Qualcomm

Qualcomm Technologies, Inc.

SA6155/SA8155/SA8195/SA8295 Automotive Camera AIS Customization Guide

80-PG469-93 Rev. G

July 27, 2022

For additional information or to submit technical questions, go to: https://createpoint.qti.qualcomm.com

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Revision history

Revision	Date	Description
A	January 2019	Initial release.
B	March 2019	Updated to include SA6155.
C	January 2021	Updated to include SA8195. Numerous changes were made to this document; it should be read in its entirety.
D	February 2021	Updated Section 5.5. Added Section 5.7.
E	August 2021	Updated custom functions for sensor_lib_t (Section 6.2.2) and added information to enable CPHY (see Section 6.3). Updated command line arguments for QCarCam test (see Section 8.1). Added ISP customization (see Chapter 12).
F	January 2022	Updated support for bulk write-read operation using CCI.
G	July 2022	Updated to include SA8295.

Contents

Figures

Tables

1 Introduction

1.1 Purpose

This document provides camera sensor and camera bridge chip driver development guidelines. It describes how to bring up the camera on all AIS supported platforms, which include Android P, Automotive Grade Linux, QNX, and Green Hills. These camera guidelines include the configuration of the following components:

- Camera sensor
- Camera bridge chip
- AIS device configuration

Most of the information in this document is specific to automotive platforms and was written based on the SA6155/SA8155/SA8195 chipset.

Driver development guidelines and bring up procedures for the related mobile camera platform has applicable camera content, described in the *Multimedia Driver Development and Bringup Guide – Camera* (80-NU323-2).

1.2 Conventions

Function declarations, function names, type declarations, attributes, and code samples appear in a different font, for example, `cp armcc armcpp`.

Code variables appear in angle brackets, for example, `<number>`.

Commands to be entered appear in a different font, for example, **`copy a:*.* b:`**.

Button and key names appear in bold font, for example, click **Save** or press **Enter**.

Shading indicates content that has been added or changed in this revision of the document.

1.3 Technical assistance

For assistance or clarification on information in this document, submit a case to Qualcomm Technologies, Inc. (QTI) at https://createpoint.qti.qualcomm.com/.

If you do not have access to the CDMATech Support website, register for access or send email to support.cdmatech@qti.qualcomm.com.

2 Qualcomm® Snapdragon™ ADP

2.1 ADP hardware topology

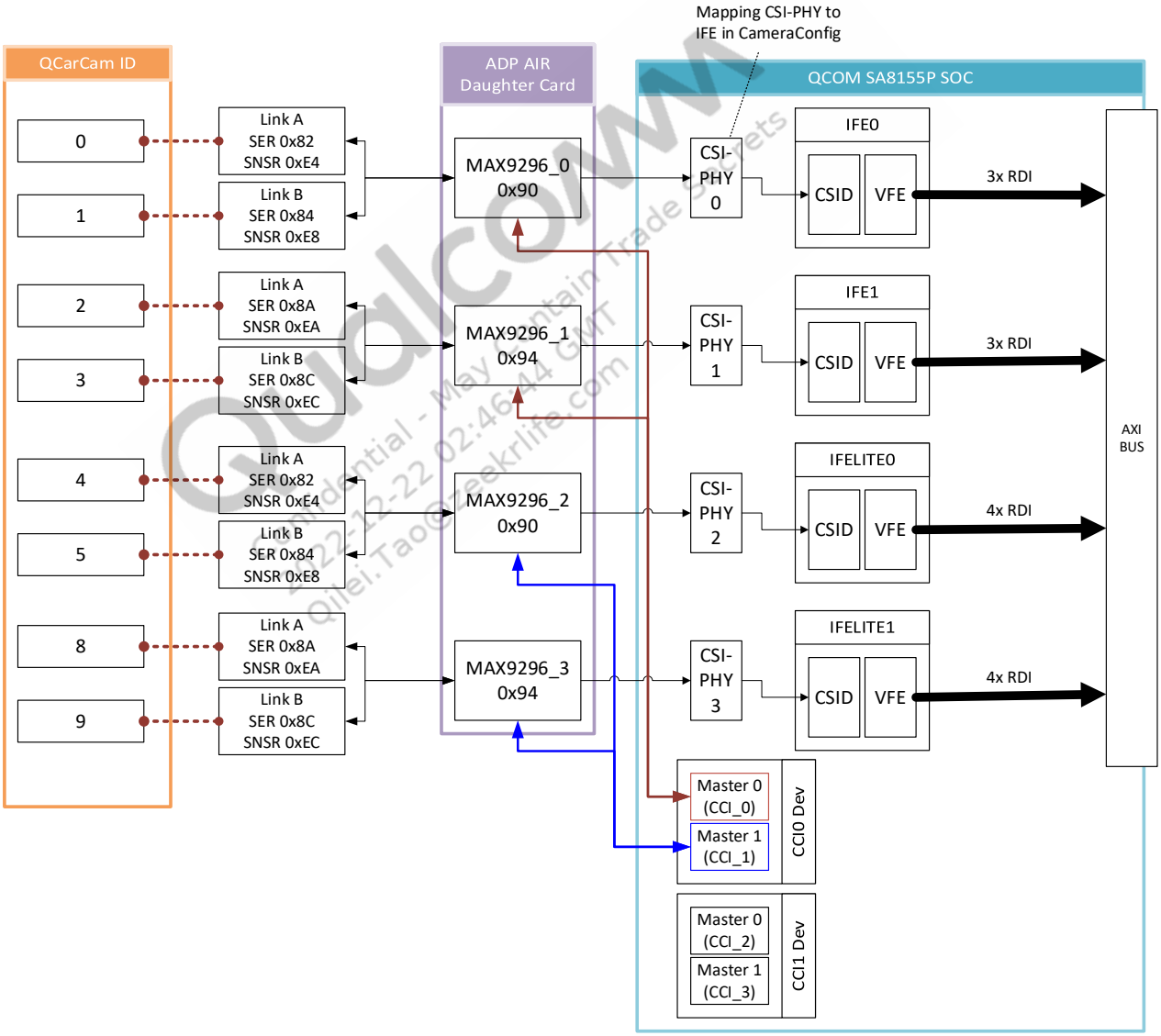

Figure 2-1 Hardware topology for SA8155 ADP reference platform

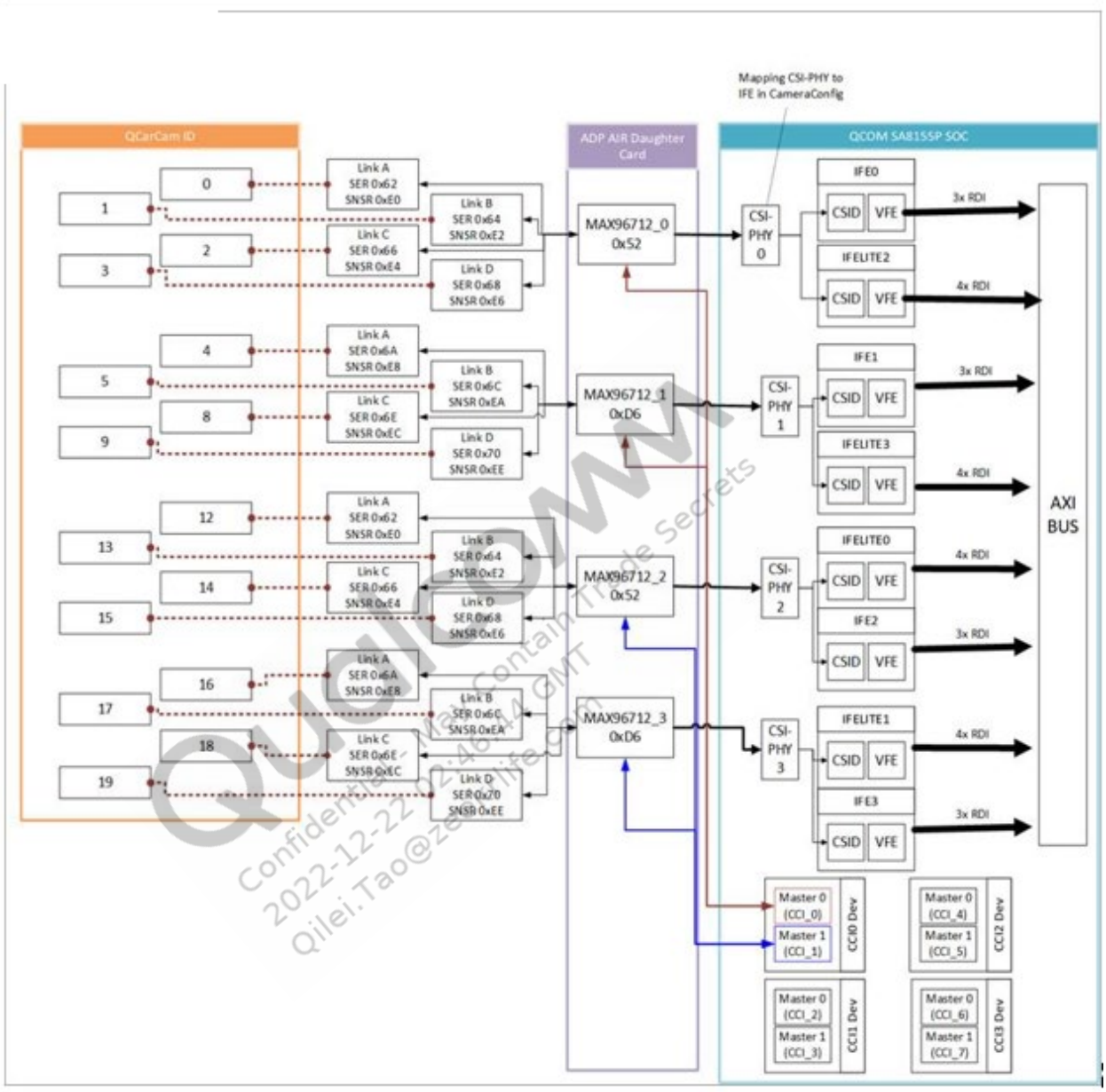

Figure 2-2 Hardware topology for the SA8295 Qualcomm® Snapdragon™ ADP reference platform

2.1.1 ADP AIR connections

The following shows the physical Qualcomm ADP AIRVideo In Card and the QCarCam ID.

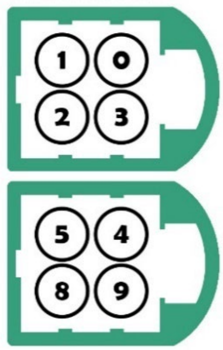

Figure 2-3 SA6155/SA8155 ADP AIR video in card connections

2.1.2 Camera input devices

Camera input devices (typically deserializers or bridge chips) can be used on any CSI. The Snapdragon ADP camera daughter card comes with four (4) MAXIM 9296 deserializers (see Figure 2-1).

This bridge chip will work out of the box without any additional customization. Analog (CVBS/HDMI input) camera bridge chips and digital camera bridge chips other than the MAXIM 9296 will require additional customization and add more development time by the OEM. The necessary changes to customize bridge chips are described later in this guide.

2.1.3 Camera sensors

The Snapdragon ADP supports the standard camera sensor AR0231-GMSL2. It comes in multiple configurations such as AR0231-GMSL2-060H and AR0231-GMSL2-200H.

These configurations can be purchased from Leopard Imaging under:
Products > SerDes Cameras > Maxim GMSL2 Camera > AR0231-GMSL2.

When testing new sensors or using new bridge chips, it is recommended to first test with the AR0231 as all the sensor initialization, mode selection, timing params, etc., are already set correctly in the AIS sensor lib driver. Use the AR0231 sensor as a reference to compare logs with the new sensor you are bringing up. Changes needed for customizations to camera sensors are described later in this guide.

The AR0231-GMSL2 module is comprised of three components:

- AR0231 sensor
- AP0200 ISP
- MAX9295 serializer

As the MAX9296 deserializer supports up to two (2) GMSL connections, two (2) sensors can be connected to each deserializer as the figure below shows.

