for success, -1 otherwise.

int max9296_setup_streaming (struct device * dev, struct device * s_dev)

Performs the internal pipeline configuration for the link in context to setup streaming and puts the deserializer link in ready to stream state.

Parameters

[in] dev Reference to deserializer device handle

[in] s_dev Reference to sensor device handle

Returns

0 for success, -1 otherwise.

int max9296_start_streaming (struct device * dev, struct device * s_dev)

Sensor client driver calls it to enable the streaming.

Parameters

[in] dev Reference to deserializer device handle

[in] s_dev Reference to sensor device handle

Returns

0 for success, -1 otherwise.

int max9296_stop_streaming (struct device * dev, struct device * s_dev)

Sensor client driver calls it to disable streaming.

Parameters

[in] dev Reference to deserializer device handle

[in] s_dev Reference to sensor device handle

Returns

0 for success, -1 otherwise.

Both max9296_start_streaming and max9296_stop_streaming are mainly added to enable and disable sensor streaming on the fly while other sensor(s) are active.

MAX9295 Serializer Driver

int max9295_reset_control (struct device * dev)

It reverts the i2c overrides and resets the SER serializer device.

Parameters

[in]	dev	Reference to serializer device handle

Returns

0 for success, -1 otherwise.

int max9295_sdev_pair (struct device * dev, struct gmsl_link_ctx * g_ctx)

When called by sensor client driver, it pairs sensor device with serializer device.

Parameters

[in]	dev	Reference to deserializer device handle
[in]	g_ctx	Reference to gmsl_link_ctx structure handle

Returns

0 for success, -1 otherwise.

int max9295_sdev_unpair (struct device * dev, struct device * s_dev)

When called by sensor client driver, it unpairs sensor device with serializer device.

Parameters

[in]	dev	Reference to serializer device handle
[in]	s_dev	Reference to sensor device handle

Returns

0 for success, -1 otherwise.

int max9295_setup_control (struct device * dev)

Powers on the serializer device and does perform the i2c overrides for sensor and serializer devices which includes setting proxy i2c slave addresses for these devices.

Client must make sure the deserializer device is in link_ex exclusive link mode by calling deserializer driver's max9296_setup_link() function API, before calling this function.

Parameters

[in]	dev	Reference to serializer device handle

Returns

0 for success, -1 otherwise.

int max9295_setup_streaming (struct device * dev)

Sets up the internal pipeline of serializer device for a given pair module of sensor and serializer pair.

Parameters

[in] dev Reference to serializer device handle

Returns

0 for success, -1 otherwise.

DEVICE TREE PROGRAMMING

The device tree describes the hardware connections to the system as well as the static device properties of the drivers.

Platform Device Tree

Hardware connections and device addressing are configured in the platform device tree.

As explained in the preceding section, sensor and serializer devices have two addresses each: one a physical address and the other a proxy address. Both are defined in the platform device tree.

In the reference GMSL module, both sensors have the physical slave address `0x1a`, but must be assigned distinct proxy addresses (user configurable, by default `0x1b` and `0x1c`).

Similarly, for serializer devices the physical address is `0x62`, and the proxy addresses are user configurable, by default `0x40` and `0x60`.

Below table shows the address assignments for given reference GMSL setup:

Table 2: I2C address assignment for reference GMSL setup

	Sensor1	Sensor2	Ser1	Ser2	Des
Physical slave address	0x1a	0x1a	0x62	0x62	0x48
Proxy slave address	0x1b	0x1c	0x40	0x60	NA

Each sensor device initiates power_on request to deserializer driver in order to perform the GMSL link configuration and power on the shared deserializer device. In this power-

on call, the deserializer driver only enables the GMSL link on which the caller sensor device is connected to, and keeps other one disabled, so that there will be no address conflicts between the two links. It then updates all devices' slave addresses on the active link to the proxy addresses defined in the device tree. Same procedure is followed for other GMSL link when deserializer driver gets power_on request from sensor device connected to other GMSL link. In this way, deserializer driver updates all the addresses of all the devices hosted by the deserializer device.

For example, in the reference GMSL setup, when the deserializer gets a power-on request from the sensor device connected to link A, it enables GMSL link A and disables GMSL link B. It first communicates to serializer and the sensor devices on link A at their respective physical addresses (`0x1a` for sensor1 and `0x62` for Ser1), and then updates those physical addresses to proxy addresses (`0x1b` for sensor1 and `0x40` for Ser1). Then it performs the same steps for GMSL link B when sensor device on GMSL link B requests power_on to deserializer driver. Finally, it sets up GMSL link in dual mode. Thus, each device is identified uniquely over the I2C bus. All further device communication takes place on the proxy addresses until reboot.

