Arm® Mali™ - IV009 Software Technical Reference Manual

Version r0p1-00eac0 / Revision r0p1

Revision Information

Date	Issue	Confidentiality	Change
23/03/2018	EAC	Confidential	Initial Release for r0p0
28/09/2018	EAC	Confidential	Initial Release for r0p1

Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20348

Confidentiality Status

This document is Confidential. This document may only be used and distributed in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Product Status

The information in this document is Final, that is for a developed product.

Contents

List of Tables

Preface

This preface introduces the Arm® Mali™ IV009 Software Technical Reference Manual. It contains the following sections:

- About this book.
- Feedback.

About this book

This book is for the Arm Mali-IV009.

Implementation obligations

This book is designed to help you implement an Arm product. The extent to which the deliverables may be modified or disclosed is governed by the contract between ARM and Licensee. There may be validation requirements, which if applicable will be detailed in the contract between Arm and Licensee and which if present must be complied with prior to the distribution of any silicon devices incorporating the technology described in this document. Reproduction of this document is only permitted in accordance with the licences granted to Licensee.

Arm assumes no liability for your overall system design and performance, the verification procedures defined by Arm are only intended to verify the correct implementation of the technology licensed by Arm, and are not intended to test the functionality or performance of the overall system. You or the Licensee will be responsible for performing any system level tests.

You are responsible for any applications which are used in conjunction with the Arm technology described in this document, and to minimize risks adequate design and operating safeguards should be provided for by you. Arm's publication of any information in this document of information regarding any third party’s products or services is not an express or implied approval or endorsement of the use thereof.

Product revision status

The rmpn identifier indicates the revision status of the product described in this book, for example, r1p2, where:

rm Identifies the major revision of the product, for example, r1.

pn Identifies the minor revision or modification status of the product, for example, p2.

Intended audience

This book is for system designers, system integrators, and verification engineers who are designing a System-on-Chip (SoC) device that uses the Mali-IV009 Image Signal Processor.

Typographic conventions

Font/text type	What it means
italic	Introduces special terminology, denotes cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.
`monospace`	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace italic`	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
`monospace bold`	Denotes language keywords when used outside example code.
`<and>`	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>`

Terms and abbreviations

Term/Abbreviation	What it means
LUT	Look-Up Table
MCFE	Multi-Channel Front End
FPGA	Field-Programmable Gate Array
ISP	Image Signal Processor
IQ	Image Quality
SBuf	Shared Buffer
AE	Auto Exposure
AWB	Auto White Balance
AF	Auto Focus
FSM	Finite State Machine
ROI	Region of Interest

Additional reading

Information published by Arm and by third parties.

See https://developer.arm.com/ for access to Arm documentation.

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to errata@arm.com. Give:

- The title.
- The number ARM-EPM-137349.
- The page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note: Arm tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

1 Introduction

1.1 Overview

The Mali-IV009 ISP Software (“The Driver”) is a cross-platform application designed to control a sensor, a lens and the Arm ISP hardware core. The software is developed to work with Arm ISP hardware only. The current version of the software is not compatible with any other ISP provided by Arm.

The driver is designed to be compatible “as is” with the Arm Juno v.2 development board and widely used flavors of Linux and most Bare-Metal systems.

The Linux version of the driver is based on the V4L2 framework and implements standard v4l2 techniques. It includes the source code for the kernel driver and the user space application which encapsulates all 3A algorithms.

The driver for bare-metal platforms has all source code in one library. The ISP Software can be directly run as one executable or be a part of a bigger application.

1.2 Features

The ISP Software provides the following functionality:

- Single-camera support: the code is designed to be used with only one instance of sensor and lens driver.
- Multi-Calibration set: the calibration set can be changed dynamically as per the requirement.
- Multi-exposure *High Dynamic Range* (HDR) support: the software supports up to 3:1 multi-exposure.
- The following algorithms are included in the standard driver delivery:
    - AWB
    - AE
    - AF
    - Gamma Contrast
    - Noise Reduction Control
    - Sharpening control
- The purpose and additional information about each algorithm will be provided in the following sections.
- Independent driver core which enables the driver to be platform agnostic and be compiled under different target platforms by providing a proper BSP layer.
- Dedicated channel which helps to update the internal parameters of the driver in real time without re-compilation thus speeding up the tuning/debugging procedure.

- V4L2 compatible layer for the Linux version of the driver and the reference example for the integration of the driver into the external framework.

2 ISP software deliverables

The ISP Software is delivered as one `tar.gz` archive which can be extracted by the standard Linux command:
`tar -zxvf ./????package_name????_`**`version`**`.tar.gz`

where

version may have one of the following values:

alpha **–** Alpha release of the ISP Driver

beta **–** Beta release of the ISP Driver

lac **–** Limited access release of the ISP Driver

eac **–** Final release of the ISP Driver

The content of the delivery archive includes source code, makefiles and scripts which are required to build the driver.

2.1 Directory structure

The directory structure of the release is as follows:

driver/	
bare-metal/	Folder with the source code for Bare-Metal platform
linux/	Folder with the files for Linux platform
acamera_lib/	User space application with software algorithms
kernel/	Folder for Kernel space drivers
v412_dev/	Main V4L2 device driver module
subdev/	Sub device modules
iq/	Sub device module for IQ calibration
lens/	Sub device module for lens driver
sensor/	Sub device module for sensor driver

Figure 1. ISP software delivery structure

3 ISP software for Linux-based platforms

3.1 Overview

The ISP software for Linux-based platform is split into the following two components:

- **Kernel space V4L2 device driver**: This low-level device driver implements the V4L2 interface, does the buffer management, manages communication with the hardware and interacts with V4L2 sub-devices.
- **User space driver**: This component is specifically designed for 3A algorithms and runs in the Linux User Space. It communicates internally with the kernel device driver and usually does not require any changes to be made on a customer side.

The following figure shows the ISP device driver stack:

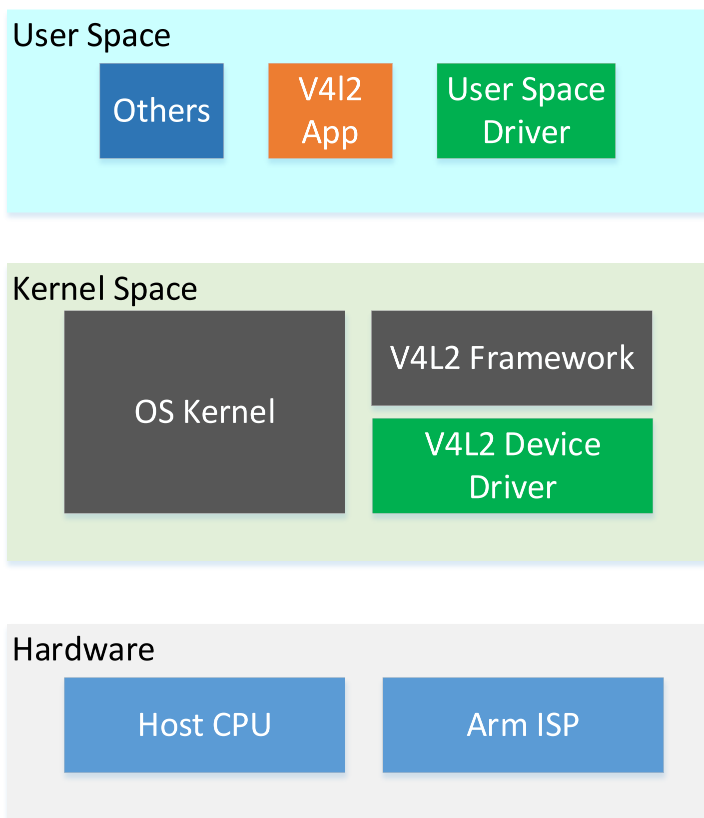


Figure 2. ISP device driver stack

The user space driver and the V4L2 device driver (shown in green in Figure 2) communicate with each other to ensure that the whole system works. The user space driver depends on kernel driver. However, the kernel space driver can work independently without the user space driver but no algorithms in kernel driver.

3.2 Architecture

The following figure shows the high-level implementation architecture on a Linux OS platform.

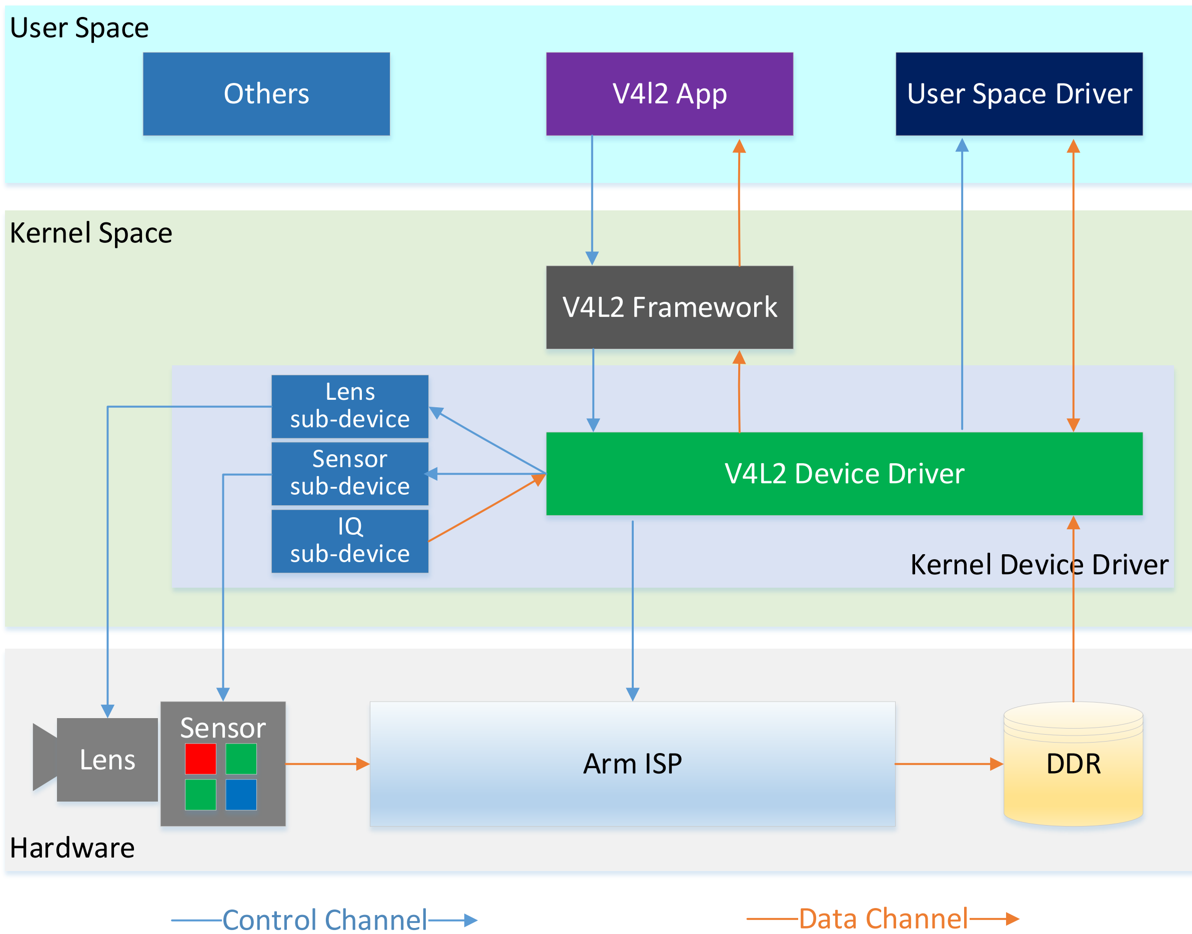


Figure 3. ISP driver architecture

In the ISP driver software, the kernel space driver has four separate parts:

- one main V4L2 device driver.
- three sub-device drivers, two of which are used to control the hardware, and one is used to provide the IQ calibration data.

The user space driver has two channels to communicate with kernel space driver. One channel is for control commands and the other channel is used to exchange statistics data and algorithm result. `V4L2_app` uses the standard V4L2 interfaces (such as /dev/video0 and ioctl command) to interact with `v4l2_framework`.

3.3 System requirements

The V4L2 Kernel space driver has the following system requirements:

Requirement	Value
Linux kernel version	4.9
DMIPS	15
Code Size	180 KB – V4L2 device only
BSS + Data	50 KB – without calibrations
Stack	8 KB
Heap	400 KB

Table 1. System requirements for Linux V4L2 ISP driver

The User space driver for ISP 3A library has the following system requirements:

Requirement	Value
DMIPS	65
Code Size	100 KB
BSS + Data	45 KB – without calibrations
Stack	8 KB
Heap	8 KB

Table 2. System requirements for Linux 3A ISP library

3.4 ISP kernel space driver

The part of the V4L2 ISP Device which is responsible for direct interaction with the ISP core, sensor and the lens is called the ISP Kernel Driver. It updates the ISP configuration space on every new frame and reads the statistic information which is sent to the algorithms running in the ISP user space driver. The kernel driver implements its own API for V4L2 or any other application and can be run independently. This is important if the V4L2 functionality is not required for some reason.

The ISP Kernel Driver interacts with the ISP User Driver through special character device created on initialization stage.

The user driver receives incoming statistic data and recalculates new parameters for the hardware. These parameters are sent back to the ISP Kernel Driver.

3.4.1 Driver components

The ISP kernel space driver is a V4L2-compliant Linux kernel driver which is delivered as the following components:

- **ISP Kernel Driver** – this module encapsulates the knowledge of the hardware and is responsible for all interactions between the software and the ISP pipeline. It handles all interrupts coming from the ISP, reads statistic information and sends it further to the ISP User Driver. This module controls the sensor and lens driver and is responsible for the buffer management on the ISP side.

- **V4L2 ISP Device** – the main purpose of this device is to provide a standard interface for external applications to communicate with and control the ISP. It does not have any knowledge of the hardware underneath and is an abstraction layer between the core logic and applications.
- **V4L2 IQ Sub-device** – the sub-device contains actual IQ calibration parameters for a given sensor/lens pair.
- **V4L2 Sensor Sub-device** – this sub-device implements a sensor driver. It is responsible for communication with the sensor hardware and guarantees the correct implementation of the `soc_sensor.h` interface routines.
- **V4L2 Lens Sub-device** – this sub-device implements a lens driver. It is responsible for communication with the lens hardware and guarantees the correct implementation of the `soc_lens.h` interface routines.

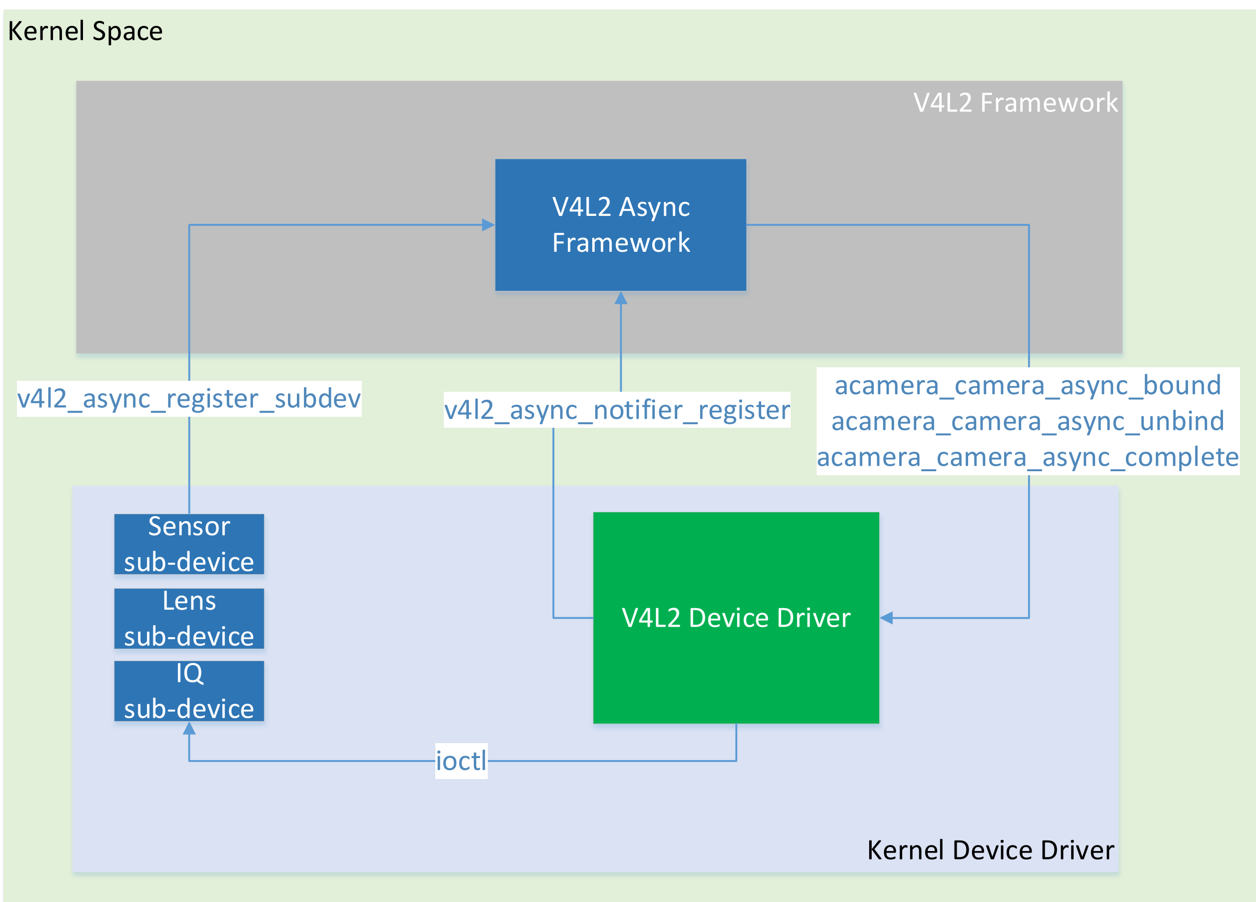


Figure 4. V4L2 ISP driver

The V4L2 main device driver uses the `v4l2_async_notifier_register` to register call-back functions into V4L2 Asynchronized Framework. After the sub-device driver registers itself via the `v4l2_async_register_subdev` API, the V4L2 main device driver gets a notification and binds the sub-device driver. The main device driver starts initialising the sensor, lens and ISP when it binds with all sub-devices it expected, it uses ioctl interface to communicate with sub-devices at runtime.

The number of expected sub-devices is predefined by the `V4L2_SOC_SUBDEV_NUMBER` parameter. The value of the `V4L2_SOC_SUBDEV_NUMBER` is 3 in the reference implementation. It can be changed to 2 if the sensor has a fixed lens, and no lens sub-device is needed.

3.4.2 V4L2 sub-devices

This section provides information about the V4L2 sub-devices, their available interfaces and their implementation.

3.4.2.1 V4L2 IQ Sub-device

This sub-device encapsulates all the calibration parameters which are required by the main V4L2 device. They include settings for the 3A library, tables to initialize ISP arrays such as mesh shading, down scaler coefficients and some settings which are used to dynamically modulate ISP registers based on the surrounding environmental conditions.

The calibration parameter set includes two main files:

- **Static calibrations**. This file is automatically generated by the ISP Calibration Tool and is usually not changed for a given sensor/lens pair.
- **Dynamic Calibrations**: This file is usually updated during the tuning session and includes more subjective parameters.

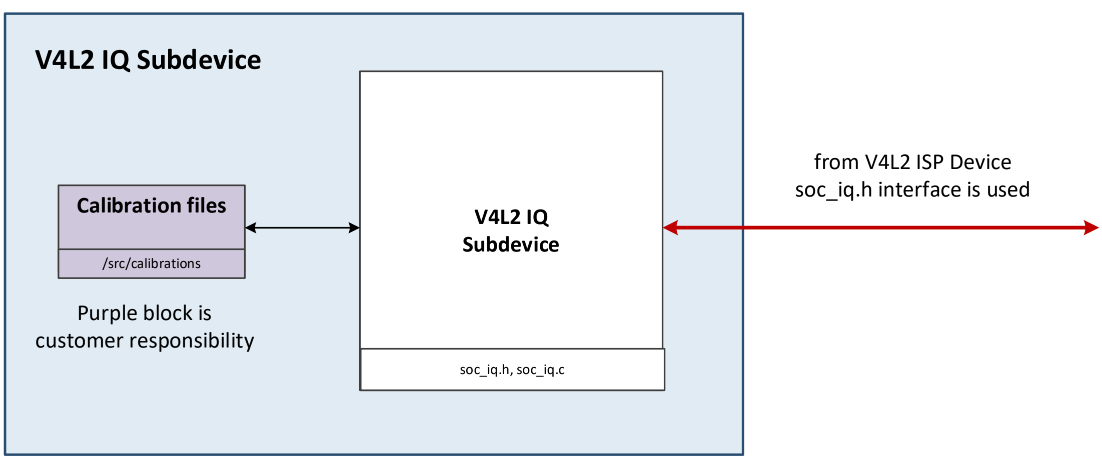


Figure 5. V4L2 IQ sub-device

For more information about the calibration parameters, refer to the Arm ISP Calibration Guide.

The following files are provided as a reference implementation of the V4L2 sub-device for calibration tables:

File	Description
`soc_iq.h`	The main interface header which is used for communication between the V4L2 ISP device and the IQ sub-device.
`soc_iq.c`	The main file with the implementation of the IQ sub-device.
`acamera_calibrations_dynamic_linear_dummy.c`	Includes dynamic calibrations for the linear mode.
`acamera_calibrations_static_linear_dummy.c`	Includes static calibrations for linear mode.
`acamera_calibrations_dynamic_fs_lin_dummy.c`	Includes dynamic calibrations for DOL mode.
`acamera_calibrations_static_fs_lin_dummy.c`	Includes static calibrations for DOL mode.
`acamera_get_calibrations_dummy.c`	The file includes the entry point function `get_calibrations` which must fill the given `AcameraCalibration` pointer with correct LUT's addresses.
`acamera_command_api.h`	The file has the full list of supported tables with their ID. You must ensure that the same version of this file is used by both, the V4L2 ISP device and the V4L2 IQ device.

Note: ***IQ calibration files are provided as reference only. If another sensor/lens pair is used all values should be recalibrated according to the Arm ISP Calibration Guide.***

3.4.2.2 Interface

The calibration subdev interface is provided in the `soc_iq.h` file and used by both, the V4L2 ISP device and the IQ V4L2 sub-device.

