SA8155/SA8155P HQX Automotive Camera Architecture Overview

80-PG469-195 Rev A

March 21, 2019

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

Revision history

Revision	Date	Description
A	March 2019	Initial release

Contents

Figures

Tables

1 Introduction

1.1 Purpose

This document describes how to configure the AIS (automotive imaging system) on HQX automotive platforms. It details the following:

- AIS architecture
- Data flow between AIS client and server running on different GVMs and PVMs
- Camera driver porting and source files
- Basic API calls

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**.

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 Overview

AIS is designed to manage video input streams (e.g., camera and HDMI). AIS is optimized for the Qualcomm® SA8155 Automotive chipset.

AIS manages memory buffers under the AIS framework. It also sets access levels for specific users and applications so that any camera input can become privileged. By using this feature, multiple users can simultaneously access the same camera input. Users can also access noncamera input sources (e.g., HDMI and video input).

AIS supports the following features:

- OS portability
- Multiclient support
- Input permissions
- Bridge chip abstraction for entity access
- Hardware assisted split stream
- Zero buffer copy
- Frame transformation
- Frame processing

2.1 SA8155 hardware architecture

Figure 2-1 shows a typical multicamera input system.

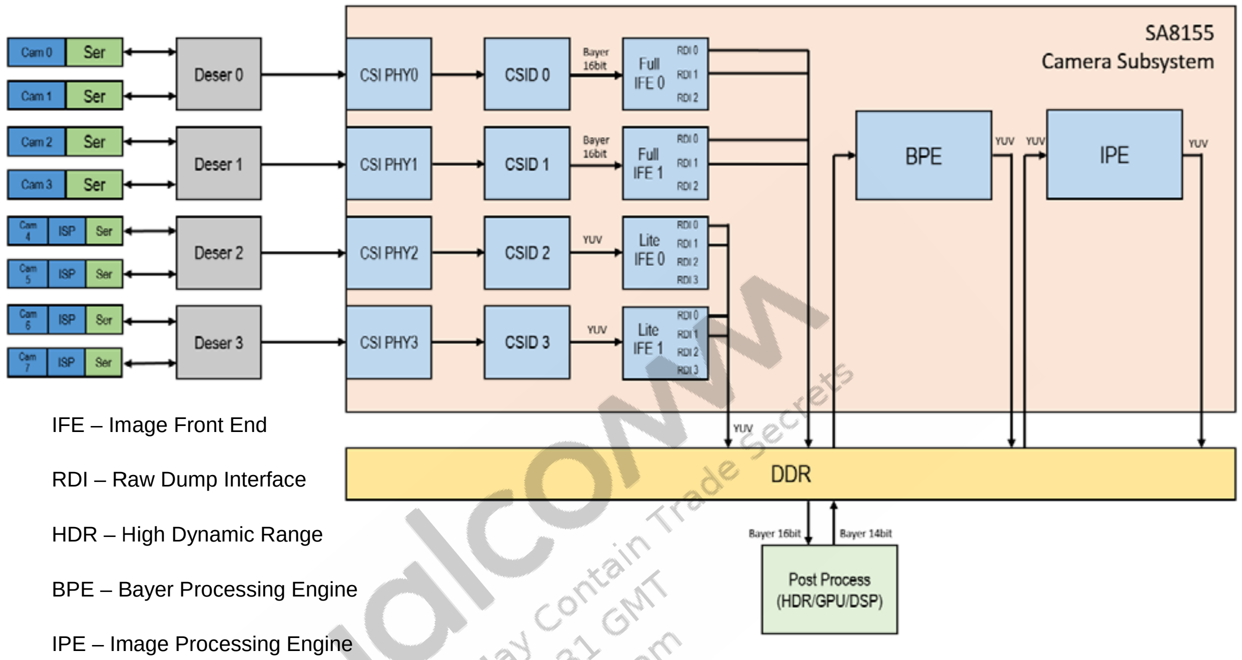
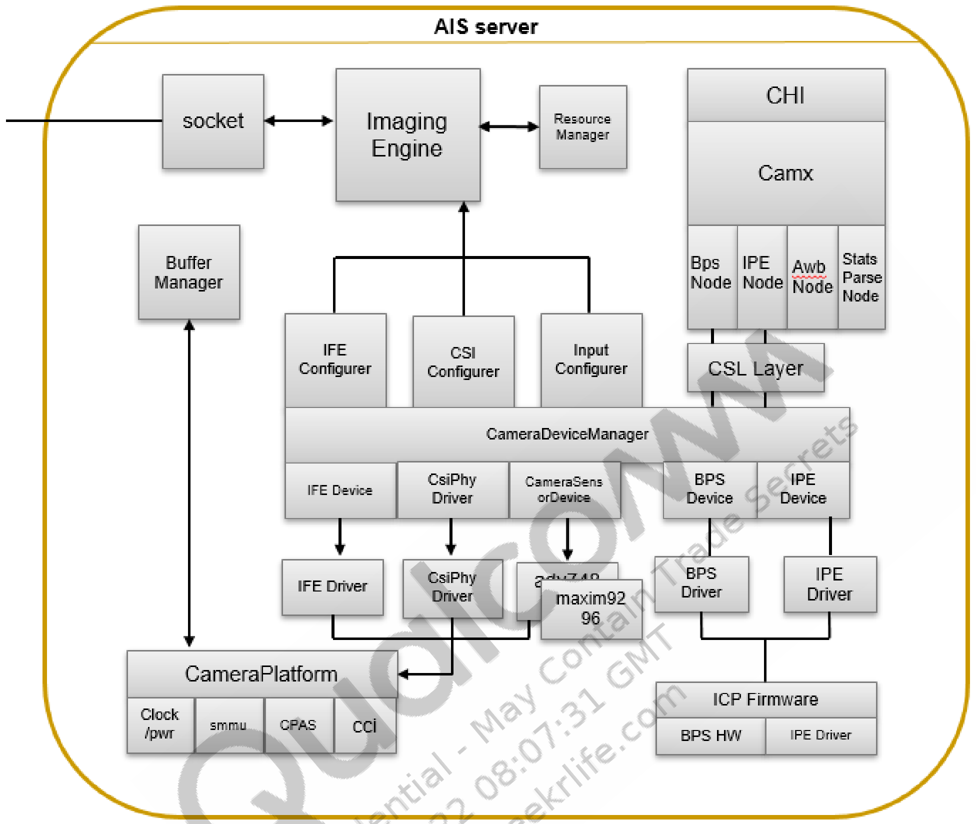