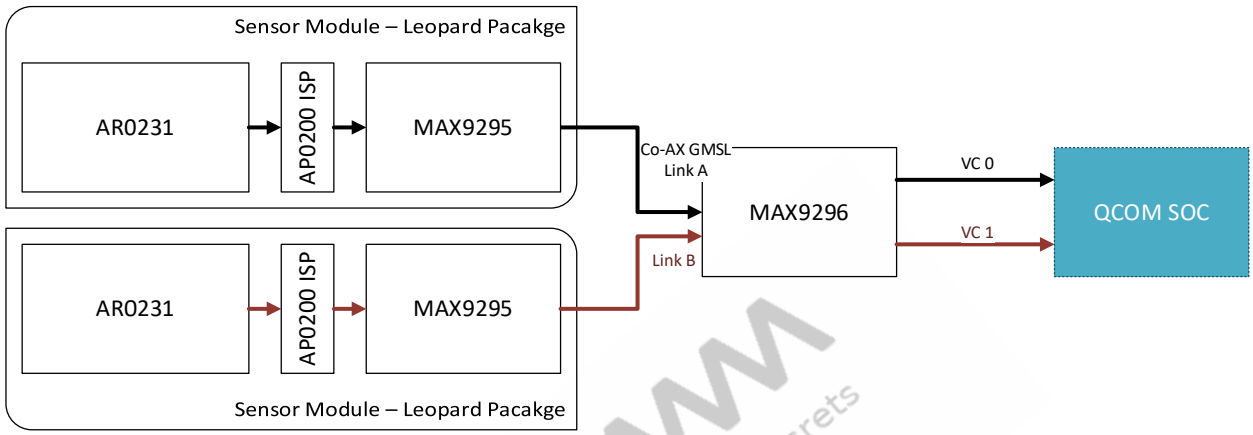

Figure 2-4 MAX9296 and AR0231-GMSL2 hardware topology ADP SA6155/SA8155

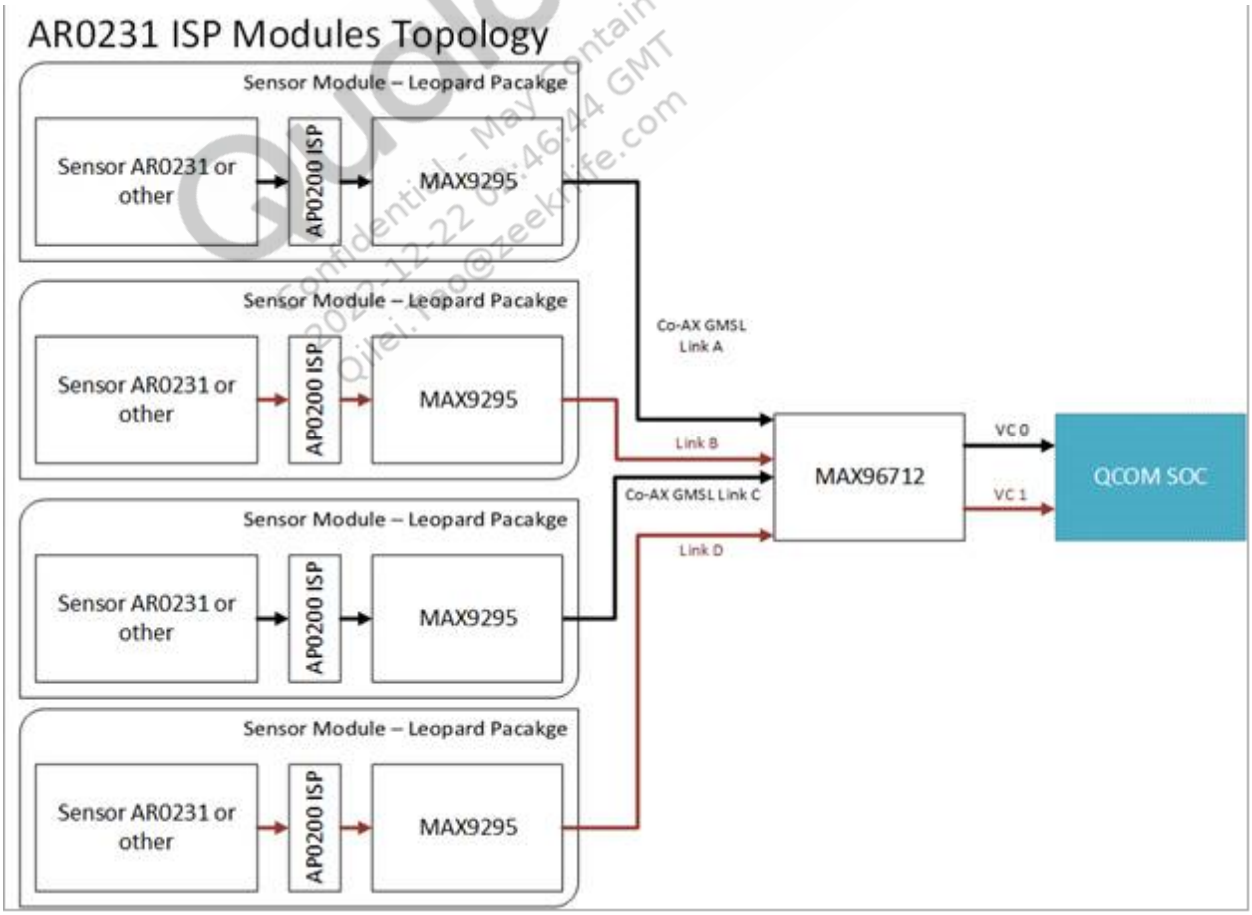

Figure 2-5 MAX96712 and AR0231-GMSL2 HW Topology ADP SA8295

2.2 ADP AIR hardware configuration

The following table details the physical configuration of the ADP AIR Platform for Gen 3 SOCs.

Table 2-1 ADP AIR hardware configuration

Deserializer	Interface	SA6155	SA8155	SA8195
MAX9296_0	CSI	CSI0 4 lanes	CSI0 4 lanes	CSI0 4 lanes
	I2C	CCI0 (gpio 32/33) Slave Addr: 0x90	CCI0 (gpio 17/18) Slave Addr: 0x90	CCI0 (gpio 17/18) Slave Addr: 0x90
	GPIOs	PDB – 28 INTR – 13	PDB – 21 INTR – 13	PDB – 21 INTR – 13
MAX9296_1	CSI	CSI1 4 lanes	CSI1 4 lanes	CSI1 4 lanes
	I2C	CCI0 (gpio 32/33) Slave Addr: 0x94	CCI0 (gpio 17/18) Slave Addr: 0x94	CCI0 (gpio 17/18) Slave Addr: 0x94
	GPIOs	PDB – 29 INTR – 14	PDB – 22 INTR – 14	PDB – 22 INTR – 14
MAX9296_2	CSI	CSI2 4 lanes	CSI2 4 lanes	CSI2 4 lanes
	I2C	CCI1 (gpio 34/35) Slave Addr: 0x90	CCI1 (gpio 34/35) Slave Addr: 0x90	CCI1 (gpio 34/35) Slave Addr: 0x90
	GPIOs	PDB – 30 INTR – 15	PDB – 23 INTR – 15	PDB – 23 INTR – 15
MAX9296_3	CSI	N/A (Only 3 CSIs on SA6155)	CSI3 4 lanes	CSI3 4 lanes
	I2C	N/A	CCI1 (gpio 34/35) Slave Addr: 0x94	CCI1 (gpio 34/35) Slave Addr: 0x94
	GPIOs	N/A	PDB – 25 INTR – 16	PDB – 25 INTR – 16

Table 2-2 ADP AIR hardware configuration

NOTE: DESERIALIZER		SA8295
MAX96712_0	CSI	CSI0 4 lanes
	I2C	CCI0 (gpio 17/18) Slave Addr: 0x90
	GPIOs	PDB – 21 INTR – 13
MAX96712_1	CSI	CSI1 4 lanes
	I2C	CCI0 (gpio 17/18) Slave Addr: 0x94
	GPIOs	PDB – 22 INTR – 14
MAX96712_2	CSI	CSI2 4 lanes

NOTE: DESERIALIZER		SA8295
	I2C	CCI1 (gpio 34/35) Slave Addr: 0x90
	GPIOs	PDB – 23 INTR – 15
MAX96712_3	CSI	CSI3 4 lanes
	I2C	CCI1 (gpio 34/35) Slave Addr: 0x94
	GPIOs	PDB – 25 INTR – 16

NOTE: CSI D-PHY also requires a clock lane.

To save time, it is recommended to use the same GPIOs as what is used on the ADP and configured in AIS.

2.3 CSI modes

The Snapdragon MIPI CSI PHY provides DPHY (2 phase) and CPHY (3 phase) support. Combo mode (2+1) is currently not supported with AIS.

2.4 Multiple camera sensors on one bridge chip

If you want to use different camera sensors on one bridge chip, contact the camera bridge chip vendor to confirm that the camera bridge chip actually supports different resolution cameras connected to the same bridge chip.

This method may affect frame timing and safety.

2.5 I2C communication

SA6155/SA8155/SA8195 supports I2C communication through CCI. There are up to eight (8) CCI busses that can be connected and used for communication with the bridge chip and sensors.

Snapdragon ADPs use a single bus to communicate with all bridge chips. Customization of which bridge chip is connected to which bus will be addressed in Chapter 5 .

It is recommended to have an I2C communication lane directly to the camera bridge chip from SA6155/SA8155. This is important for bringup purposes as all initialization in AIS for camera bridge chips is done over I2C.

Without a direct I2C connection to the camera bridge chip, runtime settings cannot be applied to the camera sensor under the current AIS framework, which limits the use case to YUV output of camera sensors where the controls are at the start and stop.

The following table lists the number of CCI busses for each Snapdragon SOC.

Snapdragon SOC	Number of CCI cores	Number of CCI masters
SA6155	1	2
SA8155	2	4
SA8195	4	8
SA8295	4	8

If connecting multiple devices to a single I2C bus, ensure they are uniquely addressed. On the ADP, the MAX9296s are uniquely addressed as 0x90 and 0x94 on each bus.

3 AIS background information

3.1 QCarCam format

The qcarcam_fmt_t is a mask comprised of three components:

- Color pattern
- Bit depth of each color channel
- Memory packing

3.1.1 QCarCam color pattern

Possible color patterns are defined by `qcarcam_color_pattern_t`.

```
/// @brief Color type
typedef enum
{
    QCARCAM_RAW = 0,

    QCARCAM_YUV_YUYV = 0x100,
    QCARCAM_YUV_YVYU,
    QCARCAM_YUV_UYVY,
    QCARCAM_YUV_VYUY,

    QCARCAM_YUV_NV12,
    QCARCAM_YUV_NV21,

    QCARCAM_BAYER_GBRG = 0x200,
    QCARCAM_BAYER_GRBG,
    QCARCAM_BAYER_RGGB,
    QCARCAM_BAYER_BGGR,

    QCARCAM_RGB = 0x300,

}qcarcam_color_pattern_t;
```

3.1.2 QCarCam color bit depth

Possible values for the bit depth of each color channel are defined in `qcarcam_color_bitdepth_t`.

```
/// @brief Bitdepth per color channel
typedef enum
{
    QCARCAM_BITDEPTH_8 = 8,
    QCARCAM_BITDEPTH_10 = 10,
    QCARCAM_BITDEPTH_12 = 12,
    QCARCAM_BITDEPTH_14 = 14,
    QCARCAM_BITDEPTH_16 = 16,
    QCARCAM_BITDEPTH_20 = 20
}qcarcam_color_bitdepth_t;
```

3.1.3 QCarCam color packing

Possible memory packing options are defined in `qcarcam_color_pack_t`.

```
/// @brief Packing type
typedef enum
{
    QCARCAM_PACK_QTI = 0,
    QCARCAM_PACK_MIPI,
    QCARCAM_PACK_DPCM6,
    QCARCAM_PACK_DPCM8,
    QCARCAM_PACK_PLAIN8,
    QCARCAM_PACK_PLAIN16,
    QCARCAM_PACK_PLAIN32,
    QCARCAM_PACK_FOURCC
}qcarcam_color_pack_t;
```

The following examples illustrate common packing and bit depth:

MIPI 10-bit

The 10-bit packing holds formats like Raw10. Four (4) pixels are held in every 5 bytes. The output width of the image must be a multiple of 4 pixels.

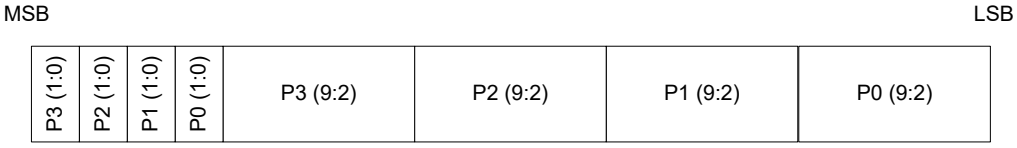

Figure 3-1 MIPI 10-bit memory packing

MIPI 12-bit

The 12-bit packing holds format like Raw12. Two (2) pixels are held in every 3 bytes. The output width of the image must be a multiple of 2 pixels.

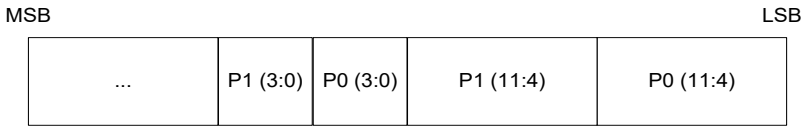

Figure 3-2 MIPI 12-bit memory packing

Plain16 10-bit:

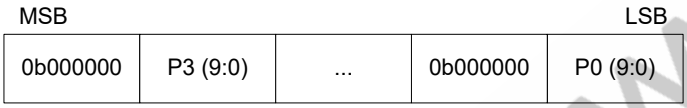

Figure 3-3 Plain16 10-bit memory packing

Plain16 12-bit:

Figure 3-4 Plain16 12-bit memory packing

3.1.4 QCarCam defined color formats

QCarCam API provides a definition of commonly used formats. This is non-exhaustive and users can use valid combinations of the above parameters to define a format using the QCARCAM_COLOR_FMT macro.

```
#define QCARCAM_COLOR_FMT(_pattern_, _bitdepth_, _pack_) \
    (((_pack_ & 0xff) << 24) | ((_bitdepth_ & 0xff) << 16) | (_pattern_ &
0xffff))
```

This results in the following `qcarcam_color_fmt_t` definition for YUV422 8 bit with UYVY ordering:

```
QCARCAM_FMT_UYVY_8 = QCARCAM_COLOR_FMT(QCARCAM_YUV_UYVY,
QCARCAM_BITDEPTH_8, QCARCAM_PACK_FOURCC),
```

qcarcam_types.h provides a defined list of commonly used formats.

3.2 Sensor format settings

Users can customize the format of the camera sensor, as long as the camera bridge chip can handle the format and output it to the ADP correctly. For example, RAW10 with 8 bit mode as it is done in the sample AR0231 driver.

It is important to track which formats are being output at each stage. The following example image shows the block chain that the Snapdragon ADP reference platform follows using AR0231.

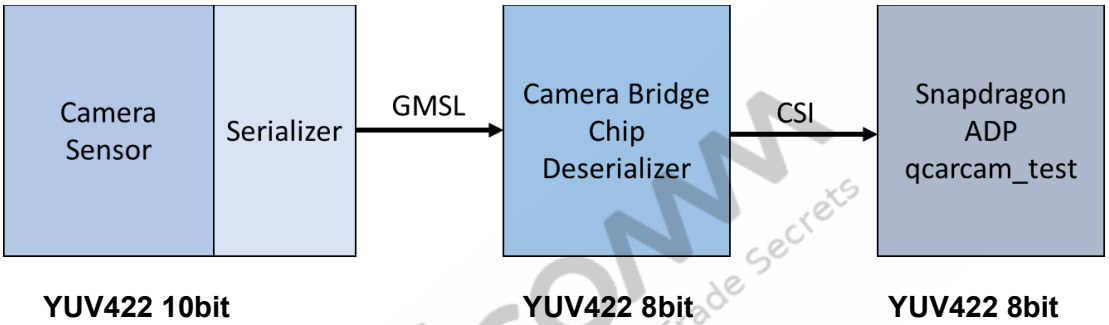

Figure 3-5 SA6155/SA8155 ADP sample camera system setup

In the reference platform, AR0231 comes with AP0200. The default output of it is YUV422 10 bit. The Serializer is programmed to cut the 2 LSBs and forward it as YUV422 8 bit. Therefore, the advertised format from the sensor driver is QCARCAM_YUYV_8 to represent YUV422 8 bit.

It is useful to set the color format that QCarCam expects in the header so that the camera bridge chip driver can correctly pass the information on.

3.3 Hardware format repacking

Snapdragon hardware provides the ability to repack some formats to memory. The AIS Engine will determine if this is possible by checking the advertised sensor format against the requested format from the client.