Note: ***Both v4l2 objects must use the same sub-device name, which is defined by macro V4L2_SOC_IQ_NAME as "SocCalibrations".***

All LUT IDs are taken from the `acamera_command_api.h` file which must be identical across all V4L2 sub-devices.

```
// This name is used by both V4L2 ISP device and
// V4L2 IQ sub-device to match the sub-device in the
// V4L2 async sub-device list.
#define V4L2_SOC_IQ_NAME "SocCalibrations"

// This is used as the main communication structure between
// V4L2 ISP Device and V4L2 IQ Sub-device
struct soc_iq_ioctl_args {
     union {
        // This struct is used to request information
```

```
        // about a LUT.
        // On this request the LookupTable lut
        // will be filled with data excluding the ptr.
        // ptr must be assigned to NULL.
        struct {
            uint32_t context; // must be 0
            uint32_t preset;  // calibration preset number.
            uint32_t id;      // LUT ID value from acamera_command_api.h
            LookupTable lut;  // lut will be filled on return but ptr must be NULL
        } request_info;
        // This struct is used to request the actual LUT dat
        // The memory must be preallocated in advance and provided
        // by *ptr pointer.
        // The IQ V4L2 sub-device will copy LUT to the given memory as output
        struct {
            uint32_t context;      // must be always 0
            uint32_t preset;       // calibration preset number
            uint32_t id;              // LUT ID value from acamera_command_api.h
            void* ptr;                 // preallocated memory for the requested LUT data
            uint32_t data_size;  // data size in bytes for ptr buffer
            uint32_t kernel;       // must be always 1
        }request_data;
     } ioctl;
};

// The enum defines possible commands ID for ioctl request from
// the V4L2 ISP device.
enum SocIQ_ioctl {


    // request the information about the LUT
    // it includes data type, number of elements
    // The given input structure will have type request_info.
    // Commonly used to calculate the size of the LUT before
    // allocating memory for the actual data.
    V4L2_SOC_IQ_IOCTL_REQUEST_INFO = 0,


    // request the whole LUT including the data.
    // The given input structure will have type request_data
    // Used to request the LUT data.
    V4L2_SOC_IQ_IOCTL_REQUEST_DATA,
};
```

This interface is used by the V4L2 ISP Device as follows:

1. The V4L2 ISP device collects information about all supported LUTs by calling the `ioctl` request with `V4L2_SOC_IQ_IOCTL_REQUEST_INFO` parameter. The IQ sub-device must return the correct description of the LUT for a given preset by filling the LookupTable structure.

2. The V4L2 ISP device requests the data for each supported LUT by calling the `ioctl` request with `V4L2_SOC_IQ_IOCTL_REQUEST_DATA` parameter. The IQ sub-device must check the given parameters inside the `request_data` structure and copy the whole array into a location provided by ***ptr** pointer memory.

> ***Note:*** ***The IQ sub-device can support as many presets as required but the number and meaning of each preset should be known by V4L2 ISP Device in advance. This means that the V4L2 ISP Device should be able to distinguish presets and use them properly.***
>
> ***For example, one preset can be used for Linear Sensor Mode of operation and another one for DOL mode of operation.***

3.4.2.3 Implementation

The IQ sub-device is implemented inside one `soc_iq.c` file as a standard Linux kernel module.

The sub-device is defined as:

```
static struct v4l2_subdev soc_iq;
```

After the IQ sub-device module is inserted to the Linux kernel the `soc_iq_probe` function is called.

Inside the function the sub-device is registered as an asynchronous V4L2 sub-device:

```
// initialize sub-device with give ops
v4l2_subdev_init(&soc_iq, &iq_ops);

// support direct access through /dev/
soc_iq.flags |= V4L2_SUBDEV_FL_HAS_DEVNODE;
soc_iq.dev = &pdev->dev;

// register async subdev in the v4l2 framework
rc = v4l2_async_register_subdev(&soc_iq);
```

The iq_ops structure has a pointer to the `iq_ioctl(struct v4l2_subdev *sd, unsigned int cmd, void* arg)` function.

This function is used as the main communication channel between the V4L2 ISP Device and the V4L2 IQ sub-device and must support commands from `SocIQ_ioctl enum` which is declared in the `soc_iq.h` file. Every time the V4L2 ISP Device needs to get information about the LUT it calls the `iq_ioctl` function.

```
static long iq_ioctl(struct v4l2_subdev *sd, unsigned int cmd, void* arg)
{
    long rc = 0;
    switch (cmd) {
        case V4L2_SOC_IQ_IOCTL_REQUEST_INFO: {
            int32_t context = ARGS_TO_PTR(arg)->ioctl.request_info.context;
            int32_t preset = ARGS_TO_PTR(arg)->ioctl.request_info.preset;
            int32_t id = ARGS_TO_PTR(arg)->ioctl.request_info.id;
            //ACameraCalibrations * luts_ptr = &g_luts_arr[context][preset];
            if ( context < FIRMWARE_CONTEXT_NUMBER
                    //&& preset < MAX_CALIBRATION_PRESETS
                    && id < CALIBRATION_TOTAL_SIZE) {
                CALIBRATION_FUNC_ARR[context](context,&g_luts_arr[context]);
```

```
                ACameraCalibrations * luts_ptr = &g_luts_arr[context];
                ARGS_TO_PTR(arg)->ioctl.request_info.lut.ptr = NULL;
                *((uint16_t*)&ARGS_TO_PTR(arg)->ioctl.request_info.lut.rows) = luts_ptr-
>calibrations[id]->rows;
                *((uint16_t*)&ARGS_TO_PTR(arg)->ioctl.request_info.lut.cols) = luts_ptr-
>calibrations[id]->cols;
                *((uint16_t*)&ARGS_TO_PTR(arg)->ioctl.request_info.lut.width) = luts_ptr-
>calibrations[id]->width;

                rc = 0;
            } else {
                rc = -1;
            }
        }
        break;
        case V4L2_SOC_IQ_IOCTL_REQUEST_DATA: {
            int32_t context = ARGS_TO_PTR(arg)->ioctl.request_data.context;
            int32_t preset = ARGS_TO_PTR(arg)->ioctl.request_data.preset;
            int32_t id = ARGS_TO_PTR(arg)->ioctl.request_data.id;
            int32_t data_size = ARGS_TO_PTR(arg)->ioctl.request_data.data_size;
            //ACameraCalibrations * luts_ptr = &g_luts_arr[context][preset];
            void* ptr = ARGS_TO_PTR(arg)->ioctl.request_data.ptr;
            if ( context < FIRMWARE_CONTEXT_NUMBER
                    //&& preset < MAX_CALIBRATION_PRESETS
                    && id < CALIBRATION_TOTAL_SIZE) {
                CALIBRATION_FUNC_ARR[context](context,&g_luts_arr[context]);
                ACameraCalibrations * luts_ptr = &g_luts_arr[context];
                if ( ptr != NULL ) {
                    if ( data_size == __GET_LUT_SIZE( luts_ptr->calibrations[id] ) ) {
                        if ( ARGS_TO_PTR(arg)->ioctl.request_data.kernel == 0 ) {
                            copy_to_user( ptr, luts_ptr->calibrations[id]->ptr,
__GET_LUT_SIZE( luts_ptr->calibrations[id] ) ) ;
                        } else {
                            memcpy( ptr, luts_ptr->calibrations[id]->ptr, __GET_LUT_SIZE(
luts_ptr->calibrations[id] ) ) ;
                        }
                        rc = 0;
                    }else {
                        rc = -1 ;
                    }
                } else {
                    LOG(LOG_ERR, "User pointer is null, lut id %d, lut preset %d", id,
preset ) ;
                    rc = -1 ;
                }
            } else {
                rc = -1;
            }
        }
        break;
        default:
            LOG(LOG_WARNING, "Unknown soc iq ioctl cmd %d", cmd);
            rc = -1;
            break;
    };

    return rc;
}
```

For more details, refer to the `soc_iq.c` file.

3.4.2.4 V4L2 Lens Sub-device

This sub-device is a wrapper around the actual lens driver. The implementation must support all standard API functions which are used to control the lens. The full list of supported commands can be found in the `soc_lens.h` file.

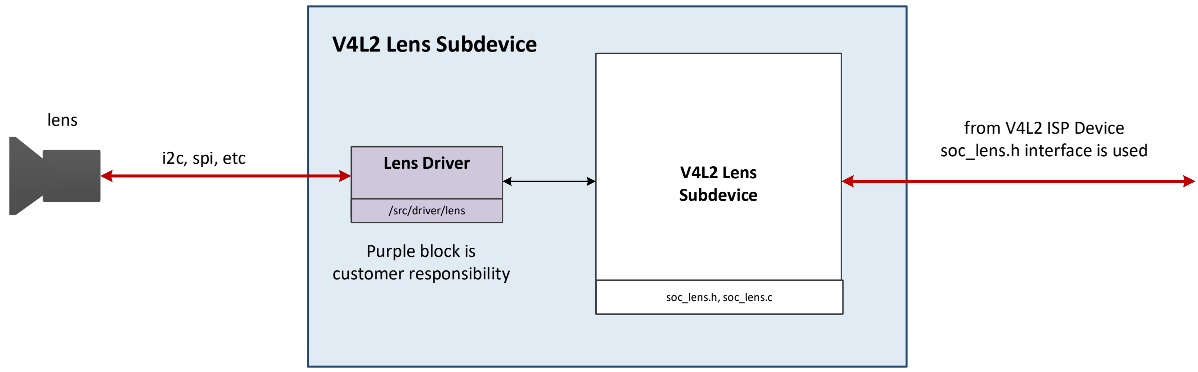


Figure 6. V4L2 Lens sub-device

The following files are provided as a reference implementation of the V4L2 lens sub-device:

File	Description
`soc_lens.h`	The main interface header file which is used for communication between the V4L2 ISP device and lens sub-device. It includes the description of all API commands which should be supported by a lens driver.
`soc_lens.c`	The main file with the implementation of the V4L2 lens sub-device.
`null_vcm.h`	The header file of the dummy lens driver which can be used for any other lens driver.
`Null_vcm.c`	The dummy implementation of the lens driver which can be used as the reference code for any other lens driver.

3.4.2.4.1 Interface

The lens subdev interface is provided in the `soc_lens.h` file and used by both the V4L2 ISP device and the V4L2 lens sub-device. It is required that both V4L2 objects use the same sub-device name `V4L2_SOC_LENS_NAME` to identify the correct sub-device in the V4L2 framework.

The V4L2 ISP Device expects the lens sub-device to support the following specific commands.

```
// This name is used by both V4L2 ISP device and
// V4L2 Lens sub-device to match the sub-device in the
// V4L2 async sub-device list.
#define V4L2_SOC_LENS_NAME "SocLens"
```

```
// This is used as the main communication structure between
// V4L2 ISP Device and V4L2 Lens Sub-device
// Parameters are used differently depending on the actual API command ID.
struct soc_lens_ioctl_args {
     uint32_t ctx_num;
     union {
        struct {
            uint32_t val_in;    // first input value for the API function (optional)
            uint32_t val_in2;   // second input value for the API function (optional)
            uint32_t val_out;   // output value returned by API function (optional)
        }general;
     } args;
};

// The enum declares the API commands ID which
// must be supported by V4L2 lens sub-device.
// This API ID will be used on each ioctl call from
// the V4L2 ISP Device.
enum SocLens_ioctl {
    // move the lens to the given position
    // input: val_in - target position for the lens
    // output: none
    SOC_LENS_MOVE = 0,

    // stop lens movement
    // input: none
    // output: none
    SOC_LENS_STOP,

    // get the current lens positon
    // input: none
    // output: val_out
    SOC_LENS_GET_POS,

    // return the lens status
    // input: none
    // output: val_out - 1 moving or 0 not moving
    SOC_LENS_IS_MOVING,


    // NOT SUPPORTED BY V4L2 ISP DEVICE
    // move zoom lens to the target position (if supported)
    // input: val_in - target position for the zoom lens
    // output: none
    SOC_LENS_MOVE_ZOOM,


    // NOT SUPPORTED BY V4L2 ISP DEVICE
    // return the zoom lens status
    // input: none
    // output: val_out - 1 zooming is in progress, 0 no active zooming
    SOC_LENS_IS_ZOOMING,

    // read the lens register
    // input: val_in - register address
    // output: val_out - register value
    SOC_LENS_READ_REG,
```

```
    // write the lens register
    // input: val_in - register address
    // input: val_in2 - register value
    // output: none
    SOC_LENS_WRITE_REG,

    // return lens type - static value
    // input: none
    // output: val_out - lens type
    SOC_LENS_GET_LENS_TYPE,

    // return minimum step size - static value
    // input: none
    // output: val_out - minimal step size
    SOC_LENS_GET_MIN_STEP,

    // NOT SUPPORTED BY V4L2 ISP DEVICE
    // next zoom lens position
    // input: none
    // output: val_out - next zoom lens position
    SOC_LENS_GET_NEXT_ZOOM,


    // NOT SUPPORTED BY V4L2 ISP DEVICE
    // current zoom value
    // input: none
    // output: val_out - current zoom value
    SOC_LENS_GET_CURR_ZOOM,

    // next lens position
    // input: none
    // output: val_out - next lens position
    SOC_LENS_GET_NEXT_POS
};
```

This interface is used by the V4L2 ISP Device as follows:

- The V4L2 ISP Device works on the assumption that the lens driver is initialized at the time the V4L2 lens sub-device is registered in the V4L2 framework. This implies that all initialization steps must be done inside the probe function.
- The V4L2 ISP device will call the `soc_lens_ioctl` function with the Lens API command ID from the `SocLens_ioctl` list and corresponding input parameters in a random manner. The V4L2 Lens sub-device should not make any assumption on the calling order.

Note: Zoom lens is not supported by this version of the V4L2 ISP driver so all related API commands will not be called and shouldn't be implemented by the driver. Such commands are marked as "// NOT SUPPORTED BY V4L2 ISP DEVICE".

3.4.2.4.2 Implementation

The reference lens sub-device module is in the file `soc_lens.c` and implements a standard Linux kernel module.

The sub-device is defined as:

`static struct v4l2_subdev soc_lens;`

After the lens sub-device module is inserted to the Linux kernel the `soc_lens_probe` function is called.

Inside the function the sub-device is registered as an asynchronous V4L2 sub-device:

```
// initialize sub-device with give ops
v4l2_subdev_init(&soc_lens, &iq_ops);

// support direct access through /dev/
soc_lens.flags |= V4L2_SUBDEV_FL_HAS_DEVNODE;
soc_lens.dev = &pdev->dev;

// register async subdev in the v4l2 framework
rc = v4l2_async_register_subdev(&soc_lens);
```

The `lens_ops` structure has a pointer to the `soc_lens_ioctl (struct v4l2_subdev *sd, unsigned int cmd, void* arg)` function.

This function is used as the main communication channel between the V4L2 ISP Device and the V4L2 Lens sub-device and must support commands from `SocLens_ioctl enum` which is declared in the `soc_lens.h` file. Every time the V4L2 ISP Device needs to call the lens driver API function it calls the `soc_lens_ioctl` routine with a specific command ID.

The reference implementation of the `soc_lens_ioctl` function is as follows.

```
static long soc_lens_ioctl(struct v4l2_subdev *sd, unsigned int cmd, void* arg)
{
    long rc = 0;

    if(ARGS_TO_PTR(arg)->ctx_num > FIRMWARE_CONTEXT_NUMBER){
        LOG(LOG_ERR, "Failed to process lens_ioctl for ctx:%d\n",ARGS_TO_PTR(arg)-
>ctx_num);
        return -1;
    }

    subdev_lens_ctx * ctx = &l_ctx[ARGS_TO_PTR(arg)->ctx_num];

    if (ctx->lens_context == NULL) {
        LOG(LOG_ERR, "Failed to process lens_ioctl. Lens is not initialized yet.
lens_init must be called before");
        rc = -1;
        return rc;
    }

    const lens_param_t* params = ctx->lens_control.get_parameters(ctx->lens_context);


    switch (cmd) {
        case SOC_LENS_MOVE:
            ctx->lens_control.move(ctx->lens_context, ARGS_TO_PTR(arg)-
>args.general.val_in);
            break;
```

```
        case SOC_LENS_STOP:
            ctx->lens_control.stop(ctx->lens_context);
            break;
        case SOC_LENS_GET_POS:
            ARGS_TO_PTR(arg)->args.general.val_out = ctx->lens_control.get_pos(ctx-
>lens_context);
            break;
        case SOC_LENS_IS_MOVING:
            ARGS_TO_PTR(arg)->args.general.val_out = ctx->lens_control.is_moving(ctx-
>lens_context);
            break;
        case SOC_LENS_MOVE_ZOOM:
            ctx->lens_control.move_zoom(ctx->lens_context, ARGS_TO_PTR(arg)-
>args.general.val_in);
            break;
        case SOC_LENS_IS_ZOOMING:
            ARGS_TO_PTR(arg)->args.general.val_out = ctx->lens_control.is_zooming(ctx-
>lens_context);
            break;
        case SOC_LENS_READ_REG:
            ARGS_TO_PTR(arg)->args.general.val_out = ctx-
>lens_control.read_lens_register(ctx->lens_context, ARGS_TO_PTR(arg)-
>args.general.val_in);
            break;
        case SOC_LENS_WRITE_REG:
            ctx->lens_control.write_lens_register(ctx->lens_context, ARGS_TO_PTR(arg)-
>args.general.val_in, ARGS_TO_PTR(arg)->args.general.val_in2);
            break;
        case SOC_LENS_GET_LENS_TYPE:
            ARGS_TO_PTR(arg)->args.general.val_out = params->lens_type;
            break;
        case SOC_LENS_GET_MIN_STEP:
            ARGS_TO_PTR(arg)->args.general.val_out = params->min_step;
            break;
        case SOC_LENS_GET_NEXT_ZOOM:
            ARGS_TO_PTR(arg)->args.general.val_out = params->next_zoom;
            break;
        case SOC_LENS_GET_CURR_ZOOM:
            ARGS_TO_PTR(arg)->args.general.val_out = params->curr_zoom;
            break;
        case SOC_LENS_GET_NEXT_POS:
            ARGS_TO_PTR(arg)->args.general.val_out = params->next_pos;
            break;
        default:
            LOG(LOG_WARNING, "Unknown lens ioctl cmd %d", cmd);
            rc = -1;
            break;
    };

    return rc;
}
```

For more information, refer to the `soc_lens.c` file.

Note: ***Zoom lens is not supported by this version of the V4L2 ISP driver so all related API commands will not be called and could be skipped in the the driver. Such commands are marked as "// NOT SUPPORTED BY V4L2 ISP DEVICE" in the soc_lens.h file.***

3.4.2.5 V4L2 Sensor Sub-device

This sub-device is a wrapper around the actual sensor driver. The implementation must support all standard API functions which are used to control the sensor. The full list of supported commands can be found in the `soc_sensor.h` file.

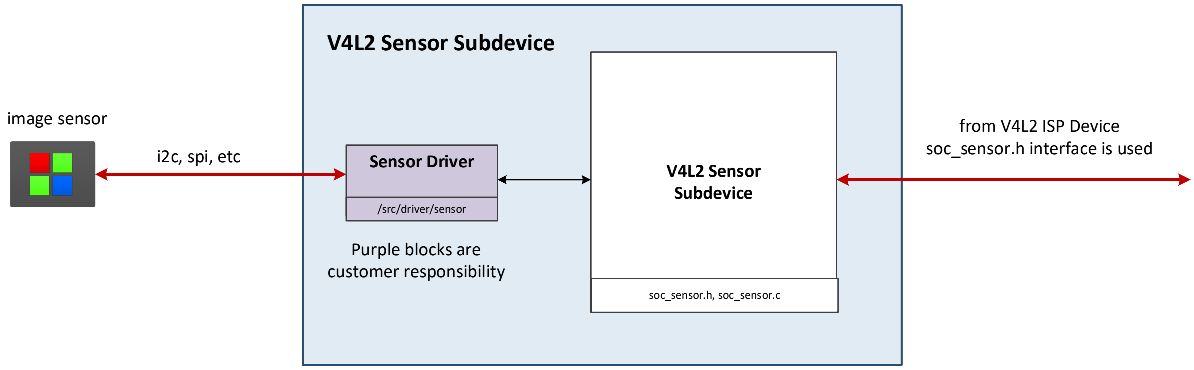


Figure 7. V4L2 sensor sub-device

The following files are provided as a reference implementation of V4L2 sensor sub-device:

File	Description
`soc_sensor.h`	The main interface header which is used for communication between the V4L2 ISP device and the sensor sub-device. It includes the description of all API commands which should be supported by the sensor driver.
`soc_sensor.c`	The main file with the implementation of the V4L2 sensor sub-device.
`dummy_seq.h`	The header of the dummy sensor driver which can be used as a reference for any other sensor driver.
`dummy_drv.c`	The dummy implementation of the sensor driver which can be used as the reference code for any other sensor driver.

3.4.2.5.1 Interface

The sensor subdev interface is provided in the `soc_sensor.h` file and used by both, the V4L2 ISP device and the V4L2 sensor sub-device. It is required that both V4L2 objects are using the same sub-device name which is `V4L2_SOC_SENSOR_NAME` to identify the correct sub-device in the V4L2 framework.

