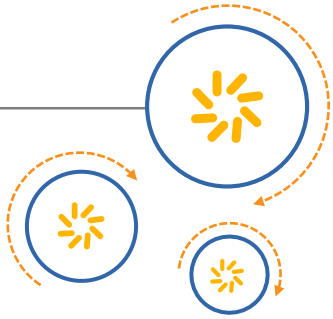
Qualcomm Technologies, Inc.



Camera Sensor Module Bringup

Application Note

80-NV354-3 Rev. A

July 16, 2015

Questions or comments: https://support.cdmatech.com/

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

Revision history

Revision	Date	Description
A	July 2015	Initial release

Contents

Tables

1 Introduction

1.1 Purpose

This document describes driver generation and integration details for the camera sensor and its associated modules within the Qualcomm® camera stack. The camera sensor relies on configuration of these receiver modules, and the driver contains configuration parameters for the modules. The camera sensor framework includes configuration of these components:

- Sensor with receiver modules including csiphy, csid
- CCI for I2C communication
- Actuator for lens movement
- Chromatix™ color optimization tool for configuring the ISP, CPP, etc.

The following sections describe how to create/configure these components to bring up the camera.

1.2 Scope

Much of the information in this document is generic to Linux camera code on all MSM8994 and MSM8992 (B-Family) chipsets, but the document has been written based on the MSM8994-92 code base.

1.3 Conventions

Function declarations, function names, type declarations, and code samples appear in a different font, e.g., `#include`.

Code variables appear in angle brackets, e.g., `<number>`.

2 Sensor Driver Bringup

This chapter covers how to bring up camera sensor hardware on the Android platform with an MSM8992/MSM8994 device.

NOTE: Unless otherwise specified, all information applies to Bayer sensors.

Sensor driver bringup comprises three tasks:

1. Collect sensor driver hardware and software requirements
2. Configure the camera-related hardware
3. Integrate the software

2.1 Collect sensor driver requirements

- Review the requirements and make sure they are achievable with the selected hardware (camera and device).
- To work with the camera SDK, we assume that the camera sensor schematics are configured the same as the Qualcomm schematics. See Section 2.2.
- We recommend not making any changes in the dtsi for the sensor bringup.

2.2 Hardware configuration

For the camera to work properly, configure a number of input signals (EXTCLK, Power supply, RESET, etc.) and output signals (CSI, CCI, flash trigger and etc.) and make them available to the camera.

In the current Android source code, those signals are grouped into slots/nodes. Each slot configuration describes the available hardware signals that are wired to it. As the number of slots and their configuration vary between hardware devices, their descriptions are device-specific and are placed in a <target>_camera*.dtsi file in kernel/arch/arm/boot/dts/qcom/.

NOTE: For a 64-bit processor, use the arm64 directory instead of arm.

```
qcom,camera@0 {
        cell-index = <0>;
        compatible = "qcom,camera";
        . . .
}
```

If the camera module is to be integrated on an already known hardware platform/device, minor or no hardware configuration should be needed.

NOTE: Incorrect changes to the hardware configuration can lead to damage to part of, or the whole device. Make such changes only after closely inspecting the hardware schematics.

2.2.1 GPIO configuration

The following snippet shows how to configure sensor-specific GPIOs based on the target board. For explanations of each property, refer to documents here:

kernel/Documentation/devicetree/bindings/media/video/

```
gpios = <&msmgpio 15 0>,
        <&msmgpio 90 0>,
        <&msmgpio 89 0>;
qcom,gpio-reset = <1>;
qcom,gpio-standby = <2>;
qcom,gpio-req-tbl-num = <0 1 2>;
qcom,gpio-req-tbl-flags = <1 0 0>;
qcom,gpio-req-tbl-label = "CAMIF_MCLK",
                          "CAM_RESET1",
                          "CAM_STANDBY";
qcom,gpio-set-tbl-num = <1 1>;
qcom,gpio-set-tbl-flags = <0 2>;
qcom,gpio-set-tbl-delay = <1000 30000>;
```

For CCI, there are dedicated GPIOs for data and clock as shown in the following MSM8916 example. Because these GPIOs are dedicated to the CCI master, the customer must use these settings if CCI is used for camera I2C.

```
gpios = <&msm_gpio 29 0>,
        <&msm_gpio 30 0>;
qcom,gpio-tbl-num = <0 1>;
qcom,gpio-tbl-flags = <1 1>;
qcom,gpio-tbl-label = "CCI_I2C_DATA0",
                      "CCI_I2C_CLK0";
```

2.2.2 Clock related settings

In the .dts file, for each sensor slot, configure the clock source as follows:

```
clocks = <&clock_gcc clk_mclk0_clk_src>,
         <&clock_gcc clk_gcc_camss_mclk0_clk>;
clock-names = "cam_src_clk", "cam_clk";
```

The ordering of the lists in the two properties is important. The Nth clock name corresponds to the nth entry in the clock's property. Thus, in the DT snippets above, cam_src_clk corresponds to clk_mclk0_clk_src; cam_clk corresponds to clk_gcc_camss_mclk0_clk, etc.

The customer does not need to change this, as it is parsed in the clock framework.

2.3 Software configuration

This is the final and main step in sensor integration. Perform this step only after the first two steps are done, i.e., the requirements are known and the hardware configuration is verified.

The source code is placed in the user space. The camera integration is a collection of numbers of libraries. The focus here is the sensor driver library.

NOTE: For a correct build, the make command must have visibility into the new integrated libraries, which is provided by adding the libraries in /vendor/qcom/proprietary/common/config/device_vendor.mk.

NOTE: The source files of the sensor driver must be placed in a sensor-specific directory in /vendor/qcom/proprietary/mm-camera/ mm-camera2/media-controller/modules/sensors/sensor_libs/.

The directory name needs to be unique and representable. A good practice is to use the sensor name as the directory name (e.g., imx214).

Each sensor driver consists of three files:

- Android.mk – Contains build rules and include paths.
- <sensor>_lib.c – Contains sensor-specific logics and functions.
- <sensor>_lib.h –Contains sensor configuration and sensor handler structure.

The sensor driver, its parameters and structures are accessed by the source code through the sensor handler:

```
static sensor_lib_t sensor_lib_ptr =
{
 .sensor_slave_info =
...
```

Its type (`sensor_lib_t`) is defined in:

includes/sensor_lib.h.

Following is a description of the sensor handler structure and its parameters. They need to be filled according to the sensor configuration and project requirements.

2.3.1 Sensor slave information

The sensor slave information structure contains the information about camera module parts, their names and configuration.

```
struct camera_sensor_slave_info {
  char sensor_name[32];
  char eeprom_name[32];
```

```
  char actuator_name[32];
  char ois_name[32];
  char flash_name[32];
  enum sensor_camera_id camera_id;
  unsigned short slave_addr;
  enum sensor_i2c_freq_mode i2c_freq_mode;
  enum camera_i2c_data_type addr_type;
  struct sensor_id_info_t sensor_id_info;
  struct camera_power_setting_array power_setting_array;
  unsigned char is_init_params_valid;
  struct sensor_init_params sensor_init_params;
  unsigned char is_flash_supported;
};
```

Sensor, EEPROM, actuator and OIS names are set as char strings. If any of the modules is not present, its char string should be set to NULL.

We support PMIC flash for the MSM8992 and MSM8994 targets; update the flash name with pmic.

`camera_id` – The hardware slot (see Section 2.2 for an explanation) is set to which camera is connected.

`slave_addr` – Sets the CCI/I2C slave address of the sensor. The 8-bit-wide read address is expected here. Its 7 MSB are the slave address given in the sensor documentation and the LSB is set to 0.

`i2c_freq_mode` and `addr_type` describes the CCI/I2C configuration.

`i2c_freq_mode` are as follows

SENSOR_I2C_MODE_STANDARD for 100 KHz

SENSOR_I2C_MODE_FAST for 400 KHz

SENSOR_I2C_MODE_CUSTOM for custom configuration in dtsi.

SENSOR_I2C_MODE_FAST_PLUS for 1 MHz

`is_init_params_valid` – A flash (TRUE/FALSE) showing whether the initial parameters are valid. If they are not, the code tries collecting them from the *.dtsi file.

`is_flash_supported` – A flag (TRUE/FALSE) showing whether the hardware supports a flash (LED) module for this sensor.

2.3.1.1 Sensor ID info

`sensor_id_info` contains the sensor id register and the expected value in it.

```
struct sensor_id_info_t {
  unsigned short sensor_id_reg_addr;
  unsigned short sensor_id;
  unsigned short sensor_id_mask;
```

```
};
```

The register address and its value (sensor-specific) should be taken from the sensor documentation. This register is used for sensor detection. If the source is able to read that value from the register, it can be certain that the sensor is present and powered up correctly.

sensor_id_mask is a number that has appropriate bits set to consider for read_id_value.