For instance, a typical use case would involve decoding MIPI RAW to PLAIN packing because it is preferred method to access pixel data in 16 bit units. Suppose we have a QCARCAM_FMT_MIPIRAW_12 input. The client can request QCARCAM_FMT_PLAIN16_12 to have the data repacked to Plain16 12 bit format described in Section 3.1.3 .

3.4 Settle count calculations

To calculate settle count, refer to the camera bridge chip data sheet to determine the preferred transmission speed and register delay when initializing the bridge chip.

The following reference formula is used to calculate settle count value for the MAX9296 sensor driver. Typically, there is a min and max settle count value that can be calculated such that:

$\text{Min}\ (K_{SETTLE\_COUNT}) > \text{Min}\ (T_{HS\_SETTLE}) / T_{TIMER}$

$\text{Max}_1\ (K_{SETTLE\_COUNT}) < (\text{Min}(T_{HS\_PREPARE} + T_{HS\_ZERO}) - 4*T_{Timer}) / T_{TIMER}$

$\text{Max}_2\ (K_{SETTLE\_COUNT}) < (\text{Max}(T_{HS\_PREPARE} + T_{HS\_ZERO}) - 4*T_{Timer}) / T_{TIMER}$

The settle count value should be a value selected between min and max_1 with the absolute maximum range for the value to be chosen from min and max_2.

Descriptions of all the formula variables are in the following table. Refer to the camera bridge chip datasheet and bridge chip vendor to confirm the tolerance range is correct for your bridge chip.

Variable	Description	Associated formulas
F_{TIMER}	Clock rate frequency in MHz	CSIPHY clock rate frequency = 200 MHz
T_{TIMER}	Period of F_{TIMER}	T_{TIMER} (ns) = 1000/ F_{TIMER} (MHz)
F_{DATA}	CSI transmission data rate specified by the camera bridge chip data sheet.	Typical values are 800 Mbps or 1600Mbps. The common transmission rate used in AIS is 800 Mbps. You can find CSI transmit values in camera bridge chip data sheet.
UI	Unit Interval	UI (ns) = 1000/F_{DATA} (Mbps)
F_{DDR}	DDR clock rate frequency in MHz	$F_{DDR} = F_{DATA}/2$ (MHz)
$T_{HS\_PREPARE}$	Time that the transmitter drives the data lane LP-00 (low power-00) line state immediately before the HS-0 (high speed-0) line starting HS mode	Min $T_{HS\_PREPARE}$ = 40ns + 4UI Max $T_{HS\_PREPARE}$ = 85ns + 6UI
T_{HS_SETTLE}	Time interval in which the HS receiver shall ignore any Data Lane transitions starting from the beginning of $T_{HS\_PREPARE}$	Min T_{HS_SETTLE} = Max $T_{HS\_PREPARE}$ Max T_{HS_SETTLE} = 145ns + 10UI
$T_{HS\_PREPARE} + T_{HS\_ZERO}$	Time that the transmitter drives the data lane LP-00 (low power-00) line state immediately before the HS-0 (high speed-0) line starting HS mode plus zero transmit time	Min ($T_{HS\_PREPARE} + T_{HS\_ZERO}$) = Max T_{HS_SETTLE} Max ($T_{HS\_PREPARE} + T_{HS\_ZERO}$) = Max T_{HS_SETTLE} +… …(Max $T_{HS\_PREPARE}$ - Min $T_{HS\_PREPARE}$)

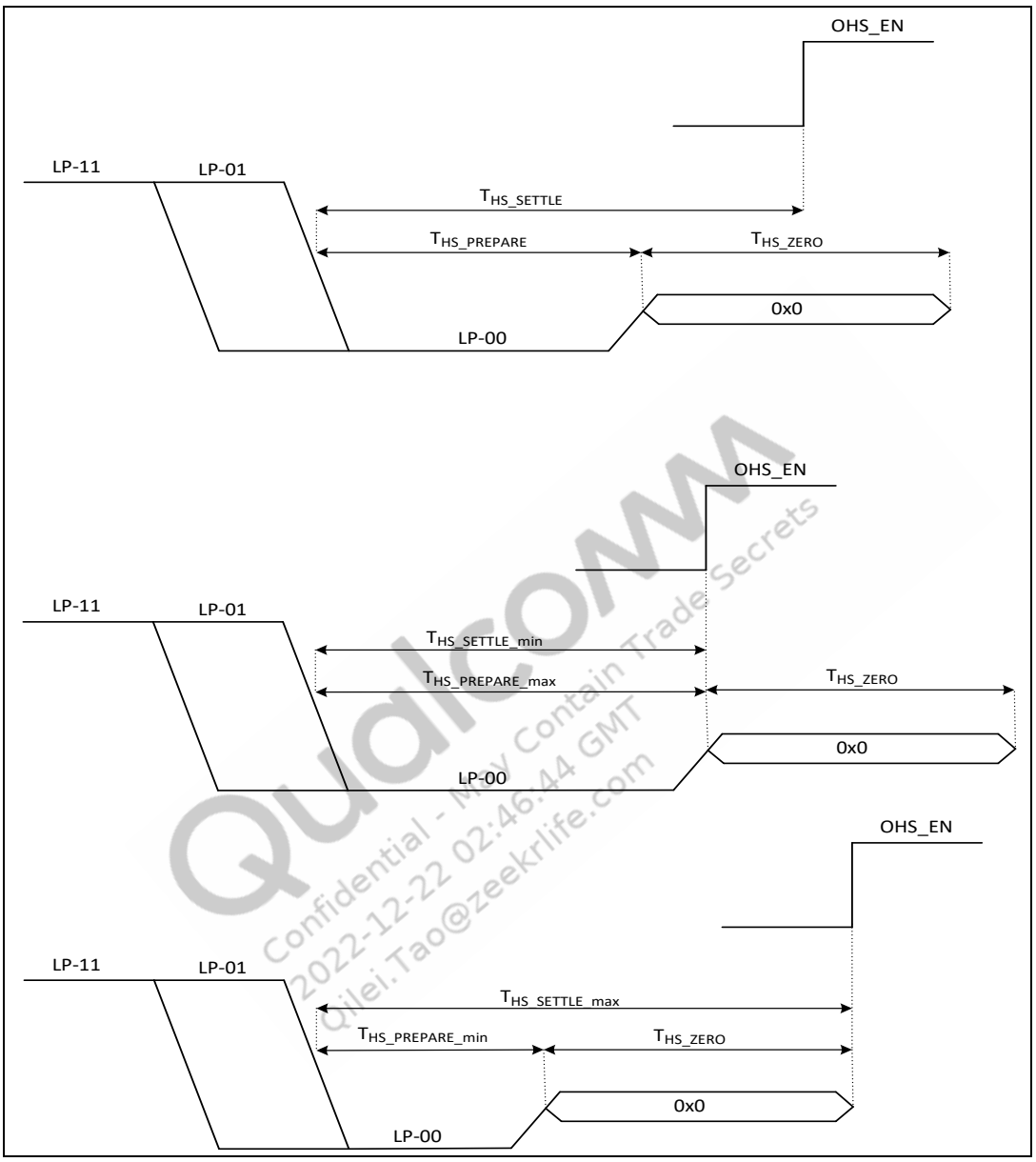

Figure 3-6 T_{HS_SETTLE} diagram for typical T_{HS_SETTLE}, $T_{HS\_SETTLE\_MIN}$, and $T_{HS\_SETTLE\_MAX}$

3.5 Calculating sensor timing parameters

To calculate all camera sensor timing parameters seen in the sensor snippet, refer to the table below.

Variable	Description	Associated formulas
Xout	Horizontal output that is twice the sensor width	xout = w*2
Yout	Vertical output which is the sensor height	yout = h
clks_per_line	Line length pixel clock multiplied by two	clks_per_line = ll_pclk*2
total_lines	▪ Total lines where each line runs horizontal ▪ Must account for horizontal blanking lines	total_lines = (w + hbl) + (hbl*2)
vt_clk	vt_pixel_clk is the sensor scanning rate (cycles / sec)	vt_pclk = ll_pclk * fll * fr
op_clk	op_pixel_clk is the actual sensor output rate (cycles / sec)	op_pclk = vt_pclk Unless otherwise specified by camera sensor datasheet
hts	Horizontal total screen is the total horizontal screen size which is the width and horizontal blanking lines combined	hts = w + hbl
ll_pclk	line_length_pclk is the number of pixels per line	ll_pclk = (w + hbl) * b/p
Fll	frame_length_lines is the number of lines per frame	fll = h + vbl
Fr	Max frame rate of camera sensor	Determined by camera sensor datasheet
W	Width of camera sensors resolution	Determined by camera sensor datasheet
H	Height of camera sensors resolution	Determined by camera sensor datasheet
Vbl	Vertical blanking lines	Determined by camera sensor datasheet
Hbl	Horizontal blanking lines	Determined by camera sensor datasheet
b/p	▪ Bits per pixel ▪ Determined by format type ▪ Example: It takes 2 pclk for 1 pixel for uyvy thus 2 bits/pixel	Determined by camera sensor output format

4 AIS server customization

4.1 Deferred input detection

One of the main features that is enabled by default is deferred input detection to optimize for early camera availability in a multicamera system.

As the AIS Engine initializes, it defers the detection and initialization of input device drivers to a background thread. Without this feature, the AIS Engine initialization would block until all inputs are detected.

As input devices are detected, their sources are advertised and clients can open streams and start a session while the engine may still be detecting the remaining input devices.

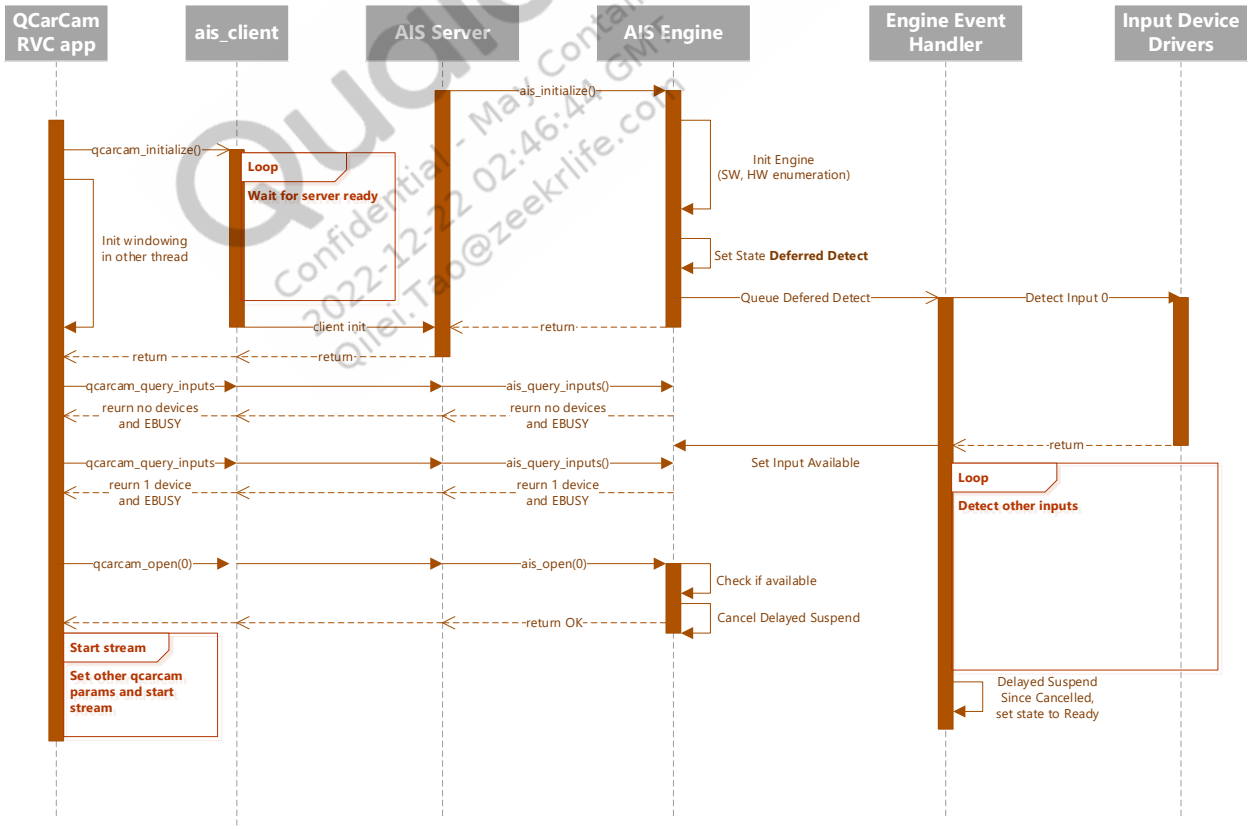

Figure 4-1 Deferred input detection sequence

This feature is enabled by default through the `qcarcam_init_t` flags parameter to `ais_initialize()` in `ais_server.c`

```
qcarcam_init_t qcarcam_init = {
    .flags = AIS_INITIALIZE_DEFER_INPUT_DETECTION |
             AIS_INITIALIZE_POWER_INPUT_GRANULAR
};
```

4.2 Input device granular power control

When enabled, the power control of input devices is done at a more granular level. On the first client of an input device, only that input device is powered on. Similarly, on the last client of a particular device, only that device is powered off.

If disabled, on the first overall client, we turn power on all input devices. The first client might incur a large cost to wait for powering on all input devices.

This feature is enabled by default through the `AIS_INITIALIZE_POWER_INPUT_GRANULAR` flag to `qcarcam_init_t` in `ais_server.c`

5 AIS CameraConfig customization

AIS customizations are primarily handled by the `CameraConfig` module. This module implements the `CameraConfig` interface which details the board configuration, engine settings, and QCarCam logical input mapping.

The interface is defined in `CameraConfig.h` (`ais/CameraConfig/inc`).

The `CameraConfig` module is compiled into a library that can either be statically linked or dynamically loaded at runtime (see Chapter 11).

To facilitate testing or development, the `CameraConfig` module also supports the dynamic parsing of an XML configuration file at runtime (see Section 5.6).

The reference `CameraConfig` module already defines and implements configurations for supported ADP platforms (SA6155/SA8155/SA8195). Customers can use these as a reference to start customization.