At present the GMSL link address are programmed at boot time, during probing, due to control configuration timing constraints. In a future release the process may be moved to power-on/power-off.

The following code section is an example of platform device configuration of the GMSL setup shown in diagrams above. For full details, see the platform device tree file `tegra186-quill-camera-imx390-a00.dtsi` (for Jetson TX2) and `tegra194-p2822-0000-camera-imx390-a00.dtsi` (for Jetson AGX Xavier).

```
tca9546@70 {
    /* The deserializer is connected to Tegra Jetson camera connector
port via MIPI adapter over pca9546 i2c expander, so all the GMSL device
nodes go under here. */
    compatible = "nxp,pca9546";
    i2c@0 {
        /* As in above setup, deserializer (which hosts all the
serializers and sensors) is connected to adaptor onto port A, so on i2c
expander all the GMSL devices go under i2c@0, the expander assigns 0x30
unique address to port 0.*/
        reg = <0>;
        dser: max9296@48 {
            /* single common deserializer at 0x48 */
            compatible = "nvidia,max9296";
            reg = <0x48>;
            csi-mode = "2x4";  /* this tells max CSI lane configuration
*/
            max-src = <2>; /* max sources */
            /* As deserializer is common among sensor devices, the
reset and power rails are controlled via deserializer */
```

```
            reset-gpios = <&tegra_main_gpio CAM0_RST_L
GPIO_ACTIVE_HIGH>;
            vdd_cam_1v2-supply = <&en_vdd_cam_1v2>;
        };

        ser_prim: max9295_prim@62 { /* Default serializer device */
            compatible = "nvidia,max9295";
            reg = <0x62>;
            is-prim-ser; /* primary ser device */
        };

        ser_a: max9295_a@40 {
            /* serializer device connected at link A at proxy address
0x40 (the proxy address are assigned runtime) */
            compatible = "nvidia,max9295";
            reg = <0x40>;
            nvidia,gmsl-dser-device = <&dser>; /* link to its
deserializer device */
        };

        ser_b: max9295_b@60 {
            /* serializer device connected at link A at proxy address
0x60 (the proxy address is assigned runtime) */
            compatible = "nvidia,max9295";
            reg = <0x60>;
            nvidia,gmsl-dser-device = <&dser>; /* link to its
deserializer device */
        };

        imx390_a@1b {
            /* sensor device connected at link A at proxy address 0x1b
(the proxy address is assigned runtime). */
            def-addr = <0x1a>; /* default slave address is 0x1a */
            nvidia,gmsl-ser-device = <&ser_a>; /* link to its
serializer device. */
            nvidia,gmsl-dser-device = <&dser>; /* link to its
deserializer device. */
        };

        imx390_b@1c {
            /* sensor device connected at link A at proxy address 0x1b
(the proxy address are assigned runtime). */
            def-addr = <0x1a>; /* Default slave address is 0x1a. */
            /* Define clocks, io pins, power sources */
            nvidia,gmsl-ser-device = <&ser_b>; /* Link to its
serializer device. */
            nvidia,gmsl-dser-device = <&dser>; /* Link to its
deserializer device. */
        };
    }; /* i2c@0 closing */
};   /* tca9546@70 closing */
```

Module Device Tree

After platform device configuration, the device module configuration must be added to the module device tree. For full details, see `tegra186-camera-imx390-a00.dtsi` for Jetson TX2 and `tegra194-camera-imx390-a00.dtsi` for Jetson AGX Xavier. A few snippets are shown here to illustrate the configuration. The virtual channels are set in the module device tree.

The sensor device acts as master device in the GMSL software framework, so in the module device tree each sensor device node contains a `gmsl-link` device node which describes the properties of the GMSL link to which the sensor is connected. In the example below, the `gmsl-link` node describes the sensor connected at GMSL link A.

```
gmsl-link {
    src-csi-port = "b";     /* Port at which sensor is connected to
 its serializer device. */
    dst-csi-port = "a";     /* Destination CSI port on the Jetson
 side, connected at deserializer. */
    serdes-csi-link = "a"; /* GMSL link sensor/serializer connected */
    csi-mode = "1x4";      /*  to sensor CSI mode. */
    st-vc = <0>;           /* Sensor source default VC ID: 0 unless
 overridden by sensor. */
    vc-id = <0>;           /* Destination VC ID, assigned to sensor
 stream by deserializer. */
    num-lanes = <2>;       /* Number of CSI lanes used. */
    streams = "ued-u1", "raw12";
/* Types of streams sensor is streaming. */
};
```

For any GMSL module configuration, all the fields shown above must be set as per connectivity.