NOTE: A successful sensor detect does not mean the sensor is working properly. It just proves that the sensor is present, but does not test its streaming capabilities at that point.

2.3.1.2 Powerup/powerdown sequence

`power_setting_array` – Contains the powerup and powerdown sequences. They need to follow the description in the sensor documentation.

The sensor power sequence is added in an array using the camera_power_setting structure in each user space sensor driver.

```
static struct camera_power_setting power_setting[] = {
. . .
}
```

Both powerup and powerdown sequences can be added in the camera_power_setting_array structure. If a power_down_setting/size_down member is not added as shown below, the powerdown sequence is the reverse of the powerup sequence.

```
.power_setting_array = {
    .power_setting = power_setting,
    .size = ARRAY_SIZE(power_setting),
 },
```

This power_setting points to an array that has information on GPIO/CLK/VREG to be used to configure each sensor.

```
static struct camera_power_setting power_setting[] = {
  {
    .seq_type = SENSOR_VREG,
    .seq_val = CAM_VDIG,
    .config_val = 0,
    .delay = 0, //this delay is in ms
  },
  {
    .seq_type = SENSOR_VREG,
    .seq_val = CAM_VANA,
    .config_val = 0,
    .delay = 0, //this delay is in ms
  },
```

```
. . .
```

This structure is used in msm_camera_power_up() in the kernel to configure the sensor for the powerup sequence.

Depending on the customer hardware design, the power source can be provided by PMIC or through GPIOs as shown in the following .dtsi file.

PMIC case

```
cam_vdig-supply = <&pm8916_s4>;
cam_vana-supply = <&pm8916_l17>;
cam_vio-supply = <&pm8916_l6>;
cam_vaf-supply = <&pm8916_l10>;
```

GPIO case

```
gpios = <&msm_gpio 27 0>,
            <&msm_gpio 28 0>,
            <&msm_gpio 33 0>,
            <&msm_gpio 114 0>,
            <&msm_gpio 110 0>;
qcom,gpio-reset = <1>;
qcom,gpio-standby = <2>;
qcom,gpio-vdig = <3>;
qcom,gpio-vana = <4>;
qcom,gpio-req-tbl-num = <0 1 2 3 4>;
qcom,gpio-req-tbl-flags = <1 0 0 0 0>;
qcom,gpio-req-tbl-label = "CAMIF_MCLK",
                          "CAM_RESET",
                          "CAM_STANDBY",
                          "CAM_VDIG",
                          "CAM_VANA";
```

- CAMERA_VANA – Analog supply voltage
- CAMERA_VDIG – Digital supply voltage
- CAMERA_VAF – Actuator supply voltage
- CAMERA_VIO – Digital input/output voltage

2.3.1.3 Sensor init parameters

`sensor_init_params` contains the camera module configuration.

- `modes_supported` – Mask showing the supported modes (2D, 3D).
- `position` – Sets sensor position on the device (back or front camera).
- `sensor_mount_angle` – Sets camera mount angle relative to the display.

NOTE: If the mount angle is unknown, set it to SENSOR_MOUNTANGLE_360, which tells the system to look into the *dtsi for the rotation.

2.3.2 Sensor output parameters

The sensor output parameters contain the information about the type and format of the data the sensor is streaming out.

```
typedef struct {
        sensor_output_format_t output_format;
        sensor_connection_mode_t connection_mode;
        sensor_raw_output_t raw_output;
        sensor_filter_arrangement filter_arrangement;
} sensor_output_t;
```

`output_format` – The sensor streams BAYER (SENSOR_BAYER) for RAW frames.

`connection_mode` – The transmission interface – MIPI (SENSOR_MIPI_CSI) or Parallel.

`raw_output` – The type of data that is transmitted – RAW10 (SENSOR_10_BIT_DIRECT), RAW12 (SENSOR_12_BIT_DIRECT), RAW8 (SENSOR_8_BIT_DIRECT), DPCM8, etc.

`filter_arrangement` – The bayer color pattern of the frames streamed out by the sensor (SENSOR_BGGR, SENSOR_RGGB, SENSOR_RGBG, SENSOR_BGRG).

NOTE: Depending on the sensor setting, the color pattern might differ from the one in the sensor datasheet.

2.3.3 Output register address

`output_reg_addr` structure contains the sensor-specific addresses of four registers. They are needed to control the sensor behavior. The register addresses are taken from the sensor documentation.

```
struct sensor_output_reg_addr_t {
  unsigned short x_output;
  unsigned short y_output;
  unsigned short line_length_pclk;
  unsigned short frame_length_lines;
};
```

`x_output` – Address of x_output register. This register contains the number of active pixels per line in the sensor output.

`y_output` – Address of y_output register. This register contains the number of active lines per frame in the sensor output.

`line_length_pclk` – Address of line_length_pclk register. This register contains the number of total pixels per line in the sensor output.

`frame_length_lines` – Address of y_output register. This registers contains the number of total lines per frame in the sensor output.

2.3.4 Exposure register address

`exp_gain_info` structure contains the addresses of exposure-related registers. Those register addresses are used in AEC functionality to control the sensor exposure and analog gain.

```
struct sensor_exp_gain_info_t {
  unsigned short coarse_int_time_addr;
  unsigned short global_gain_addr;
  unsigned short vert_offset;
};
```

`coarse_int_time_addr` – Address of coarse_integration_time register. This register contains the frame exposure in number of lines.

`global_gain_addr` – Address of global_gain or analog_gain register. This register contains the frame analog gain. Its value is calculated by a sensor-specific function.

`vert_offset` – Contains the value of max_coarse_integration_margin, which is a sensor-specific value. It is the max margin between exposure and frame length. Exposure cannot be bigger than frame length minus vert_offset.

max_exp <= frame_length – vert_offset

2.3.5 AEC info

`aec_info` structure contains the addresses of exposure-related registers. Those register addresses are used in AEC functionality to control the sensor exposure and analog gain.

```
typedef struct {
  float min_gain;
  float max_gain;
  float min_analog_gain;
  float max_linecount;
}sensor_aec_data_t;
```

`min_gain` – Sets the lowest acceptable value for gain from the sensor, in normalized units (1.00x). Set to 1.00x, if not otherwise specified.

NOTE: Values under 1.00x are unacceptable.

`max_gain` – Sets the highest acceptable value for gain, in normalized units (1.00x). It is the result of both analog and digital gains.

`max_analog_gain` – Sets the highest acceptable value for analog gain from the sensor, in normalized units (1.00x). If no digital gain from the sensor, max_analog_gain and max_gain should be equal.

`max_linecount` – Sets the highest acceptable value for exposure from the sensor. This value depends on coarse_integration_time register resolution and max_coarse_integration_margin (set in vert_offset).

2.3.6 Frame skips

When the sensor is reconfigured, a number of frames might need to be streamed out before the sensor can provide a stable image. These parameters are used to describe the number of needed frames in different use cases.

`unsigned short sensor_num_frame_skip;` - Sets the number of frames to be skipped after sensor streaming is turned on.

`unsigned short sensor_num_HDR_frame_skip;` - Sets the number of frames to skip after turning on HDR sensor streaming.

`unsigned int sensor_max_pipeline_frame_delay;` - Sets the number of frames that the pipeline can have.

`unsigned short sensor_num_fast_aec_frame_skip;` - Sets the number of frames to be skipped when fast AEC is used.

2.3.7 Pipeline delay

When the sensor configuration is changed (exposure change, etc.), this indicates the number of frames after which the sensor streams the affected frames.

`unsigned int sensor_max_pipeline_frame_delay;` - Sets the number of frames to which the sensor applies the change.

2.3.8 Default lens info

`default_lens_info` structure contains the physical description of the lens used with the sensor. As those settings are lens-specific, extract their values from the lens and camera module documentation.

```
typedef struct {
  float focal_length;
  float pix_size;
  float f_number;
  float total_f_dist;
  float hor_view_angle;
  float ver_view_angle;
  sensor_sensing_method_type_t sensing_method;
  float crop_factor; //depends on sensor physical dimentions
  float min_focus_distance;
} sensor_lens_info_t;
```

`focal_length` – Sets lens focal length.

`pix_size` – Sets lens unit cell size.

`f_number` – Sets F number of the lens.

`Total_f_dist` – Sets total focus distance. The value is in meters.

`hor_view_angle` – Sets horizontal view angle of the lens. The value is in degrees.

`ver_view_angle` – Sets horizontal view angle of the lens. The value is in degrees.

`sensing_method` – Sets sensing method. If the method is unknown, set to SENSOR_SMETHOD_NOT_DEFINED.

`crop_factor` – Sets lens crop factor. This value depends on lens and sensor physical dimensions, showing whether the lens is cropping out from the sensor matrix.