5.1 CameraConfig Interface

```
/**
 * This structure defines an interface for camera board specific
configurations
 * and methods.
 */
typedef struct
{
    /**
     * This method initializes the camera config
     * @return 0 SUCCESS and -1 otherwise
     */
    int (*CameraConfigInit)(void);

    /**
     * This method deinits the camera config
     * @return 0 SUCCESS and -1 otherwise
     */
    int (*CameraConfigDeInit)(void);

    /**
     * This method returns the CameraBoardType for the current hardware
platform.
     * @return CameraBoardType Camera board configuration information.
     */
```

```
    CameraBoardType const* (*GetCameraBoardInfo)(void);

    /**
     * This method returns the Camera Board Config version
     * @return version
     */
    int (*GetCameraConfigVersion)(void);

    /*
     * This method returns the array of channel config info and its
     * size in no. of elements (channels).
     */
    int (*GetCameraChannelInfo)(CameraChannelInfoType const
**ppChannelInfo, int *nChannels);
} ICameraConfig;
```

The CameraConfig library must expose a function `GetCameraConfigInterface` of type `GetCameraConfigInterfaceType` to return the `ICameraConfig` structure above.

```
/*
 * Defined type for method GetCameraConfigInterface.
 */
typedef ICameraConfig const* (*GetCameraConfigInterfaceType)(void);
```

5.2 CameraConfig board configuration

The `CameraBoardType` describes the hardware configuration of what is connected to the Camera subsystem, how it is connected, and AIS Engine settings.

```
/**
 * This structure defines the board specific configurable items related
to the
 * camera subsystem.
 */
typedef struct
{
    CameraHwBoardType boardType;

    /**< MultiSoc Enviroment */
    uint32 multiSocEnable;

    /**< Board type name */
    char boardName[MAX_CAMERA_DEVICE_NAME];

    /**< Camera configurations */
    CameraSensorBoardType camera[MAX_NUM_CAMERA_INPUT_DEVS];

    /**< I2C device configurations*/
    CameraI2CDeviceType i2c[MAX_NUM_CAMERA_I2C_DEVS];
```

```
    /**< Engine settings */
    CameraEngineSettings engineSetting;
} CameraBoardType;
```

5.3 CameraSensorBoardType

Each input device is described as a `CameraSensorBoardType`. It includes descriptions for each input device such as a unique device ID, CSI configuration, I2C configuration, GPIO and interrupt configuration, driver library loading information, and default driver configuration.

```
/**
 * This structure defines the board specific configurable items for a
given
 * camera sensor.
 */
typedef struct
{
    /**< unique device id */
    CameraDeviceIDType devId;

    /**< Driver config */
    CameraDeviceConfigType    devConfig;

    /**< Driver loading info */
    CameraDeviceDriverInfoType  driverInfo;

    /**< Driver detection thread id */
    uint32 detectThrdId;

    /**< CSI info */
    CameraCsiInfo csiInfo;

    /*GPIO config*/
    CameraGPIOPinType gpioConfig[CAMERA_GPIO_MAX];

    /*Interrupt config*/
    CameraSensorGPIO_IntrPinType intr[MAX_CAMERA_DEV_INTR_PINS];

    /**< I2C port */
    CameraI2CPortType i2cPort[SOC_ID_MAX];

} CameraSensorBoardType;
```

Each I2C bus is described as a `CameraI2CDeviceType`.

```
/**
 * This structure describes I2C Devices
 */
typedef struct {
    CameraI2CType i2ctype;      /**< CCI or I2C port */
    uint32 device_id;           /**< CCI device number */
    uint32 port_id;             /**< I2C port number */
    CameraGPIOPinType sda_pin;  /**< I2C SDA pin configuration */
    CameraGPIOPinType scl_pin;  /**< I2C SCL pin configuration */
    char i2cDevname[MAX_I2C_DEVICE_NAME];    /**< i2c port used by
sensor */
} CameraI2CDeviceType;
```

The table below contains all instances and attributes for setting `CameraBoardType` correctly. You can also refer to CameraConfig.h.

Instance	Attributes	Descriptions
`camera`	Top camera instance for all camera devices to be defined below	

	`devId`	Unique Device ID assigned to each input device.
		Ideally can enumerate starting with CAMERADEVICEID_INPUT_0 but in theory can be any unique number.
		
		Gen 3 default configuration is as follows:
		`CAMERADEVICEID_INPUT_0 : MAX9296 on CSI0`
		`CAMERADEVICEID_INPUT_1 : MAX9296 on CSI1`
		`CAMERADEVICEID_INPUT_2 : MAX9296 on CSI2`
		`CAMERADEVICEID_INPUT_3 : MAX9296 on CSI3`

	`devConfig`	Device configuration initialization parameters that are passed as an argument to the device driver instance. This includes parameters such as the subdevId to distinguish multiple devices of the same driver. If also includes default configuration settings that the device driver may make use of such as number of sensors, operation mode, type of deserializer, type of sensor, default mode for each sensor, etc. The input device driver is free to interpret this information for its own purposes. Refer to `CameraDeviceConfigType` for full definitions.
	`driverInfo`	Specifies the input device library to be loaded for this device and the Open function symbol exposed by the library. Refer to `CameraDeviceDriverInfoType` and section below for more details.
	`detectThrdId`	Specifies the thread ID which will perform the detection of the input device. Refer to Section 5.3.3 for more details on detection schemes

Instance	Attributes	Descriptions
	csiInfo	Specifies the CSI connection information such as the CSI id, number of lanes, and lane mapping. Also specified if a CSI is secure. This must be configured to match the programming in ARM® TrustZone® for secure/content protection feature. It also specifies the and CSI to IFE mapping. By default, it is a 1:1 mapping of CSI to IFEs. However, there may be use cases that a single CSI may be routed to multiple IFEs and this allows this customization. Refer to CameraCsiInfo or section below for more details.
	gpioConfig	Table of GPIO configurations for the device. Each GPIO id can be programmed with a particular GPIO pin and configuration. The device driver would use the GPIO id to refer to it. Refer to CameraGPIOPinType
	intr	Specifies the interrupt pins and their configuration. Each interrupt pin is referred to using a unique GPIO id Refer to CameraSensorGPIO_IntrPinType
I2C	i2cDevname	Name of device
	i2ctype	Type of I2C device (CameraI2CType). Only CAMERA_I2C_TYPE_CCI supported for now.
	device_id	In case of CCI, the CCI Core ID.
	port_id	In case of CCI, the master ID within a CCI Core.
	sda_pin	SDA pin configuration
	scl_pin	SCL pin configuration

5.3.1 CameraDeviceDriverInfoType

This section details the driverInfo parameter (CameraDeviceDriverInfoType).

If dynamic loading is enabled, the library will be dynamically loaded and the open function symbol will be loaded. In case of static linking, AIS will use the function pointer to the open function.

Instance	Attributes	Description
driverInfo	deviceCategory	▪ Specifies device category ▪ Must be set to CAMERA_DEVICE_CATEGORY_SENSOR for input devices
	strDeviceLibraryName	Filename of library that will be dynamically loaded through dlopen for this instance of the device
	strCameraDeviceOpenFn	▪ String of open function library symbol that will be dynamically loaded through dlsym ▪ Must be implemented by the library
	pfnCameraDeviceOpen	Function pointer to the open function if using static linking instead of dlsym

5.3.2 CameraCsiInfo

This section will detail the `csiInfo` parameter.

Instance	Attributes	Description
`csiInfo`	`csiId`	Physical SOC CSI port to which the input device is connected
	`isSecure`	Set to 1 if the CSI is secure or content protected
	`numLanes`	Number of MIPI CSI lanes physically connected
	`laneAssign`	▪ CSI Lane mapping configuration where each nibble (4 bits), starting from the LSB up to numLanes, assigns the physical lane ▪ Default is 0x3210 (no swaps) ▫ Example: 0x3120 can be set in case of lane 1 and 2 are swapped
	`ifeMap`	▪ Mappings to IFE interfaces where each IFE index is expressed as a nibble (4 bits) ▪ IFE priority is set in priority starting from the nibble at the LSB ▫ Example: 0x31 has IFE 1 then IFE 3
	`numIfeMap`	▪ Number of IFE mappings in ifeMap (number of nibbles valid from LSB) ▪ If 0, ifeMap is ignored and the IFE mapping will default to csiId

5.3.3 Detection order and schemes

Input devices are probed and detected in the same order as the `CameraSensorBoardType camera[MAX_NUM_CAMERA_INPUT_DEVS]` list.

In addition, our AIS Engine instantiates four threads to perform this operation in parallel. Each input device instance can specify a detectThrdId to specify which of these threads will perform the operation. It is recommended to assign the threadId based on a unique I2C/CCI bus. Assigning multiple devices to the same thread will queue them to that thread so that they are probed sequentially.

If two devices are on the same bus, it is preferable (some exceptions may exist) to assign them to the same thread for the following reasons:

- They will compete on the same bus and be similar to running sequentially on one thread.
- There might be a case where there are inherent dependencies that require two devices to be probed sequentially.
  - For example, on Gen3 with MAX9296 daughter cards, there is an inherent dependency that the initialization of devices on the same bus is done sequentially so that we can remap the sensor/serializers to unique I2C addresses. If not, there would be conflict on the bus as to which serializer we are communicating with during initialization because all serializers/sensors would power on with the same I2C address.

The following figures illustrate different detection schemes that can be configured in CameraConfig and how they translate to the AIS Engine detection threads.

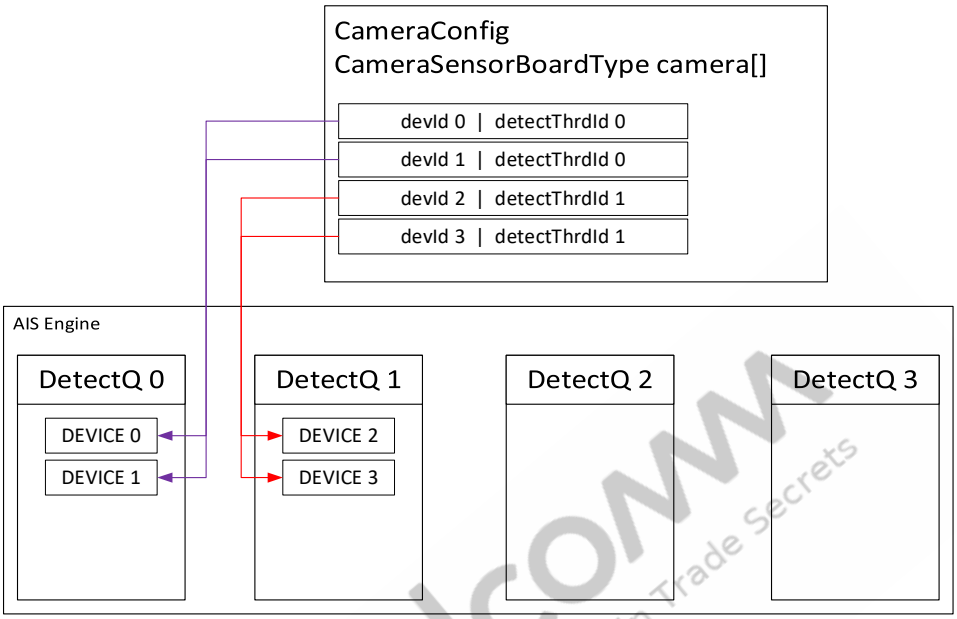

Figure 5-1 Camera Config: Parallel detection scheme for SA8155

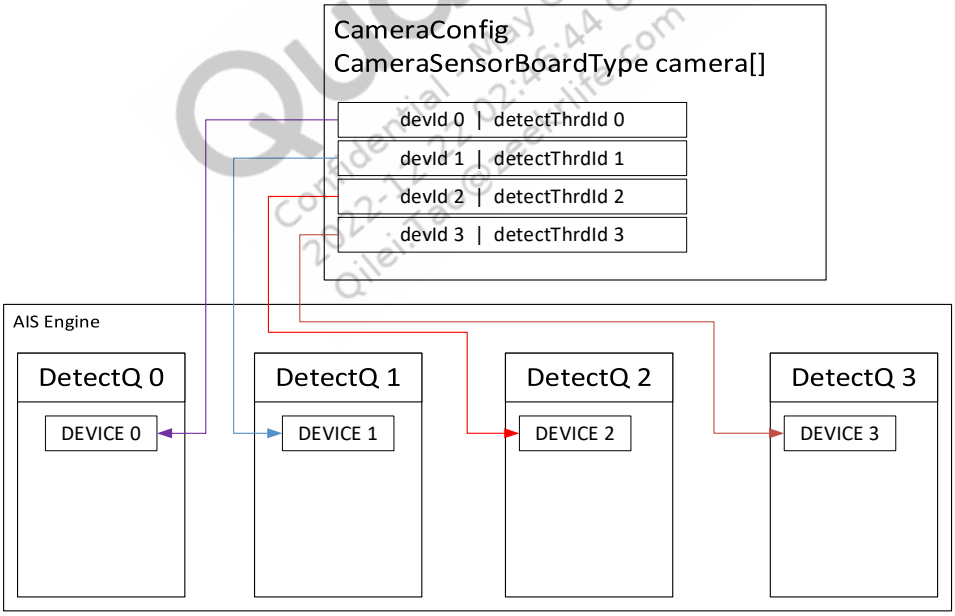

Figure 5-2 Parallel detection scheme using all four detect threads

The following figure shows the AIS Engine initialization flow.