Figure 2-1 SA8155 typical hardware architecture

SA8155 supports four CSI input ports with up to four lanes for each. Each port connects to a camera or bridge chip. CSI0-4 connects to a bridge chip. The bridge chip collects two camera data streams respectively to one CSI port through MIPI lanes. This example is shown with four Bayer camera sensors and four YUV camera sensors.

2.2 AIS software architecture



CHI – Camera Hardware Interface
AWB – Auto white balance
CSL – Camera Service Layer
ICP – Image Control Processor
CPAS – Camera Peripheral and Support
IFE/VFE – Image Front End/Video Front End
CCI – Camera Control Interface
BPS – Bayer Processing Segment
IPE – Image Processing Engine

Figure 2-2 AIS software architecture

The AIS server runs on user space as a daemon. The server communicates with kernel space hardware drivers through platform-related libraries. The server executes most of the management tasks. AIS clients use the socket protocol to communicate with the AIS server to query camera information, control camera handle, and retrieve camera data.

3 PVM/Hypervisor/GVM solution

3.1 High-level software design

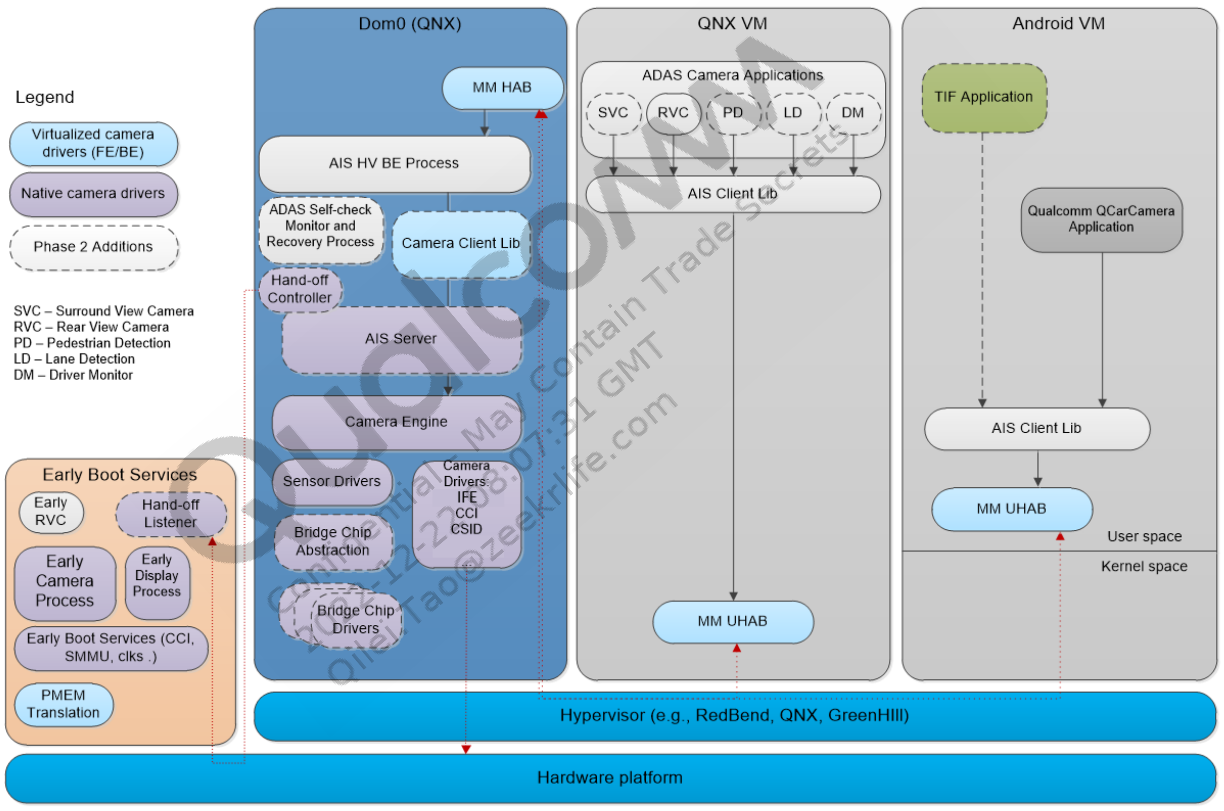


Figure 3-1 Camera virtualization with QNX PVM, Android, and QNX GVM

The camera back-end and front-end communication depends on mmHAB functions such as open, close, send, receive, export, and import (both on GVM and on PVM).

- The camera back-end driver depends on the QNX-to-the metal camera driver and firmware.
- The camera front-end driver depends on the HLOS memory (e.g., ION in LA, GBM in LV, PMEM in QNX) for memory allocation and sharing with the back-end driver.
- The camera front-end driver depends on the native user mode media framework implemented within AIS.

3.2 Sequence of camera hypervisor communication call

The sequence of the camera hypervisor communication call is as follows:

- The QNX camera back-end server is a 64-bit application that keeps listening to any FE connection request. It calls the Dom 0 Hypervisor Abstract Communication driver (HAB) using the `habmm_socket_open` function with the MM_CAMERA tag.
- The HAB abstracts the details of the hypervisor connection, send, and receive functions. The call waits until the HAB detects a camera FE connection request.
- The FE connection request is caused by the `habmm_socket_open` function with the MM_CAMERA tag.
- Upon receiving the “be virtual channel address” from the `habmm_socket_open` function, the camera BE Master creates a thread to handle the client requests for this session.
- Once the communication channel is up, the FE sends messages to the BE process and then blocks to wait for the return value.
- The BE process receives the hypervisor message and translates the message into the AIS client calls to the local AIS server for processing. This is the same as the normal call process from a local client.
- The call returns parameters from camera IP to the BE process. The parameters are transled into hypervisor messages before being sent back to the FE.
- The FE translates the returned parameters from the hypervisor message to the return data to the caller, using the QCarCam API that the clients use.
- At the end of the communication, the FE synchronizes with the BE process to close the virtual channel using the `habmm_socket_close` function.
- The BE thread exits and cleans up any leftover items.
- The original BE process keeps watching for any new connections.