`min_focus_distance` – Sets lens min step of movement. If there are no special limitations from the lens, sets to 0.1.

2.3.9 Sensor physical info

`sensor_phy_info` structure contains information about physical dimensions of the sensor color matrix.

```
typedef struct {
  float pix_width;
  float pix_height;
}sensor_physical_dimensions_t;
```

`pix_width` – Sets the sensor pixel array width in millimeters. (cell size * effective pixel width)

`pix_height` – Sets the sensor pixel array height in millimeters. (cell size * effective pixel height)

Sensor documentation usually has sensor pixel array diagonal length and a single pixel dimension. Calculate the width and height based on the pixel dimensions and the pixel array size.

NOTE: Sensor documentation usually has the sensor pixel array diagonal length for the effective pixel array; this should not be mistaken for active pixel array.

2.3.10 Pixel array size info

`pixel_array_size_info` structure contains information about sensor pixel array. The active pixel array dimensions and the border pixels on each side are set in the structure.

```
typedef struct {
  sensor_dimension_t active_array_size;
  unsigned short left_dummy;
  unsigned short right_dummy;
  unsigned short top_dummy;
  unsigned short bottom_dummy;
} sensor_imaging_pixel_array_size;

typedef struct {
  int width;
  int height;
} sensor_dimention_t;
```

`active_array_size.width` – Sets sensor active pixel array width in number of pixels.

`active_array_size.height` – Sets sensor active pixel array height in number of lines.

`left_dummy` – Sets sensor buffer between effective and active pixel array on the left side. The value is in number of pixels.

`right_dummy` – Sets sensor buffer between effective and active pixel array on the right side. The value is in number of pixels.

`top_dummy` – Sets sensor buffer between effective and active pixel array on the top. The value is in number of lines.

`bottom_dummy` – Sets sensor buffer between effective and active pixel array on the bottom. The value is in number of lines.

2.3.11 Color level info

`color_level_info` structure contains information about sensor color limits.

```
typedef struct {
  unsigned short white_level;
  unsigned short r_pedestal;
  unsigned short gr_pedestal;
  unsigned short gb_pedestal;
  unsigned short b_pedestal;
} sensor_color_level_info;
```

`white_level` – Sets the value of sensor DAC for white. This is usually the max possible value that the DAC can output (1023 for RAW10).

`r_pedestal` – Sets black level/data pedestal for red channel. The value is for RAW10 data. For RAW12 multiply by 4.

`gr_pedestal` – Sets black level/data pedestal for green (on red line) channel. The value is for RAW10 data. For RAW12 multiply by 4.

`gr_pedestal` – Sets black level/data pedestal for green (on blue line) channel. The value is for RAW10 data. For RAW12 multiply by 4.

`b_pedestal` – Sets black level/data pedestal for blue channel. The value is for RAW10 data. For RAW12 multiply by 4.

2.3.12 Sensor stream info

`sensor_stream_info_array` structure contains information about sensor streams. It shows the number of stream types that the sensor is sending out.

Depending on its configuration the sensor can stream out a number of different data types (DT). That data is wrapped in different streams. In a stream there can be one or more different DTs. A Virtual Channel (VC) is assigned to each stream. The combination of DT and VC should be unique and is assigned a Channel ID (CID).

There are requirements/limitations on how the CID is given. The current MIPI CSI_Rx supports four VC, and it can have up to four CIDs for each VC, as shown in this table.

VC	CID			
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15

```
typedef struct _sensor_stream_info_array_t {
  sensor_stream_info_t sensor_stream_info[MAX_SENSOR_STREAM];
  unsigned short size;
} sensor_stream_info_array_t;

typedef struct _sensor_stream_info_t {
  unsigned short vc_cfg_size;
  struct sensor_csid_vc_cfg vc_cfg[MAX_SENSOR_DATA_TYPE];
  sensor_output_format_t pix_data_fmt[MAX_SENSOR_DATA_TYPE];
} sensor_stream_info_t;

struct sensor_csid_vc_cfg {
   unsigned char cid;
   unsigned char dt;
   unsigned char decode_format;
};
```

`size` – Sets the number of different streams the sensor provides.

`sensor_stream_info.vc_cfg_size` – Sets the number of data types in the current stream.

`sensor_stream_info.pix_fmt_fourcc` – Sets the format of the data in the current stream.

`sensor_stream_info.vc_cfg_size.cid` – Set the unique CID for the current data.

`sensor_stream_info.vc_cfg_size.dt` – Sets the current data type.

`sensor_stream_info.vc_cfg_size.decode_format` – Sets the current data decode format.

Example – Sensor with two streams. Stream 1 contains two data types: RAW10 and 0x35. Stream 2 contains only one data type: 0x35.

```
.sensor_stream_info_array =
  {
    .sensor_stream_info =
    {
      {
        .vc_cfg_size = 2,
        .vc_cfg =
        {
          {
            .cid = 0,
            .dt = CSI_RAW10,
            .decode_format = CSI_DECODE_10BIT
          },
          {
```

```
            .cid = 1,
            .dt = 0x35,
            .decode_format = CSI_DECODE_8BIT
          },
        },
        . pix_data_fmt =
        {
          SENSOR_BAYER,
          SENSOR_META,
        },
      },
      {
        .vc_cfg_size = 1,
        .vc_cfg =
        {
          {
            .cid = 1,
            .dt = 0x35,
            .decode_format = CSI_DECODE_8BIT
          },
        },
        . pix_data_fmt =
        {
          SENSOR_META,
        },
      },
    },
    .size = 2,
},
```


2.3.13 Sensor control settings

Sensor setting structures contain specific configurations. Each configuration is a collection of sensor registers and their values. A combination of registers and values commands the sensor to perform predefined tasks.

The groups are:

`start_settings` – Register set to command the sensor to start streaming out frames.

`stop_settings` – Register set to command the sensor to stop streaming out frames.

`groupon_setting` – Register set to command the sensor to set group parameter hold logic.

`groupoff_settings` – Register set to command the sensor to release group parameter hold logic.

`embedded_data_enable_settings` – Register set to command the sensor to enable streaming out of embedded data.

`embedded_data_disable_settings` – Register set to command the sensor to disable streaming out of embedded data.

All the configurations have the same structure:

```
struct camera_i2c_reg_setting {
  struct camera_i2c_reg_array *reg_setting;
  unsigned short size;
  enum camera_i2c_reg_addr_type addr_type;
  enum camera_i2c_data_type data_type;
  unsigned short delay;
};
struct camera_i2c_reg_array {
 unsigned short reg_addr;
 unsigned short reg_data;
 unsigned int delay;
};
```

`size` – Sets the register number that is set in the current configuration.

`addr_type` – Sets I2C register address length.

`data_type` – Sets I2C register data length.

`delay` – Sets delay (in ms) to wait after the configuration is done.

`reg_setting.reg_addr` – Sets the sensor register address of the register that will be updated.

`reg_setting.reg_data` – Sets the data that need to be written in the sensor register.

`reg_setting.delay` – Sets delay (in ms) to wait after the register configuration is done.

NOTE: If `reg_setting.delay` is not set, the code assumes it equals 0.

2.3.14 Sensor init settings

When a sensor is brought out of shutdown, its registers are reset to their default values. Not all of the default values are suitable for the sensor to stream out frames. An init configuration of the sensor is needed to prepare it to work. This configuration is done by setting the registers in `init_setttings_array`. Its structures are almost identical to sensor control settings structures. The main difference is that in the init settings we can have more than one set of settings.

```
struct sensor_lib_reg_settings_array {
  struct camera_i2c_reg_setting reg_settings[MAX_RESOLUTION_MODES];
  unsigned int                          size;
};
```

NOTE: In current source code `MAX_RESOLUTION_MODES` is set to 10.

`reg_settings` – Has the same configuration setup as sensor control settings. Review Section 2.3.1.2 for more details.

`size` – Sets the number of init configurations. The size should equal the available configurations and cannot be bigger than `MAX_RESOLUTION_MODES`.

2.3.15 Sensor resolution settings

Once the sensor is brought out of shutdown and has been configured initially, it is ready for work. Depending on the use case we want the sensor to achieve, a different sensor mode is selected and the sensor is configured with it. The sensor mode configurations are kept in `res_settings_array`, which uses the same structure as `init_setttings_array`.

```
struct sensor_lib_reg_settings_array {
  struct camera_i2c_reg_setting reg_settings[MAX_RESOLUTION_MODES];
  unsigned int                          size;
};
```

NOTE: In current source code `MAX_RESOLUTION_MODES` is set to 10.

`reg_settings` – It has the same configuration setup as sensor control settings. Review Section 2.3.1.2 for a detailed explanation.

`size` – Sets the number of sensor mode configurations. The size should equal the available configurations and cannot be bigger than `MAX_RESOLUTION_MODES`.