Virtual Channel Programming in the Module Device Tree

You must add the same `vc-id` property value to the respective CSI and VI channel nodes in the same module device tree file. This property is used for vi-mode use cases.

You must set the `vc-id` property in the `gmsl-link` node for each sensor to match the `vc_id` property in the sensor mode device node.

This mode `vc_id` property is used for vi bypass mode use cases. It is read and set by PCL in user space. For example:

```
mode0 { /*mode IMX390_MODE_1920X1080_CROP_30FPS*/
      mclk_khz = "24000";
      num_lanes = "2";
      tegra_sinterface = "serial_a";
      vc_id = "0";
```

VI and CSI Channel Nodes in the Module Device Tree

The port binding to VI and CSI is also done in module device tree for given reference GMSL setup. The number of VI and CSI channels would be 2 just like any other dual sensor setup, but the "`port-index`" field would be set as per the hardware connectivity.

```
vi@15700000 {
      num-channels = <2>;
      ports {
            #address-cells = <1>;
            #size-cells = <0>;
            port@0 {
                  reg = <0>;
                  imx390_vi_in0: endpoint {
                        vc-id = <0>;
                        port-index = <0>;
                        bus-width = <2>;
                        remote-endpoint = <&imx390_csi_out0>;
                  };
            };
            port@1 {
                  reg = <1>;
                  imx390_vi_in1: endpoint {
                        vc-id = <1>;
                        port-index = <0>;
                        bus-width = <2>;
                        remote-endpoint = <&imx390_csi_out1>;
                  };
            };
      };
};

nvcsi@150c0000 {
      num-channels = <2>;
      #address-cells = <1>;
      #size-cells = <0>;
      channel@0 {
            reg = <0>;
            ports {
                  #address-cells = <1>;
                  #size-cells = <0>;
                  port@0 {
                        reg = <0>;
                        imx390_csi_in0: endpoint@0 {
                              port-index = <0>;
                              bus-width = <2>;
                        remote-endpoint = <&imx390_imx390_out0>;
                        };
```

```
                };
                port@1 {
                        reg = <1>;
                        imx390_csi_out0: endpoint@1 {
                        remote-endpoint = <&imx390_vi_in0>;
                        };
        };
                };
        };
        channel@1 {
                reg = <1>;
                ports {
                        #address-cells = <1>;
                        #size-cells = <0>;
                        port@0 {
                                reg = <0>;
                                imx390_csi_in1: endpoint@2 {
                                        port-index = <0>;
                                        bus-width = <2>;
                                remote-endpoint = <&imx390_imx390_out1>;
                                };
                        };
                        port@1 {
                                reg = <1>;
                                imx390_csi_out1: endpoint@3 {
                                remote-endpoint = <&imx390_vi_in1>;
                                };
                        };
                };
        };
};
```

In this configuration, for VI channel 0 and channel 1 nodes both, the "`port-index`" property is set to 0, means both the sensors using VI in0 stream or PP0 stream.

For CSI channel 0 and channel 1 nodes, same the "`port-index`" property is set to 0, means in both the channels, the sensors are connected to CSI port A.

Also, both VI and CSI channel nodes, contain new property "`vc-id`" used in vi mode use cases as mentioned above, it must match with "gmsl-link" device node's "`vc-id`" property.

The "`bus-width`" property tells the num of CSI lanes used.