The V4L2 ISP Device expects the sensor sub-device to support some specific commands which are as follows.

```
#ifndef __SOC_SENSOR_H__
#define __SOC_SENSOR_H__

// This name is used by both V4L2 ISP device and
// V4L2 sensor sub-device to match the sub-device in the
// V4L2 async sub-device list.
#define V4L2_SOC_SENSOR_NAME "SocSensor"
```

```
// This is used as the main communication structure between
// V4L2 ISP Device and V4L2 Sensor Sub-device
// Parameters are used differently depending on the actual API command ID.
struct SOC_SENSOR_ioctl_args {
         uint32_t ctx_num;
     union {
        // This struct is used to set the new integration time parameters
        // for the sensor.
        // The Interface supports up to 4 different exposures for HDR scenes.
        // This structure is used only for SOC_SENSOR_ALLOC_IT API call.
        struct {
            uint16_t it_short;  // short integration time
            uint16_t it_long;   // long integration time
            uint16_t it_medium; // medium integration time
        }integration_time;
        // This struct is used for all API commands except SOC_SENSOR_ALLOC_IT
        // It contains some general parameters the meaning of which is different
        // and depends on the specific API ID.
        struct {
            uint32_t val_in;    // first input value
            uint32_t val_in2;   // second input value
            uint32_t val_out;   // output value
        }general;
     } args;
};

// The enum declares the API commands ID which
// must be supported by V4L2 sensor sub-device.
// This API ID will be used on each ioctl call from
// V4L2 ISP device
enum SocCamera_ioctl {
    //########## CONTROLS ###########//

    // Enable sensor data streaming
    // input: none
    // output: none
    SOC_SENSOR_STREAMING_ON = 0,

    // disable sensor data streaming
    // input: none
    // output: none
    SOC_SENSOR_STREAMING_OFF,

    // set a new sensor preset
    // Each sensor driver can support any number of different presets. A preset is a combination
    // of a resolution, fps and wdr mode. V4L2 ISP driver should know in advance about the presets
    // used and will use them by number from 0 till PR_NUM-1.
    //
    // input: val_in: change sensor preset
    // output: none
    SOC_SENSOR_SET_PRESET,

    // allocate new analog gain.
    // The V4L2 ISP device will try to set a new analog gain on every frame
    // V4L2 sensor sub-device should save the closest possible value which is less \
    // or equal then requested one.
    // The saved value must be applied on the time when SOC_SENSOR_UPDATE_EXP is called.
    // The meaning of this command is to get the real possible analog gain
    // based on sensor driver limitations.
    // The returned value will be used to adjust other gains if requested value cannot be
    // used exactly.
    //
    // input: val_in: requested analog gain
    // output: val_out: actual analog gain value which can be used by the sensor.
    SOC_SENSOR_ALLOC_AGAIN,

    // allocate new digital gain.
```

```
    // The V4L2 ISP device will try to set a new digital gain on every frame
    // V4L2 sensor sub-device should save the closest possible value which is less
    // or equal then requested one.
    // This saved value must be applied on the time when SOC_SENSOR_UPDATE_EXP is called.
    // The meaning of this command is to get the real possible dital gain
    // based on sensor driver limitations.
    // The returned value will be used to adjust other gains if requested value cannot be
    // used exactly.
    //
    // input: val_in: requested digital gain
    // output: val_out: actual digital gain value which can be used by the sensor.
    SOC_SENSOR_ALLOC_DGAIN,

    // allocate new integration time
    // The V4L2 ISP device will try to set a new integration time on every frame
    // That is the only command which uses integration_time structure as input
    // parameter for ioctl call.
    // Integration time is a combination of 1, 2 or 3 different integration times.
    // The number depends on the current sensor preset. For example if the sensor
    // is working in linear mode only it_short is used. If sensor is initialized
    // in DOL 3Exp mode all it_short, it_medium and it_long will be send.
    //
    // input: it_short: integration time for short exposure ( should be used for linear mode)
    // input: it_medium: integration time for medium exposure
    // input: it_long: integration time for long exposure
    // output: none
    SOC_SENSOR_ALLOC_IT,

    // The function is called every frame.
    // All previously set parameters for Analog/Digital gain and Integration time
    // must be send to the sensor on this call at the same time.
    // input: none
    // output: none
    SOC_SENSOR_UPDATE_EXP,

    // read sensor register value
    // input: val_in - register address
    // output: val_out - register value
    SOC_SENSOR_READ_REG,

    // write a value to the sensor register
    // input: val_in - register address
    // input: val_in2 - register value
    // output: none
    SOC_SENSOR_WRITE_REG,

    //########## STATIC PARAMETERS ###########//

    // Return the number of supported presets.
    // This call is used by V4L2 ISP device to understand how many
    // and what kind of presets are supported by the sensor driver
    // input: none
    // output: val_out - number of supported presets. Minimum 1
    SOC_SENSOR_GET_PRESET_NUM,

    // Get a sensor image widht for a given preset
    // input: val_in - preset number
    // output: val_out - image width for a given preset
    SOC_SENSOR_GET_PRESET_WIDTH,

    // Get a sensor image height for a given preset
    // input: val_in - preset number
    // output: val_out - image height for a given preset
    SOC_SENSOR_GET_PRESET_HEIGHT,

    // Get a sensor fps for a given preset
    // input: val_in - preset number
```

```
    // output: val_out - fps for a given preset
    SOC_SENSOR_GET_PRESET_FPS,

    // Get a sensor mode for a given preset
    // input: val_in - preset number
    // output: val_out - WDR_MODE_LINEAR or WDR_MODE_FS_LIN (DOL)
    SOC_SENSOR_GET_PRESET_MODE,

    //########## DYNAMIC PARAMETERS ###########//

    // return current number of different exposures
    // This command should return actual number of different
    // exposures from the sensor
    // input: none
    // output: val_out - number of exposures. Min 1, Max 3.
    SOC_SENSOR_GET_EXP_NUMBER,

    // return maximum integration time in lines for the current mode
    // input: none
    // output: val_out - maximum integration time in lines
    SOC_SENSOR_GET_INTEGRATION_TIME_MAX,

    // return maximum integration time for the long exposure
    // input: none
    // output: val_out - maximum long integration time
    SOC_SENSOR_GET_INTEGRATION_TIME_LONG_MAX,

    // return current minimum integration time in lines
    // input: none
    // output: val_out - min it in lines
    SOC_SENSOR_GET_INTEGRATION_TIME_MIN,

    // return current maximum integration time limit
    // input: none
    // output: val_out - maximum limit for it.
    SOC_SENSOR_GET_INTEGRATION_TIME_LIMIT,

    // return current maximum possible analog gain value
    // The returned value must be in log2 format with LOG2_GAIN_SHIFT bits precesion.
    // input: none
    // output: val_out - maximum analog gain
    SOC_SENSOR_GET_ANALOG_GAIN_MAX,

    // return current maximum possible digital gain value
    // The returned value must be in log2 format with LOG2_GAIN_SHIFT bits precesion.
    // input: none
    // output: val_out - maximum digital gain
    SOC_SENSOR_GET_DIGITAL_GAIN_MAX,

    // return integration time latency
    // It means number of frames which is required for the sensor
    // between SOC_SENSOR_UPDATE_EXP is called and the actual value applied on the sensor side.
    // input: none
    // output: val_out - integration time latency
    SOC_SENSOR_GET_UPDATE_LATENCY,

    // return lines per second for the current mode
    // input: none
    // output: val_out - number of lines per second
    SOC_SENSOR_GET_LINES_PER_SECOND,

    // return current fps
    // input: none
    // output: val_out - current fps with 8 bits precesion
    SOC_SENSOR_GET_FPS,

    // return active image height
```

```
    // input: none
    // output: val_out - active height
    SOC_SENSOR_GET_ACTIVE_HEIGHT,

    // return active image width
    // input: none
    // output: val_out - active width
    SOC_SENSOR_GET_ACTIVE_WIDTH
};

#endif //__SOC_SENSOR_H__
```

This interface is used by V4L2 ISP Device in the following manner:

- The V4L2 ISP Device works with the assumption that the sensor driver is initialized at the time the V4L2 sensor sub-device is registered in the V4L2 framework. This implies that all initialization steps must be done inside the probe function.
- The V4L2 ISP device will call the `soc_sensor_ioctl` function with the Sensor API command ID from the `SocSensor_ioctl` list and corresponding input parameters.
- The V4L2 ISP device will always call `SOC_SENSOR_ALLOC_AGAIN`, `SOC_SENSOR_ALLOC_DGAIN`, `SOC_SENSOR_ALLOC_IT` followed by `SOC_SENSOR_UPDATE_EXP`. The V4L2 sensor sub-device must save the requested Again, Dgain and Integration Time parameters and apply them at once at the time the `SOC_SENSOR_UPDATE_EXP` is called.
- The V4L2 ISP device may change a supported preset. This means that the main device might change the resolution, fps or HDR mode based on the its own logic.

Note: The V4L2 sensor sub-device and V4L2 ISP device must be compiled with the same headers which are stored under the /inc folder.

3.4.2.5.2 Implementation

The reference sensor sub-device module is in the file `soc_sensor.c` and implements a standard Linux kernel module.

The sub-device is defined as:

```
static struct v4l2_subdev soc_sensor;
```

After the sensor sub-device module is inserted to the Linux kernel the `soc_sensor_probe` function will be called.

Inside the function the sub-device is registered as an asynchronous V4L2 sub-device:

```
v4l2_subdev_init(&soc_camera, &camera_ops);

  soc_camera.flags |= V4L2_SUBDEV_FL_HAS_DEVNODE;

  soc_camera.dev = &pdev->dev;
  rc = v4l2_async_register_subdev(&soc_camera);
```

The `sensor_ops` structure has a pointer to the `soc_sensor_ioctl(struct v4l2_subdev *sd, unsigned int cmd, void* arg)` function.

This function is used as the main communication channel between the V4L2 ISP Device and the V4L2 sensor sub-device and must support commands from the `SocSensor_ioctl` enum which is declared in the `soc_sensor.h` file. Every time the V4L2 ISP Device needs to call the sensor driver API function it calls the `soc_sensor_ioctl` routine with specific command ID.

The reference implementation of the `soc_sensor_ioctl` function is as follows.

```
 static long camera_ioctl(struct v4l2_subdev *sd, unsigned int cmd, void* arg)
{
    long rc = 0;

    if(ARGS_TO_PTR(arg)->ctx_num > FIRMWARE_CONTEXT_NUMBER){
        LOG(LOG_ERR, "Failed to process camera_ioctl for ctx:%d\n",ARGS_TO_PTR(arg)->ctx_num);
        return -1;
    }

    subdev_camera_ctx * ctx = &s_ctx[ARGS_TO_PTR(arg)->ctx_num];

    if (ctx->camera_context == NULL) {
        LOG(LOG_ERR, "Failed to process camera_ioctl. Sensor is not initialized yet. camera_init must
be called before");
        rc = -1;
        return rc;
    }

    const sensor_param_t* params = ctx->camera_control.get_parameters(ctx->camera_context);

    switch (cmd) {
        case SOC_SENSOR_STREAMING_ON:
                ctx->camera_control.start_streaming(ctx->camera_context);
            break;
        case SOC_SENSOR_STREAMING_OFF:
                ctx->camera_control.stop_streaming(ctx->camera_context);
            break;
        case SOC_SENSOR_SET_PRESET:
                ctx->camera_control.set_mode(ctx->camera_context, ARGS_TO_PTR(arg)-
>args.general.val_in ) ;
            break;
        case SOC_SENSOR_ALLOC_AGAIN:
            ARGS_TO_PTR(arg)->args.general.val_out = ctx->camera_control.alloc_analog_gain(ctx-
>camera_context, ARGS_TO_PTR(arg)->args.general.val_in ) ;
            break;
        case SOC_SENSOR_ALLOC_DGAIN:
            ARGS_TO_PTR(arg)->args.general.val_out = ctx->camera_control.alloc_digital_gain(ctx-
>camera_context, ARGS_TO_PTR(arg)->args.general.val_in ) ;
            break;
        case SOC_SENSOR_ALLOC_IT:
                ctx->camera_control.alloc_integration_time(ctx->camera_context, &ARGS_TO_PTR(arg)-
>args.integration_time.it_short, &ARGS_TO_PTR(arg)->args.integration_time.it_medium,
&ARGS_TO_PTR(arg)->args.integration_time.it_long) ;
            break;
        case SOC_SENSOR_UPDATE_EXP:
                ctx->camera_control.sensor_update(ctx->camera_context);
            break;
        case SOC_SENSOR_READ_REG:
            ARGS_TO_PTR(arg)->args.general.val_out = ctx->camera_control.read_sensor_register(ctx-
>camera_context, ARGS_TO_PTR(arg)->args.general.val_in ) ;
            break;
        case SOC_SENSOR_WRITE_REG:
```

```
                ctx->camera_control.write_sensor_register(ctx->camera_context, ARGS_TO_PTR(arg)-
>args.general.val_in, ARGS_TO_PTR(arg)->args.general.val_in2 ) ;
            break;
        case SOC_SENSOR_GET_PRESET_NUM:
            ARGS_TO_PTR(arg)->args.general.val_out = params->modes_num;
            break;
        case SOC_SENSOR_GET_PRESET_WIDTH:{
            int preset = ARGS_TO_PTR(arg)->args.general.val_in;
            if (preset < params->modes_num) {
                ARGS_TO_PTR(arg)->args.general.val_out = params-
>modes_table[preset].resolution.width;
            } else {
                LOG(LOG_ERR, "Preset number is invalid. Available %d presets, requested %d", params-
>modes_num, preset);
                rc = -1;
            }
            }
            break;
        case SOC_SENSOR_GET_PRESET_HEIGHT: {
            int preset = ARGS_TO_PTR(arg)->args.general.val_in;
            if (preset < params->modes_num) {
                ARGS_TO_PTR(arg)->args.general.val_out = params-
>modes_table[preset].resolution.height;
            } else {
                LOG(LOG_ERR, "Preset number is invalid. Available %d presets, requested %d", params-
>modes_num, preset);
                rc = -1;
            }
            }
            break;
        case SOC_SENSOR_GET_PRESET_FPS: {
            int preset = ARGS_TO_PTR(arg)->args.general.val_in;
            if (preset < params->modes_num) {
                ARGS_TO_PTR(arg)->args.general.val_out = params->modes_table[preset].fps;
            } else {
                LOG(LOG_ERR, "Preset number is invalid. Available %d presets, requested %d", params-
>modes_num, preset);
                rc = -1;
            }
            }
            break;
        case SOC_SENSOR_GET_PRESET_MODE:{
            int preset = ARGS_TO_PTR(arg)->args.general.val_in;
            if (preset < params->modes_num) {
                ARGS_TO_PTR(arg)->args.general.val_out = 0;
            } else {
                LOG(LOG_ERR, "Preset number is invalid. Available %d presets, requested %d", params-
>modes_num, preset);
                rc = -1;
            }
            }
            break;
         case SOC_SENSOR_GET_EXP_NUMBER:
            ARGS_TO_PTR(arg)->args.general.val_out = params->sensor_exp_number;
            break;
        case SOC_SENSOR_GET_INTEGRATION_TIME_MAX:
            ARGS_TO_PTR(arg)->args.general.val_out = params->integration_time_max;
            break;
        case SOC_SENSOR_GET_INTEGRATION_TIME_MIN:
            ARGS_TO_PTR(arg)->args.general.val_out = params->integration_time_min;
            break;
        case SOC_SENSOR_GET_INTEGRATION_TIME_LONG_MAX:
            ARGS_TO_PTR(arg)->args.general.val_out = params->integration_time_long_max;
            break;
        case SOC_SENSOR_GET_ANALOG_GAIN_MAX:
            ARGS_TO_PTR(arg)->args.general.val_out = params->again_log2_max;
            break;
```

```
        case SOC_SENSOR_GET_DIGITAL_GAIN_MAX:
            ARGS_TO_PTR(arg)->args.general.val_out = params->dgain_log2_max;
            break;
        case SOC_SENSOR_GET_UPDATE_LATENCY:
            ARGS_TO_PTR(arg)->args.general.val_out = params->integration_time_apply_delay;
            break;
        case SOC_SENSOR_GET_LINES_PER_SECOND:
            ARGS_TO_PTR(arg)->args.general.val_out = params->lines_per_second;
            break;
        case SOC_SENSOR_GET_FPS: {
            int mode = ARGS_TO_PTR(arg)->args.general.val_in;
            ARGS_TO_PTR(arg)->args.general.val_out = params->modes_table[mode].fps;
            }
            break;
        case SOC_SENSOR_GET_ACTIVE_HEIGHT: {
            ARGS_TO_PTR(arg)->args.general.val_out = params->active.height;
            }
            break;
        case SOC_SENSOR_GET_ACTIVE_WIDTH: {
            ARGS_TO_PTR(arg)->args.general.val_out = params->active.width;
            }
            break;
        default:
            LOG(LOG_WARNING, "Unknown soc sensor ioctl cmd %d", cmd);
            rc = -1;
            break;
    };

    return rc;
}
```

For more information, refer to the `soc_sensor.c` file.

3.4.3 V4L2 device driver

The V4L2 ISP Device is the main device which controls the Arm ISP hardware, provides statistics data for user driver 3A algorithms and communicates with the given sensor/lens devices through the V4L2 sub-devices.

The V4L2 ISP Device includes V4L2 Interface files plus the ISP Kernel Driver code. The ISP Kernel Driver can work independently through its own API. The V4L2 related files only use the API functions from the `/inc/api` folder to communicate with the kernel driver.

The V4L2 ISP Device provides pointers to the V4L2 sub-devices for the core driver to communicate with the sensor, lens and calibration sub-devices directly.

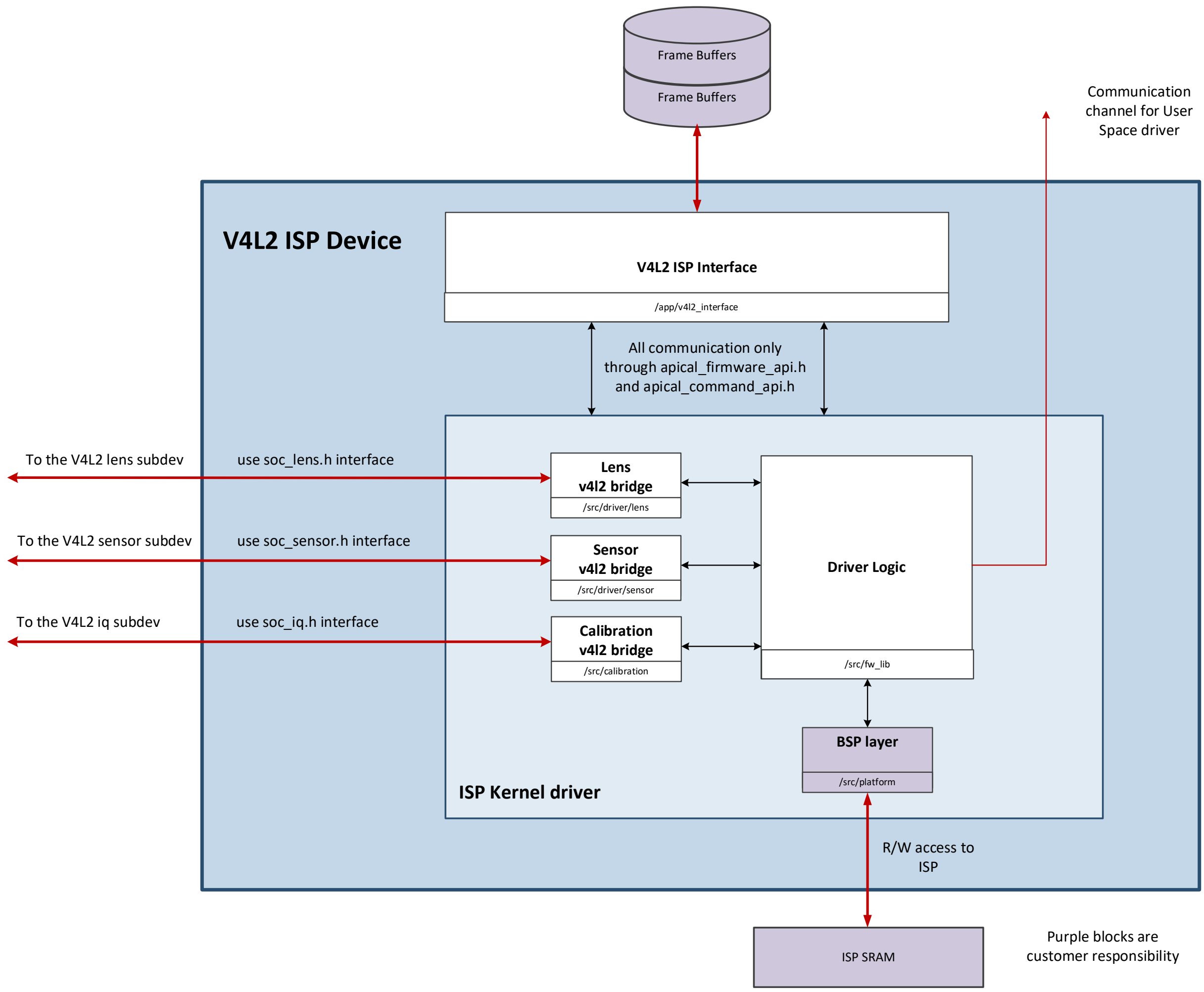


Figure 8. V4L2 ISP Device Diagram

3.4.3.1 V4L2 layer

All V4L2 ISP device files are located under `/delivery/app/v4l2_interface` folder.

File	Description
`isp-v4l2.c`	The logic to register V4L2 device and handle V4L2 interfaces.
`fw-interface.c`	The Wrapper layer between the V4L2 interface and the ISP driver, contains a kernel thread to handle the sensor streaming.
`main_kernel_juno_vl42.c`	The main file to initialize the V4L2 device and handle top level routines.
`main_firmware.c`	This file contains firmware initialize, de-initialize and interrupt handling.

These files include `acamera_firmware_api.h` and `acamera_command_api.h` to get access to the core driver functionality.

3.4.3.2 Driver file structure

The ISP Kernel Driver files are located under the `/src` and `/inc` folders. The `/inc` folder contains all headers required for interaction between the driver and external application. You must not call any kernel driver functions directly if they are not declared in the API headers from the `/inc` folder.

File/Folder	Description
`inc/api/acamera_firmware_api.h`	This is the main header file with the driver API routines. In most cases it is sufficient to use only this file if no extra functionality is required on the application side.
`inc/api/acamera_command_api.h`	This file has the driver command API to control its behavior. There are special functions which allow to change the state of algorithms, update IQ tables, and so on.
`inc/api/acamera_lens_api.h`	The driver expects that any connected lens driver would implement a predefined number of routines which are called directly by the driver.
`inc/api/acamera_sensor_api.h`	The driver expects that any connected sensor driver would implement a predefined number of routines which are called directly by the driver.
`inc/api/acamera_types.h`	This file defines standard types which are used inside the whole driver source code.
`inc/api/acamera_sbus_api.h`	This file defines a standard API to access i2c, spi bus by the sensor or lens driver. The reference sensor/lens driver implementation uses routines from the header to get access to the hardware registers.
`inc/api/acamera_firmware_settings.h`	The `acamera_init` function from the `acamera_firmware_api.h` file expects to receive the initialization structure with the configuration parameters such as sensor and lens references, output pipe addresses and so on.

File/Folder	Description
`/inc/isp/*.h`	Files in this folder provide a simple read/write routine for every single ISP register available for a customer. For “read only” registers only the read function is implemented. Note: All register access routines rely on the `system_hw_io.c` and `system_sw_io.c` BSP files.
`inc/sys/*.h`	Files in this folder define the BSP layer.
`src/calibration/*.c`	The ISP Kernel Driver uses calibration tables for internal algorithms and control parameters. All files related to the calibration are in this folder. For V4L2 these files do not have any actual data because all values of the LUT are requested from the V4L2 IQ sub-device.
`src/driver/sensor`	Files for reference sensor driver implementation. For V4L2 these files do not have any actual sensor control logic because it is implemented in the V4L2 Sensor sub-device. In the V4L2 use-case the files are working like a bridge between the ISP Kernel Driver and the hardware Sensor.
`src/driver/lens`	This folder contains files for the reference lens driver implementation. For V4L2 these files do not have any actual lens control logic because it is implemented in the V4L2 Lens sub-device. In the V4L2 use-case the files work like a bridge between the ISP Kernel Driver and the hardware Lens.
`src/fw_lib`	This folder includes all the ISP Driver logic. It implements the main API routines defined in the `/inc/api/` directory, interacts with the ISP hardware and communicates with the ISP User Driver for 3A parameters.
`src/platform`	This folder implements the BSP layer for a given platform. The BSP layer may not change much for different Linux platforms but it is the customer’s responsibility to ensure that all the files are working properly. Especially, it is important to update the `system_hw_io.c` file as it interacts with the ISP configuration space directly.