API

The API is the same for all applications (either in HV or metal). BE in a client of the AIS server is the same as in any other client.

3.3 Communication implementation details

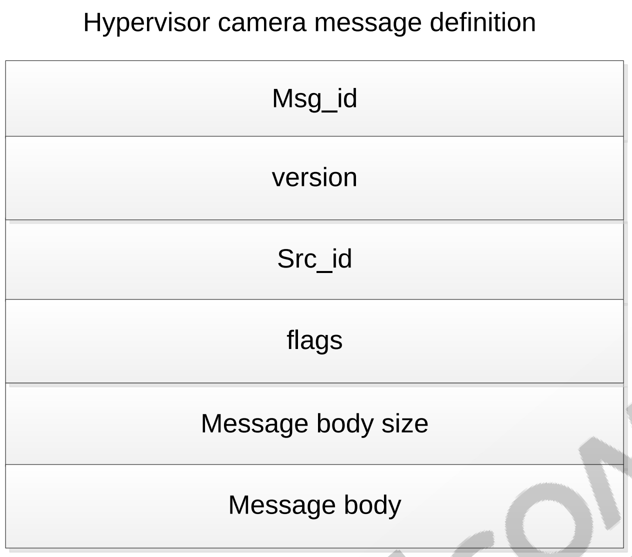


Figure 3-2 Hypervisor camera message

Table 3-1 Hypervisor command message structure

hypcamera_msg_id_type	Description	Details

`CAM_MSG_OPEN`	Opens a camera or other input devices	`typedef struct`
		`{`
		`    /**< The input desc to open.  This was returned by query input call */`
		`    qcarcam_input_desc_t desc;`
		`}cam_vm_cmd_open_t;`

`CAM_MSG_QUERY_INPUTS`	Queries what inputs are available	`typedef struct`
		`{`
		`    /**< flag to indicate if p_inputs was set */`
		`    unsigned int is_p_inputs_set;`
		
		`    /**< Number of elements in input structure */`
		`    unsigned int size;`
		`}cam_vm_cmd_query_inputs_t;`

`CAM_MSG_GET_PARM`	Gets a parameter	
`CAM_MSG_SET_PARM`	Sets a parameter	

hypcamera_msg_id_type	Description	Details

CAM_MSG_SET_BUFFERS	Sets the buffer list to write camera data into	typedef struct cam_vm_cmd_s_buffer_t
		{
		    /**< handle returned during open*/
		    qcarcam_hndl_t hndl;
		
		    /**< color format*/
		    qcarcam_color_fmt_t color_fmt;
		
		    /**< The buffers to use*/
		    qcarcam_buffer_t buffers[QCARCAM_MAX_NUM_BUFFERS];
		
		   /**< The number of buffers*/
		    unsigned int n_buffers;
		}cam_vm_cmd_s_buffer_t;

CAM_MSG_START		typedef struct
		{
		    /**< handle returned during open*/
		    qcarcam_hndl_t hndl;
		}cam_vm_cmd_start_t;

CAM_MSG_STOP		See CAM_MSG_START
CAM_MSG_PAUSE		See CAM_MSG_START
CAM_MSG_RESUME		See CAM_MSG_START

CAM_MSG_GET_FRAME	Blocks until a frame is ready in a buffer	typedef struct
		{
		    /**< handle returned during open*/
		    qcarcam_hndl_t hndl;
		
		    /**< timeout*/
		    unsigned long long int timeout;
		
		    /**< flags*/
		    unsigned int flags;
		}cam_vm_cmd_get_frame_t;

hypcamera_msg_id_type	Description	Details

CAM_MSG_RELEASE_FRAME	Releases a buffer back to queue so that hardware can write to it again	`typedef struct`
		`{`
		`    /**< handle returned during open*/`
		`    qcarcam_hndl_t hndl;`
		