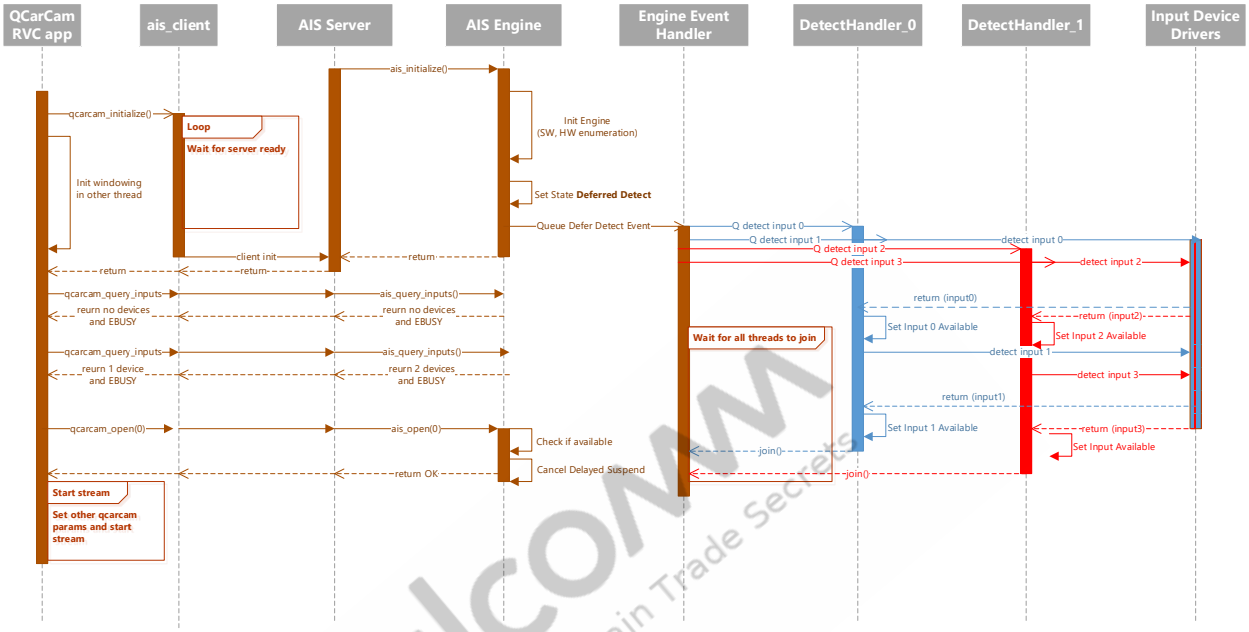

Figure 5-3 Early RVC client, AIS initialization, and parallel input device detection call sequence

5.3.4 Example code

The following example shows a MAX9296_0 bridge chip from the SA8155 Topology (see Figure 2-1).

```
    CameraBoardType cameraBoardDefinition =
    { // Configuration
        .camera =
         {
          {
            .devId = CAMERADEVICEID_INPUT_0,
            .detectThrdId = 0,
            .driverInfo = {
                .deviceCategory = CAMERA_DEVICE_CATEGORY_SENSOR,
                .strDeviceLibraryName = "ais_max9296",
                .strCameraDeviceOpenFn = "CameraSensorDevice_Open_max9296",
#ifdef AIS_BUILD_STATIC_DEVICES
                .pfnCameraDeviceOpen = CameraSensorDevice_Open_max9296,
#endif
            },
            .devConfig = {
                    /*2x AR0231 YUV*/
                    .subdevId = 0,
                    .type = 1,
```

```
                        .numSensors = 2,
                        .powerSaveMode = CAMERA_POWERSAVE_MODE_POWEROFF,
                        .sensors = {
                                {.type = 2, .snsrModeId = 0},
                                {.type = 2, .snsrModeId = 0}
                        }
                },
                .csiInfo = {
                    .csiId = 0,
                    .isSecure = 0,
                    .numLanes = 4,
                    .laneAssign = CAMERA_DEFAULT_CSI_LANE_ASSIGN,
                    .ifeMap = 0x0,
                    .numIfeMap = 1
                },
                .gpioConfig = {
                        [CAMERA_GPIO_RESET] = {
                            .num = 21,
                            .config = GPIO_PIN_CFG(GPIO_OUTPUT, GPIO_PULL_UP,
GPIO_STRENGTH_2MA, 0),
                        }
                },
                .i2cPort = {
                    .i2ctype = CAMERA_I2C_TYPE_CCI,
                    .device_id = 0,
                    .port_id = 0,
                },
                .intr = {
                        {.gpio_id = CAMERA_GPIO_INTR, .pin_id = 13, .intr_type
= CAMERA_GPIO_INTR_TLMM,
                                .trigger =
CAMERA_GPIO_TRIGGER_FALLING, .gpio_cfg = {0, 0x30, 0, 0}},
                        {.gpio_id = CAMERA_GPIO_INVALID,},
                        {.gpio_id = CAMERA_GPIO_INVALID,},
                }
            },
```

5.3.5 Content protected camera

The CSI for content protection must be set as secure in both TrustZone and Camera Config.

TrustZone customization

Modify secure_camera_config.xml for the particular SOC.

- SA8155
  `trustzone_images\ssg\securemsm\seccam\hw\855\inc\secure_camera_config.xml`

- SA8195
  `trustzone_images\ssg\securemsm\seccam\hw\1000\inc\secure_camera_config.xml`

1. Set the list of CSIs that are protected.

   For example, to set CSI1 and 3 as protected:

   ```
   <var_seq name="cam_phys_seq" type=DALPROP_DATA_TYPE_UINT32_SEQ> 1, 3,
   end</var_seq>
   ```

2. Enable the CSI protection property by setting it to 1.

   ```
   <props name="enable_cam_phys_boot_protection"
   type=DALPROP_ATTR_TYPE_UINT32>
               1
           </props>
   ```

3. Recompile `devcfg_auto.mbn` image

CameraConfig customization

In CameraConfig, set the csiInfo.isSecure property of the input device to “1” if the CSI is protected.

For example, the following code shows a section of input device 0, that is connected to a protected CSI1.

```
.devId = CAMERADEVICEID_INPUT_0,
...
csiInfo = {
                .csiId = 1,
                .isSecure = 1,
                .numLanes = 4,
                .laneAssign = CAMERA_DEFAULT_CSI_LANE_ASSIGN,
                .ifeMap = 0x0,
                .numIfeMap = 1
            },
```

5.4 QCarCam logical input mapping

In AIS, mapping QCarCam IDs to particular streams is done through a mapping table defined by `CameraChannelInfoType` (see Figure 2-1).

The client application will use the logical QCarCam ID to refer to a particular source. A single QCarCam ID can refer to multiple sources for use cases such as paired input stream (Dual-CSI).

```
/**
 * This structure defines the mapping of unique qcarcam descriptor to
 * input sources as well as the default operation mode.
 */
typedef struct
{
```

```
    /**
     * unique qcarcam inputId that maps to <devId, srcId>
     */
    uint32 aisInputId;

    /**
     * unique <devId, srcId> of input id and default operation mode
     */
    struct
    {
        uint32 devId;
        uint32 srcId;
    }inputSrc[MAX_CHANNEL_INPUT_SRCS];

    qcarcam_opmode_type opMode;

} CameraChannelInfoType;
```

The SA8155 reference uses the following mapping definition:

```
static const CameraChannelInfoType InputDeviceChannelInfo[] =
{
    { /*max9296_0*/
        .aisInputId = 0,
        .inputSrc = {
            { .devId = CAMERADEVICEID_INPUT_0, .srcId = 0 }
        },
        .opMode = QCARCAM_OPMODE_RAW_DUMP
    },
    { /*max9296_0*/
        .aisInputId = 1,
        .inputSrc = {
            { .devId = CAMERADEVICEID_INPUT_0, .srcId = 1 }
        },
        .opMode = QCARCAM_OPMODE_RAW_DUMP
    },
    { /*max9296_1*/
        .aisInputId = 2,
        .inputSrc = {
            { .devId = CAMERADEVICEID_INPUT_1, .srcId = 0 }
        },
        .opMode = QCARCAM_OPMODE_RAW_DUMP
    },
    { /*max9296_1*/
        .aisInputId = 3,
        .inputSrc = {
            { .devId = CAMERADEVICEID_INPUT_1, .srcId = 1 }
        },
```

```
            .opMode = QCARCAM_OPMODE_RAW_DUMP
        },
        { /*max9296_2*/
            .aisInputId = 4,
            .inputSrc = {
                { .devId = CAMERADEVICEID_INPUT_2, .srcId = 0 }
            },
            .opMode = QCARCAM_OPMODE_RAW_DUMP
        },
        { /*max9296_2*/
            .aisInputId = 5,
            .inputSrc = {
                { .devId = CAMERADEVICEID_INPUT_2, .srcId = 1 }
            },
            .opMode = QCARCAM_OPMODE_RAW_DUMP
        },
        { /*max9296_3*/
            .aisInputId = 8,
            .inputSrc = {
                { .devId = CAMERADEVICEID_INPUT_3, .srcId = 0 }
            },
            .opMode = QCARCAM_OPMODE_RAW_DUMP
        },
        { /*max9296_3*/
            .aisInputId = 9,
            .inputSrc = {
                { .devId = CAMERADEVICEID_INPUT_3, .srcId = 1 }
            },
            .opMode = QCARCAM_OPMODE_RAW_DUMP
        }
    #endif
    };
```

5.5 AIS engine settings

The `CameraEngineSettings` structure sets AIS Engine-specific settings.

```
typedef struct
{
    /**< Maximum number of buffers per client */
    uint32 numBufMax;

    /**< power suspend/resume manager policy */
    CameraPowerManagerPolicyType powerManagementPolicy;

    /**< customize matching functions for events. This matching criteria is
          used to determine impacted clients for a particular event. */
```

```
    CameraConfigCustomMatchFunc
customMatchFunctions[CAMERA_CONFIG_EVENT_NUM];

    /**< How many milliseconds to wait between calling start and stop of a
     * user context while in recovery.
    */
    uint32 recoveryRestartDelay;

    /**< For how many milliseconds to wait to receive a frame done event,
     * after calling Start() on a user context. Not receiving this event in
     * this time frame means that the recovery has failed.
     */
    uint32 recoveryTimeoutAfterUsrCtxtRestart;

    /**< For how many milliseconds to wait after a recovery failure before
     * attempting another recovery.
     */
    uint32 recoveryRetryDelay;

    /**< Maximum number of times the engine will try to recover from a
     * single error.
     */
    uint32  recoveryMaxNumAttempts;
} CameraEngineSettings;
```

5.5.1 PowerManagement policy

```
/**
 * camera power management policies supported
 *
 * +---------------------------------------+------------------+------------------+
 * |              PM Policy                |  SOC Pwr/Clks    |    Bridge Chip   |
 * +---------------------------------------+------------------+------------------+
 * | CAMERA_PM_POLICY_NO_LPM               | First/Last Client | First/Last Client |
 * | CAMERA_PM_POLICY_LPM_EVENT_FOR_INPUTS | First/Last Client | PM Event          |
 * | CAMERA_PM_POLICY_LPM_EVENT_ALL        | PM Event          | PM Event          |
 * +---------------------------------------+------------------+------------------+
 */
typedef enum
{
    CAMERA_PM_POLICY_NO_LPM = 0,
    CAMERA_PM_POLICY_LPM_EVENT_FOR_INPUTS,
    CAMERA_PM_POLICY_LPM_EVENT_ALL,
    CAMERA_PM_POLICY_TYPE_MAX
} CameraPowerManagerPolicyType;
```

5.5.2 AIS recovery customization

For AIS Recovery, the following settings can be adjusted:

- `customMatchFunctions`: Customize handling of particular AIS engine errors and which streams will be affected by particular error and severity of such error.
- `recoveryRestartDelay`: Number of milliseconds to wait before restarting a context during recovery.
- `recoveryTimeoutAfterUsrCtxtRestart`: After restarting the stream, time in milliseconds to wait for a new frame. If no new frames, recovery failed.
- `recoveryRetryDelay`: Time in milliseconds to wait before attempting another retry if recovery failed.
- `recoveryMaxNumAttempts`: Maximum number of retries.

5.6 CameraConfig XML

To speed up development and customization work, the `CameraConfig` module supports the dynamic parsing of an XML file to override compiled settings.

The XML schema and a sample SA8155 XML file are included in `ais/CameraConfig/xml`.

The XML path on the target is defined by `CAMERA_CONFIG_XML_FILE` macro in CameraConfig.h.

6 AIS camera sensor driver customization

When writing a new camera sensor or bridge chip driver in AIS, refer to the existing reference drivers found in `ais/ImagingInputs/SensorLibs/`.

The sensor drivers must implement two interfaces that the AIS sensor framework will use to communicate with the driver:

- `CameraDevice`
- `sensor_lib`

6.1 CameraDevice interface

The sensor driver must define and expose an open function based on the CameraDeviceOpenType function prototype. This is the entry point to the library and dynamic symbol that is configured and loaded by CameraConfig. The sensor driver must call into CameraSensorDevice_Open and provide its own SensorOpenLibType open function.

The Sensor Driver Framework will call the provided SensorOpenLibType open function to open an instance of the driver.

This call flow ensures that a separate instance of the driver framework is loaded for each device driver while also enabling the static linking of all device driver libraries.

6.2 sensor_lib Interface

The `sensor_lib` API is defined in sensor_lib.h (`ais/ImagingInputs/ImagingInputDriver/inc`). This section will cover requirements for AIS.

6.2.1 SensorOpenLibType

The entry point to this interface is defined by `SensorOpenLibType` and called by the Sensor Driver Framework. The two arguments:

- `ctrl`: Pointer to sensor driver framework context. This is to be provided by the sensor driver library as an argument to any framework API.
- `arg`: pointer to `sensor_open_lib_t`. This structure contains CameraConfig deviceConfig parameters that can be used by the driver to initialize its instance.

The return value of the open function must be a pointer to the `sensor_lib_t` object representing the sensor driver library. This object will be passed as an argument to the sensor driver APIs from the framework.

There are two ways to store additional driver-specific context structure with `sensor_lib_t.priv_ctxt`.

- In the following example, users can make `sensor_lib_t` the first member in the context structure and cast it as a `sensor_lib_t`. `max9296_context_t` can then be cast to `sensor_lib_t`.
- `sensor_lib_t` has member `priv_ctxt` where the `max9296_context_t` pointer can be stored.

Example

`max9296_context_t` is defined as:

```
struct max9296_context_t
{
  /*must be first element*/
  sensor_lib_t sensor_lib;

  /*ADD SOME OTHER CONTEXT MEMBERS HERE*/

  /*framework object*/
  void* ctrl;
};
```

6.2.2 sensor_lib_t

The `sensor_lib_t` structure represents the sensor driver library. The framework uses information in the `sensor_lib_t` object in its operation.

The following table describes the required and optional parameters in this object as well as their effect on call flows.