3.4.3.3 Driver API

The driver API defined in the `acamera_firmware_api.h` and includes the following functions:

Function	Description
`acamera_init(acamera_settings* settings, uint32_t ctx_num);`	This function is used to initialize the firmware. `ctx_num` must be 1 since only one context is supported presently.
`int32_t acamera_process();`	The driver must be given a CPU processing time to fulfil all tasks it has at any given moment. This function must be called to process all contexts as frequently as possible to avoid delays.
`int32_t acamera_interrupt_handler();`	This function must be called from the external application interrupt handler when any of ISP interrupts happen. It will process IRQ properly to guarantee the correct control behavior.

Note: The multi-context is not supported in the current release of the ISP Software, so ctx_num must always be 1.

3.4.3.3.1 acamera_init

```
/**
 *   Initialize one instance of firmware context
 *
 *   The firmware can control several contexts at the same time. Each context must be initialized
with its own
 *   set of setting and independently from all other contexts. A pointer will be returned on valid
context on
 *   successful initialization.
 *
 *   @param settings - a structure with setting for context to be initialized with
 *   @param ctx_num - number of contexts to be initialized
 *
 *   @return 0 - success
 *          -1 - fail.
 */
int32_t acamera_init( acamera_settings* settings, uint32_t ctx_num ) ;
```

The function must be called before any other API routines are called by the application. The return code must be checked and if any error returned the driver should not be used any time after.

The function `acamera_init` is responsible for the ISP driver initialization and initial state set up. It uses parameters from `acamera_settings` structure to establish connection to the sensor and lens drivers, get information about buffers available for ISP processing.

Note: MMU is not supported so all input and output buffers must be contiguous and be enough to store an output frame.

The list of input parameters and their description is as follows:

File	Type	Description
`sensor_init`	Mandatory	Sensor initialization entry.
`sensor_deinit`	Mandatory	Used to close the sensor instance.
`get_calibrations`	Mandatory	Used to get access to the calibration entry function.
`isp_base`	Reserved	Reserved
`hw_isp_addr`	Reserved	Reserved
`lens_init`	Optional	Used to initialize lens if exists.
`lens_deinit`	Optional	Used to de-initialize lens if exists.
`callback_metadata`	Optional	Metadata information for the frame.
`callback_dma_alloc_coherent`	Mandatory	Allocate DMA-able contiguous and cache-coherent buffers.
`callback_dma_free_coherent`	Mandatory	Free allocated DMA buffers.
`callback_stream_get_frame`	Mandatory	Get frame buffer from application, such as v4l2 layer.
`callback_stream_put_frame`	Mandatory	Return frame buffer to application, such as v4l2 layer.

Table 3. acamera_settings structure

3.4.3.3.2 acamera_interrupt_handler

The ISP Driver logic is driven by interrupts. Every time the ISP generates an event the `acamera_interrupt_handler` function must be called with minimum latency time.

Note: It is essential not to miss and not to delay the function call to guarantee the correct driver behavior.

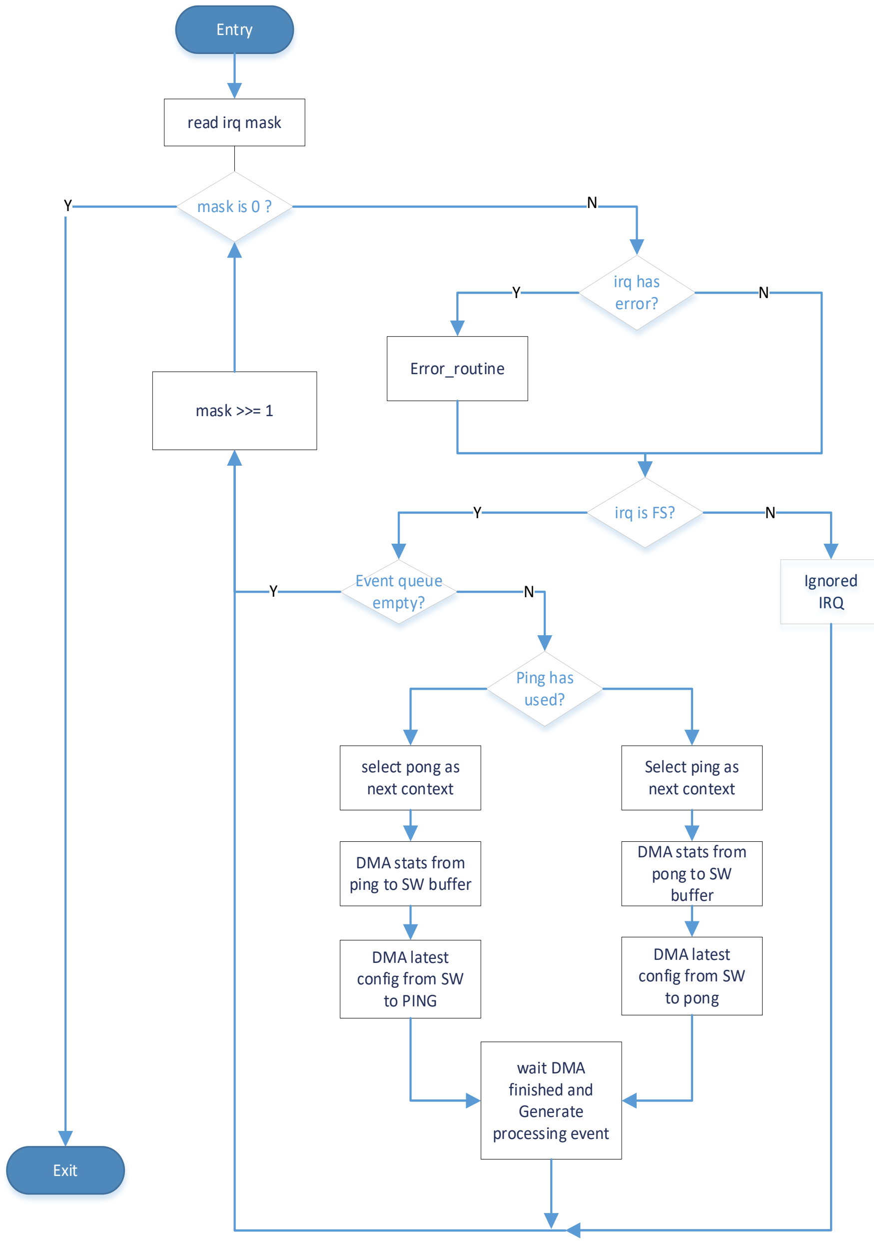


Figure 9. acamera_interrupt_handler

The Frame Start is the main event handled by the ISP Driver:

- **Frame Start**: mainly used for sensor/lens parameters update and DMA new parameters to hardware ping/pong buffers. When an interrupt occurs, the ISP driver issues the commands to update the image sensor. This is the synchronization point between ISP and external sensor/lens hardware.

This interrupt is also used to read statistic from ISP configuration space by DMA transfers. On the time DMA transfers finished the special software events are created to start 3A processing. When the `ISP_FRAME_START` interrupt request is detected inside the `acamera_interrupt_handler`, the software swaps ping/pong buffers. For example, if the current frame has been processed by the ping configuration space the software will set up the pong config for the next frame. Since the configuration space for the next frame is fixed the software initiates 2 DMA transactions from the last config to update statistics.

3.4.3.3.3 acamera_process

The function processes all software events one by one until the even queue is empty. It must be called by the application in a separate thread because it can sleep to save CPU performance. An event has a unique ID and one or more handler. The full list of supported events is defined in the `/src/acamera_fsm_mgr.c` file in the `event_name` array.

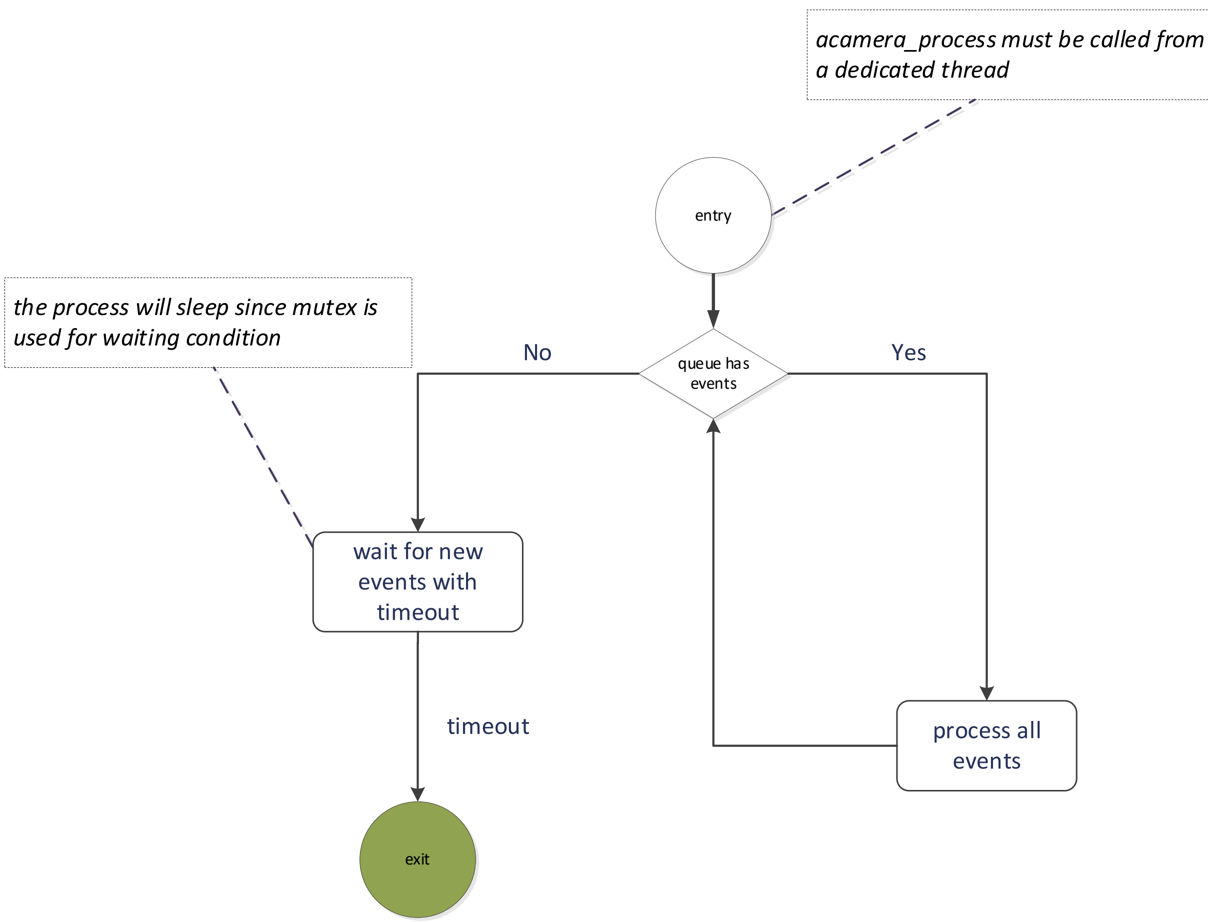


Figure 10. acamera_process

***Note:** There are two sources for new events. They are created inside the acamera_interrupt_handler or alternatively some of them can create new events.*

3.4.4 Command API

All command API routines are defined in the `acamera_command_api.h` file. They are used to control the ISP driver, change algorithm parameters, update calibration tables, and so on.

3.4.4.1 acamera_command

This function should be called by the application at any time after `acamera_init` has successfully finished.

```
uint8_t acamera_command(uint8_t command_type,
                        uint8_t command,
                        uint32_t value,
                        uint8_t direction,
                        uint32_t *ret_value);
```

Parameter	Description
`command_type`	A value from the COMMAND TYPE LIST. All control commands are divided in several groups. The full list of groups can be found in the file `acamera_command_api.h`.
`command`	Specific command ID from the given group. The full list of commands can be found in the file `acamera_command_api.h`.
`direction`	Can be one of `COMMAND_SET` to write a value or `COMMAND_GET` to request a value.
`value`	Input Parameter value. Used only when direction is `COMMAND_SET`.
`ret_value`	Output Parameter value. Used only when direction is `COMMAND_GET`.

Table 4. acamera_command routine

3.4.4.2 acamera_api_calibration

The function is mainly used to adjust some calibration parameters during the IQ tuning session for a new sensor/lens pair.

The function should be called by the application at any time after `acamera_init` has successfully finished. It is used to update calibration LUTs in real-time.

```
uint8_t acamera_api_calibration( uint8_t type,
                                 uint8_t id,
                                 uint8_t direction,
                                 void* data,
```

```
                                uint32_t data_size,
                                uint32_t* ret_value);
```

The following table provides the full list of input parameters with the description:

Parameter	Description
`type`	Reserved. Must be 0
`id`	Calibration LUT ID. The full list of supported tables is in the `acamera_command_api.h` file under `STATIC_CALIBRATIONS` and `DYNAMIC_CALIBRATIONS` sections.
`direction`	Can be one of `COMMAND_SET` to write a value or `COMMAND_GET` to request a value.
`data`	Pointer to the new LUT for the driver if direction is `COMMAND_SET` or pointer to the memory for the requested LUT if direction is `COMMAND_GET`.
`dats_size`	Size of the allocated memory in bytes.
`ret_value`	Internal error code or 0 if success.

Table 5. acamera_calibration routine

3.4.5 ISP register access

This section provides information about the register memory layout, routines to access registers, and register DMA transactions.

3.4.5.1 Register memory layout

There are two types of registers:

- Hardware registers which are located inside the ISP.
- Software registers, which reside in the host DDR memory and are managed by the driver.

The memory layout for ISP hardware registers and software registers is shown in the following figure.

Figure 11. Memory lay out of Registers

3.4.5.2 Routines to access registers

The ISP Driver includes a read/write routine for every ISP register. There are two types of routines.

- Routines providing direct access to the hardware ISP configuration space by using the `system_hw_read` and `system_hw_write` implementation from the `system_hw_io` file.

- Functions providing access to the software representation of the ISP config (SW registers). Due to such functions the HW ISP state will not change immediately, but waits for the next DMA transfer to happen inside the driver.

This split of routines is required because internally the ISP includes one common block of registers and two identical contexts which are called ping and pong. Every frame the software switches the context from ping to pong and from pong to ping accordingly.

The common block is accessed directly from the ISP driver but the ping/pong contexts are updated via DMA transfers from one buffer allocated inside the driver.

For example:

- Direct access to the software registers. Please note that both functions rely on `system_sw_read_32` and `system_sw_write_32` to get access to the corresponding register.

```
static __inline void acamera_isp_top_active_width_write(uintptr_t base, uint16_t data) {
    uint32_t curr = system_sw_read_32(base + 0x0);
    system_sw_write_32(base + 0x0, (((uint32_t) (data & 0xffff)) << 0) | (curr & 0xffff0000));
}
static __inline uint16_t acamera_isp_top_active_width_read(uintptr_t base) {
    return (uint16_t)((system_sw_read_32(base + 0x0) & 0xffff) >> 0);
}
```

- Direct access to the hardware registers. Please note that both functions rely on `system_hw_read_32` and `system_hw_write_32` to get access to the corresponding register.

```
static __inline void acamera_isp_input_port_preset_write(uintptr_t base, uint8_t data) {
    uint32_t curr = system_hw_read_32(0x6cL);
    system_hw_write_32(0x6cL, (((uint32_t) (data & 0xf)) << 0) | (curr & 0xfffffff0));
}
static __inline uint8_t acamera_isp_input_port_preset_read(uintptr_t base) {
    return (uint8_t)((system_hw_read_32(0x6cL) & 0xf) >> 0);
}
```

File	Access type	Description
`acamera_aexp_hist_stats_mem_config.h`	software	Routines to access the auto exposure statistic memory.
`acamera_ca_correction_filter_mem_config.h`	software	Routines to access chromatic aberration filer coefficients.
`acamera_ca_correction_mesh_mem_config.h`	software	Routines to access chromatic aberration mesh table.

File	Access type	Description
acamera_cmd_queues_config.h	hardware	Routines to access command queue memory. This is the internal memory used for communication between the ISP driver and ARM Control Tool only.
acamera_dpc_mem_config.h	software	Routines to access the DPC memory.
acamera_ds1_lut_arbiter_rgb1_mem_config.h	software	Routines to access RGB Gamma for the DS output pipe.
acamera_fr_lut_arbiter_rgb1_mem_config.h	software	Routines to access RGB Gamma for the FR output pipe.
acamera_ihist_stats_mem_config.h	software	Routines to access iridix history statistic memory.
acamera_isp1_config.h	software	Routines to access the ISP software context inside the driver. This context is transferred via DMA to ping and pong hardware memory spaces.
acamera_isp_config.h	hardware	Routines to access ISP registers which are common for ping and pong contexts.
acamera_lut3d_mem_config.h	software	Routines to access lut3d filter coefficients.
acamera_lut_arbiter_iridix_fp1_mem_config.h	software	Routines to access iridix fp1 memory.
acamera_lut_arbiter_iridix_rp_mem_config.h	software	Routines to access iridix rp memory.
acamera_lut_arbiter_shading_mem_config.h	software	Routines to access arbiter shading memory.
acamera_lut_arb_decompander0_mem_config.h	software	Routines to access decompander0 memory.
acamera_lut_arb_decompander1_mem_config.h	software	Routines to access decompander1 memory.
acamera_mesh_shading_mem_config.h	software	Routines to access shading memory.
acamera_metering_stats_mem_config.h	software	Routines to access statistic memory.

Table 6. Register access routines

3.4.5.3 Register DMA transaction

DMA transactions happen in the driver for every frame.

There are two types of DMA:

- From HW registers to SW registers to dump the statistics data.
- From SW registers to HW registers to apply new ISP parameters.

Figure 12 and Figure 13 show both directions for ping and pong buffers.

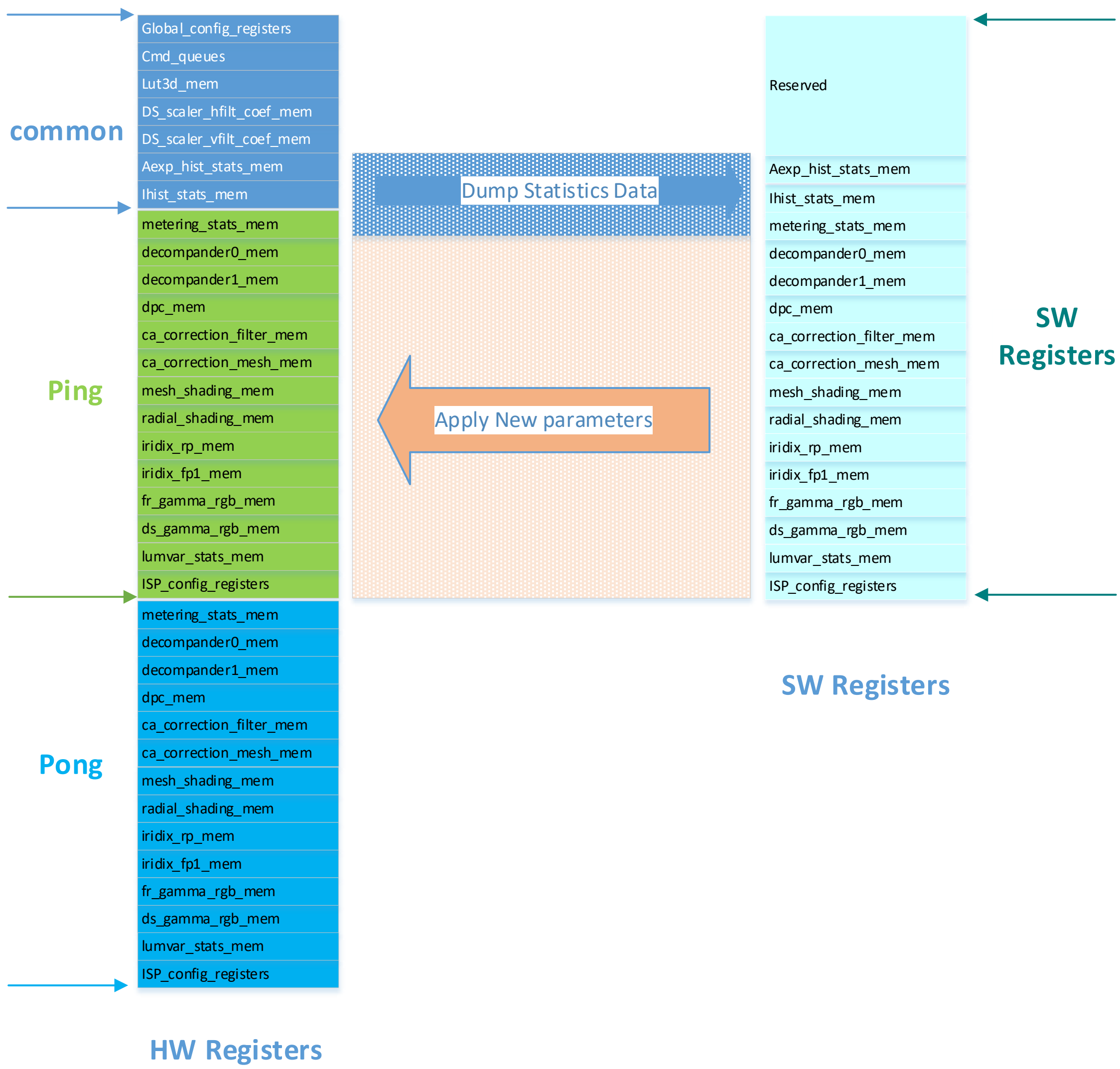


Figure 12. DMA from/to ping buffer

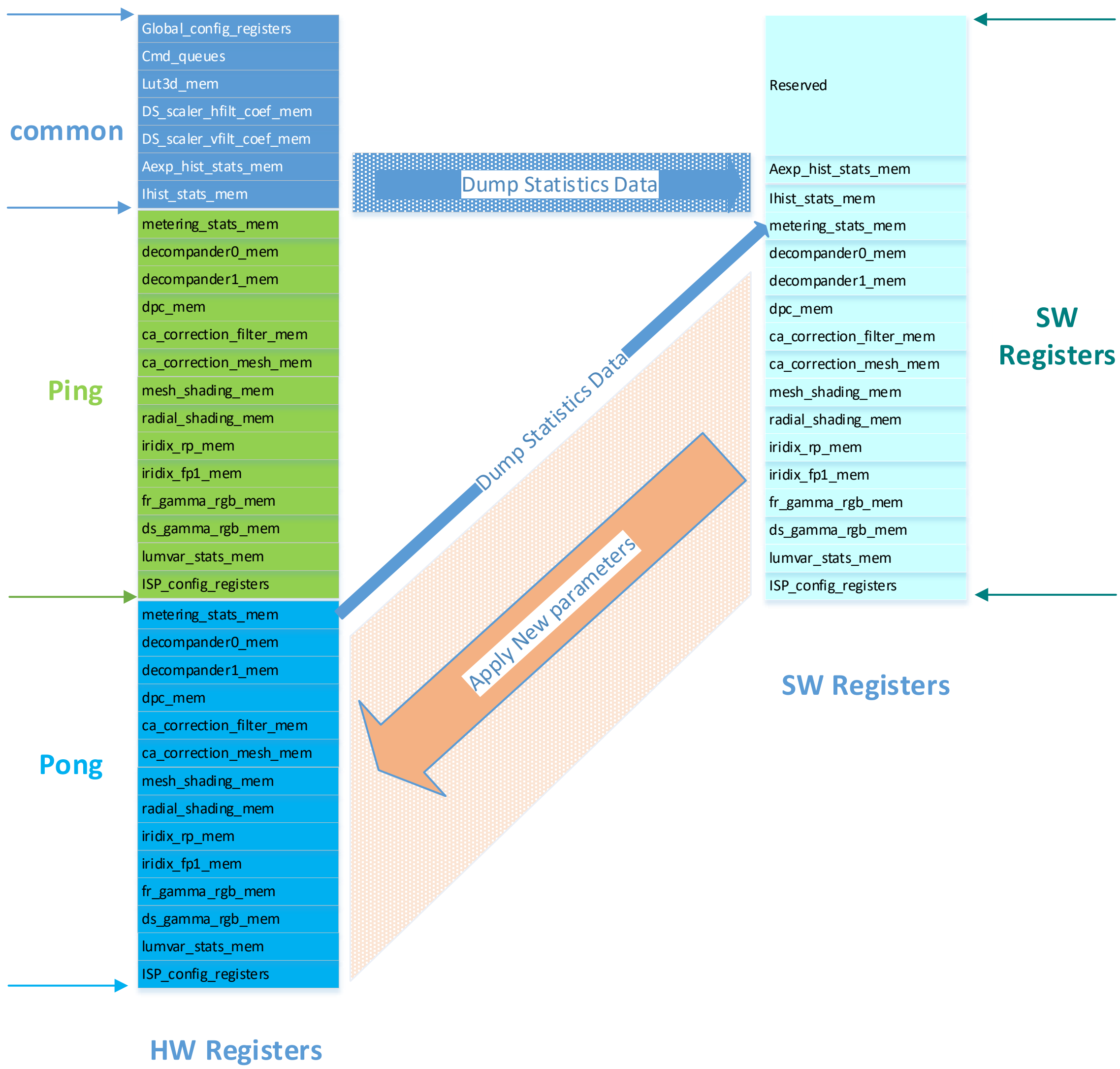


Figure 13. DMA from/to pong buffer

Note: ***As the address of statistics data from the pong buffer is not contiguous, scatter and gather DMA is required for pong buffer statistics data. Customers must implement it carefully in the target platform.***

3.4.6 ISP initialization sequence

During the initialization process the ISP configuration space is updated with a sequence using the `acamera_init` function.