		`    /**< buffer index to release back to ais*/`
		`    int idx;`
		`}cam_vm_cmd_release_frame_t;`

CAM_MSG_CLOSE	Specifies the close device call. The data stores the corresponding handle.	`typedef struct`
		`{`
		`    /**< The opened service handle. */`
		`    qcarcam_hndl_t handle;`
		`}cam_vm_cmd_close_t ;`

Table 3-2 Hypervisor response message structure

hypcamera_msg_id_type	Description	Details

CAM_MSG_OPEN	Returns the status and a handle to the newly opened input	`typedef struct`
		`{`
		`    /**< The status of open. */`
		`    unsigned int status;`
		
		`    /**< The opened service handle. */`
		`    qcarcam_hndl_t handle;`
		`}cam_vm_cmd_open_rsp_t`

CAM_MSG_QUERY_INPUTS	Queries what inputs are available.
	For more details, see `qcarcam_input_t`.
		`typedef struct`
		`{`
		`    /**< Available input structure */`
		`    qcarcam_input_t p_inputs[MAX_NUM_AIS_INPUTS];`
		
		`    /**< Filled size */`
		`    unsigned int ret_size;`
		
		`    /**< status */`
		`    qcarcam_ret_t status;`
		`}cam_vm_cmd_query_inputs_rsp_t;`

CAM_MSG_GET_PARM	Gets a parameter	–
CAM_MSG_SET_PARM	Sets a parameter	–

hypcamera_msg_id_type	Description	Details
CAM_MSG_SET_BUFFERS	–	typedef struct { /**< The status of call. */ unsigned int status; }cam_vm_cmd_s_buffer_rsp_t;
CAM_MSG_START	–	typedef struct { /**< The status of start. */ unsigned int status; }cam_vm_cmd_start_rsp_t;
CAM_MSG_STOP	–	See CAM_MSG_START
CAM_MSG_PAUSE	–	See CAM_MSG_START
CAM_MSG_RESUME	–	See CAM_MSG_START
CAM_MSG_GET_FRAME	Blocks until a frame is ready in a buffer	typedef struct { /**< The status of get frame. */ unsigned int status; /**< frame info*/ qcarcam_frame_info_t frame_info; }cam_vm_cmd_get_frame_rsp_t;
CAM_MSG_RELEASE_FRAME	Releases a buffer back to queue so that hardware can write to it again	typedef struct { /**< The status of release frame. */ unsigned int status; }cam_vm_cmd_release_frame_rsp_t;
CAM_MSG_CLOSE	Specifies the close device call. The data stores the corresponding handle.	typedef struct { /**< The status of close. */ unsigned int status; }cam_vm_cmd_close_rsp_t;