Table 6-1 Required and optional parameters

Instance	Attributes	Description
`priv_ctxt`	–	▪ Private context of the sensor driver library ▪ Unused by framework and can be used by driver to store private information
`sensor_close_lib`	–	▪ Close function for the library ▪ Must be provided to clean up the library
`sensor_slave_info`	`sensor_name`	Sensor name used by framework for logging
	`camera_id`	▪ Camera ID must be initialized to `sensor_open_lib_t.cameraId` parameter ▪ Used by framework to uniquely identify the `sensor_lib` instance
	`slave_addr`	Default slave address of device

Instance	Attributes	Description
	i2c_freq_mode	▪ I2C frequency mode ▪ Typically set to SENSOR_I2C_MODE_CUSTOM to enable clock stretching
	addr_type	Default number of bytes for I2C address
	data_type	Default number of bytes for I2C data
	is_init_params_valid	Legacy (unused)
	sensor_id_info	Specifies sensor ID information for detection (sensor_id, sensor_id_reg_addr and sensor_id mask) **NOTE:** This is used to detect sensor if custom detect function is not defined.
	power_setting_array	Sets power up and power down sequence for the sensor **NOTE:** This sequence will be used for power on, power off, power suspend, and power resume if a custom function is not defined.
num_channels	–	Number of channels needed which correlates to the number of supported sensors
channels	output_mode	▪ Details channel's current output mode ▪ Includes color format, resolution, frame rate, and VC/DT
	num_subchannels	▪ Number of subchannels defined for each channel ▪ Typical use is one subchannel per channel **NOTE:** Multiple subchannels for single channel is unsupported for now.
	subchan_layout	Details the src_id of the subchannel and layout within the channel
num_subchannels	–	Total number of defined subchannels
subchannels	src_id	Defines source identifier for the subchannel **NOTE:** This is referenced in the subchan_layout of the channel and the srcId in CameraConfig QCarCam ID Mapping.
	modes	▪ List of supported modes for the source ▪ Includes color format, resolution, VC/DT, frame rate, and crop information
	num_modes	Number of modes defined

Instance	Attributes	Description
src_id_enable_mask	-	▪ Mask of available sources ▪ Used by framework to check availability of the source when it is requested
sensor_output	-	Legacy (unused)
sensor_num_frame_skip	-	Legacy (unused)
sensor_num_HDR_frame_skip	-	Legacy (unused)
sensor_max_pipeline_frame_delay	-	Legacy (unused)
pixel_array_size_info	-	Legacy (unused)
out_info_array	-	Legacy (unused)
csi_params	lane_cnt	Number of CSI lanes
	settle_cnt	See Section 3.4
	combo_mode	Should only be set to 1 if using 2 lane + 1 combo mode **NOTE:** This mode is not supported. Must be set to 0.
	is_csi_3phase	▪ Specifies CSI type ▪ Set to 0 for DPHY (2 phase) ▪ Set to 1 for CPHY (3 phase)
sensor_close_lib	-	Specifies sensor_close_lib function for the camera bridge chip driver
exposure_func_table	-	▪ Exposure control function table ▪ Should only be used if bridge chip supports exposure control
sensor_capability	-	▪ Bit mask for supported features of the driver ▪ Example: If sensor and bridge chip support exposure then the value is set to 1 left bit shifted by the SENSOR_CAPABILITY_EXPOSURE_CONFIG. For full list of capabilities, see sensor_lib.h.
init_settings_array	-	Initialization sequence if custom init function is not defined
res_settings_array	-	Mode initialization sequence if custom set channel mode is not defined
start_settings	-	Start stream sequence if custom start function not defined
stop_settings	-	Stop stream sequence if custom stop function not defined
sensor_custom_func	-	Custom functions to override default settings (defined in Table 6-2)
use_sensor_custom_func	-	▪ Defines if the driver should use the custom functions referenced in sensor_custom_func ▪ Set to TRUE if sensor_custom_func are defined

Table 6-2 Custom functions

Custom Function	Description
sensor_set_platform_func_table	Sets sensor_platform_func_table_t to sensor driver. These functions can be used by the driver for platform operations such as I2C, GPIO, and interrupt operations. It also defines event callback function for sensor events, such as, change of lock status, errors, frame freeze, etc.
sensor_power_on sensor_power_off sensor_power_suspend sensor_power_resume	Custom function to power on/off or suspend/resume. **NOTE:** If not defined, power_setting_array is used by the framework as the on/off sequence
sensor_detect_device	Custom function to detect device. **NOTE:** If not defined, sensor_slave_info and sensor_id_info is used by the framework to detect
sensor_detect_device_channels	Custom function to detect subdevices. **NOTE:** If not defined, will not be called.
sensor_init_setting	Custom init function. **NOTE:** If not defined, init_settings_array will be used.
sensor_set_channel_mode	Custom function to set mode for particular source.
sensor_start_stream	Custom start stream function. **NOTE:** If not defined, start_settings will be used.
sensor_stop_stream	Custom stop stream function. **NOTE:** If not defined, stop_settings will be used.
sensor_s_param	Custom function to set parameter to a source. Used to set parameters such as AEC, color hue/saturation, etc.
sensor_g_param	Custom function to query a parameter.
sensor_config_resolution	Custom function to set resolution. Used for devices such as HDMI input.
sensor_query_field	Custom function to query field information for interlaced content.
sensor_process_frame_data	Custom function to process or parse raw image data. This is typically used to parse embedded data that may be part of the image data. Information can also be used to determine frozen frame or other events and propagate them as event callbacks to the clients.
sensor_error_recovery	Custom function to handle any specific routine that might be required on performing recovery.

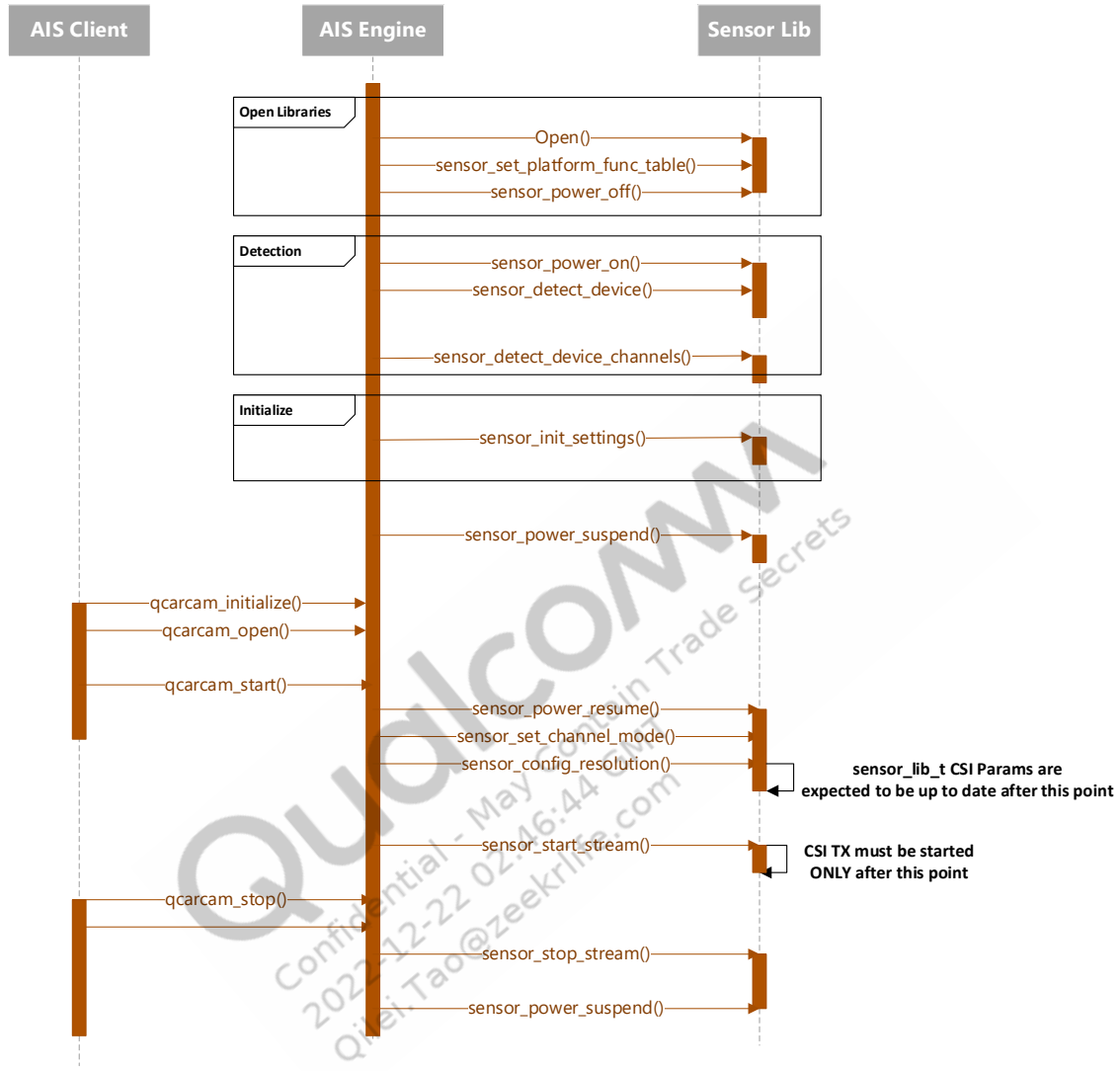

Figure 6-1 Sensory library call sequence

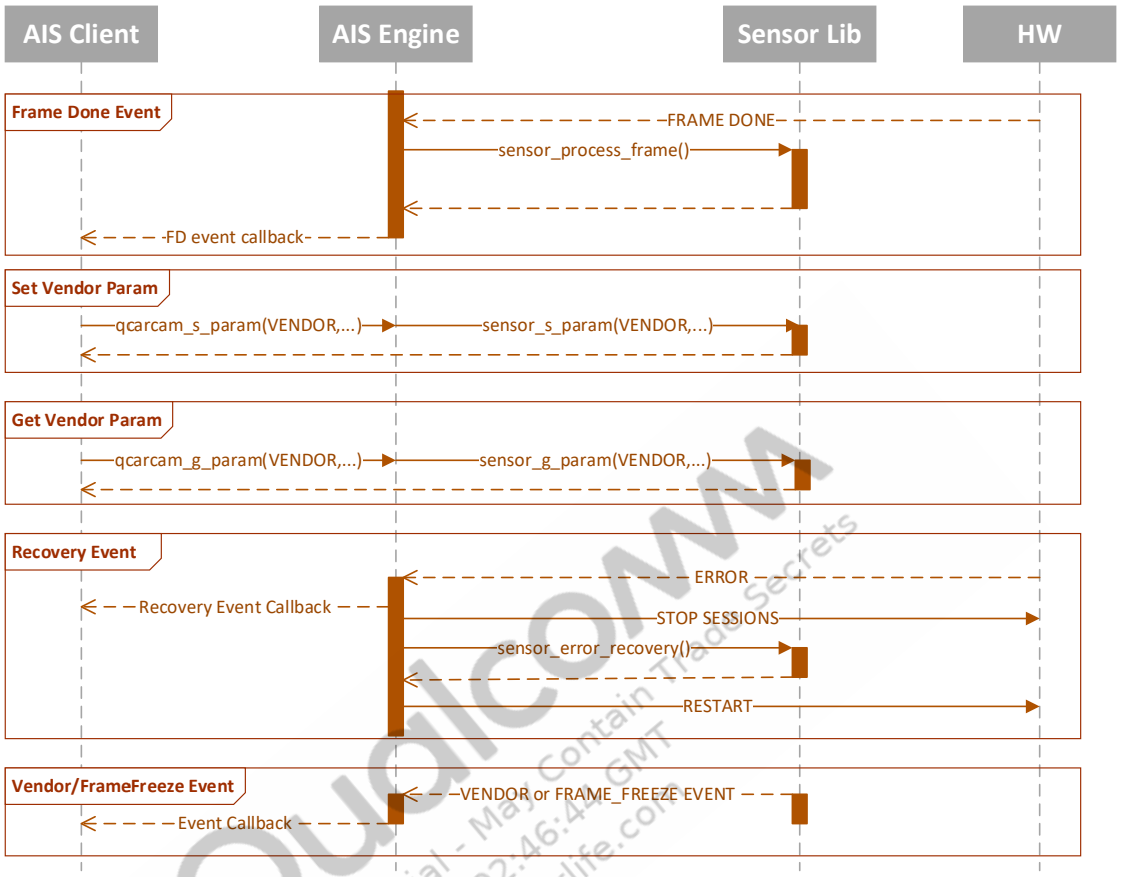

Figure 6-2 Sensory library event calls

6.2.3 I2C bulk write-read operation using CCI customization

This feature provides support for CCI hardware to perform burst read and burst write. It also provides support for a custom burst write and then burst read operation without sending an I2C stop in between, which can be used to support request-response protocol over I2C.

Burst read and burst write can support up to 65535 bytes of data to be read or written over an I2C lane. Select clock mode and clock configuration in such a way that clock stretch bit is set. Customer-specific sensor libraries, like max9296_lib.c contain struct sensor_lib_t with member.sensor_slave_info.i2c_freq_mode. This 'i2c_freq_mode' can be configured with the following possible values:

- SENSOR_I2C_MODE_STANDARD
- SENSOR_I2C_MODE_FAST
- SENSOR_I2C_MODE_CUSTOM
- SENSOR_I2C_MODE_FAST_PLUS

Of these, except for MODE_STANDARD, the remaining modes have clock stretch bit set. By default, MODE_CUSTOM is selected. The available APIs are below.

Burst write

To perform burst write, refer to sample function max9296_test_i2c_bulk_write() in max9296_lib.c. It is accessed using function pointer .i2c_slave_write_bulk.

Function:

```
int (*i2c_slave_write_bulk)(void* ctxt,
            unsigned short slave_addr,
            struct camera_i2c_bulk_reg_setting *reg_setting,
            boolean exec_pending_i2ccmds);
```

Parameter	Description
ctxt*	Void pointer used to hold the pointer to the SensorDriver class. It contains the public variables: ▪ SensorPlatform* m_pSensorPlatform ▪ CameraSensorDevice* m_pDeviceContext ▪ sensor_lib_t* m_pSensorLib
slave_addr	Unsigned short parameter containing the slave address.

reg_setting*	struct camera_i2c_bulk_reg_setting {
	unsigned int *reg_data;
	unsigned int reg_addr;
	unsigned int size;
	enum camera_i2c_reg_addr_type addr_type;
	enum camera_i2c_data_type data_type;
	};

exec_pending_i2ccmds	Bool value TRUE. When set to TRUE, the CCI bulk operation will wait for any previous CCI requests to be completed. In the CCI driver, it will wait for queue 1 to execute its commands before queue 0 locks the bus for the requested bulk opn, which is time consuming

Return:

0 or CAMERA_SUCCESS when successful. A positive value for failure.

Burst read

To perform burst read, refer to sample function max9296_test_i2c_bulk_read() in max9296_lib.c. It is accessed using function pointer .i2c_slave_read_bulk.