> ***Note:*** ***The sequence changes the default state of the ISP configuration space and may change the original hardware register values.***

That is done in the function `acamera_init_context` (this is part of the `acamera_init` function) which is in the `src/fw_lib/acamera_fw.c` file.

```
acamera_load_isp_sequence(0, p_ctx->isp_sequence, SENSOR_ISP_SEQUENCE_DEFAULT_SETTINGS);

#if defined(SENSOR_ISP_SEQUENCE_DEFAULT_SETTINGS_CONTEXT)
acamera_load_sw_sequence( p_ctx->settings.isp_base, p_ctx->isp_sequence,
SENSOR_ISP_SEQUENCE_DEFAULT_SETTINGS_CONTEXT ) ;
#endif
```

All ISP sequences are defined in the `/src/fw_lib/isp_config_seq.h` file

Each sequence is represented by an array of the `acam_reg_t` elements which is defined as:

```
typedef struct acam_reg_t {

    uint32_t address;

    uint32_t value;

    uint32_t mask;

    uint32_t len;

};
```

For example, the sequence to initialize the ISP when the sensor mode is linear:

```
static acam_reg_t linear[] = {

    { 0x18f98, 0x20000L, 0x70007,4 },

    { 0x18eac, 0x30L, 0x30,1 },

    { 0x18e8c, 0x0L, 0x3000000,4 },

    //stop sequence - address is 0x0000

    { 0x0000, 0x0000, 0x0000, 0x0000 }

};
```

These configuration sequences are used inside the ISP Kernel Driver at the following main points:

- /src/general_fun.c: general_set_wdr_mode function
  This is called by the driver to reconfigure the ISP according to the current sensor mode every time the sensor changes its mode. For example, when sensor changes HDR mode to Linear some registers are required to be updated in the ISP pipeline to support it. This function can be called as many times as the sensor changes its mode.

- /src/acamera_fw.c: acamera_init_context.
  This is called only once during the driver initialization stage and its main purpose is to reconfigure the ISP according to the image quality requirements.

3.4.7 Customize ISP default values

The reference implementation provides some default values for ISP register configuration. Customers can override the default values or add new values as per their requirements by adding appropriate statements into the calibration item in the dynamic calibration file `subdev/iq/src/calibration/acamera_calibrations_dynamic_linear_dummy.c` as follows:

```
static uint32_t _calibration_custom_settings_context[][4] = {

//addr, value, mask, length

//stop sequence - address is 0x0000

{0x0000, 0x0000, 0x0000, 0x0000}

};
```

An example to enable the test_pattern by default is shown in the following code snippet:

```
static acam_reg_t custom_settings_context[] = {

//{address, value, mask, length}

{ 0x18ed8, 0x1L, 0x1,1 },   // ← Enable test pattern.

{ 0x18edc, 0x3L, 0xff,1 },  // ← Set pattern type to 0x3.

//stop sequence - address is 0x0000

{ 0x0000, 0x0000, 0x0000, 0x0000 }

};
```

Customers must find out the registers address and mask from the corresponding ISP configuration header files. The addresses and masks of the example can be found from the header file `v4l2_dev/inc/isp/acamera_isp1_config.h` as follows:

```
// ------------------------------------------------------------------------------ //

// Test pattern off-on: 0=off, 1=on

// ------------------------------------------------------------------------------ //

#define ACAMERA_ISP_VIDEO_TEST_GEN_CH0_TEST_PATTERN_OFF_ON_MASK (0x1)

static __inline uint8_t acamera_isp_video_test_gen_ch0_test_pattern_off_on_read(uintptr_t
base) {

return (uint8_t)((system_sw_read_32(base + 0x18ed8L) & 0x1) >> 0);

}
```

```
#define ACAMERA_ISP_VIDEO_TEST_GEN_CH0_PATTERN_TYPE_MASK (0xff)

static __inline uint8_t acamera_isp_video_test_gen_ch0_pattern_type_read(uintptr_t base)
{

return (uint8_t)((system_sw_read_32(base + 0x18edcL) & 0xff) >> 0);

}
```

This calibration item is applied in the function `acamera_init_context()` in the `v4l2_dev\src\fw_lib\acamera_fw.c` file, configure parameter FW_HAS_CUSTOM_SETTINGS can be used to enable/disable this code snippet as per customer's requirements.

```
int32_t acamera_init_context()

{

#if FW_HAS_CUSTOM_SETTINGS

        // the custom initialization may be required for a context

        const uint32_t *p_custom_settings_context = _GET_UINT_PTR (p_ctx,
CALIBRATION_CUSTOM_SETTINGS_CONTEXT);

        acamera_load_sw_sequence(p_ctx->settings.isp_base, (const acam_reg_t **)
&p_custom_settings_context, 0);

#endif

}
```

3.4.8 Event queue processing

The internal driver logic is based on events processing which are generated by the interrupt handler and algorithms.

The queue initialization and deinitialization functions are called by the FSM manager in its `acamera_fsm_mgr_init` and `acamera_fsm_mgr_deinit` interfaces. The event queue is protected by a spinlock since it is used in the interrupt handler and another kernel thread.

At the beginning the event queue is empty and the driver is in idle state until the first interrupt comes. When the Frame Start interrupt occurs, the driver updates the ISP configuration space and generates the first event:
`event_id_new_frame,acamera_fw_raise_event( p_ctx, event_id_new_frame);`

Any new event is put into the event queue FIFO by the function `acamera_event_queue_push`. For example:

```
void acamera_event_queue_push(acamera_event_queue_ptr_t p_queue, int
event)
```

All events are handled inside `acamera_process` function one by one until the FIFO is empty.

Note: ***An event processing handler can add new events to the FIFO and the events will be processed at the same call of the acamera_process function.***

When the next Frame Start interrupt comes, the driver first checks that the event FIFO is empty and there are no unprocessed events left in the queue. That means that all events should be handled for one frame period otherwise the 3A algorithms may not work properly.

Each FSM component has its own event processing flow which can be reviewed in the file `##fsm_name##_process_event(##fsm_name##_fsm_t *p_fsm,event_id_t event_id)`

For example, the event flow for auto exposure algorithm is located inside the function:

```
uint8_t AE_fsm_process_event(AE_fsm_t *p_fsm,event_id_t event_id)

{

    uint8_t b_event_processed=0;

    switch(event_id)

    {

    default:

        break;

    case event_id_ae_stats_ready:

        ae_calculate_target(p_fsm);

        ae_calculate_exposure(p_fsm);

        fsm_raise_event(p_fsm,event_id_exposure_changed);

        AE_request_interrupt(p_fsm, ACAMERA_IRQ_MASK(ACAMERA_IRQ_AE_STATS));

        b_event_processed=1;

        break;

    }

    return b_event_processed;

}
```

From the preceding code snippet, you can see the entry event on which the exposure calculation logic is started is `event_id_ae_stats_ready`. After all the parameters are calculated another event is generated with the name `event_id_exposure_changed`. The FSM manager notifies all other FSMs that depend on this event that the event has occurred.

Note: Since the 3A Algorithms are running in the user space library, the event flow is different as compared with the bare-metal version of the driver.

The typical event flow for AE/AWB/AF and other algorithms from the library is as follows:

1. The FSM Manager calls the `proc_interrupt` routine for every FSM.
2. If the interrupt type is IRQ_AE_STATS/AWB_STATS/AF_STATS then the 3A FSMs request a new buffer from the SBUF manager to read statistic information into it.
3. FSM marks the buffer as SBUF_STATUS_DATA_DONE.
4. SBUF manager checks if any filled buffers are available and sets a notification flag to let the user space 3A Library know that new statistic is ready.
5. 3A Library uses the statistic information in SBUF to calculate the output parameters. It puts the results back into the shared memory and notifies the kernel driver that the result is ready.
6. SBUF manager calls the FSM manager to update the result for the specific FSM.
7. FSM manager raises an event to notify the FSM that the new result is available.

As an example of this flow please refer to the functions `ae_read_full_histogram_data`, `sbuf_mgr_apply_new_param`, and `ae_set_new_param`.

3.4.9 Calibration access and update

The calibration data can be accessed via the following interfaces in the `acamera_calibrations.h` file:

```
const void *_GET_LUT_PTR( void *p_ctx, uint32_t idx );

uint8_t *_GET_UCHAR_PTR( void *p_ctx, uint32_t idx );

uint16_t *_GET_USHORT_PTR( void *p_ctx, uint32_t idx );

uint32_t *_GET_UINT_PTR( void *p_ctx, uint32_t idx );
```

The `p_ctx` is a pointer which points to the target context of `struct acamera_context_t`, `idx` is the index of the LUT which is defined in the file `acamera_command_api.h`. The returned pointer points to the binary data of the LUT. The pointer can be converted to some other structure pointer and driver should use new structure pointer according to its definition. Wrong usage may cause undefined behaviour.

For example, in CMOS FSM, it accesses the CALIBRATION_CMOS_CONTROL data to decide whether the `manual_integration_time` is enabled.

```
void cmos_alloc_integration_time( cmos_fsm_ptr_t p_fsm, int32_t int_time )
```

```
{
    cmos_control_param_t * param = (cmos_control_param_t *)_GET_UINT_PTR(
ACAMERA_FSM2CTX_PTR( p_fsm ), CALIBRATION_CMOS_CONTROL );

    if ( param->global_manual_integration_time ) {

        new_integration_time_short = param->global_integration_time;

    } else {

        new_integration_time_short = acamera_math_exp2( int_time, LOG2_GAIN_SHIFT, 0 );

    }

}
```

The calibration data can be updated at runtime from the control tool via the API command `acamera_calibration_update()`.

3.4.10 Calibration switch logic

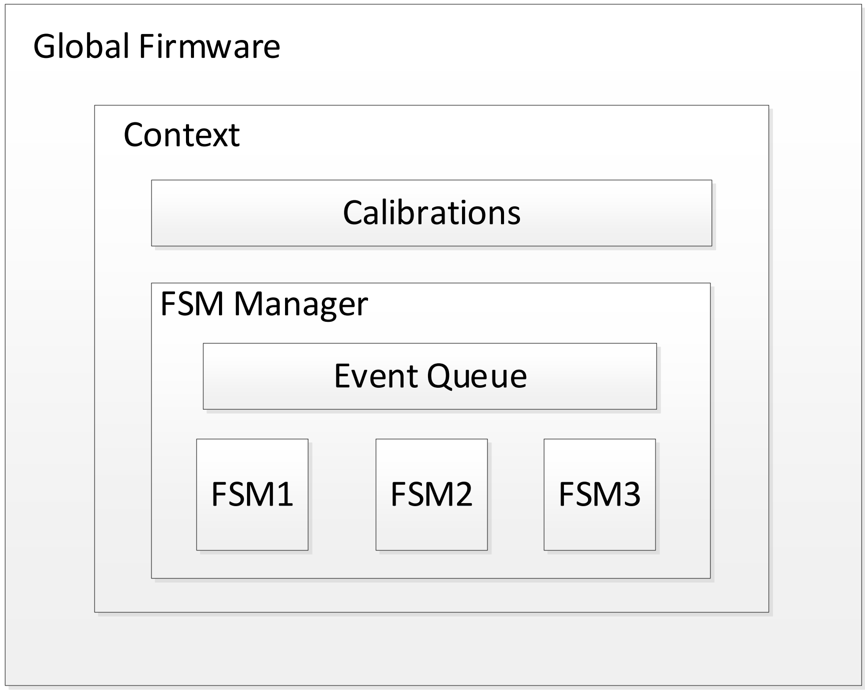
The calibration switch logic is in a single place inside the `/src/calibrations/acamera_get_calibrations.c` file.

Every time the external application changes the sensor mode, the main driver calls the function `get_calibrations` by calling the `acamera_command_api` routine with the `SENSOR_PRESET` parameter. Based on the input sensor preset mode, the driver returns different calibration sets.

The switching logic may be customized if required or disabled if only one calibration set is used.

3.4.11 Main structures

Inside the ISP Kernel Driver there are several layers of levels of abstractions which are shown in the following diagram:



3.4.11.1 Global Firmware

The global firmware is the top level structure for the whole driver logic. It keeps information about all contexts and DMA channels.

```
struct _acamera_firmware_t {
  #if ISP_DMA_RAW_CAPTURE
  // dma_capture
  dma_raw_capture_t dma_raw_capture ;
  #endif

  uint32_t api_context ;      // the active context for API and to display
  uint32_t isp_running_mode ; // online or offline for multicontext isp
  // context instances
  uint32_t context_number ;
  acamera_context_t fw_ctx[ FIRMWARE_CONTEXT_NUMBER ] ;

  void* dma_chan_isp_config ;
  uint32_t dma_flag_isp_config_completed;
  void* dma_chan_isp_metering ;
  uint32_t dma_flag_isp_metering_completed;
  uint32_t irq_fs_happend;
  uint32_t start_processing_on_fs;


  uint32_t initialized ;

  semaphore_t sem_evt_avail;
} ;
```

Note: Multi-context is not supported by the driver so FIRMWARE_CONTEXT_NUMBER parameter must be initialized to one.

3.4.11.2 Context

This is the global structure used to represent one sensor context.

The context maintains information about the FSM Manager, calibration parameters and global configuration settings.

It is represented by the structure `acamera_context_t` defined in the `acamera_fw.h` file.

The following table describes the main fields of the context structure:

Name	Type	Definition
`fsm_mgr`	`acamera_fsm_mgr_t`	Pointer to the FSM manager.
`p_gfw`	`acamera_firmware_t`	Pointer to the global firmware structure described in the previous paragraph.
`acameraCalibrations`	`ACameraCalibrations`	Calibration structure which includes static and dynamic LUTs.
`context_id`	`uint32_t`	Reserved. Must be zero.
`settings`	`acamera_settings`	Camera settings from the application.
`isp_sequence`	`sensor_reg_t`	ISP initialization sequence.

3.4.11.3 FSM Manager

The FSM Manager is the main component of the ISP Kernel Driver and it:

- controls all FSM components through unified interfaces.
- handles events.
- implements inter-FSM communication channels.

The `_acamera_fsm_mgr_t` structure is defined in the `acamera_fsm_mgr.h` file.

```
struct _acamera_fsm_mgr_t
{
    uint8_t ctx_id;
    uintptr_t isp_base;
    int index;
    acamera_context_ptr_t p_ctx;
    isp_info_t info;
    fsm_common_t *fsm_arr[FSM_ID_MAX];
    acamera_event_queue_t event_queue;
```

```
    uint8_t event_queue_data[ACAMERA_EVENT_QUEUE_SIZE];
    uint32_t reserved;
};
```

Name	type	Definition
ctx_id	uint8_t	Reserved.
uintptr_t	isp_base	Reserved.
p_ctx	acamera_context_ptr_t	Pointer to the context the FSM manager belongs to.
info	isp_info	ISP information structure.
fsm_arr	fsm_common_t	Array of FSM components handled by the manager.
event_queue	acamera_event_queue_t	Sync primitive to sync the queue.
event_queue_data	uint8_t	Internal events FIFO.
reserved	uint32_t	Reserved.

3.4.11.4 FSM Interface

The driver includes:

- source code to deal with interrupts, events, and calibrations.
- a set of FSM components which implement the main logic.

Each FSM has a standard interface which is defined in the `fsm_intf.h` file.

The FSM manager treats all components in the same way and controls it by means of `fsm_ops_t` functions.

```
typedef struct _fsm_common_t_ {
void *      p_fsm;
void *         p_fsm_mgr;
uintptr_t  isp_base;
uint8_t     ctx_id;
fsm_ops_t   ops;
} fsm_common_t;
```

Name	type	Definition
p_fsm	void*	Pointer to a specific FSM. The FSM manager doesn't know anything about the data hidden behind the pointer. It treats all fsm in the same way.
p_fsm_mgr	void*	Pointer to FSM manager. All FSMs have this pointer and share one FSM manager.
isp_base	uintptr_t	Reserved.
ctx_id	uint8_t	Reserved.

Name	type	Definition
ops	fsm_ops_t	List of pointers to the control functions.

The control functions are defined by the following structure:

```
typedef struct _fsm_ops_t_ {

      FUN_PTR_INIT        init;
      FUN_PTR_DEINIT      deinit;

      FUN_PTR_RUN         run;
      FUN_PTR_SET_PARAM   set_param;
      FUN_PTR_GET_PARAM   get_param;

      FUN_PTR_PROC_EVENT  proc_event;
      FUN_PTR_PROC_INT    proc_interrupt;
} fsm_ops_t;
```

Name	Type	Definition
init	FUN_PTR_INIT	This function is called by FSM manager on initialization stage when acamera_init is called.
deinit	FUN_PTR_DEINIT	This function is called to de-initialize an FSM.
run	FUN_PTR_RUN	This function is called to give an FSM processing time for internal logic. That is usually the entry point for an FSM to process events and calculate the output parameters.
set_param	FUN_PTR_SET_PARAM	This function is called to change an FSM settings.
get_param	FUN_PTR_GET_PARAM	This function is called to request current FSM settings.
proc_event	FUN_PTR_PROC_EVENT	FSM manager calls this function to indicate that an even has happened.
proc_interrupt	FUN_PTR_PROC_INT	This function is called in the interrupt context to handle time-critical events.

3.4.11.5 Event Queue

Event queue is used to save all the events in the driver. Events are generated from the interrupt handler and some event handlers. The task of the main thread is to handle all the events in the queue. All events generated for one frame should be finished within the frame period. The event related structures are as follows:

```
struct _acamera_loop_buf_t {

    volatile int head;

    volatile int tail;

    uint8_t *p_data_buf;
```

```
    int data_buf_size;

};

typedef struct acamera_event_queue {

    sys_spinlock lock;

    acamera_loop_buf_t buf;

} acamera_event_queue_t;
```

The buffer of the event queue is not included in the `acamera_event_queue_t` structure, instead, it is defined at `struct _acamera_fsm_mgr_t` and initialized in the function `acamera_fsm_mgr_init()`.

```
struct _acamera_fsm_mgr_t {

    acamera_event_queue_t event_queue;

    uint8_t event_queue_data[ACAMERA_EVENT_QUEUE_SIZE];

};
```

3.4.11.6 Exposure partition table

The dynamic calibration file has the table to control how the target exposure is converted to integration time and gains. Gains include sensor analog gain, sensor digital gain and ISP digital gain.

```
// *** NOTE: to add/remove items in partition luts, please also update
SYSTEM_EXPOSURE_PARTITION_VALUE_COUNT.

static uint16_t _calibration_cmos_exposure_partition_luts[][10] = {

    // {integration time, gain }

    // value: for integration time - milliseconds, for gains - multiplier.

    //         Zero value means maximum.


    // LUT partitions_balanced

    {

        10, 2,

        30, 4,

        60, 6,

        100, 8,

        0, 0,
```

```
    },


    // LUT partition_int_priority
    {
        0, 0,
        0, 0,
        0, 0,
        0, 0,
        0, 0,
    },
};
```

Every two data in the LUT is a pair of integration time and gain settings, as the above comments indicate. The length of the LUT should match another parameter SYSTEM_EXPOSURE_PARTITION_VALUE_COUNT which is used in the CMOS FSM.

The integration time in the LUT is in milliseconds and it will be converted to sensor lines when running. The function `cmos_update_exposure_partitioning_lut()` shows the detailed information of this partition table.

Note: ***Customers can change the LUT based on the product requirement. The API function*** **`ae_split_preset()`** ***must be updated if a new LUT is needed.***

3.4.12 Communication with User Driver

The ISP Driver uses the shared buffer (SBUF) channel to transfer data to the ISP User Driver and gets back the result from it. The SBUF is used to share data between the kernel-space and the user-space with zero-copy.

The SBUF channel is managed by `sbuf_mgr` in SBUF FSM, which belongs to a ISP Driver context.