3.4 Data flow of a hypervisor camera application

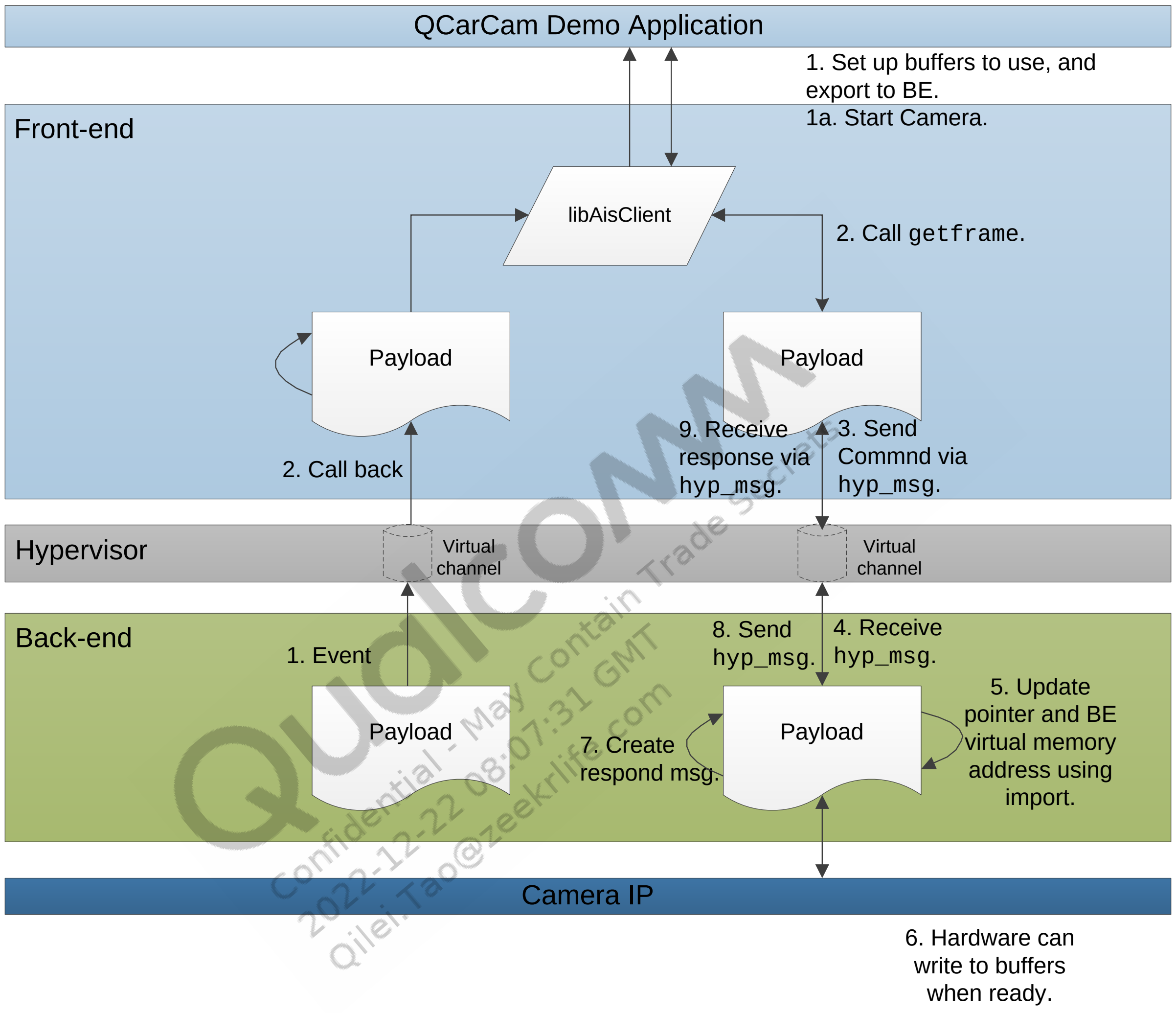


Figure 3-3 Data flow of a hypervisor IOCTL call

Camera data flow is transparent to the message channel. Memory is mapped during the `set_buffer()`. The `Getframe()` call returns which buffer contains data, allowing the application to use the data and then call `releaseFrame()` to give it back to the system to use again.

4 AIS client application

4.1 Typical client workflow

Figure 4-1 shows the recommended call flow when using the QCarCam API.

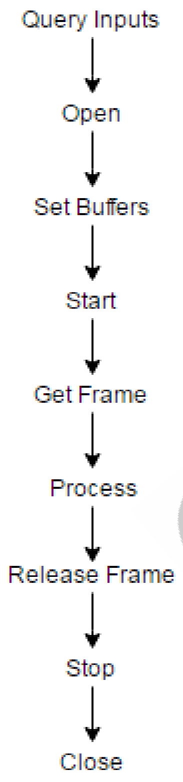


Figure 4-1 QCarCam call flow

4.2 Demo application

The `qcarcam_test` native demo applications are executable over the terminal to test the QCarCam API. This demo is available on Android O, QNX, and AGL with root permissions.