2.3.16 Sensor resolution configuration table

`out_info_array` – structure contains detailed information about each sensor mode/resolution configuration we set in `res_settings_array`. The system uses this information to select the sensor mode that provides the use case requirements (frame rate, resolutions, etc.) with best possible Image Quality (IQ).

NOTE: The sensor mode/resolution descriptions need to be in the same order as in `res_settings_array`. Description 1 must be for resolution 1.

```
struct sensor_lib_out_info_array {
  /* sensor output for each resolutions */
  struct sensor_lib_out_info_t out_info[MAX_RESOLUTION_MODES];

  /* Number of valid entries in out_info array */
  unsigned short size;
};

struct sensor_lib_out_info_t {
  unsigned short x_output;
  unsigned short y_output;
  unsigned short line_length_pclk;
  unsigned short frame_length_lines;
  unsigned int vt_pixel_clk;
  unsigned int op_pixel_clk;
  unsigned short binning_factor;
  float    min_fps;
  float    max_fps;
  unsigned int mode;
  unsigned int offset_x;
  unsigned int offset_y;
  unsigned int scale_factor;
};
```

`size` – Sets the number of sensor mode configurations. The size should equal the available configurations and cannot be bigger than `MAX_RESOLUTION_MODES`.

`out_info.x_output` – Sets the number of active pixels the sensor outputs per line. The value is number of pixels.

`out_info.y_output` – Sets the number of active lines the sensor outputs per frame. The value is number of lines.

`out_info.line_length_pclk` – Sets the total number of pixels the sensor outputs per line. This includes the active pixels and the horizontal blanking. The value is number of pixels.

`out_info.frame_length_lines` – Sets the total number of lines the sensor outputs per frame. This includes the active lines and the vertical blanking. The value is number of lines.

`out_info.vt_pixel_clk` – Sets video domain timing clock value, which is generated by the sensor internal PLL module. To calculate its value, use the external clock (EXTCLK) frequency and the PLL settings. The sensor uses this clock for several internal tasks, and its value controls several sensor parameters – shutter/exposure time, frame rate, sensor internal ISP processing, etc.

`out_info.op_pixel_clk` – Sets output domain timing clock value, which is generated by the sensor internal PLL module. To calculate its value, use the external clock (EXTCLK) frequency and the PLL settings. The sensor uses this clock for its output modules. This clock is used as a base for the MIPI/CSI_Tx speeds.

NOTE: `vt_pixel_clk` and `op_pixel_clk` are calculated based on PLL settings. Do not calculate them based on parameter values, which are derived from the clock value (frame rate, CSI speed, etc.)

`out_info.binning_factor` – Sets sensor mode binning factor.

`out_info.min_fps` – Sets sensor mode/resolution minimum frame rate. The value is used by AEC and Auto Frame Rate (AFR) to control the frame rate and exposure times.

`out_info.max_fps` – Sets sensor mode/resolution maximum frame rate. This is the sensor mode/resolution frame rate, when the AFR is not triggered.

`out_info.mode` – Sets sensor mode/resolution type. The system uses this parameter to sort out the sensor mode/resolution purpose. Currently available mode types are:

- `SENSOR_DEFAULT_MODE` – Normal sensor modes – preview, video, snapshot, ZSL, etc.
- `SENSOR_HFR_MODE` – High frame rate mode – modes with a frame rate greater than 30 fps are used for high-speed video recording.
- `SENSOR_HDR_MODE` – High Dynamic Range (HDR) mode – use for HDR video and image recording.

`out_info.offset_x` – Sets sensor mode/resolution horizontal offset of the frame output compared to the pixel array start.

`out_info.offset_y` – Sets sensor mode/resolution vertical offset of the frame output compared to the pixel array start.

`out_info.scale_factor` – Sets sensor scale factor. The value is a ratio, and the sensors are able only to downscale; it must be 1.00 or greater at all times.

2.3.17 CSI parameters configuration

`csi_params_array` structure contains detailed information for CSI2_Rx configurations. These configurations overlap to some degree with Section 2.3.12 Sensor stream info configuration, but should not be equated.

`sensor_stream_info_array` contains information about the streams sent by the sensor, where `csi_params_array` contains information about the configuration of the CSI2_Rx receiver.

```
struct sensor_lib_csi_params_array {
  struct sensor_csi2_params csi2_params[MAX_RESOLUTION_MODES];
  unsigned short size;
};
```

`size` – Sets the number of sensor mode configurations. The size should equal the available configurations and cannot be bigger than `MAX_RESOLUTION_MODES`.

CSI2_Rx configuration has two parts:

- CSID – Contains the logical configuration.
- CSIHPY – Contains the physical configuration.

```
struct sensor_csi2_params {
    struct sensor_csid_params csid_params;
    struct sensor_csiphy_params csiphy_params;
};
```

2.3.17.1 CSI CSID configuration

```
struct sensor_csid_params {
  unsigned char lane_cnt;
  unsigned short lane_assign;
  unsigned char phy_sel;
  unsigned int csi_clk;
  struct sensor_csid_lut_params lut_params;
};
```

`lane_cnt` – Sets CSI2 lane count. Depending on the sensor mode configuration, one, two, or four lanes can be used for streaming out.

`lane_assign` – For CSI2 transmission of data, a clock and at least one data lane are needed. Depending on the hardware layout, the used lanes can be connected to different lanes of CSI2_Rx PHY. To do so requires configuration information.

Bit position	Represents
15:12	Camera sensor lane number connected to data lane 4 of a given PHY on the MSM™ device side
11:8	Camera sensor lane number connected to data lane 3 of a given PHY on the MSM side
7:4	Camera sensor lane number connected to data lane 2 of a given PHY on the MSM side
3:0	Camera sensor lane number connected to data lane 0 of a given PHY on the MSM side

NOTE: Lane 1 is reserved for the clock lane. Wiring and setting it to a data lane is prohibited.

`phy_sel` – Sets which CSI2_Rx PHY is used. As this is the physical connection, this value must be verified with the hardware schematics.

`lut_params` – Set data steam info for the current mode.

```
struct sensor_csid_lut_params {
  unsigned char num_cid;
  struct sensor_csid_vc_cfg vc_cfg_a[MAX_CID];
  struct sensor_csid_vc_cfg *vc_cfg[MAX_CID];
};
```

`lut_params.num_cid` – Sets the number of unique CID in the current mode data stream.

`lut_params.vc_cfg` – Sets configuration for each data type into the data stream. See Section 2.3.12 for a more detailed explanation of the structure elements.

NOTE: The maximum number of CIDs is 16.

2.3.17.2 CSI CSIPHY configuration

```
struct sensor_csiphy_params {
  unsigned char lane_cnt;
  unsigned char settle_cnt;
  unsigned short lane_mask;
  unsigned char combo_mode;
  unsigned char csid_core;
  unsigned int csiphy_clk;
};
```

`lane_cnt` – Sets CSI2 lane count. Depending on the sensor mode configuration one, two or four lanes can be used for streaming out.

`settle_cnt` – For CSI_Tx (the sensor) and CRI_Rx (the device) to work properly, a period for syncing between them is required. This time is set here as number of timer clock ticks. It has to be between the MIN and MAX values calculated by the formulas:

MIN [Settle count * T(Timer clock)] > T(HS_SETTLE)_MIN

MAX [Settle count * T(Timer clock)] < T(HS-PREPARE)+T(HS_ZERO) - 4*T(Timer clock)

`lane_mask` – Sets which of the CSI2 lanes is used in the current configuration. The value is a combination of flags.

Bit position	Represents
7:5	Reserved
4	Is data lane 3 valid? ▪ 0 – No ▪ 1 – Yes
3	Is data lane 2 valid? ▪ 0 – No ▪ 1 – Yes

Bit position	Represents
2	Is data lane 1 valid? ▪ 0 – No ▪ 1 – Yes
1	Is clock lane (lane 1) valid? ▪ 0 – No ▪ 1 – Yes **Note:** This should always be set to 1.
0	Is data lane 0 valid? ▪ 0 – No ▪ 1 – Yes

`combo_mode` – Set CSI PHY combo mode. >>>>>>>>>>>

`csid_core` – Set which CSI2_Rx core is used. Usually it is set to correspond to `csid_params.phy_sel`.

2.3.18 Sensor crop array configuration

`crop_params_array` structure contains information about sensor modes/resolutions cropping configuration.

```
struct sensor_lib_crop_params_array{
  struct sensor_crop_parms_t crop_params[MAX_RESOLUTION_MODES];
  unsigned short size;
};

struct sensor_crop_parms_t {
  unsigned short top_crop;
  unsigned short bottom_crop;
  unsigned short left_crop;
  unsigned short right_crop;
};
```

`size` – Sets the number of configuration steps. The size should equal the available configurations, and cannot be bigger than `MAX_RESOLUTION_MODES`.