Based on the customer's requirements, new data structure can be added into the SBUF.

Note: ***You must keep the same SBUF structure in both, the kernel driver and the user driver. Failure to do so can lead to undefined problems with the kernel firmware and user application behavior.***

The following figure illustrates the structure of the SBUF channel.

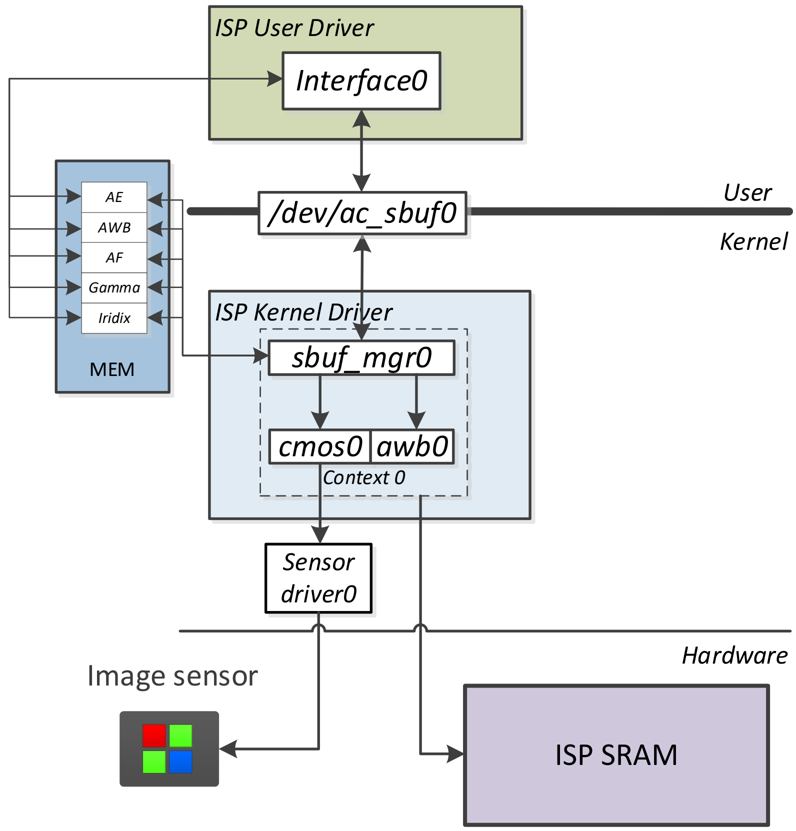


Figure 14. SBUF channel

3.4.12.1 Interface

The SBUF channel is created when the SBUF FSM is initialized inside the `acamera_init` routine. The FSM exports the interface via the device node, such as `/dev/ac_sbuf0`. The shared memory will be allocated at initialization stage.

ISP User Driver maps the memory into its own memory space after opening the device node. It can read the SBUF index array from the device to get the information about which type of `sbuf` has new data to be shared, then application can access the data directly. The ISP User Driver needs to return the buffer to the Kernel Driver after its usage, otherwise, the kernel driver will not provide new SBUF item with the same SBUF type.

The SBUF status transition is shown in the following figure:

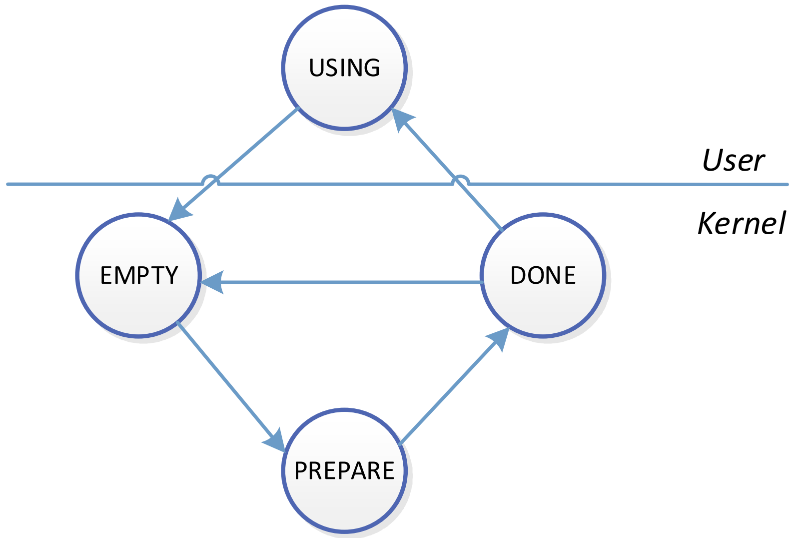


Figure 15. SBUF buffers state

The DONE to EMPTY transition will happen if there is no EMPTY buffer but `sbuf_mgr` is required to give one EMPTY buffer.

3.4.12.2 Shared buffer state flow

The SBUF is shared by the Kernel Driver and the User Driver. It is used by several components in both drivers. The following diagram illustrates the detailed flow process of the `ae_sbuf`. Other SBUFs, such as `awb_sbuf`, `af_sbuf` and `gamma_sbuf` have the same process.

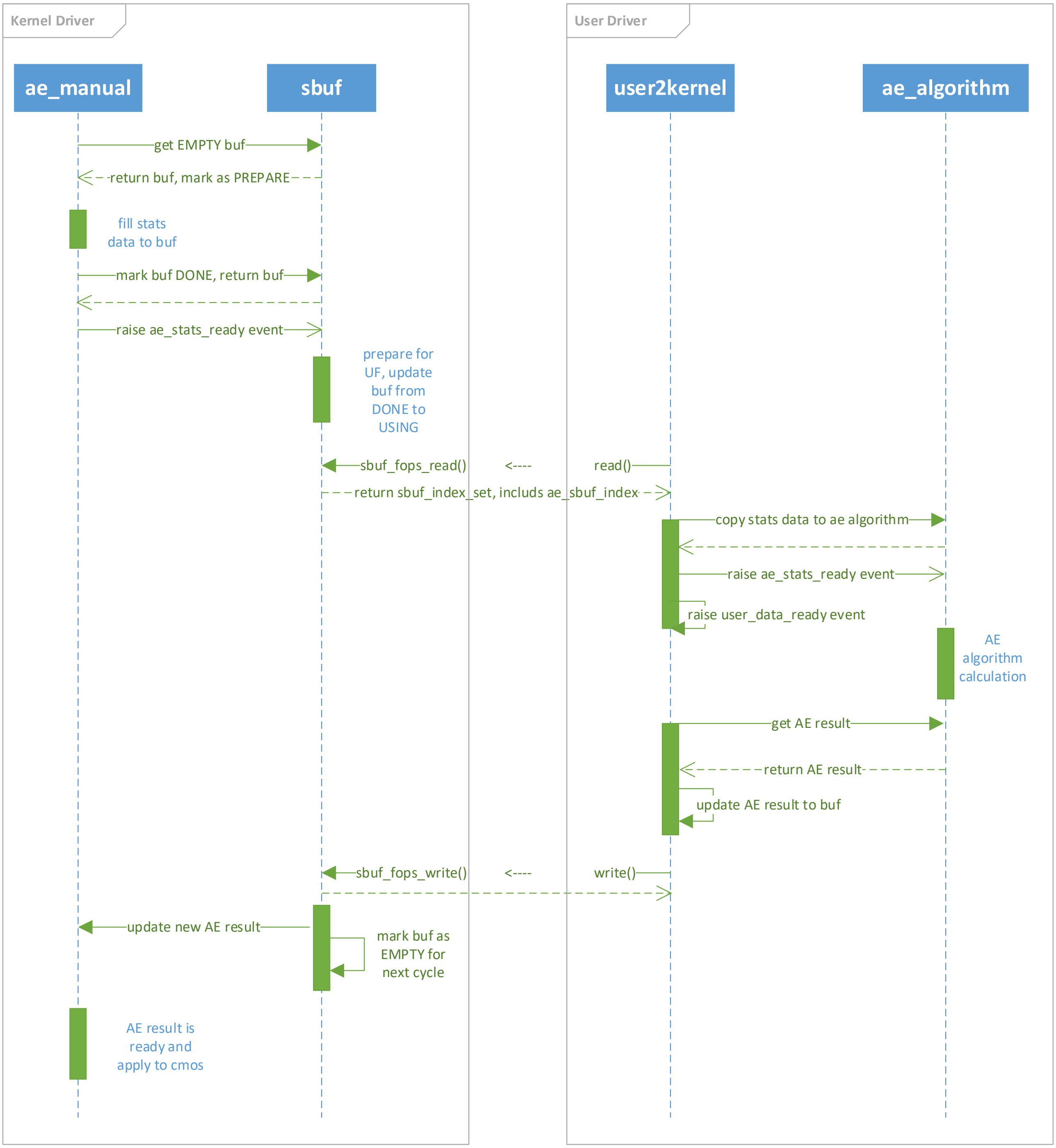


Figure 16. AE SBUF cycle between ISP kernel driver and user driver

3.4.12.3 Shared IQ calibration data

The SBUF channel also supports the shared calibration data between the ISP kernel driver and ISP user driver. Instead of including the calibration data in the driver, it is more flexible to share the IQ calibration data between kernel driver and user driver so that when data is changed, you don't need to recompile the user driver to use the new data.

The calibration data size is a 128KB array in SBUF for easy communication between kernel driver and user driver, it can be changed via the macro `ISP_MAX_CALIBRATION_DATA_SIZE` in the project setting file. Another data member `is_fetched` is used to synchronize kernel driver and user driver when the calibration is changed. The structure is as follows:

```
struct calibration_info {

    uint8_t is_fetched;

    uint8_t cali_data[ISP_MAX_CALIBRATION_DATA_SIZE];

};
```

Another new structure `sbuf_lookup_table` is introduced for SBUF calibration data, it is based on LookupTable in `acamera_types.h` but does not have const qualifier so that kernel driver can prepare the data for user driver, these two structures should have the same memory layout, otherwise, user driver may not work as expected.

3.4.12.3.1 Control flows

There are some different use cases of shared calibration data, the control flows are introduced in this section.

The control flow of initialization is as follows:

1. The kernel driver creates a device node, prepares the LUT array, and copies the calibration data into the LUT array.
2. User driver opens the divide node and memory mapping into SBUF.
3. The user driver reconstructs the LUT because it has different memory space and there is a pointer data member in LUT item.
4. The kernel driver and user driver use different copy of the same calibration data.

The control flow when sensor preset mode changed is as follows:

1. The kernel driver API command handler receives the change sensor preset mode command.
2. The kernel driver API command handler stops the ISP.
3. The kernel driver API command handler forwards the command to the user driver.
4. The kernel driver sensor module changes the sensor preset mode and starts streaming.
5. The kernel driver updates to new calibration data.
6. The kernel driver SBUF module waits until the user driver finishes the use of the old calibration data.

7. The kernel driver SBUF module constructs a new LUT array and copies the new calibration data.

8. The kernel driver SBUF module marks a new flag and waits for the user driver to fetch new data.

9. The user driver API command handler receives the change sensor preset mode command.

10. The user driver sensor module changes the internal sensor mode.

11. The user driver sensor module updates new calibration data from the user2kernel module.

12. The user driver user2kenrel module waits for the kernel driver to update the calibration data and mark the flag.

13. The user driver user2kenel module reconstructs the new calibration LUTs and changes the flag.

14. The kernel driver and user driver are synchronized with the new calibration data.

NOTE: The kernel driver and user driver have different threads, so the order of steps 4) and step 9) cannot be predicted, for this reason a flag is added for synchronisation purpose.

3.4.13 Log system

Adding logging in the driver is straightforward, for example, in the `acamera.c` file there are many log statements:

```
// Information log

LOG(LOG_INFO, "IRQ MASK is %d", irq_mask );

// Critical error log

LOG(LOG_CRIT, "Software Context %d failed to allocate", idx );
```

The `LOG_INFO` and `LOG_CRIT` is the log level which is used to control whether this log is output to the terminal or ignored. This depends on the configuration parameter values used when building the driver.

The main implementation of the log system is located in the files `acamera_logger.h` and `acamera_logger.c`. The kernel driver log system also supports several configurable parameters which can be used to customize logs for different requirements.

```
FW_LOG_FROM_ISR

FW_LOG_HAS_SRC

FW_LOG_HAS_TIME

FW_LOG_LEVEL
```

```
FW_LOG_MASK

FW_LOG_REAL_TIME
```

3.4.13.1 Log configuration parameters

The usage of each log configuration parameter is given in this section.

- **FW_LOG_REAL_TIME**

  Values: [0: Disable, 1: Enable]

  Default: 0

This configuration parameter is used to control the log output level. The masking can be changed in real-time or fixed at compile time.

If this parameter is enabled, the output log level and mask can be dynamically changed via `TSYSTEM` API commands `system_logger_level()` and `system_logger_mask()`.

Note: Enable DEBUG level will generate too many logs and may reduce system performance. This should be used only for debug purpose.

If this parameter is disabled, the output log level and masking is decided by the parameter FW_LOG_LEVEL and FW_LOG_MASK.

- **FW_LOG_LEVEL**

  Values: [LOG_DEBUG, LOG_INFO, LOG_NOTICE, LOG_WARNING, LOG_ERR, LOG_CRIT, LOG_NOTHING]

  Default: LOG_NOTHING

This configuration parameter is used to control the output log level when the parameter FW_LOG_REAL_TIME is 0. You can change this parameter to enable some logs.

- **FW_LOG_MASK**

  Values: [0x0 - 0xFFFFFFFF]

  Default: 0xFFFF

This configuration parameter is used to control the log mask of modules when the parameter FW_LOG_REAL_TIME is 0. There is a definition for each module, such as LOG_MODULE_CMOS, LOG_MODULE_SENSOR and so on. Each module corresponds to one bit in this mask.

- **FW_LOG_HAS_TIME**

  Values: [0: Disable, 1: Enable]

  Default: 0

This configuration parameter controls the timestamp in the log. The driver adds a timestamp if it is enabled, otherwise, no timestamp is added to the log.

- **FW_LOG_HAS_SRC**

  Values: [0: Disable, 1: Enable]

  Default: 0

This configuration parameter controls the file name and function name in the log. The driver adds those fields in the log if it is enabled, otherwise, no information about the file and function is added in the log.

- **FW_LOG_FROM_ISR**

  Values: [0: Disable, 1: Enable]

  Default: 0

This configuration parameter is not used in the Linux system.

3.4.14 Configuration file

The ISP Kernel Space Driver is delivered with predefined configuration parameters. All parameters are collected and located in the file `/inc/acamera_firmware_config.h`

Some parameters should not be changed because they describe specific hardware configuration but some of them can be updated at the compilation stage.

The following table lists the important parameters with their descriptions:

Name	Type	Default Value	Possible values	Description
`KERNEL_MODULE`	Fixed	1	1	Enable the driver to be used in the Linux Kernel.
`ACAMERA_EVENT_QUEUE_SIZE`	Flexible	256	>=128	Maximum possible events to be stored in the queue.
`ACAMERA_IRQ_FRAME_START`	Fixed	7	7	Frame Start IRQ number.
`ACAMERA_IRQ_FRAME_END`	Fixed	0	0	Frame End IRQ number.
`CONNECTION_BUFFER_SIZE`	Flexible	26 KB	>1024	Buffer used to get/set data to driver, such as read register values, send API commands, small buffer size may cause command failed.
`CONNECTION_IN_THREAD`	Flexible	1	0 / 1	Command is handled in a separate thread or in a dedicated thread.
`FIRMWARE_CONTEXT_NUMBER`	Fixed	1	1	The current driver version supports only 1 context.

Name	Type	Default Value	Possible values	Description
FW_HAS_CONTROL_CHANNEL	Fixed	1	1	Control channel for user driver.
FW_LOG_LEVEL	Flexible	LOG_NOTHING	Refer to "Log system" for possible values	Control the log system output level when FW_LOG_REAL_TIME is 0.
ISP_MAX_CALIBRATION_DATA_SIZE	Flexible	128 KB	>= sizeof(calibration data)	The calibration data size used by shared buffer which depends on the real calibration size. If this parameter is less than the real calibration size, the user driver may not work properly.

3.4.15 Main application example

The `main_firmware.c` file implements the reference application which shows the basic principle of the ISP driver usage. The V4L2 ISP Device uses the same file to initialize the ISP Kernel Driver.

The following list indicates the key steps involved in using the ISP driver:

1. Ensure that the system BSP layer is ready to be used by the driver.
2. Initialize with `acamera_init`.
3. Create a thread and call `acamera_process` until the application is closed.
4. Call the `acamera_interrupt_handler` on every event from the ISP hardware.
5. Close the driver with `acamera_terminate`.

The following code snippet is a reference example from the `main_firmware.c` file. Important places are indicated in bold.

```
// The driver can provide the full set of metadata parameters
// The callback_meta function should be set in initialization settings to support it.
void callback_meta( uint32_t ctx_num,  const void *fw_metadata)
{
}

static int isp_fw_process(void *data)
{
  LOG(LOG_CRIT,"isp_fw_process start" ) ;

  while (!kthread_should_stop()) {
     acamera_process( ) ;
   }

  LOG(LOG_CRIT,"isp_fw_process stop" ) ;
  return 0;
}
```

```
// this is a main application IRQ handler to drive firmware
// The main purpose is to redirect irq events to the
// appropriate firmware context.
// There are several type of firmware IRQ which may happen.
// The most basic one is ACAMERA_IRQ_ISP which means that interrupt from ISP happened.
// The other IRQ events are platform specific. The firmware can support several external irq events
or
// does not support any. It totally depends on the system configuration and firmware compile time
settings.

static void interrupt_handler( void* data, uint32_t mask ) {
    acamera_interrupt_handler( ) ;
}

int isp_fw_init(void)
{
    int result = 0 ;

    bsp_init();

    LOG(LOG_INFO,"fw_init start" ) ;

    // The firmware supports multicontext.
    // It means that the customer can use the same firmware for controlling
    // several instances of different sensors/isp. To initialise a context
    // the structure acamera_settings must be filled properly.
    // the total number of initialized context must not exceed FIRMWARE_CONTEXT_NUMBER
    // all contexts are numerated from 0 till ctx_number - 1
    result = acamera_init( settings, FIRMWARE_CONTEXT_NUMBER );

    if ( result == 0 ) {
        uint32_t rc = 0;
        uint32_t ctx_num;

        // set the interrupt handler. The last parameter may be used
        // to specify the context. The system must call this interrupt_handler
        // function whenever the ISP interrupt happens.
        // This interrupt handling procedure is only advisable and is used in ACamera demo
application.
        // It can be changed by a customer discretion.
        system_interrupt_set_handler( interrupt_handler, NULL ) ;

        // start streaming for sensors
        for(ctx_num = 0; ctx_num < FIRMWARE_CONTEXT_NUMBER; ctx_num++) {
          application_command(TSENSOR, SENSOR_STREAMING, ON, COMMAND_SET, &rc);
        }

    } else {
         LOG(LOG_INFO,"Failed to start firmware processing thread. " ) ;
    }

    LOG(LOG_INFO,"isp_fw_init result %d", result ) ;
    if ( result == 0 ) {

      LOG(LOG_INFO,"start fw thread %d", result ) ;
      isp_fw_process_thread = kthread_run(isp_fw_process, NULL, "isp_process");
    }

  return PTR_RET(isp_fw_process_thread);
}

void isp_fw_exit(void)
{

  if(isp_fw_process_thread) {
    kthread_stop(isp_fw_process_thread);
```

```
  }

  // this api function will free
  // all resources allocated by the firmware
  acamera_terminate() ;

  bsp_destroy();
}
```

3.5 ISP user space driver

The ISP User Driver is based on the same principle as the ISP Kernel Driver with the following exceptions:

- It does not access the ISP or any other hardware directly.
- It implements the 3A algorithms.

3.5.1 Algorithm list

The user driver implements the following algorithms:

- **AE**

  The AE algorithm controls the brightness of the image based on target brightness in calibration data.

  The input data is AE statistics data from kernel driver, the algorithm output is the `ae_exposure` and `ae_exposure_ratio`.

- **AWB**

  The AWB algorithm adjusts the white balance to get correct neutral tones of the image.

  The input data is AWB statistics data from kernel driver, the algorithm output is `awb_red_gain` and `awb_blue_gain` and some other `light_source` related information.

- **AF**

  The AF algorithm calculates the best lens position to get a clear image.

  The input data is AF statistics data from kernel driver, the algorithm output is lens position and sharpness value.

- **Gamma_contrast**

  The Gamma_contrast algorithm adjusts auto level to get the desired image contrast.

  The input data is gamma statistics data from kernel driver, the algorithm output is `gamma_gain` and `gamma_offset`.

- **Iridix**

  The Iridix algorithm uses local tone mapping to increase the range of tones of the image.

  The input data is AE statistics data from AE algorithm, the algorithm output is iridix strength and contrast.

3.5.2 Communication with kernel driver

The user driver communicates with the kernel driver through the device node `/dev/ac_sbuf0` which is created by the kernel driver. The user driver opens the device node when initialised.

Note: ***The user driver exits if the open device node operation fail. This can happen if the kernel driver is not ready or another user driver is already running.***

The user driver maps the shared memory which is allocated by the kernel driver into its own memory space. The shared memory layout is defined as a `struct fw_sbuf` in the header file `sbuf.h`.

The following code segment show that each algorithm has its own array of shared buffer.

```
#define SBUF_STATS_ARRAY_SIZE 4


struct fw_sbuf {

    struct kf_info kf_info;

#if defined( ISP_HAS_AE_BALANCED_FSM ) || defined( ISP_HAS_AE_MANUAL_FSM )

    sbuf_ae_t ae_sbuf[SBUF_STATS_ARRAY_SIZE];

#endif

#if defined( ISP_HAS_AWB_MESH_FSM ) || defined( ISP_HAS_AWB_MESH_NBP_FSM ) || defined(
ISP_HAS_AWB_MANUAL_FSM )

    sbuf_awb_t awb_sbuf[SBUF_STATS_ARRAY_SIZE];

#endif

#if defined( ISP_HAS_AF_LMS_FSM ) || defined( ISP_HAS_AF_MANUAL_FSM )

    sbuf_af_t af_sbuf[SBUF_STATS_ARRAY_SIZE];

#endif

#if defined( ISP_HAS_GAMMA_CONTRAST_FSM ) || defined( ISP_HAS_GAMMA_MANUAL_FSM )

    sbuf_gamma_t gamma_sbuf[SBUF_STATS_ARRAY_SIZE];
```

```
#endif

#if defined( ISP_HAS_IRIDIX_HIST_FSM ) || defined( ISP_HAS_IRIDIX8_FSM ) || defined(
ISP_HAS_IRIDIX_MANUAL_FSM ) || defined( ISP_HAS_IRIDIX8_MANUAL_FSM )

    sbuf_iridix_t iridix_sbuf[SBUF_STATS_ARRAY_SIZE];

#endif

};
```

Note: ***It is important to use the same sbuf.h file in both user driver and kernel driver, otherwise, algorithms may not be working correctly and may cause undefined behavior.***

The shared memory is zero-copy between user driver and kernel driver, the only thing copied between them is the `sbuf_idx_set` which also defined at file `sbuf.h`.