Function:

```
int (*i2c_slave_read_bulk)(void* ctxt,
            unsigned short slave_addr,
            struct camera_i2c_bulk_reg_setting *reg_setting,
            boolean exec_pending_i2ccmds);
```

Parameter	Description
ctxt*	Void pointer used to hold the pointer to the SensorDriver class. It contains the public variables: ▪ SensorPlatform* m_pSensorPlatform ▪ CameraSensorDevice* m_pDeviceContext ▪ sensor_lib_t* m_pSensorLib
slave_addr	Unsigned short parameter containing the slave address.

reg_setting*	struct camera_i2c_bulk_reg_setting {
	unsigned int *reg_data;
	unsigned int reg_addr;
	unsigned int size;
	enum camera_i2c_reg_addr_type addr_type;
	enum camera_i2c_data_type data_type;
	};

exec_pending_i2ccmds	Bool value TRUE. When set to TRUE, the CCI bulk operation will wait for any previous CCI requests to be completed. In the CCI driver, it will wait for queue 1 to execute its commands before queue 0 locks the bus for the requested bulk opn, which is time consuming

Return:

0 or CAMERA_SUCCESS when successful. A positive value for failure.

Burst write then burst read without I2C stop

To perform burst write and then burst read without I2C stop, refer to sample function imx_gw5200_test_bulk_write_then_read () in imx_gw5200.c. It is accessed using function pointer .i2c_slave_bulk_write_then_read.

Function:

```
int(*i2c_slave_bulk_write_then_read)(void* ctxt,

            unsigned short slave_addr,

            struct camera_i2c_bulk_reg_setting *write_reg_setting,

            struct camera_i2c_bulk_reg_setting *read_reg_setting,

            boolean exec_pending_i2ccmds);
```

Parameter	Decription
ctxt*	Void pointer used to hold the pointer to the SensorDriver class. It contains the public variables: ▪ SensorPlatform* m_pSensorPlatform ▪ CameraSensorDevice* m_pDeviceContext ▪ sensor_lib_t* m_pSensorLib
slave_addr	Unsigned short parameter containing the slave address.

Parameter	Decription

write_reg_setting*	struct camera_i2c_bulk_reg_setting {
	unsigned int *reg_data;
	unsigned int reg_addr;
	unsigned int size;
	enum camera_i2c_reg_addr_type addr_type;
	enum camera_i2c_data_type data_type;
	};

read_reg_setting*	Same struct as above.
exec_pending_i2ccmds	Bool value TRUE. When set to TRUE, the CCI bulk operation will wait for any previous CCI requests to be completed. In the CCI driver, it will wait for queue 1 to execute its commands before queue 0 locks the bus for the requested bulk opn, which is time consuming.

Return:

0 or CAMERA_SUCCESS when successful. A positive value for failure

Currently this burst write and then burst read without I2C stop API is only supported on QNX.

6.3 Changes to enable CPHY

The sensor_lib_t.csi_params must be set to:

```
lane_cnt = 3;
lane_mask = 0x7;
is_csi_3phase = 1;
```

7 MAX9296 driver overview

This chapter details the `max9296_lib` sensor driver, which uses the MAXIM 9296 bridge chip on the Snapdragon ADP.

7.1 Topology

The reference MAX9296 driver supports a several sensors:

- AR0231 Bayer sensor
- AR0231 with AP0200 external ISP
- AR0820

The driver abstracts the interface to these sensors in a `max9296_sensor_t` structure.

```
/**
 * MAXIM Slave Description
 */
typedef struct
{
    maxim_sensor_id_t id;

    int (*detect)(max9296_context_t* ctxt, uint32 link);
    int (*get_link_cfg)(max9296_context_t* ctxt, uint32 link,
max9296_link_cfg_t* p_cfg);

    int (*init_link)(max9296_context_t* ctxt, uint32 link);
    int (*start_link)(max9296_context_t* ctxt, uint32 link);
    int (*stop_link)(max9296_context_t* ctxt, uint32 link);

    int (*calculate_exposure)(max9296_context_t* ctxt, uint32 link,
sensor_exposure_info_t* exposure_info);
    int (*apply_exposure)(max9296_context_t* ctxt, uint32 link,
sensor_exposure_info_t* exposure_info);

    int (*apply_hdr_exposure)(max9296_context_t* ctxt, uint32 link,
qcarcam_hdr_exposure_config_t* hdr_exposure);

    int (*apply_gamma)(max9296_context_t* ctxt, uint32 link,
qcarcam_gamma_config_t* gamma_info);

}max9296_sensor_t;
```

Individual sensors implement such functions, as needed. For example, AR0231 with external ISP sensor (ar0231_ext_isp.c / ar0231_ext_isp.h):

```
static max9296_sensor_t ar0231_ext_isp_info = {
    .id = MAXIM_SENSOR_ID_AR0231_EXT_ISP,
    .detect = ar0231_ext_isp_detect,
    .get_link_cfg = ar0231_ext_isp_get_link_cfg,

    .init_link = ar0231_ext_isp_init_link,
    .start_link = ar0231_ext_isp_start_link,
    .stop_link = ar0231_ext_isp_stop_link,
    .apply_exposure = ap0200_apply_exposure,
    .apply_gamma = ap0200_apply_gamma
};
```

7.1.1 Hardware topology

MAX9296 is set up to route frame data from LINK A to pipeline X and LINK B to pipeline Y as shown in the following illustration:

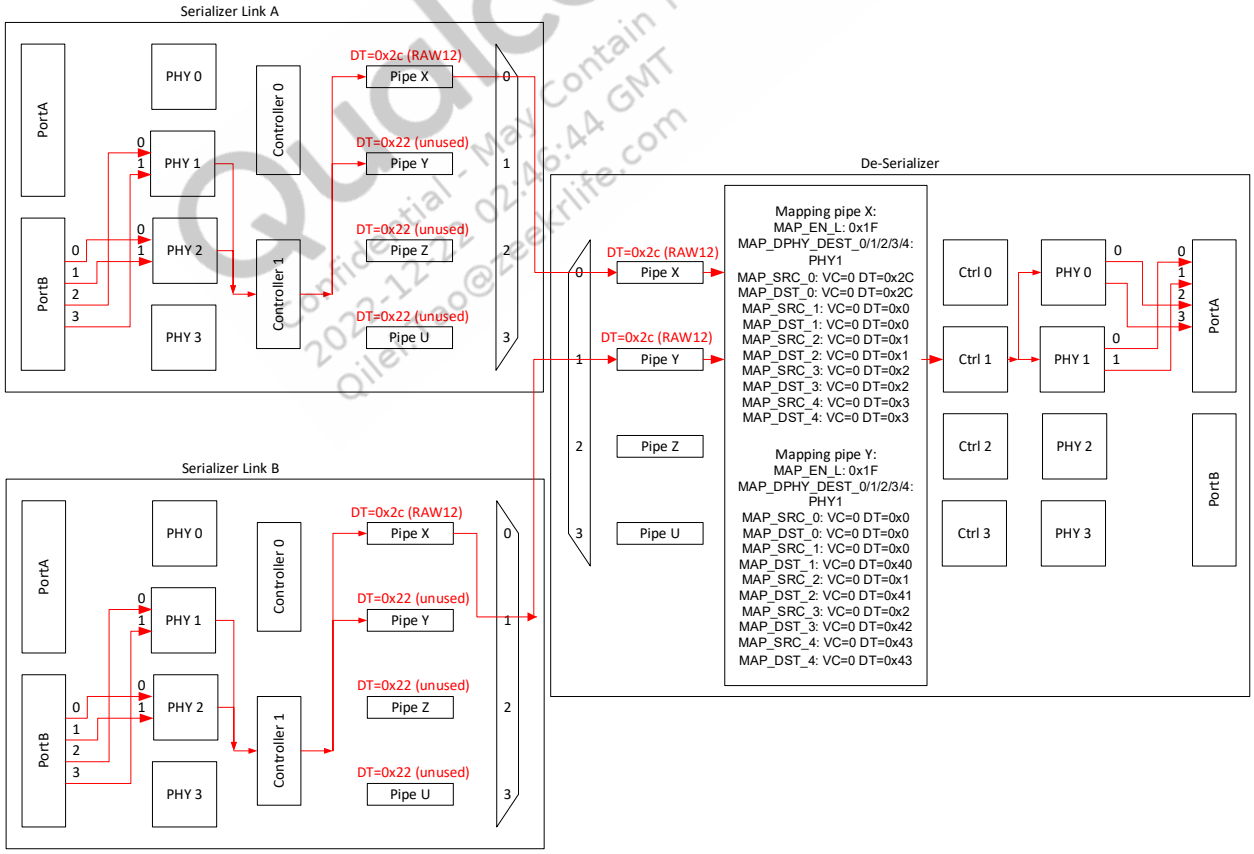

Figure 7-1 MAX9296 hardware topology

7.2 CSI parameters

The following example shows how to set settle count in max9296_lib.c (found in `ais/ImagingInputs/SensorLibs/max9296_lib/src/`):

```
.csi_params =
  {
    .lane_cnt = 4,
    .settle_cnt = 0xE,
```

You can also set parameters for I2C write delays in max9296_lib.h (found in `ais/ImagingInputs/SensorLibs/max9296_lib/src/`).

```
#define _reg_delay_ 0
```

7.3 Bridge chip slave address and model

Each camera bridge chip has a unique slave address which needs to be accessed over I2C via the camera bridge chip driver.

Below is the definition of the slave address for the MAX9296 bridge chip on CSI0 found in `max9296_lib.h` in `ais/ImagingInputs/SensorLibs/max9296_lib/src`.

```
#define MSM_DESER_0_SLAVEADDR        0x90
```

There are four MAX9296 bridge chips on the SA6155/SA8155 ADP reference platform – one on each of the CSIDs. If all four bridge chips are used, there will be four 8 bit slave address definitions in the bridge chip driver header file:

- 0x90 address
- 0x94 slave address
- 0xD0 slave address
- 0xD8 slave address

There are also associated serializer alias and sensor alias referenced for each bridge chip.

There should be a sensor model name defined in the camera bridge header file. The camera bridge header file should be named such that the name is the camera bridge chip followed by `_lib.h`. For example, if the camera bridge chip driver header is named `max9296_lib.h`, the model will be:

```
#define SENSOR_MODEL "max9296"
```

7.4 Channel setup

A channel is defined by a MIPI-CSI Virtual Channel (VC) and Data Type (DT) pair. The following sample code shows a channel that represents VC 1 and DT CSI RAW12.

```
#define CID_VC0          0
#define CID_VC1          4
#define CID_VC2          8
#define CID_VC3          12

#define DT_LINK     CSI_DT_RAW12

        .mode =
        {
            .fmt = FMT_LINK,
            .res = {.width = SENSOR_WIDTH, .height =
SENSOR_HEIGHT,   .fps = 30.0f},
            .channel_info = {.vc = 1, .dt = DT_LINK, .cid = CID_VC1},
        },
```

Each unique combination of the VC should map to a unique channel identifier (CID). Possible CID values for a given VC are:

- 0 – 0, 1, 2, 3
- 1 – 4, 5, 6, 7
- 2 – 8, 9, 10, 11
- 3 – 12, 13, 14, 15

Valid values for VCs are 0, 1, 2, and 3. It is recommended to use the same CID values for each VC, such that VC0 : CID 0, VC1 : CID 4, VC2 : CID 8, VC3 : CID 12.

7.5 Port and alias setup

Camera sensors have hardcoded I2C slave addresses. To individually communicate with these sensors, the bridge chip driver aliases them to an arbitrary sensor and serializer. Ensure that these aliased addresses do not collide.

The following example from the `max9296_lib` header details the deserializer, sensor alias, and serializer alias addresses which the MAX9296 bridge chip driver uses.

```
#define MSM_DESER_0_SLAVEADDR           0x90
#define MSM_DESER_0_SLAVEADDR           0x90
#define MSM_DESER_1_SLAVEADDR           0x94

#define MSM_DESER_0_ALIAS_ADDR_CAM_SER_0        0x82
#define MSM_DESER_0_ALIAS_ADDR_CAM_SER_1        0x84
#define MSM_DESER_1_ALIAS_ADDR_CAM_SER_0        0x8A
#define MSM_DESER_1_ALIAS_ADDR_CAM_SER_1        0x8C
#define MSM_DESER_0_ALIAS_ADDR_CAM_SNSR_0       0xE4
#define MSM_DESER_0_ALIAS_ADDR_CAM_SNSR_1       0xE8
#define MSM_DESER_1_ALIAS_ADDR_CAM_SNSR_0       0xEA
#define MSM_DESER_1_ALIAS_ADDR_CAM_SNSR_1       0xEC
```

The two sensor alias each have a serializer within the camera sensor. On this bridge chip, you can have up to two sensors per deserializer address, and therefore, two serializer alias.

7.6 Camera bridge chip driver init

The camera bridge chip driver should initialize registers based on the registers detailed in the associated camera bridge chip data sheet.

The `INIT_DESERIALIZER` array in the reference camera bridge chip drivers (found in `camx/ais/ImagingInputs/SensorLibs/`) details example init sequences for the reference camera bridge chips. However, expect there to be differences in what values and registers to write to when referencing the init sequences caused by bridge chip differences.

Initialization the bridge chip in AIS assumes I2C access to the camera bridge chip. The following example shows the initialization of a single register during the init process can:

```
#define INIT_DESERIALIZER \
{ \
    { 0x0330, 0x04, _reg_delay_ }, \
```

There will be many I2C writes to various registers during initialization. These writes during init could include tasks such as:

- Writing to RX0 to configure what DT is transmitted over CSI
- Configuring the decoder for an I2C slave dev attached to a remote deserializer
- Configuring the physical I2C address of the remote I2C slave device attached to the remote serializer
- Setting the frame valid time

Consult with the bridge chip vendor and bridge chip data sheet to confirm that all necessary settings to initialize a bridge chip are known.

NOTE: Refer to max96712_lib sensor driver, which uses the MAXIM 96712 bridge chip on SA8295.

8 QCarCam tests

The `qcarcam_test` binary application is a cross-OS development application to test multiple cameras and render them on display. The OS specific buffer and windowing functions are abstracted in the `test_util` library.

8.1 Command line arguments

Table 8-1 Command line arguments for QCarCam test

Parameter	Description	Example
`-config`	Specify `qcarcam_config.xml` file location	`config=/bin/camera/qcarcam_test/qcarcam_config.xml`
`-dumpFrame`	Enable frame dump every X frames	`dumpFame=50`
`-startStop`	Start/Stop every X frames. Waits 500ms after stop.	`startStop=50`
`-pauseResume`	Pause/Resume every X frames. Waits 500ms after pause.	`pauseResume=50`
`-noDisplay`	Runs offscreen with no display	`noDisplay`
`-singlethread`	Run `qcarcam_test` on a single thread (polls for frames)	`singlethread`
`-printfps`	Print average frames per second every X seconds	`printfps=10`
`-allocator`	Specifies buffer allocator to be used for offscreen streams. QNX: 0 – pmem v2, 1 – pmem v1	`allocator=1`

8.2 XML configuration file

The `qcarcam_test` takes in an XML configuration file as an argument. The XML specifies the camera streams and their properties. All sample configuration files are located in: `ais/test/test_util/config/`.