`crop_params.top_crop` – Sets the crop from the top side for the current mode.

`crop_params.bottom_crop` – Sets the crop from the bottom side for the current mode.

`crop_params.left_crop` – Sets the crop from the left side for the current mode.

`crop_params.right_crop` – Sets the crop from the right side for the current mode.

2.3.19 Chromatix configuration

The Chromatix headers contain IQ tuning of the hardware and software blocks used in frame processing. Each Chromatix header is built in its unique library, and the headers are dynamically loaded and unloaded depending on the use case and the selected sensor mode/resolution.

`chromatix_array` structure contains information about which Chromatix headers are used for each sensor mode/resolution in the different use cases.

```
struct sensor_lib_chromatix_array {
  struct sensor_lib_chromatix_t sensor_lib_chromatix[MAX_RESOLUTION_MODES];
  unsigned short size;
};

struct sensor_lib_chromatix_t {
  char *common_chromatix;
  char *camera_preview_chromatix;
  char *camera_snapshot_chromatix;
  char *camcorder_chromatix;
  char *liveshot_chromatix;
  char *cpp_chromatix;
  char *cpp_snapchromatix;
  char *cpp_videochromatix;
  char *cpp_liveshotchromatix;
  char *postproc_chromatix;
};
```


`size` – Sets the number of configuration steps. The size should equal the available configurations, and cannot be bigger than `MAX_RESOLUTION_MODES.`

`sensor_lib_chromatix.common_chromatix` – Sets pointer to common (VFE) chromatix for the current sensor mode/resolution.

`sensor_lib_chromatix.camera_preview_chromatix` – Sets pointer to preview chromatix for the current sensor mode/resolution. It is loaded when the mode is used for image preview.

`sensor_lib_chromatix.camera_snapshot_chromatix` – Sets pointer to snapshot chromatix for the current sensor mode/resolution. It is loaded when the mode is used for snapshot.

`sensor_lib_chromatix.camcorder_chromatix` – Sets pointer to camcorder chromatix for the current sensor mode/resolution. It is loaded when the mode is used for video preview/recording.

`sensor_lib_chromatix.liveshot_chromatix` – Sets pointer to liveshot chromatix for the current sensor mode/resolution. It is loaded when liveshot (snapshot while video recording) is made.

`sensor_lib_chromatix.cpp_chromatix` – Sets pointer to cpp chromatix for the current sensor mode/resolution. It will be is loaded when the mode is used for image preview.

`sensor_lib_chromatix.cpp_snapchromatix` – Sets pointer to cpp snapshot chromatix for the current sensor mode/resolution. It is loaded when the mode is used for snapshot.

`sensor_lib_chromatix.cpp_videochromatix` – Sets pointer to cpp video chromatix for the current sensor mode/resolution. It is loaded when the mode is used for video preview/recording.

`sensor_lib_chromatix.cpp_liveshotchromatix` – Sets pointer to cpp liveshot chromatix for the current sensor mode/resolution. It is loaded when liveshot (snapshot while video recording) is made.

`sensor_lib_chromatix.postproc_chromatix` – Sets pointer to postprocessing chromatix for the current sensor mode/resolution. It is used for software postprocessing.

Setting of the pointer to the chromatix headers is done, by using define - `SENSOR_LOAD_CHROMATIX (name, mode)`, where `name` is sensor name and `mode` is the chromatix name.

Example:

```
        .common_chromatix =
  SENSOR_LOAD_CHROMATIX(SENSOR_MODEL, common),
.camera_preview_chromatix =
  SENSOR_LOAD_CHROMATIX(SENSOR_MODEL, snapshot),
.camera_snapshot_chromatix =
  SENSOR_LOAD_CHROMATIX(SENSOR_MODEL, snapshot),
.camcorder_chromatix =
  SENSOR_LOAD_CHROMATIX(SENSOR_MODEL, default_video),
.liveshot_chromatix =
  SENSOR_LOAD_CHROMATIX(SENSOR_MODEL, liveshot),
.cpp_chromatix =
  SENSOR_LOAD_CHROMATIX(SENSOR_MODEL, cpp),
```

2.3.20 Exposure control configuration

The AEC works by setting exposure and gain values. Both are sent to the sensor to set them. On its side the sensor driver calculates how best to achieve the required values from the AEC, and configure the proper register and their values.

The exposure is passed by the AEC as number of lines, so it can be directly applied in the sensor registers.

The gain passed by the AEC is real gain, but the sensor does not work with real gain and needs it to be converted to register gain. This conversion is sensor-specific and is based on a specific formula, which should be taken from the sensor documentation. Usually only the formula for converting register gain into real gain is given, so its opposite (from real to register) needs to be generated. Based on those two formulas functions need to be written for the conversions. Those functions are called <sensor>_real_to_register_gain() and <sensor>_register_to_real_gain(), and their source code is placed in the <sensor>_libs.c file.

`exposure_func_table` structure contains pointers to these sensor-specific functions.

```
typedef struct {
  int (*sensor_calculate_exposure) (float, unsigned int,
    sensor_exposure_info_t *);
  int (*sensor_fill_exposure_array)(unsigned int, unsigned int, unsigned int,
unsigned int,
    int, unsigned int, struct camera_i2c_reg_setting *);
} sensor_exposure_table_t;
```

`sensor_calculate_exposure` – Sets pointer to a sensor-specific function which calculates exposure and gain values based on sensor-specific formulas.

`sensor_fill_exposure_array` – Sets pointer to sensor-specific function which sets the exposure configuration array with register addresses and values.

NOTE: The functions must be manually generated and integrated.

`unsigned char sync_exp_gain; -` This flag shows the system whether the sensor takes care of exposure and gain synchronization or the system is expected to do so.

`int64_t adc_readout_time; -` Time needed by the sensor ADC to read out and convert an analog signal from the pixel array into a digital value.

2.3.21 Metadata configuration

`meta_data_out_info_array` structure contains information about sensor metadata.

```
struct sensor_lib_meta_data_info_array {
 struct sensor_meta_data_out_info_t meta_data_out_info[MAX_META_DATA_SIZE];
 unsigned short size;
};

struct sensor_meta_data_out_info_t {
  unsigned int width;
  unsigned int height;
  enum sensor_stats_type stats_type;
};
```

`size` – Sets the number of different metadata configurations that are available.

NOTE: In the current source code, the maximum number of metadata configurations is 3.

`meta_data_out_info.width` – Sets width of metadata. The value is in number of pixels.

`meta_data_out_info.height` – Sets height of metadata. The value is in number of lines.

`meta_data_out_info.stats_type` – Sets metadata data type.

2.3.22 HDR AWB LSC configuration

With the sensor development, directly streaming out HDR frames becomes available. When this sensor functionality is used, the system must pass to the sensor the HDR configuration parameters, and set the HDR related registers with proper values.
`video_hdr_awb_lsc_func_table` structure contains configuration information for HDR capabilities of the sensor.

```
typedef struct {
  int (*sensor_fill_awb_array)(unsigned short, unsigned short,
    struct camera_i2c_seq_reg_setting *);
  unsigned short awb_table_size;
  int video_hdr_capability;
} sensor_video_hdr_table_t;
```

`sensor_fill_awb_array` – Sets pointer to sensor-specific function which set the HDR configuration array with register addresses and values.

NOTE: The function must be manually generated and integrated.

`awb_table_size` – Sets the size if the HDR configuration array. It must match the number of registers set in `sensor_fill_awb_array`.

`video_hdr_capability` – Sets the sensor HDR video capabilities. This value is the sum of different system dependent flags.

Example:

```
.video_hdr_capability = (1 << 8) | (1 << 15) | 0,
```

NOTE: The flags values used for setting `video_hdr_capability` are defined inside the camera hall in cam_types.h. The file path is : /hardware/qcom/camera/Qcamera2/stack/common/cam_types.h

2.3.23 Sensor rolloff configuration

Because of the physical limitation of the sensor and the lens, there is lens shading, which leads to lower light levels in the corner pixels of the sensor. Depending on the quality of the lens and the camera module configuration, the lens shading can be higher or lower. To compensate for this a Lens Shading Correction (LSC) or rolloff compensations is performed from the system. Some sensors can perform such LSC in their internal ISP. `rolloff_config` structure contains configuration information for LSC capabilities of the sensor.

```
typedef struct {
    unsigned char enable;
    sensor_full_size_info_t full_size_info;
} sensor_rolloff_config;

typedef struct {
  unsigned int full_size_width;
```

```
  unsigned int full_size_height;
  unsigned int full_size_left_crop;
  unsigned int full_size_top_crop;
} sensor_full_size_info_t;
```

`enable` – Sets flag (TRUE/FALSE) to show the system whether the sensor LSC logic is used.

`full_size_info.full_size_width` – Sets the width of the LSC table that can be used.

`full_size_info.full_size_height` – Sets the height of the LSC table that can be used.