The following struct indicates the buffer index and buffer index validation.

```
struct sbuf_idx_set {

    uint8_t ae_idx;

    uint8_t ae_idx_valid;


    uint8_t awb_idx;

    uint8_t awb_idx_valid;


    uint8_t af_idx;

    uint8_t af_idx_valid;


    uint8_t gamma_idx;

    uint8_t gamma_idx_valid;


    uint8_t iridix_idx;

    uint8_t iridix_idx_valid;

};
```

Based on the validation of index, the user driver updates the related algorithm’s statistics data from the shared memory and trigger algorithm to calculate the new result. The user driver the updates the new result into the shared buffer and notifies the kernel driver. The kernel driver the applies the new result to corresponding places.

3.6 Porting to the target platform

Porting is the final step that you must perform before you can start using the ISP software driver. The user space driver normally does not require porting in Linux-based platforms, the following sections are mainly for kernel space driver porting.

Note: It is important to port the software to your target platform for it to work.

3.6.1 BSP layer

The BSP layer should be implemented on the target platform only for V4L2 ISP device driver. The reference implementation of the layer is provided within the release but some changes are still required to make it work.

Note: The reference implementation of the V4L2 Sensor Driver and V4L2 Lens Driver rely on the same BSP layer. This scenario is valid only for the Arm Development platform. This may not necessarily work as is for customer scenarios. Customers MUST make necessary changes depending on their specific scenarios to ensure that devices communicate properly.

3.6.1.1 system_hw_io.c

This file contains the implementation for read/write access routines to the ISP configuration space.

```
/**
 *   Read 32 bit word from isp memory
 *
 *   This function returns a 32 bits word from ISP memory with a given offset.
 *
 *   @param addr - the offset in ISP memory to read 32 bits word.
 *                 Correct values from 0 to ACAMERA_ISP_MAX_ADDR
 *
 *   @return 32 bits memory value
 */
uint32_t system_hw_read_32(uintptr_t addr) ;


/**
 *   Write 32 bits word to isp memory
 *
 *   This function writes a 32 bits word to ISP memory with a given offset.
 *
 *   @param addr - the offset in ISP memory to write data.
 *                 Correct values from 0 to ACAMERA_ISP_MAX_ADDR.
 *   @param data - data to be written
 */
```

```
void system_hw_write_32(uintptr_t addr, uint32_t data) ;
```

Only `system_hw_read_32` and `system_hw_write_32` must be properly implemented to provide the access from the ISP Kernel Driver to the ISP configuration space memory.

The input addr parameter is the offset inside the ISP configuration address space. For example, if addr is equal 0 the `system_hw_read_32` function should return the first 4 bytes of the ISP configuration memory.

> ***Note:*** ***There is no initialisation routine defined in the system_hw_io.h file so it is assumed that read/write routines work before any of the ISP driver functions are called. This means that the ISP Driver is not responsible for BSP initialisation process and it should be done by the external application. Please refer to the bsp_init function for the reference implementation.***

The implementation for the Arm Development System gets the ISP configuration memory address from the DTS file. You MUST change the DTS file according to the target platform.

> ***Warning:*** ***Usage of wrong address can damage the system and lead to undefined system behaviour.***

3.6.1.2 system_dma.c

The ISP Kernel driver depends on system DMA engines for data transfers between the ISP configuration space and internal buffers. This functionality is system dependent so must be developed on the target platform.

3.6.1.3 system_interrupt.c

The ISP Kernel driver depends on the Linux IRQ implementation. The V4L2 ISP Device initializes the interrupts by calling the `system_interrupt_init` routine and then assigning the callback by a `system_interrupt_set_handler` call. After the callback is set the V4L2 ISP Device expects the callback to be called on every interrupt from the ISP hardware.

It is very likely that on a customer platform the interrupt handling process will be different so this logic must be revised to guarantee proper behavior.

3.6.1.4 system_*.c

For all other `system_*.c` files, the reference implementation is compatible with most Linux-based systems, porting is needed if customers find that their systems are not compatible with the target platform.

3.6.2 Linux DTS table update

The reference implementation of the V4L2 layer relies on the device tree for Arm FPGA Development platform. That is required to update the device tree table of the target Linux system and match it with the V4L2 source code.

For example, the `dts` file for Arm Development system is located under Linux directory `/linux_kernel/linux/arch/arm64/boot/dts/arm/juno-r2.dts`.

The file includes the description of v4l2 main device and its sub-devices. It is very likely that the target system will use the same approach but all names and base addresses will be different. Customers should update the similar file during the driver porting stage.

```
isp: isp@0x64000000 {
        compatible = "arm,isp";
        reg = <0x0 0x64000000 0x0 0x00300000>;
        interrupts = <0 168 1>;
        interrupt-names = "ISP";
};

sensor: soc_sensor@0x64300000 {
        compatible = "soc,sensor";
        reg = <0x0 0x64300000 0x0 0x001000>;
};

lens: soc_lens@0x64301000 {
        compatible = "soc,lens";
        reg = <0x0 0x64301000 0x0 0x001000>;

iq: soc_iq@0x64302000 {
        compatible = "soc,iq";
        reg = <0x0 0x64302000 0x0 0x001000>;
};

reserved-memory {

         isp_reserved: frame_buffer@64400000 {

                 compatible = "shared-dma-pool";

                 no-map;

                 #address-cells = <2>;

                 #size-cells = <2>;

                 reg = <0x0 0x64400000 0x0 0xB000000>;

         };

 };
```

The next step is to be sure that the names provides in the DTS file name exactly match the names in the driver source code.

For V4L2 Lens Sub-device the device tree structure is declared in the `soc_sensor.c` file and looks like the following code snippet:

```
static const struct of_device_id isp_dt_match[] = {
    { .compatible = "soc,lens" },
    {}
};


static struct platform_driver soc_lens_driver = {
    .driver = {
        .name = "soc,lens",
        .owner = THIS_MODULE,
        .of_match_table = isp_dt_match,
    },
};
```

Note: ***The name "soc,lens" must match the name from Linux dts file.***

For V4L2 Sensor Sub-device the device tree structure is declared in the `soc_sensor.c` file and looks like the following code snippet:

```
static const struct of_device_id isp_dt_match[] = {
    { .compatible = "soc,sensor" },
    {}
};


static struct platform_driver soc_sensor_driver = {
    .driver = {
        .name = "soc,sensor",
        .owner = THIS_MODULE,
        .of_match_table = isp_dt_match,
    },
};
```

Note: ***The name "soc,sensor" must match the name from the Linux dts file.***

3.6.3 Linux V4L2 support

The V4L2 Framework must be enabled in the Linux configuration file. For example, the Linux config for the Arm Development platform includes the following parameters:

```
CONFIG_VIDEO_DEV=y

CONFIG_VIDEO_V4L2_SUBDEV_API=y

CONFIG_VIDEO_V4L2=y

CONFIG_VIDEOBUF_GEN=y

CONFIG_VIDEOBUF2_CORE=y

CONFIG_VIDEOBUF2_MEMOPS=y
```

```
CONFIG_VIDEOBUF2_VMALLOC=y
CONFIG_VIDEOBUF2_DMA_CONTIG=y
CONFIG_VIDEOBUF2_DMA_SG=y
```

3.6.4 Frame buffers

The Arm ISP does not support MMU. Due to this you cannot use virtual memory and pointers allocated by Linux framework directly. Instead the V4L2 Device uses contiguous memory reserved from the system in advance. The reserved memory is specified in `juno-r2.dts` file based on Arm Juno hardware environment, customer needs to port the reserved memory or uses MMU based on the customer's system.

```
reserved-memory {
                isp_reserved: frame_buffer@64400000 {
                        compatible = "shared-dma-pool";
                        no-map;
                        #address-cells = <2>;
                        #size-cells = <2>;
                        reg = <0x0 0x64400000 0x0 0xB000000>;
                };
        };
```

To keep the device driver as independent as possible of the platforms, the frame buffer interfaces are moved to the application, such as bare metal application, v4l2 memory management etc. The interfaces for the frame buffers are:

```
// Allocate DMA-able contiguous and cache-coherent buffers for temper and dma_writer(default buffer)
void *callback_dma_alloc_coherent(uint32_t ctx_id, uint64_t size, uint64_t *dma_addr)
// Free allocated DMA buffers
void callback_dma_free_coherent(uint32_t ctx_id, uint64_t size, void *virt_addr, uint64_t dma_addr);
// Get frame buffer from application, such as v4l2 layer.
int callback_stream_get_frame(uint32_t ctx_id, acamera_stream_type_t type, aframe_t *aframes,
uint64_t num_planes);
// Return frame buffer to application, such as v4l2 layer.
int callback_stream_put_frame(uint32_t ctx_id, acamera_stream_type_t type, aframe_t *aframes,
uint64_t num_planes);
```

The interfaces are set in runtime_initialization_settings.h:

```
static acamera_settings settings[ FIRMWARE_CONTEXT_NUMBER ] = { {
.callback_dma_alloc_coherent = callback_dma_alloc_coherent,
```

```
.callback_dma_free_coherent = callback_dma_free_coherent,

.callback_stream_get_frame = callback_stream_get_frame,

.callback_stream_put_frame = callback_stream_put_frame,

},
```

3.6.5 Sensor driver

The Sensor driver should be implemented based on the interface in the `soc_sensor.h` file.

The main V4L2 device issues the sequence of calls ALLOC_IT, ALLOC_DGAIN, ALLOC_AGAIN, UPDATE_EXP on every frame. The sensor driver is responsible to return the actual values for the integration time and gains which the sensor can accept.

The main driver uses the returned parameters to recalculate the amount of each gain and guarantee that the full exposure is applied.

For example:

The Main driver wants to apply the exposure value 100 which can be represented as:

IT * SDG * SAG * IDG

where

IT is integration time

SDG is sensor digital gain

SAG is sensor analog gain

IDG is ISP digital gain

The Main driver is trying to split the EV into the components above as:

IT = 10

SAG = 5

SDG = 2

IDG = 1

In the first step the call `ioctl(SOC_SENSOR_ALLOC_IT , 10 )` is made and the sensor driver has to return the closest possible accepted value. Assume the returned amount of IT is the same as requested, that is, 10.

The second step for the main driver is to try to understand what is the closest possible amount of analog gain we can apply on a sensor side. To do this the driver calls:

`again = ioctl( SOC_SENSOR_ALLOC_AGAIN, 5)`. Assume that 5 is not a possible value for a sensor and it returns 4 as the best closest gain which is not greater than requested.

The main driver has to recalculate the amount of the digital gains to be applied since the target is to utilize the full exposure value (100). To do this SDG must be equal 2.5 since 100 = 10 (IT) * 4 (SAG) * 2.5 (SDG),

`dgain = ioctl(SOC_SENSOR_ALLOC_DGAIN, 2.5)`.

If the driver returns the requested amount 2.5 then the allocation phase is completed. Otherwise, a gain equivalent to the difference amount will be applied on the ISP side by changing ISP DGAIN parameter.

After the full exposure value is utilized the main driver logic will call:

`ioctl(SOC_SENSOR_UPDATE_EXP)` to apply the previously requested values on the sensor side.

Note: The main driver does all these calls in specific time to synchronize the sensor and ISP parameters.

An important parameter in the sensor driver is the `integration_time_apply_delay`. It affects how the CMOS FSM in the main driver synchronizes sensor integration time and ISP digital gain.

The CMOS FSM splits AE exposure into IT, SAG, SDG and IDG. To avoid frame flicker, the main driver needs to ensure that all the factors take effect at the same frame. This synchronization process, based on the parameter `integration_time_apply_delay` of 2, is shown in the following figure

Frame_ID	1	2	3	4	5	6	7	8
CMOS Exposure_Set	Default	F1_exp_set	F2_exp_set	F3_exp_set	F4_exp_set	F5_exp_set	F6_exp_set	F7_exp_set
ISP Effective DIgital Gain	Default	Default	Default	Default	F1_isp_DG	F2_isp_DG	F3_isp_DG	F4_isp_DG
Sensor Effective Integration_time	Default	Default	Default	Default	F1_int_time	F2_int_time	F3_int_time	F4_int_time

Figure 17. IT and IDG synchronization process

Referring to Figure 17,

1. The driver reads the Frame1 AE statistics data at FrameStart of Frame2.

2. The AE algorithm calculates the result and puts the new exposure to CMOS FSM, CMOS splits the new exposure into IT, SAG, SDG, IDG and saves them into F1_exp_set.

3. CMOS FSM updates the sensor integration time and the ISP digital gain from F1_exp_set at the FrameStart of Frame3.
4. F2_exp_set will be updated at Frame4.
5. The sensor integration time and the ISP digital gain of F1_exp_set, will take effect at Frame5.
6. F2_exp_set will take effect at Frame6.

For detailed information of the synchronization logic, please refer to the CMOS FSM in the kernel driver.

NOTE: Due to the delay of 2 to apply the ISP hardware register, the minimum sensor driver integration_time_apply_delay is 2. Customers should implement the sensor driver properly to avoid image flicker problem.

3.6.6 Lens driver

The Lens driver should be implemented based on the interface in the `soc_lens.h` file.

The lens driver sequence is as follows:

1. The main driver initializes the lens device by calling the init function.
2. It gets the information about the lens characteristics such as minimum moving step, lens type, and so on.
3. After this is done the main driver calls `SOC_LENS_MOVE` ioctl to move the lens to the desired position when needed.

The Lens Driver is an optional driver and customers can change the V4L2 device driver to remove this sub-deivce driver. If the sensor module has a fixed lens, macro V4L2_SOC_SUBDEV_NUMBER needs to be changed to 2 and the `.lens_init` and `.lens_deinit` data members in struct `acamera_settings` in the file `runtime_initialization_settings.h` must be set to NULL.

3.6.7 Calibration files

All calibration files are wrapped into the V4L2 IQ sub-device. The main V4L2 ISP driver requests tuning tables from this device and copies them into the internal tables.

Note: The main v4l2 device and v4l2 iq sub-device must share the same version of the acamera_command_id.h and soc_iq.h files.

When the main device needs to update the calibration parameters it sends the ioctl request of type `V4L2_SOC_IQ_IOCTL_REQUEST_INFO` to get the information about LUTs. Later another ioctl may be issued with the type `V4L2_SOC_IQ_IOCTL_REQUEST_DATA` on which the sub-device must return the requested data.

Note: The calibration LUT data needs to be tuned based on sensor and customer requirements.

3.7 Running the ISP Software

The V4L2 ISP Driver should be run in the following order:

1. Add the V4L2 ISP Device to the Linux kernel using the command:

   ```
   # insmod iv009_isp.ko
   ```

2. Add the V4L2 Sensor Driver to the Linux kernel using the command:

   ```
   # insmod iv009_sensor.ko
   ```

3. Add the V4L2 Lens Driver to the Linux kernel using the command:

   ```
   # insmod iv009_lens.ko
   ```

4. Add the V4L2 IQ Driver to the Linux kernel using the command:

   ```
   # insmod iv009_iq.ko
   ```

5. Run the ISP User Driver by executing the command:

   ```
   # ./ iv009_isp.elf
   ```

After the last sub-device is added to the system, the v4l2 driver initializes the sensor, lens and the ISP based on the provided calibration parameters.

After the initialization process has finished the ISP starts generating interrupts every frame. These interrupts are handled by the V4L2 driver.

Note: All source code which is running in the kernel space is distributed under standard GPLv2 license.
The user space driver is distributed under the Arm proprietary license.

4 ISP Software for bare metal platforms

4.1 Overview

A separate version of the ISP Software for bare metal platform is provided. It includes all hardware related code and algorithms in the same library. It allows the code to be easily recompiled for the desired target platform.

The consumer of the driver is responsible for correct implementation of the system dependent layers which are essential for the ISP software. They are:

- **Sensor driver**
  The ISP Software is provided with a reference example of the sensor and implements two supported presets (3 exposures) for DOL and Linear mode (1 exposure)
- **Lens driver**
  The ISP Software supports AF algorithm based on the statistic collected by the ISP hardware core.
- **Calibration tables**
  Each combination of sensor/lens should be tuned based on the procedure described in the Arm ISP Calibration Guide.
- **BSP layer**
  This set of functions should isolate the target platform specifics from the ISP Software.

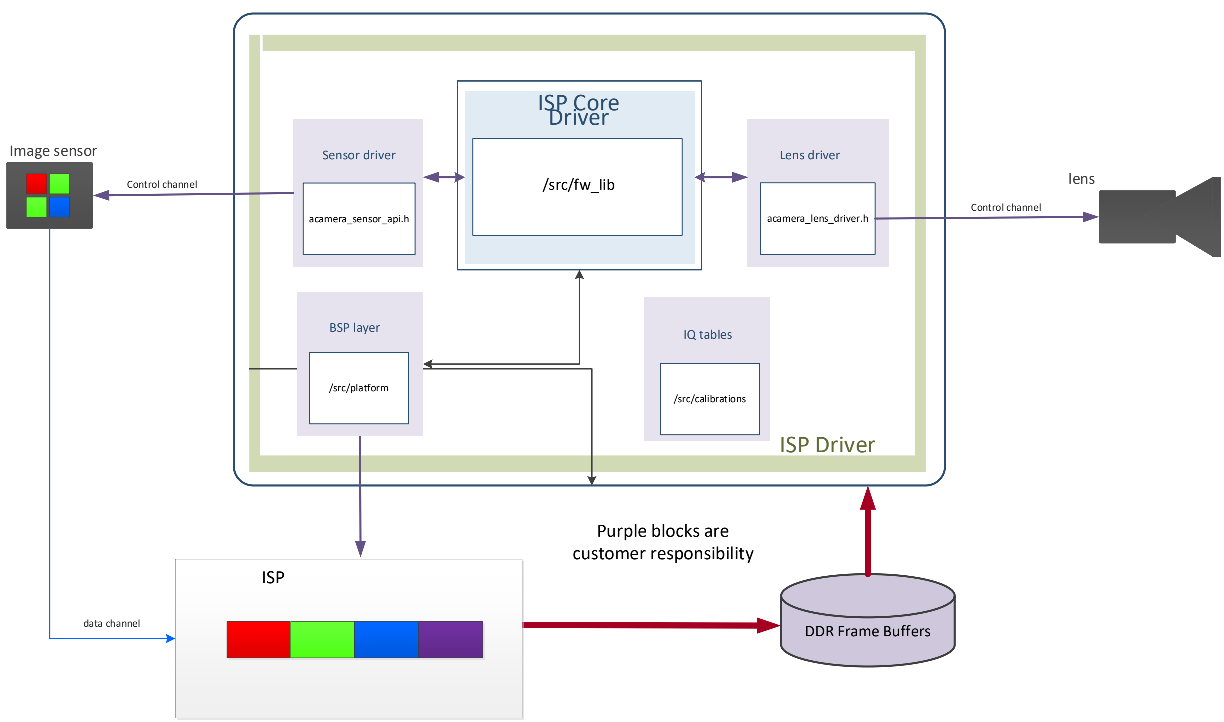


Figure 18. ISP software for bare metal platform

4.2 System requirements

The Bare-Metal firmware has the following system requirements:

Requirement	Value
DMIPS	80
Code Size	110 KB – without calibrations
BSS + Data	80 KB
Stack	8 KB

Table 7. ISP Driver System requirements for Bare-Metal platforms

4.3 ISP Driver architecture

4.3.1 ISP Driver API

The driver APIs for bare-metal architectures are presented the same commands as for Linux version of the driver. Please refer to the ISP Driver API for details.