Tegra-camera-platform node configuration in module device tree

Tegra-camera-platform device node contains the sensor modules details. Below configuration is for reference GMSL module:

```
tegra-camera-platform {
        compatible = "nvidia, tegra-camera-platform";
```

```
      num_csi_lanes = <2>;
      max_lane_speed = <4000000>;
      min_bits_per_pixel = <10>;
      vi_peak_byte_per_pixel = <2>;
      vi_bw_margin_pct = <25>;
      isp_peak_byte_per_pixel = <5>;
      isp_bw_margin_pct = <25>;

      modules {
            module0 {
                  badge = "imx390_rear";
                  position = "rear";
                  orientation = "1";
                  drivernode0 {
                        pcl_id = "v4l2_sensor";
                        /* Driver v4l2 device name */
                        devname = "imx390 30-001b";
                        proc-device-tree = "/proc/device-
tree/i2c@3180000/tca9546@70/i2c@0/imx390_a@1b";
                  };
            };
            module1 {
                  badge = "imx390_front";
                  position = "front";
                  orientation = "1";
                  drivernode0 {
                        pcl_id = "v4l2_sensor";
                        /* Driver v4l2 device name */
                        devname = "imx390 30-001c";
                        proc-device-tree = "/proc/device-
tree/i2c@3180000/tca9546@70/i2c@0/imx390_b@1c";
                  };
            };
      };
 };
```

- `num_csi_lanes` is set to 2, as both the sensors are connected to single CSI port A in x2 lane fashion. This is unlike of other dual sensors where `num_csi_lanes` is set to total number of CSI lanes used by both the sensors, because they both use different CSI ports.
- `max_lane_speed` is set considering 2 sensors are streaming through single shared port.
- The sensor device nodes are defined using sensor proxy addresses instead of the physical address, and their paths are set in `proc-device-tree` property, and similar naming is used for `devname` property.

For rest of configuration, follow the same guidelines as for other sensors.

Refer to the following module device tree files:

For Jetson TX2, <tegra186-camera-imx390-a00.dtsi>

For Jetson AGX Xavier, <tegra194-camera-imx390-a00.dtsi>

Plugin Manager Device Tree

The device tree additions for Plugin Manager are the same as for any other sensor module.

See node fragment-imx390@0 in `tegra186-quill-camera-plugin-manager.dtsi` (TX2) or `tegra194-camera-plugin-manager.dtsi`(Xavier) for reference GMSL setup plugin manager support.

Camera Modules Device Tree

All the device nodes in the I2C node of the platform's camera module device tree, `tegra186-quill-camera-modules.dtsi`(TX2) or `tegra194-p2822-camera-modules.dtsi`(Xavier), are defined in the same way as in the platform device tree. As in the reference GMSL setup, they all go to `i2c@0`.

Note:	**The default "vc-id" is set to 0 for all VI ports in modules device tree file, so you must assign/override correct "vc-id" in plugin-manager and sensor specific module device tree file.**

CONSTRAINTS

Sensor streams connected to same CSI aggregator and sharing a CSI port must use the same lane configuration. The deserializer driver fails to register a stream if its lane configuration does not match other streams that share the same deserializer device.

Apart from that, CSI aggregators generally support multiple identical cameras running with the same frame rate and sensor mode. Review your aggregator data sheet for additional restrictions if you are using a more complicated configuration.

VALIDATION

Dual GMSL sensor streaming (preview/capture) sharing CSI port A is validated using an Argus camera and the V4L2 application.

The 2x CSI deserializer/aggregator is validated. The 4x CSI aggregator is not validated, as its hardware module is not available at this time.

KNOWN ISSUES

This section summarizes known issues in the GMSL camera framework.

General Issues

- Preview flickers on scene change.
- Sensor tuning and image quality are substandard.

Xavier Specific

Note:	**This limitation and rework only applies to older GMSL kits which do not draw the on-board(DESER) 1.2V supply.**

The 1.2v power supply needs rework. The MAX9296 deserializer board used in the reference setup does not draw the required on-board 1.2v power, and Jetson AGX Xavier developer kit carrier board has dropped support for supplying 1.2v to the camera connector (Jetson TX2 does have this support).

The following rework is required on Jetson AGX Xavier for vendor modules that expect 1.2v from Jetson AGX Xavier:

- Use spare LDO for the SATA controller, which can take max 1.21v. The `Vmin` requirement from the IMX390 reference sensor is 1.14v, and the range is 1.14 to 1.26v, so this LDO supply is enough to supply 1.2v to the Jetson AGX Xavier carrier board's camera connector.

Plugin Manager Board ID

The reference GMSL module currently does not have a unique board ID. Use the MIPI adapter's board ID, LPRD-001. This ID is used by other sensor modules which use similar MIPI adapters, such as IMX274 and IMX185. To work around this issue until it is

resolved, the IMX390 module is disabled in these sensor modules; see `tegra186-quill-camera-plugin-manager.dtsi`.

Notice

Trademarks

Copyright