`full_size_info.full_size_left_crop` – Sets the crop on left size, between sensor array and LSC table.

`full_size_info.full_size_top_crop` – Sets the crop on top size, between sensor array and LSC table.

Rolloff is configured and controlled through structures in the sensor-specific Chromatix headers. Those structures are filled by IQ personnel after tests are performed with the camera module. If sensor LSC is available and is used, it had to be communicated with the IQ personnel, as the structures in the Chromatix need to be updated accordingly. Otherwise we have double LSC which introduces additional image distortions.

2.3.24 Test pattern configuration

The sensor might have a built-in pattern generator. By setting a dedicated register, the sensor streams out generated patterns.

```
typedef struct {
  sensor_test_pattern_settings
test_pattern_settings[SENSOR_TEST_PATTERN_MAX];
  unsigned char size;
  struct sensor_test_mode_addr_t solid_mode_addr;
} sensor_test_info;
```

`test_pattern_settings` – Sets test pattern mode and its register.

```
typedef struct {
  sensor_test_pattern_t mode;
  struct camera_i2c_reg_setting_array settings;
} sensor_test_pattern_settings;
```

```
typedef enum {
  SENSOR_TEST_PATTERN_OFF,
  SENSOR_TEST_PATTERN_SOLID_COLOR,
  SENSOR_TEST_PATTERN_COLOR_BARS,
  SENSOR_TEST_PATTERN_COLOR_BARS_FADE_TO_GRAY,
  SENSOR_TEST_PATTERN_PN9,
  SENSOR_TEST_PATTERN_CUSTOM1,
  SENSOR_TEST_PATTERN_MAX,
```

```
} sensor_test_pattern_t;
```

For solid color test pattern mode, we can set the frame color.

```
struct sensor_test_mode_addr_t {
  unsigned short r_addr;
  unsigned short gr_addr;
  unsigned short gb_addr;
  unsigned short b_addr;
};
```

2.3.25 ADC readout time

This is the readout time (in nanoseconds) of the sensor's analog-to-digital converter. Usually it is the minimum line time when the sensor is running at the maximum pixel clock.

NOTE: This is the sensor module's own information. Refer to the sensor vendor for more information.

2.3.26 Exposure setting delay

The sensor exposure register can be set in a different frame time.

```
unsigned char app_delay[SENSOR_DELAY_MAX];


typedef enum {
  SENSOR_DELAY_EXPOSURE,          /* delay for exposure*/
  SENSOR_DELAY_ANALOG_SENSOR_GAIN,  /* delay for sensor analog gain*/
  SENSOR_DELAY_DIGITAL_SENSOR_GAIN, /* delay for sensor digital gain*/
  SENSOR_DELAY_ISP_GAIN,          /* delay for sensor ISP (error) gain*/
  SENSOR_DELAY_MAX,
} sensor_delay_type_t;
```

`SENSOR_DELAY_EXPOSURE` – Sets the exposure of frame N at frame N + delay
`SENSOR_DELAY_ANALOG_SENSOR_GAIN` – Sets the analog gain register at frame N + delay
`SENSOR_DELAY_DIGITAL_SENSOR_GAIN` – Sets the digital gain register at frame N + delay
`SENSOR_DELAY_ISP_GAIN` – Passes the isp digital gain to the isp module at frame N + delay

2.3.27 Noise profile

This is for noise model coefficients for each color filter channel corresponding to the sensor amplification (S) and sensor readout noise (O). These coefficients are used in a two-parameter noise model to describe the amount of noise present in the image for each color channel. The noise model used here is: N(x) = sqrt(Sx + O)

```
struct sensor_noise_coefficient_t {
  double gradient_S;
  double offset_S;
  double gradient_O;
  double offset_O;
};
```

NOTE: These parameters are generated by the tuning team.

3 EEPROM Driver Bringup

This chapter provides information necessary for camera EEPROM driver bringup on the Android platform with an MSM8994/92 device.

The EEPROM driver bringup includes the following tasks/steps:

1. EEPROM driver requirements.
2. Configuring the DTSI.
3. Software driver development and integration.

3.1 EEPROM driver requirements

The following documents are needed for the EEPROM driver bringup:

- EEPROM datasheet
  - Powerup sequence
  - Slave address
  - Read / write sequence
- Schematics
  - Regulators for powerup
  - Clock
  - GPIO pins
- Memory map
  - Describes the memory layout and the address / offset where the calibration data is programmed

3.2 Configuring the DTSI

EEPROM data is read during the bootup of the device. The memory map needs to be translated to the appropriate entries in the dtsi. The regulators, clock signal, the powerup sequence, slave address and the reading sequence must be specified in the <msm8994/92>-camera-sensor-mtp.dtsi file in kernel/arch/arm/boot/dts/qcom/.

Example:

```
eeprom0: qcom,eeprom@0 {
        cell-index = <0>;
        reg = <0>;
        qcom,eeprom-name = "onsemi_cat24c32";
        compatible = "qcom,eeprom";
        qcom,slave-addr = <0xa0>;
```

```
        qcom,cci-master = <0>;
        qcom,num-blocks = <1>;
        qcom,page0 = <0 0 0 0 0 0>;
        qcom,poll0 = <0 0 0 0 0 0>;
        qcom,saddr0 = <0xa0>;
        qcom,mem0 = <2245 0x00 2 0 1 0>;

        cam_vio-supply = <&pm8994_lvs1>;
        qcom,cam-vreg-name = "cam_vio";
        qcom,cam-vreg-min-voltage = <0>;
        qcom,cam-vreg-max-voltage = <0>;
        qcom,cam-vreg-op-mode = <0>;
        qcom,cam-power-seq-type = "sensor_vreg";
        qcom,cam-power-seq-val = "cam_vio";
        qcom,cam-power-seq-cfg-val = <1>;
        qcom,cam-power-seq-delay = <1>;
};
```

- `cell-index = <0>;`
  The index is assigned to subdev_id for eeprom subdev. This should be unique means we should not use the already existing values.
- `qcom,eeprom-name = "onsemi_cat24c32";`
  The name must match that specified in corresponding sensor driver in order to link it to the sensor. Example: imx230_lib.h

```
.sensor_slave_info =
{
.sensor_name = SENSOR_MODEL,
.eeprom_name = " onsemi_cat24c32",
```

  /*This name should be the same as the name defined in the property“qcom,eeprom-name” in dtsi.*/

```
}
```

- `qcom,slave-addr = <0xa0>;`
  specifies the 8-bit slave address as 0xa0.
- `cam_vio-supply = <&pm8994_lvs1>;`
  specifies the power source as `pm8994_lvs1`
- `qcom,mem0 = <2245 0x00 2 0 1 0>;`
  `= <num_bytes address address-type n/a n/a delay>`
  `= < address-type : 1 byte : 2 word>`
  specifies the reading 2245 bytes from eeprom address 0x00.
- `qcom,cam-power-seq-type = "sensor_vreg";`

```
qcom,cam-power-seq-val = "cam_vio";
qcom,cam-power-seq-cfg-val = <1>;
qcom,cam-power-seq-delay = <1>;
```

  specifies the powerup sequence. In this example, one regulator is used for powering up.

Other supported statement for an eeprom node:

- `qcom,page0 = <1 0x0a02 2 0x00 1 5>;`

  = <*valid address address-type data data-type delay*>

  = < *address-type : 1 byte : 2 word*>

  = < *data-type : 1 byte : 2 word*>

  set the page number by writing page number (0x00) to address (0xa02).
- `qcom,pageen0 = <1 0x0a00 2 0x01 1 5>;`

  enable read mode by writing data (0x01) to address (0xa00)
- `qcom,poll0 = <1 0x0a01 2 0x01 1 5>;`

  Poll read mode – polling address (0x0a01) for data (0x01)

3.3 Software driver development and integration

The EEPROM software driver is placed in the user space and configures how EEPROM data is used for calibration. The EEPROM driver implements a list of functions, which are assigned to the following function pointer structure.

- `.get_calibration_items = cat24c32_get_calibration_items`, – Sets the availability of each type of data
  - `.get_calibration_items` – The function pointer name
  - `cat24c32_get_calibration_items` - The new function the user needs to add

    ```
    void cat24c32_get_calibration_items(void *e_ctrl)
    {
     sensor_eeprom_data_t *ectrl = (sensor_eeprom_data_t *)e_ctrl;
     eeprom_calib_items_t *e_items = &(ectrl->eeprom_data.items);

     e_items->is_wbc = datapresent ? TRUE : FALSE;
     e_items->is_afc = datapresent ? TRUE : FALSE;
     e_items->is_lsc = datapresent ? TRUE : FALSE;
     e_items->is_dpc = FALSE;
     e_items->is_insensor = datapresent ? TRUE : FALSE;
     e_items->is_ois = FALSE;

    }
    ```

    This function is to specify which calibration is available or enabled for this eeprom. For example, when ‘e_items->is_wbc = TRUE’, white balance calibration is executed during camera stream on. Otherwise, that calibration is not performed.
- `.format_calibration_data` – The user needs to write functions to link to it, typically consisting of these functions:

- `format_afdata`
  - fill struct `e_ctrl->eeprom_data.afc` with eeprom data

```
typedef struct {
  unsigned short    macro_dac;
  unsigned short    infinity_dac;
  unsigned short    starting_dac;
} afcalib_data_t;
```

Eeprom typically contains AF macro, infinity and starting DAC value. If only macro and infinity data is available, assign starting_dac as infinity DAC.