Table 4-1 Command-line arguments for each OS

Parameter	Description	Example
`config`	Specifies `qcarcam_config.xml` file location	`config=/bin/camera/qcarcam_test/qcarcam_config.xml`
`dumpFrame`	Enables frame dump every N frames	`dumpFame` = `50`
`startStop`	Starts/Stops every N frames	`startStop` = `50`
`pauseResume`	Pauses/Resumes every N frames	`pauseResume` = `50`
`noDisplay`	Runs without displaying frames on the display	`noDisplay`
`singlethread`	Runs `qcarcam_test` on a single thread	`singlethread`
`printfps`	Prints average frames per second every N seconds	`printfps` = `10`

Android O

To run the demo application on Android O:

```
adb root
adb remount
adb shell
ais_server &
//one camera test for camera at channel 0
qcarcam_test -config=/system/bin/qcarcam_config_single.xml
//multi camera test can also be used for one camera
qcarcam_test -config=/system/bin/qcarcam_config.xml
```

AGL

To run the demo application on AGL:

```
ais_server &
// one camera test for camera at channel 0
qcarcam_test -config=/data/misc/camera/qcarcam_config_one.xml
```

QNX

To run the demo application on QNX:

```
ais_server &
cd bin/camera/qcarcam_test
// one camera test for camera at channel 0
./qcarcam_test -config=qcarcam_config_single.xml
//multi camera test can also be used for one camera
./qcarcam_test -config=/qcarcam_config.xml
```

QNX hypervisor

To run the demo application on QNX + hypervisor + AGL (e.g., with AGL running as GVM):

```
//Make sure to enable the camera demon on QNX side:

//push qcarcam_config_single_0.xml to AGL side /usr/bin

//Run test in AGL GVM:
qcarcam_test -config=/usr/bin/qcarcam_config_single_0.xml
//Result: Preview camera CH0 success, and the camera CH0 show on panel
```

5 Customization

5.1 Requirements

Most customization requirements are to change the types of camera and bridge chips. Figure 2-2 shows a submodule, sensor drivers, which contains the following:

- Max9296_lib
  - AR0231

Max9296_lib is the bridge chip driver. AR0231 is the sensor driver. Users can modify these source files to port to a new type of bridge chip and camera module. The file `camera_config.c` contains structures to define camera information including ID and connection relationship.

5.2 File locations

File locations in this section use LA as an example. Most software architecture from the user space is located in /vendor/qcom/proprietary/camx/ais/.

For each block in the user space, refer to the following locations:

- Sensor driver location: /vendor/qcom/proprietary/camx/ais/ImagingInputs/SensorLibs/
- `Camera_config.so` is built from /vendor/qcom/proprietary/camx/ais/CameraConfig/
- `AIS_client` and `AIS_server`: /vendor/qcom/proprietary/camx/ais/CameraMulticlient/
- Camera platform lib: /vendor/qcom/proprietary/camx/ais/CameraPlatform/
- Camera imaging engine, managers, and configurers: /vendor/qcom/proprietary/camx/ais/Engine/
- Hardware driver API: /vendor/qcom/proprietary/camx/ais/HWDrivers/
- Application tests (e.g., `qcarcam_test`): /vendor/qcom/proprietary/camx/ais/test/

Kernel-level blocks are in:

- kernel/MSM-4.4/drivers/media/platform/MSM/camera_v2/

A References

A.1 Related documents

Title	Number
Qualcomm Technologies, Inc.	
QCarCam API Automotive Overview	80-P2310-25
SA8155 Automotive Camera AIS Customization Guide	80-PG469-93

A.2 Acronyms and terms

Acronym or term	Definition
AGL	Automotive Grade Linux
AIS	Automotive Imaging System
API	Application Programming Interface
BE	Back-End
CCI	Camera Control Interface
CSI	Camera Serial Interface
CVBS	Composite Video Broadcast Signal
DM	Driver Monitor
FE	Front-End
FPD-Link	Flat Panel Display Link
GBM	Generic Buffer Management
GVM	Guest Virtual Machine
HDMI	High-Definition Multimedia Interface
HLOS	High-Level Operating System
HV	HyperVisor
IOCTL	I/O ConTroL
ION	IP Service Over AllJoyn
ISP	Image Signal Processor
LA	Linux Android
LD	Lane Detection
LV	Linux Vehicle
MIPI	Mobile Industry Processor Interface
PD	Pedestrian Detection
PMEM	Persistent MEMory

Acronym or term	Definition
PVM	Physical Virtual Machine
RVC	Rear View Camera
SMMU	System Memory Management Unit
SVC	Surround View Camera
VFE	Video Front-End