The `qcarcam_test` configuration can be used to enable as many camera streams as required. Multiple streaming windows can also be configured to a single camera input instead of dedicating each stream to a unique camera, as seen in the example below:

```
<?xml version="1.0" encoding="UTF-8"?>

<qcarcam_inputs>
     <input_device>
          <properties input_id="0" />
          <display_setting display_id="4" nbufs="3"
         window_width="0.3" window_height="0.3" window_pos_x="0"
         window_pos_y="0"
         zorder="1"/>
          <output_setting width="1280" height="720" nbufs="5" />
     </input_device>
     <input_device>
          <properties input_id="0"  event_callback="1"/>
          <display_setting display_id="4" nbufs="3"
        window_width="0.6" window_height="0.6" window_pos_x="0.2"
        window_pos_y="0.2"
         src_width="0.5" src_height="0.5" src_x="0.25" src_y="0.25"
         zorder="0"/>
          <output_setting width="1280" height="720" nbufs="5" />
     </input_device>
     <input_device>
          <properties input_id="0" />
          <display_setting display_id="4" nbufs="3"
         window_width="0.3" window_height="0.3" window_pos_x="0.7"
         window_pos_y="0.7"
         zorder="1"/>
          <output_setting width="1280" height="720" nbufs="5" />
     </input_device>
</qcarcam_inputs>
```

The configuration file above requests three camera streams to QCarCam input ID 0 with different windowing properties. The following diagram describes how the display properties are applied.

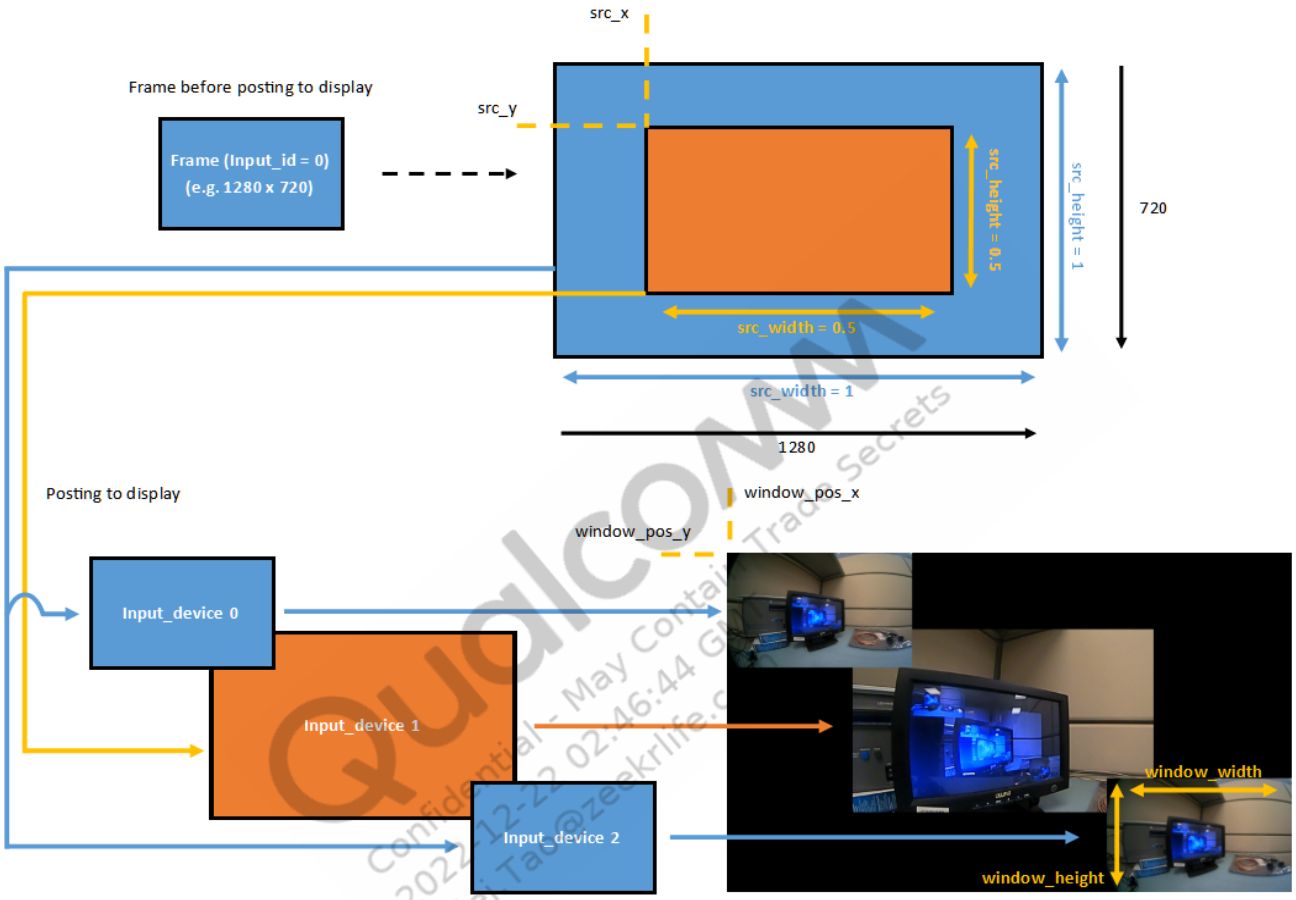

Figure 8-1 SA6155/SA8155/SA8195 ADP sample camera system setup

In each `qcarcam_test` configuration XML file, there are some required and optional components that must be set. If an optional attribute is not present, it is set to the default value.

When writing a custom `qcarcam_test` configuration XML file, refer to the XML Schema Definition (XSD) file (`ais/test/test_util/config/config_schema.xsd`) for the latest definitions. The following table describes a recent version of the schema:

Instance	Attributes	Description	Required	Default
`<qcarcam_inputs>`	-	Instance contains configuration for different input	✓	
`<input_device>`	-	One or more instances for each requested stream	✓	
`<properties>`	-	Camera properties	✓	
	`input_id`	QCarCam input ID (refer to CameraConfig QCarCam input mapping for which sensor driver source is requested)	✓	
	`event_callback`	Select to use event callback or not	✗	Enabled
	`frame_timeout`	Timeout for getting frames from camera if polling for frames	✗	500 ms
	`op_mode`	Set property QCARCAM_PARAM_OPMODE	✗	Not set
`<display_setting>`	-	Display properties (see Figure 8-1)	✓	
	`display_id`	Displays where image will be displayed (0 - main, 1 - secondary, etc.)	✓	
	`nbufs`	Number of buffers for display if need to postprocess in qcarcam_test to display buffers.	✗	5
	`window_width, height`	Size of display window (normalized value 0 to 1.0f with respect to display size)	✗	1.0f
	`window_pos_x, y`	Offset of display window (normalized value 0 to 1.0f with respect to display size)	✗	0.0f
	`src_width, height`	Size of input buffer source region (normalized value 0 to 1.0f with respect to buffer size)	✗	1.0f
	`src_x, y`	Offset in input buffer to source region (normalized value 0 to 1.0f with respect to buffer size)	✗	0.0f
	`zorder`	Set z order of the window	✗	Not set
	`pipeline`	Set windowing pipeline ID	✗	Not set
`<output_setting>`	-	Output buffer settings	✗	
	`nbufs`	Number of buffers	✗	5
	`width, height`	Size of output buffers	✗	QCarCam Query

Instance	Attributes	Description	Required	Default
	`stride`	Override stride of buffer	✗	Calculated
	`format`	Set buffer output format	✗	QCarCam Query

	`framedrop_mode`
	`framedrop_pattern`
	`framedrop_period`
		Sets frame dropping mode for the stream	✗	Not set

To create your own application based on the QCarCam API, see *QCarCam API Automotive Overview* (80-P2310-25).

9 Linux customization

For the Linux code base, the following additional customizations are done in the kernel.

GPIO and CCI (I2C) configuration

- SOC specific DTS file
  - Example: `sa8155-camera-sensor.dtsi` is for SA8155 ADP
  - Modify the specific `qcom,cam-sensor` node
- CCI
  - Set `cci-device` and `cci-master` attributes in a similar way as `i2c.device_id` and `port_id` in CameraConfig
- GPIO
  - List GPIOS in the `gpio*` attributes

10 AIS debugging

10.1 AIS log

The default logging level and output is defined in the ais_log.c file.

```
#define AIS_LOG_DEFAULT_CONF AIS_LOG_CONF_MAKE(AIS_LOG_MODE_OS,
AIS_LOG_LVL_HIGH)
```

To customize output, modify the macro of this log. You can also push a configuration file to override logging globally, or for a particular module.

Use `ais_log.conf` as an example.

10.2 Linux

10.2.1 Kernel log

For Linux, enable debug level logs for camera kernel drivers by echoing a bit mask of specific modules. The module bits are defined in cam_debug_util.h.

For example, you can enable debug level logging for CAM_SENSOR module with the following command:

```
echo 0x20 > /sys/module/cam_debug_util/parameters/debug_mdl
```

Enabling too many debug logs to console might lead the system to become very slow.

10.2.2 Enable DEVMEM

Enable DEVMEM to read hardware registers if needed or requested by our CE team.

Set “CONFIG_DEVMEM=y” in defconfig for a particular SOC (e.g., `kernel/msm-4.14/arch/arm64/configs/vendor/sa8155_defconfig`).

10.2.3 R tool

Once DEVMEM is enabled, use the R tool provided in the build (`system/core/toolbox/r.c`).

To make, go to `system/core/toolbox` and `mm`.

10.3 CCI (I2C) debugger

To facilitate debugging across CCI, a “ccidbgr” tool is provided to facilitate read/write I2C commands.

Refer to the ReadMe file for usage instructions in the ccidbgr location (`ais/test/ccidbgr`).

10.4 CSI debugging

CSI errors can indicate either an error on the transmitter side (sensor or bridge chip) or a misconfigured CSI timing parameter.

Ensure that no CSI errors are reported on the transmitter side. If this is a bring up of a new bridge chip, ensure that the CSI settle count is accurate. You might need to sweep across a range of timing values to get it right.

11 AIS build customization

This section describes the AIS build system customizations that are necessary to adjust when you are ready for commercialization.

The top level Makefile in AIS will contain the bulk of these customizations.

Static linking of libraries

Dynamic loading of AIS libraries is enabled by default, and to facilitate rapid development (`ais_config` and sensor libraries are dynamically loaded by `ais_server`).

To reduce bootup time overhead, enable static linking of all AIS libraries into `ais_server` by setting the `AIS_BUILD_STATIC` flag.

For example, for Linux Android, modify `ais/Android.mk` :

```
#Option to compile all libs statically into ais_server
AIS_BUILD_STATIC := true
```

12 ISP customization

Configuring different ISP use cases

Qualcomm ISP supports different use cases that can be set by a combination of qcarcam API and sensor binaries. This functionality is supported only if the ISP feature is enabled.

Table 12-1 Supported ISP use cases

ISP use case	Description
QCARCAM_ISP_USECASE_SHDR_BPS_IPE_AEC_AWB	Represents ISP pipeline that supports SHDR sensor and has the SHDR node.
QCARCAM_ISP_USECASE_BPS_IPE_AEC_AWB	Represents ISP pipeline that supports non-SHDR sensor and does not have the SHDR node.

12.1 QCarCam API

Users can call `qcarcam_s_param` to configure an ISP use case or `qcarcam_g_param` to query the current configuration.

The configuration of the ISP use case is a two-step process that must be done before calling qcarcam_start API.

1. Choose operational mode QCARCAM_OPMODE_ISP using `qcarcam_s_param` with QCARCAM_PARAM_OPMODE.
2. Choose ISP use case using `qcarcam_s_param` with QCARCAM_PARAM_ISP_USECASE. Current ISP use case values are listed in Table 12-1.

12.2 Changing sensor configuration to support non-SHDR ISP use case

An ISP use case might require a different set of sensor binaries to be created and downloaded to the target. For example, current default sensor binaries support an SHDR use case. The user must create a separate set of sensor binaries to reflect a non-SHDR use case.

The non-SHDR use case requires the user to remove SHDR capabilities from a sensor XML file:

~~`<capability>SHDR</capability>`~~

After removing this line, rebuild the sensor binaries and then push them to the target. This is required for 2A algorithms, which cannot use SHDR stats and or run SHDR related submodules.

12.3 camera_config XML modification to enable ISP use case

The user must modify `opMode` to be QCARCAM_OPMODE_ISP for a particular input in `camera_config_saXXX.xml` and push it to the target.

For example, to change opMode of input 0 in SA8155 hardware, the modify the following line in `camera/ais/CameraConfig/xml/camera_config_sa8155.xml`:

```
<inputMap qcarcamId = "0" opMode = "QCARCAM_OPMODE_ISP">
```

For additional camera configuration details, see Chapter 5 .

12.4 qcarcam_test configuration XML

The ISP use case can be configured using qcarcam_test XML if the qcarcam_test application is used to operate cameras.

There are two attributes controlling ISP use case configuration in qcarcam_test XML:

- `op_mode`: Chooses an operational mode for specific input.
- `isp_usecase`: Chooses an ISP use case.

For example, the user must configure or modify the following line in the qcarcam_test XML file to configure input 0 so that it works in non-SHDR mode:

```
<properties input_id="0" op_mode="7" isp_usecase="1" />
```

A References

A.1 Related documents

Title	DCN or URL
Qualcomm Technologies, Inc.	
QCarCam API Automotive Overview	80-P2310-25
Multimedia Driver Development and Bringup Guide – Camera	80-NU323-2
Additional resources	
Leopard Imaging	https://www.leopardimaging.com/

A.2 Acronyms and terms

Acronym or term	Definition
AIS	Automotive Imaging System
ADP	Automotive Development Platform
CCI	Camera Control Interface
CID	Channel identifier
CSI	Camera Serial Interface
CVBS	Composite Video Broadcast Signal. Used for analog camera stream case.
DT	Data type
INT	Interrupt pin
PDB	Power Down mode input pin
PHY	MIPI Alliance Physical Layer standard. There are three PHY types, C, D and M representing 100, 500 and 1000.
VC	Virtual channel