- `format_wbdata`
  - fill struct `e_ctrl->eeprom_data.wbc` with eeprom data

```
typedef struct {
  float   r_over_g[AGW_AWB_MAX_LIGHT];
  float   b_over_g[AGW_AWB_MAX_LIGHT];
  float   gr_over_gb;
} wbcalib_data_t;
```

White balance data is stored based on lighting condition type. Populate the struct based on the data light types which are available from the eeprom.

Calibration lighting condition for AWB	**Assignment to calibration struct**
One type only	Assign to all types
Two types 5000 K and 3000 K	Assign 3000 K to the following: AGW_AWB_A, AGW_AWB_HORIZON, AGW_AWB_CUSTOM_A, AGW_AWB_U30
	Assign 5000 K to the rest.
Three types D50, TL84, A	Assign D50 to the following: AGW_AWB_D50, AGW_AWB_D65, AGW_AWB_D75, AGW_AWB_NOON, AGW_AWB_CUSTOM_DAYLIGHT
	Assign TL84 to the following: AGW_AWB_WARM_FLO AGW_AWB_COLD_FLO, AGW_AWB_CUSTOM_FLO, AGW_AWB_WARM_FLO
	Assign A to the following: AGW_AWB_A, AGW_AWB_HORIZON, AGW_AWB_CUSTOM_A, AGW_AWB_U30

- format_lsc

  fill struct e_ctrl->eeprom_data.lsc with eeprom data

```
typedef struct{
  unsigned short        mesh_rolloff_table_size;     // TableSize
  float                  r_gain[MESH_ROLLOFF_SIZE];   // RGain
  float                  gr_gain[MESH_ROLLOFF_SIZE];  // GRGain
  float                  gb_gain[MESH_ROLLOFF_SIZE];  // GBGain
  float                  b_gain[MESH_ROLLOFF_SIZE];   // BGain
} mesh_rolloff_array_type;
```

Lens shading table is typical 13x17 in size and contains r, gr, gb, and b gain values for each grid. Lens shading data is also stored based on light types. Populate each light types with same principle as for AWB data.

- format_pdafdata

  fill struct e_ctrl->eeprom_data.pdafc with eeprom data, in case PDAF is supported for the camera module. Populate the corresponding PDAF calibration struct according to PDAF types.

- .get_raw_data, and .get_ois_raw_data – Functions to support in-sensor calibration (HW calibration)
  - Format functions for in-sensor calibration should store data in static memory
  - get_raw_data or get_ois_raw_data passes formatted calibration data to sensor module SW.
  - Sensor module SW performs in-sensor calibration via I2C
- .do_[xxx]_calibration – Performs calibration on chromatix/tuning headers
  - Common function available in eeprom.c
  - Users can also write their own functions for calibration

4 Actuator Driver Bringup

This chapter provides the necessary information for camera actuator driver bringup on the Android platform with an MSM8994/92 device.

The actuator driver bringup includes the following steps:

1. Actuator driver requirements.
2. Software driver development and integration.

4.1 Actuator driver requirements

The following documents are needed for the actuator driver bringup.

- Actuator datasheet
  - Slave address
  - Registers needed for configuring and moving lens.
- Actuator application note
  - Lens driving instructions

4.2 Software driver development and integration

The source code is placed in the user space. The actuator driver is created and compiled to create a library which is then loaded at run time.

To load the actuator library that corresponds with a sensor, the actuator library name configured in Android.mk should match the actuator name defined in a sensor library [2.3.1].

Access the parameters and structures for the actuator driver using the source code through the actuator handler:

```
static actuator_driver_ctrl_t actuator_lib_ptr =
{
        . actuator_params =
        ...
}
```

Its type (`actuator_driver_ctrl_t`) is defined in :

vendor/qcom/proprietary/mm-camerasdk/sensor/includes/actuator_driver.h.

Following is a description of the actuator driver structure and its parameters.

4.2.1 Actuator driver parameters

The actuator driver parameters contain actuator module and basic tuning (damping) information.

```
typedef struct _actuator_driver_params {
    actuator_params_t actuator_params;
    actuator_tuned_params_t actuator_tuned_params;
} actuator_driver_params_t;
```

4.2.1.1 Actuator parameters

`actuator_params` contains the actuator module information and initial register setting data. Refer to the actuator specification.

```
typedef struct _actuator_params {
  char module_name[MAX_NAME_SIZE];
  char actuator_name[MAX_NAME_SIZE];
  unsigned int i2c_addr;
  enum camera_i2c_data_type i2c_data_type;
  enum camera_i2c_reg_addr_type i2c_addr_type;
  enum actuator_module_type act_type;
  unsigned short data_size;
  struct actuator_reg_tbl_t reg_tbl;
  unsigned short init_setting_size;
  struct actuator_reg_settings_t init_settings[MAX_ACTUATOR_INIT_SET];
} actuator_params_t;
```

Structure = actuator_params		
Structure members	**Definition**	**Example**
module_name	Defines the module name.	*.module_name = "onsemi",*
actuator_name	Defines the actuator name.	*.actuator_name = "lc898122",*
i2c_addr	Slave address in 8 bits.	*.i2c_addr = 0x48,*
i2c_data_type	Data type (BYTE / WORD)	*.i2c_data_type = CAMERA_I2C_WORD_DATA,*
i2c_addr_type	Address type (BYTE / WORD)	*.i2c_addr_type = CAMERA_I2C_WORD_ADDR,*
act_type	Defines the actuator type. VCM BIVCM PIEZO	*.act_type = ACTUATOR_TYPE_VCM,*
data_size	Number of bits used to program the DAC.	*.data_size = 10,*
reg_tbl	The actuator table information. 1. reg_write_type can have the following values for VCM type:	*.reg_tbl =* *{* *.reg_tbl_size = 1,*

Structure = actuator_params		
Structure members	**Definition**	**Example**
	ACTUATOR_WRITE_HW_DAMP (Configure Damping parameters) ACTUATOR_WRITE_DAC (Position control) 2. hw_mask is the bit mask for hw_params in damping_params when HW damping is used. 3. reg_addr is the actuator control register address, 0xFFFF for no specific register address. If the MSB and LSB DAC registers are both available, the MSB DAC register address comes first, followed by the LSB DAC register address 4. hw_shift is the number of bits shift needed for hw_params before write to register. 5. Data_shift is the number of bits shift needed for data before write.	`.reg_params = { { .reg_write_type = ACTUATOR_WRITE_DAC, .hw_mask = 0x00000000, .reg_addr = 0x0380, .hw_shift = 0, .data_shift = 6, } } }`
init_setting_size	Defines the initial register settings.	`.init_setting_size = 3,`
init_settings	Defines the initial table size.	`.init_settings = { { .reg_addr = 0x0302, .addr_type = CAMERA_I2C_WORD_ADDR, .reg_data = 0x01, . data_type = CAMERA_I2C_BYTE_DATA, .i2c_operation = ACTUATOR_I2C_OP_WRITE, .delay = 0, } }`

NOTE: Check the register format in the actuator specification to program the DAC.

4.2.1.2 Actuator tuned parameters

`actuator_tuned_params` contains tuning related values including a range of actuator position, ringing and damping control.

```
typedef struct {
  unsigned short scenario_size[NUM_ACTUATOR_DIR];
  unsigned short ringing_scenario[NUM_ACTUATOR_DIR][MAX_ACTUATOR_SCENARIO];
  short initial_code;
  unsigned short region_size;
  struct region_params_t region_params[MAX_ACTUATOR_REGION];
  struct damping_t damping[NUM_ACTUATOR_DIR][MAX_ACTUATOR_SCENARIO];
} actuator_tuned_params_t;
```

Structure = actuator_tuned_params		
Structure member	**Definition**	**Example**
scenario_size	The “scenario” is for the damping configuration which depends on the jump step and direction of lens displacement. The damping parameters also depend on the region in which a particular scenario is handled. Defines scenario size in both directions (move from near to far and from far to near).	*.scenario_size =* *{* *1, /* MOVE_NEAR */* *1, /* MOVE_FAR */* *},*
ringing_scenario	Defines the ringing parameters - boundary step size (based on DAC table) of causing a ringing effect.	*.ringing_scenario =* *{* */* MOVE_NEAR */* *{* *200,* *},* */* MOVE_FAR */* *{* *200,* *},* *},*
initial_code	DAC code of the initial step.	*.initial_code = 70,*
region_size	Defines the number of regions through which the actuator moves linearly.	*.region_size = 1,*
region_params	Defines the range of steps in the region and DAC code size for each step.	*.region_params =* *{* *{* *.step_bound =* *{* *200, /* Macro step boundary*/* *0, /* Infinity step boundary*/* *},* *.code_per_step = 1,* *.qvalue = 128,*

Structure = actuator_tuned_params		
Structure member	**Definition**	**Example**
		`},` `},`

Damping	Defines the damping control values for each region and direction.
	Damping_delay: Delay in microseconds after register write is performed.
	
	Hw_params: Damping value to be programmed for HW damping.
	
	Damping_step: If damping step is less than MAX steps then SW treats as SW damping. So to use hw damping, damping_step is sufficiently large (in general > 1024).
		`.damping =`
		`{`
		`/* damping[MOVE_NEAR] */`
		`{`
		`/* Scenario 0 */`
		`{`
		`.ringing_params =`
		`{`
		`/* Region 0 */`
		`{`
		`.damping_step =`
		`0x3FF,`
		`.damping_delay =`
		`6000,`
		`.hw_params =`
		`0x00000000,`
		`},`
		`},`
		`},`
		`},`
		`},

NOTE: The actuator moves linearly only in a certain range (e.g., lc898122, from DAC code ~ 70 to DAC code ~ 400). Usually experiment with a laser device.

NOTE: The actuator has several configurations to control a ringing effect. Experiment with an oscilloscope to see the ringing effect and pick the proper value for the given lens specification.

4.2.2 Actuator algorithm tuning files

Each actuator has tuning files to support use cases (camera/camcorder). The tuning file contains several values for the AF algorithm's operation. As with Chromatix files in a sensor library, each actuator algorithm tuning header is built in its unique library and dynamically loaded and unloaded depending on the use case.

Create two libraries for two modes, camera and camcorder from the reference driver.

- Android.mk – Contains build rules and include paths.
- <actuator_name>_<camera/camcorder>.c – Simply returns AF algorithm library pointer.

- <actuator_name>_<camera/camcorder>.h – Contains AF algorithm tuning parameters. (Later the AF tuning team configures to proper tuning values.)

5 Flash Driver Bringup

This chapter provides the necessary information for camera Flash driver bringup on the Android platform with an MSM8994/92 device.

The Flash driver bringup includes the following tasks/steps:

1. Flash driver requirements.
2. Configuring the DTSI.
3. Software driver development and integration.

5.1 Flash driver requirements

The following documents are needed for the Flash driver bring up.

- Flash datasheet (for I2C-based flash)
  - Powerup sequence
  - Slave address
  - Read / write sequence
- Schematics (for I2C-based flash)
  - Regulators for powerup.
  - Gpio Pins.

5.2 Configuring the DTSI

The PMIC or I2C is used to drive the Flash. Target 8992/94 is supported with the PMIC flash. The dtsi entries are available as a part of the build. The DTSI entries are needed for I2C based flash with the below examples.

5.2.1 PMIC-based Flash driver

The DTS node contains the flash source and its capabilities (max current, max duration).

```
&soc {
led_flash0: qcom,camera-led-flash {
        cell-index = <0>;
        compatible = "qcom,camera-led-flash";
        qcom,flash-type = <1>;
        qcom,flash-source = <&pmi8994_flash0>
```

```
   };
};
```

5.2.2 QUP-/I2C-based LED Flash driver

The DTS node should be in this format if the flash is I2C-based and is connected to QUP.

```
&i2c {
led_flash0: qcom,led-flash@60 {
       cell-index = <0>;
       reg = <0x60>;
       qcom,slave-id = <0x60 0x00 0x0011>;
       compatible = "qcom,led-flash";
       qcom,flash-name = "adp1600";
       qcom,flash-type = <1>;
       qcom,gpio-no-mux = <0>;
       gpios = <&msmgpio 18 0>,
              <&msmgpio 19 0>;
       qcom,gpio-flash-en = <0>;
       qcom,gpio-flash-now = <1>;
       qcom,gpio-req-tbl-num = <0 1>;
       qcom,gpio-req-tbl-flags = <0 0>;
       qcom,gpio-req-tbl-label = "FLASH_EN",
              "FLASH_NOW";
       qcom,max-current = <750>;
       qcom,max-duration = <1600>;
};
};
```

5.2.3 CCI-based LED Flash driver

The DTS node should be in this format if the flash is I2C based and is connected to CCI.

```
&cci {
led_flash0: qcom,led-flash@60 {
       cell-index = <0>;
       reg = <0x60>;
       qcom,slave-id = <0x60 0x00 0x0011>;
       compatible = "qcom,led-flash";
       label = "adp1660";
       qcom,flash-type = <1>;
       qcom,cci-master = <0>;
       gpios = <&msmgpio 23 0>,
```

```
                <&msmgpio 24 0>;
        qcom,gpio-flash-en = <0>;
        qcom,gpio-flash-now = <1>;
        qcom,gpio-req-tbl-num = <0 1>;
        qcom,gpio-req-tbl-flags = <0 0>;
        qcom,gpio-req-tbl-label = "FLASH_EN",
                "FLASH_NOW";
        qcom,max-current = <750>;
        qcom,max-duration = <1600>;
};
```

NOTE: For any of the three LED Flash drivers above, the following entry must be added to associate it with the sensor:

```
qcom,camera@0 {
        qcom,led-flash-src = <&led_flash0>;
  };
```

For I2C-based flash gpois for flash-en, flash-now must be updated.

5.3 Software driver development and integration

The source code is placed in the user space. The flash driver is created and compiled to create a library that is then loaded at run time.

Access the parameters and structures for the flash driver using the source code through the flash handler:

```
static flash_lib_t flash_lib_ptr = {
{
        . flash_name =
        ...
}
```

Its type (`flash_lib_t`) is defined in:

vendor/qcom/proprietary/mm-camerasdk/sensor/includes/flash_lib.h.

Following table describes the structure in detail.

Structure	Definition	Example
flash_name	Gives the flash name.	`.flash_name = "bd7710",`
flash_type	Defines the type of flash and can be one of the following: FLASH_TYPE_LED	`.flash_type = FLASH_TYPE_LED,`

Structure	Definition	Example
flash_driver_type	Defines the type of flash driver and can be one of the following: FLASH_DRIVER_TYPE_PMIC (for pmic-based flash) FLASH_DRIVER_TYPE_I2C (for i2C-based flash) FLASH_DRIVER_TYPE_GPIO (for CPIO-based flash) FLASH_DRIVER_TYPE_DEFAULT (for default flash type present in the build)	*.flash_driver_type = FLASH_DRIVER_TYPE_DEFAULT*
power_setting_array	Defines the power up and power down settings for the flash.	`.power_setting_array = { .power_setting_a = { .seq_type = CAMERA_POW_SEQ_GPIO, .seq_val = CAMERA_GPIO_RESET, .config_val = GPIO_OUT_LOW, .delay = 1, } .size = 1, }`
i2c_flash_info	Defines slave address, address type, flash register settings for init, high, low and off modes. This is needed if the flash is I2C-based.	`.i2c_flash_info = { .slave_addr = 0x66, .i2c_addr_type = CAMERA_I2C_BYTE_ADDR, .flash_init_settings = { .reg_setting_a = { {0x00, 0x10, 0x00}, }, .size = 1, .addr_type = CAMERA_I2C_BYTE_ADDR, .data_type = CAMERA_I2C_BYTE_DATA, .delay = 0, } }`
num_of_flash	The number of available flashes.	`.num_of_flash = 1,`
max_flash_current	The max flash current supported.	`.max_flash_current = {1000, 0, 0},`
Max_torch_current	The max torch current supported.	`.max_torch_current = {200, 0, 0},`
Max_flash_duration	The max flash duration.	`.max_flash_duration = {1200, 0, 0},`

Structure	Definition	Example
main_flash_on_frame_skip	The number of frames to be skipped from main led on.	`. main_flash_on_frame_skip = 1,`
main_flash_off_frame_skip	The number of frames to be skipped from main led off.	`. main_flash_off_frame_skip = 1,`
torch_on_frame_skip	The number of frames to be skipped from estimation start.	`. torch_on_frame_skip = 1,`
torch_off_frame_skip	The number of frames to be skipped from estimation stop.	`. torch_off_frame_skip = 1,`

A References

The references listed in Table A-1 provide additional information about topics discussed in this document.

Table A-1 References

Title	DCN
Qualcomm	
Camera SDK User Guide	80-NV354-1