4.3.2 Sensor driver API

The ISP Driver uses the sensor driver through the API interface from the `/inc/api/acamera_sensor_api.h` file. The ISP driver uses this interface to initialize the sensor, request available modes and change sensor parameters. Arm provides a reference implementation of the driver located under the `/src/driver/sensor` folder.

```
// this structure represents image resolution
// it is used in sensor driver to keep information
// about the frame width and frame height
typedef struct _image_resolution_t {
    uint16_t width;
    uint16_t height ;
} image_resolution_t ;


// a sensor can support several different predefined modes.
// this structure keeps all neccessary information about a mode
typedef struct _sensor_mode_t {
    uint8_t wdr_mode ;                  // The wdr mode.
    uint32_t fps ;                      // Fps value multiplied by 256
    image_resolution_t resolution ;     // Resolution of the mode
    uint8_t exposures ;                 // How many exposures this mode supports
} sensor_mode_t ;


// sensor parameters structure keeps information about the current
// sensor state.
typedef struct _sensor_param_t {
    image_resolution_t total ;                // Total resolution of the image with blanking
    image_resolution_t active ;               // Active resolution without blanking
    uint16_t pixels_per_line;                 // Actual pixels per line after scaling/binning
    int32_t again_log2_max ;                  // Maximum analog gain value in log2 format
    int32_t dgain_log2_max ;                  // Maximum digital gain value in log2 format
    int32_t again_accuracy ;                  // Precision of the gain - If required gain step is
less then this do not try to allocate it
    uint32_t integration_time_min ;           // Minimum integration time for the sensor in lines
```

```
    uint32_t integration_time_max ;           // Maximum integration time for the sensor in lines
without dropping fps
    uint32_t integration_time_long_max ;      // Maximum integration time for long
    uint32_t integration_time_limit ;         // Maximum possible integration time for the sensor
    uint16_t day_light_integration_time_max ; // Limit of integration time for non-flickering light
source
    uint8_t integration_time_apply_delay ;    // Delay to apply integration time in frames
    uint8_t isp_exposure_channel_delay ;      // Select which WDR exposure channel gain is delayed
0-none, 1-long, 2-medium, 3-short (only 0 and 1 implemented)
    int32_t xoffset ;                         // Used for image stabilization
    int32_t yoffset ;                         // Used for image stabilization
    uint32_t lines_per_second ;               // Number of lines per second used for antiflicker
    int32_t sensor_exp_number ;               // Number of different exposures supported by the
sensor
    sensor_mode_t* modes_table ;              // Table of predefined modes which are supported by
the sensor
    uint32_t modes_num ;                      // The number of predefined modes
    uint8_t mode ;                            // Current mode. This value is from the range [ 0 :
modes_num - 1 ]
    void *sensor_ctx ;                        // Conext to a sensor structure. This structure is not
available to firmware
} sensor_param_t ;


// sensor control structure implements sensor API which is used by firmware
typedef struct _sensor_control_t {
         /**
         *   Allocate analog gain
         *
         *   This function sets the sensor analog gain.
         *   Gain should be just saved here for the future.
         *   The real sensor analog gain update must be implemented in
         *   sensor_update routine.
         *
         *   @param gain - analog gain value in log2 format precision is defined by LOG2_GAIN_SHIFT
         *          ctx - pointer to the sensor context
         *
         *   @return the real analog gain which will be applied
         */
    int32_t ( *alloc_analog_gain )( void* ctx, int32_t gain ) ;


         /**
         *   Allocate digital gain
         *
         *   This function sets the sensor digital gain.
         *   Gain should be just saved here for the future.
         *   The real sensor digital gain update must be implemented in
         *   sensor_update routine.
         *
         *   @param gain - analog gain value in log2 format precision is defined by LOG2_GAIN_SHIFT
         *          ctx - pointer to the sensor context
         *
         *   @return the real digital gain which will be applied
         */
    int32_t ( *alloc_digital_gain )( void* ctx, int32_t gain ) ;


         /**
         *   Allocate integration time
         *
         *   This function sets the sensor integration time.
         *   Integration time should be just saved here for the future.
         *   The real time update must be implemented in
         *   sensor_update routine.
         *
         *   @param int_time - integration time if one exposure is used or short exposure for multi-
```

```
expsoure sensors.
         *          int_time_M - medium integration time if several exposures are supported.
         *          int_time_L - long integration time if several exposures are supported.
         *          ctx - pointer to the sensor context
         *
         */
    void ( *alloc_integration_time )( void* ctx, uint16_t *int_time, uint16_t* int_time_M, uint16_t*
int_time_L ) ;


        /**
         *   Update all sensor parameters
         *
         *   The function is called from IRQ thread in vertical blanking.
         *   All sensor parameters must be updated here.
         *   @param ctx - pointer to the sensor context
         *
         */
    void ( *sensor_update )( void* ctx ) ;


    /**
         *   Set horizontal offset
         *
         *   Set sensor horizontal offset to implement crop funcionality.
         *
         *   @param ctx - pointer to the sensor context
         *
         *   @return amount of offseted pixels
         */
    uint32_t ( *set_xoffset )( void* ctx, uint32_t xoffset ) ;


    /**
         *   Set vertical offset
         *
         *   Set sensor vertical offset to implement crop funcionality.
         *
         *   @param ctx - pointer to the sensor context
         *
         *   @return amount of offseted pixels
         */
    uint32_t ( *set_yoffset )( void* ctx, uint32_t xoffset ) ;


     /**
         *   Set the sensor mode
         *
         *   Sensor can support several modes. This function
         *   is used to switch among them.
         *
         *   @param mode - the new mode to set
         *          ctx - pointer to the sensor context
         *
         */
    void (*set_mode)( void* ctx, uint8_t mode ) ;


     /**
         *   Start sensor data output
         *
         *   This function is called from system to
         *   enable video stream from sensor.
         *
         *   @param ctx - pointer to the sensor context
         *
         */
```

```
void (*start_streaming)( void* ctx ) ;


 /**
     *   Stop sensor data output
     *
     *   This function is called from system to
     *   disable video stream from sensor.
     *
     *   @param ctx - pointer to the sensor context
     *
     */
void (*stop_streaming)( void* ctx ) ;


 /**
     *   Set the new fps mode
     *
     *   This function is used to update sensor fps
     *
     *   @param fps - the new fps to set
     *          ctx - pointer to the sensor context
     *
     */
uint8_t ( *fps_control )( void* ctx, uint8_t fps ) ;


/**
     *   Get sensor id
     *
     *   This function returns sensor id if sensor has this option; if not -1 is returned
     *
     *   @param ctx - pointer to the sensor context
     */
uint16_t ( *get_id )( void* ctx ) ;


    /**
     *   Get sensor parameters
     *
     *   This function returns a pointer to a sensor parameter structure
     *
     *   @param ctx - pointer to the sensor context
     */
const sensor_param_t* ( *get_parameters )( void* ctx ) ;


/**
     *   disable on-sensor isp
     *
     *   @param ctx - pointer to the sensor context
     */
void ( *disable_sensor_isp )( void* ctx ) ;


/**
     *   read on-sensor register
     *
     *   @param ctx - pointer to the sensor context
     *      address - address of register
     */
uint32_t ( *read_sensor_register )( void* ctx, uint32_t address ) ;


/**
     *   write on-sensor register
     *
```

```
         *   @param ctx - pointer to the sensor context
         *      address - address of register
         *         data - data to write to register location
         */
    void ( *write_sensor_register )( void* ctx, uint32_t address, uint32_t data ) ;

} sensor_control_t ;
```

4.3.3 Lens Driver API

The Lens driver should be developed separately to benefit from AF algorithm. The ISP Driver uses the interface defined in the `/inc/api/acamera_lens_driver.h` file to control the lens.

It is a customer's responsibility to implement the lens driver according to the interface specification and guarantee that it works properly.

```
//useful information and state of the lens
typedef struct _lens_param_t {
    uint16_t lens_type;     //lens type which assigns one of the enum type after probing
    uint16_t min_step;      //lens step resolution
    uint16_t next_zoom;     //next assigned zoom if zoom if available
    uint16_t curr_zoom;     //current zoom positon if zoom if available
    uint16_t next_pos;      //lens position

} lens_param_t ;


//lens API implementation
typedef struct _lens_control_t {

    /**
     *   Move lens to the desired postion
     *
     *   @param position - value of which will be assigned to parameter next_pos
     *          ctx - pointer to the lens internal context or data
     *
     */
    void (*move)( void* ctx,uint16_t position ) ;


    /**
     *   Stop lens move
     *
     *   @param ctx - pointer to the lens internal context or data
     *
     */
    void (*stop)( void* ctx ) ;


    /**
     *   Api to check if lens is moving
     *
     *   @param ctx - pointer to the lens internal context or data
     *
     *   @return true if lens is moving
     */
    uint8_t (*is_moving)( void* ctx ) ;


    /**
     *   Get curent position of lens
```

```
     *
     *   @param ctx - pointer to the lens internal context or data
     *
     *   @return lens position
     */
    uint16_t (*get_pos)( void* ctx ) ;


    /**
     *   Write to lens register
     *
     *   @param ctx - pointer to the lens internal context or data
     *      address - address of register
     *         data - data to write to register location
     *
     */
    void (*write_lens_register)( void* ctx, uint32_t address, uint32_t data ) ;


    /**
     *   Read on lens register
     *
     *   @param ctx - pointer to the lens internal context or data
     *      address - address of register
     *
     *   @return the register value
     */
    uint32_t (*read_lens_register)( void* ctx, uint32_t address ) ;


    /**
     *   Get lens parameters
     *
     *   This function returns a pointer to a lens parameter structure
     *
     *   @param ctx - pointer to the lens internal context or data
     */
    const lens_param_t* ( *get_parameters )( void* ctx ) ;


    /**
     *   Move zoom to the next zoom
     *
     *   @param next_zoom - value of which will be assigned to parameter next_zoom
     *          ctx - pointer to the lens internal context or data
     *
     */
    void (*move_zoom)( void* ctx,uint16_t next_zoom ) ;


    /**
     *   Api to check if lens is zooming
     *
     *   @param ctx - pointer to the lens internal context or data
     *
     *   @return true if zooming
     */
    uint8_t (*is_zooming)( void* ctx ) ;

} lens_control_t ;
```

Note: The current implementation of the Auto Focus algorithm does not support zoom lens.

4.3.4 Calibration Files

The ISP Driver algorithms and the ISP pipeline itself use calibration tables declared under `/src/calibration` folder. Customers should update all tables based on their internal requirements but the common rule is that the number of LUTs must stay the same.

The application should provide a pointer to the function

```
uint32_t get_calibrations ( uint32_t ctx_id, ACameraCalibrations* c )
```

when `acamera_init` is called. This pointer will be called by the ISP driver to initialize the structure `ACameraCalibrations`.

A reference implementation can be found in the file:
`/src/calibartions/acamera_get_calibrations.c`

```
uint32_t get_calibrations( uint32_t ctx_id, ACameraCalibrations* c ) {

    uint32_t wdr_mode;
    uint8_t ret=0;


    if(acamera_command( TSENSOR, SENSOR_WDR_MODE, 0,COMMAND_GET ,&wdr_mode) != SUCCESS){
        wdr_mode=ISP_WDR_DEFAULT_MODE;
        LOG( LOG_CRIT, "SENSOR_WDR_MODE command failed: switching to default WDR_MODE_LINEAR " ) ;
    }

    //logic which calibration to apply
   switch(wdr_mode)
    {
        case WDR_MODE_LINEAR:
            LOG( LOG_INFO, "calibration switching to WDR_MODE_LINEAR %d ", (int)wdr_mode ) ;
            ret += (get_calibrations_dynamic_linear(c)+get_calibrations_static_linear(c));
        break;
        case WDR_MODE_NATIVE:
            LOG( LOG_INFO, "calibration switching to WDR_MODE_NATIVE %d ", (int)wdr_mode ) ;
            //ret += (get_calibrations_dynamic_wdr(c)+get_calibrations_static_wdr(c));
        break;
        case WDR_MODE_FS_LIN:
            LOG( LOG_INFO, "calibration switching to WDR mode on mode %d ", (int)wdr_mode ) ;
            ret += (get_calibrations_dynamic_fs_lin(c)+get_calibrations_static_fs_lin(c));
        break;
        default:
            LOG( LOG_INFO, "calibration switching to WDR_MODE_LINEAR %d ", (int)wdr_mode ) ;
            ret += (get_calibrations_dynamic_linear(c)+get_calibrations_static_linear(c));
        break;
    }

    return ret;
}
```

The function `get_calibrations` initializes the input structure based on the current sensor mode. For example, we can have different IQ sets for HDR and Linear mode of operation. It is possible to support as many cases as customers want by implementing additional logic in the function.

4.3.5 ISP register access

Register access for bare metal platforms works in a similar way as that for the ISP V4L2 Driver for Linux based system. Please refer to the ISP Register access section for details.

4.3.6 Calibration switch logic

Calibration switch logic is implemented in the same way as that for the ISP V4L2 Driver. Please refer to the Calibration switch logic section for details.

4.3.7 Internal event processing

The event flow is implemented in the similar way as that for the Linux version of the driver. The key difference is that the bare-metal implementation keeps all algorithms inside the source code tree. This simplifies the usage of the event flow because the communication with external 3A Library is not required.

The typical event flow for AE/AF/AWB and other algorithms is the following:

1. The FSM Manager calls `proc_interrupt` routine for every FSM.
2. If the event type is `FRAME_END` then the FSMs read the statistic information into internal buffers.
3. When statistic reading has finished, a new event is raised by the FSM. Usually it is `fsm_name_stats_ready`.
4. The Event Manager puts the new event into the FIFO for future processing.
5. When the event is ready to be processed it is sent to the FSM.
6. The FSM uses the statistic to calculate its output parameters.
7. When new parameters have been calculated the FSM generates a new notification event to indicate this to all modules.

4.3.8 Main application

The reference application source code for ISP driver is as follows:

```
// The firmware supports multicontext.
    // It means that the customer can use the same firmware for controlling
    // several instances of different sensors/isp. To initialise a context
    // the structure acamera_settings must be filled properly.
    // the total number of initialized context must not exceed FIRMWARE_CONTEXT_NUMBER
    // all contexts are numerated from 0 till ctx_number - 1
        result = acamera_init( settings, FIRMWARE_CONTEXT_NUMBER );

    if ( result == 0 ) {
        uint32_t rc = 0;
        uint32_t ctx_num;
        uint32_t prev_ctx_num = 0;

        application_command(TGENERAL, ACTIVE_CONTEXT, 0, COMMAND_GET, &prev_ctx_num);
```

```
        // set the interrupt handler. The system must call this interrupt_handler
        // function whenever the ISP interrupt happens.
        // This interrupt handling procedure is only advisable and is used in ACamera demo
application.
        // It can be changed by a customer discretion.
        system_interrupt_set_handler( interrupt_handler, NULL ) ;

        // start streaming for sensors
        for(ctx_num = 0; ctx_num < FIRMWARE_CONTEXT_NUMBER; ctx_num++) {
          application_command(TGENERAL, ACTIVE_CONTEXT, ctx_num, COMMAND_SET, &rc);
          application_command(TSENSOR, SENSOR_STREAMING, ON, COMMAND_SET, &rc);
        }

        application_command(TGENERAL, ACTIVE_CONTEXT, prev_ctx_num, COMMAND_SET, &rc);

        // acamera_process function must be called on every incoming interrupt
        // to give the firmware the possibility to apply
        // all internal algorithms and change the ISP state.
        // The external application can be run in the same loop on bare metal systems.
        while ( acamera_main_loop_active )
        {
            // acamera_process must be called for each initialised context
            acamera_process() ;
            #if ISP_HAS_STREAM_CONNECTION && !CONNECTION_IN_THREAD
            // acamera_connection_process is used for communication between
            // firmware and ACT through different possible channels like
            // cmd_queue memory in ISP, socket, UART, chardev etc.
            // Different channels can be supported depending on the target
            // platform. The common case when cmd_queue buffer is used
            // (see acamera_isp_config.h )
            acamera_connection_process();
            #endif
        }
    } else {
         LOG(LOG_ERR,"Failed to start firmware processing thread. " ) ;
    }

    // this api function will free
    // all resources allocated by the firmware
    acamera_terminate() ;
```

4.4 Porting to the target platform

Porting is the final step that you must perform before you can start using the ISP software driver.

Note: It is important to port the software to your target platform for it to work.

4.4.1 BSP layer

Some system interfaces are mandatory and some are optional. The mandatory interfaces are used everywhere across the ISP Software core and cannot be omitted. They must be implemented and verified properly on the target platform to guarantee the overall software stability.

The optional interfaces are used across the code which is provided as a reference implementation for some parts. These parts, for example, sensor driver or interrupt routines handling, are likely to be changed on a customer side. . The ISP Software is not connected directly with such interfaces and can work in a normal way without them.

Note: Optional interfaces are used only for the Arm reference platform and can be omitted on other systems.

4.4.1.1 Mandatory interfaces

The following functions should be correctly implemented on a target platform to enable the bare-metal version of the IV009 software driver to access the ISP configuration space:

- `system_isp_read_32` – read 32 bits from ISP configuration space by a given offset.
- `system_isp_write_32` – write 32 bits data by a given offset.
- `system_memset` – analogue of libc memset function.
- `system_memcpy` – analogue of libc memcpy function

4.4.1.2 Interrupts

The ISP provides four interrupt outputs. These allow external software to be synchronized with the ISP processing state. Internally, 47 event sources are monitored and each can be assigned to one of four output lines. The four lines enable you to group interrupts according to level of priority or some other logical relationship.

Interrupts operate on the control interface clock and are level triggered. They are cleared by writing to the interrupt controller status register.

4.4.2 Sensor Driver

The Sensor driver should be implemented based on the interface in the `Dummy_drv.c` file. The `dummy_drv.c` file only provides the interface; the reference system has no implementation of sensor driver. Customers must implement the sensor driver for the target platform.

The firmware issues the sequence of calls `alloc_integration_time`, `alloc_digital_gain`, `alloc_analog_gain`, `sensor_update` on every frame. The sensor driver is responsible to return the actual value for integration time and gains which the sensor can accept.

The firmware will use the returned parameters to recalculate the amount of each gain and guarantee that the full exposure is applied.

4.4.3 Lens driver

The Lens driver should be implemented based on the interface in the `null_vcm.h` and `null_cvm.c` files. These files only provide the interface. The reference system has no implementation of the lens driver and customers must implement the lens driver for the target platform.

The lens driver sequence is as follows:

1. The main driver initializes the lens device by calling the init function.
2. It gets the information about the lens characteristics such as minimum moving step, lens type, and so on.
3. After this is done the firmware calls the `lens_ctrl.move` interface to move the lens to the desired position when needed.

The Lens Driver is an optional driver. To disable this driver customers can change the `.lens_init` and `.lens_deinit` data members of `struct acamera_settings` in the `runtime_initialization_settings.h` to NULL.

4.4.4 Calibration files

All calibration files are built into the firmware and stored as binary data. The calibration LUT data must be tuned based on the sensor and customer requirements.

5 Command API

The driver supports API commands to change the software and hardware behaviour. It includes commands to control Auto Exposure, Auto Focus, AWB algorithms, modulation parameters, and other parameters.

These commands can help to change the calibration LUTs in real-time with the `acamera_api_calibration()` routine.

This section provides the full list of supported commands with a brief explanation of each tab of the API commands.

5.1 TSYSTEM

The TSYSTEM API commands are used to control the system level behaviors, such as:

- the system dynamical logger levels (refer to the section Log system for details).
- system temper mode.
- system exposure time apply control (manual integration time, manual ISP digital gain, manual sensor analog gain, and so on.).
- system calibration update status.

Name	Type	Description
`system_logger_level`	SET/GET	Control the log level of current running firmware.
`system_logger_mask`	SET/GET	Control the log mask of current running firmware.
`buffer_data_type`	GET	Returns calibration item description information (width | rows | cols).
`test_pattern`	SET/GET	Get/Set test pattern type.
`test_pattern_enable`	SET/GET	Enable/Disable test pattern.
`temper_mode`	SET/GET	Switch between temper2 and temper3 mode.
`system_freeze_firmware`	SET/GET	Freeze/Unfreeze firmware.
`system_manual_exposure`	SET/GET	Enable/Disable manual exposure.
`system_manual_integration_time`	SET/GET	Enable/Disable manual integration time.
`system_manual_max_integration_time`	SET/GET	Enable/Disable manual max integration time.
`system_manual_sensor_analog_gain`	SET/GET	Enable/Disable manual sensor analog gain.
`system_manual_sensor_digital_gain`	SET/GET	Enable/Disable manual sensor digital gain.
`system_manual_isp_digital_gain`	SET/GET	Enable/Disable manual ISP digital gain.

Name	Type	Description
`system_manual_exposure_ratio`	SET/GET	Enable/Disable manual exposure ratio.
`system_max_exposure_ratio`	SET/GET	Get/Set max exposure ratio.
`system_exposure`	SET/GET	Control the AE exposure value.
`system_integration_time`	SET/GET	Control the AE integration time.
`system_exposure_ratio`	SET/GET	Control the AE exposure ratio.
`system_max_integration_time`	SET/GET	Get/Set max integration time.
`system_sensor_analog_gain`	SET/GET	Get/Set sensor analog gain.
`system_max_sensor_analog_gain`	SET/GET	Get/Set max sensor analog gain.
`system_sensor_digital_gain`	SET/GET	Get/Set sensor digital gain.
`system_max_sensor_digital_gain`	SET/GET	Get/Set max digital gain.
`system_isp_digital_gain`	SET/GET	Control the ISP digital gain.
`system_max_isp_digital_gain`	SET/GET	Get/Set max ISP digital gain.
`system_short_integration_time`	GET	Get the `short_integration_time` parameter.
`system_long_integration_time`	GET	Get the `long_integration_time` parameter when HDR mode is active.
`system_antiflicker_enable`	SET/GET	Enable/Disable anti flicker.
`system_anti_flicker_frequency`	SET/GET	Get/Set anti flicker frequency.
`system_exposure_priority`	SET/GET	System exposure priority 0: means frame rate is constant 1: frame rate could change.
`system_manual_awb`	SET/GET	Enable/Disable manual white balance.
`system_awb_red_gain`	SET/GET	Control the AWB red gain.
`system_awb_blue_gain`	SET/GET	Control the AWB blue gain.
`system_manual_saturation`	SET/GET	Enable/Disable manual saturation.
`calibration_update`	SET/GET	Notify calibration update when IQ subdev inserted.

5.2 TISP_MODULES

The TISP_MODULES API commands are used to control the manual mode of some hardware blocks. The driver does not update any registers of the hardware in the manual mode. These API commands can be used to debug the hardware blocks manually.

Name	Type	Description
`isp_modules_manual_frame_stitch`	SET/GET	Enable/Disable `frame_stitch` manual mode.
`isp_modules_manual_raw_frontend`	SET/GET	Enable/Disable `raw_frontend` manual mode.
`isp_modules_manual_sinter`	SET/GET	Enable/Disable sinter manual mode.
`isp_modules_manual_temper`	SET/GET	Enable/Disable temper manual mode.

Name	Type	Description
`isp_modules_manual_auto_level`	SET/GET	Enable/Disable auto level manual mode.
`isp_modules_manual_black_level`	SET/GET	Enable/Disable `black_level` manual mode.
`isp_modules_manual_shading`	SET/GET	Enable/Disable shading manual mode.
`isp_modules_manual_demosaic`	SET/GET	Enable/Disable demosaic manual mode.
`isp_modules_manual_cnr`	SET/GET	Enable/Disable cnr manual mode.
`isp_modules_manual_sharpen`	SET/GET	Enable/Disable sharpen manual mode.
`isp_modules_manual_iridix`	SET/GET	Enable/Disable iridix manual mode.

5.3 TALGORITHMS

The TALGORITHMS API commands are used to control the algorithm behaviors, such as algorithm mode, algorithm internal parameters control. It supports the control of algorithm of AE, AWB and AF.

Name	Type	Description
`ae_mode`	SET/GET	Control the AE mode: Auto/Manual.
`ae_split_preset`	SET/GET	Control the strategy when split exposure value (BALANCED or INTEGRATION_PRIORITY) .
`ae_gain`	SET/GET	Control the total gain.
`ae_exposure`	SET/GET	Control the integration time in ms of Exp1.
`ae_roi`	SET/GET	AE exposure ROI setting.
`ae_compensation`	SET/GET	Adjust AE compensation to an under/over-expose image.
`awb_mode`	SET/GET	Select an AWB mode (AUTO, MANUAL, INCANDESCENT, FLOURESCENT, DAY_LIGHT, CLOUDY).
`awb_temperature`	GET	Returns the current color temperature being used by the AWB algorithm, in kelvin [K], divided by 100.
`awb_light_source`	GET	Returns the current light source candidate and p_high being used by the AWB algorithm.
`af_mode`	SET/GET	Sets the mode of operation for the AF algorithm(Single/CAF/Manual).
`af_manual_control`	SET/GET	Manually set the focal length, only available when AF_MODE_ID is set to AF_MANUAL.

Name	Type	Description
af_range_low	SET/GET	Sets the lowest value that the AF algorithm can select.
af_range_high	SET/GET	Sets the highest value that the AF algorithm can select.
af_roi	SET/GET	Select ROI which is used to gather AF statistics.
af_lens_status	GET	Get AF lens driver initialized status(SUCCESS/FAILED).
antiflicker_mode	SET/GET	Set the frequency for the anti-flicker to remove.

5.4 TIMAGE

The TIMAGE API commands are used to control the properties of output image, such as image orientation, image size, image format, and so on.

Name	Type	Description
orientation_hflip	SET/GET	Horizontally flip the output image.
orientation_vflip	SET/GET	Vertically flip the output image.
fr_format_base_plane	SET/GET	Select the FR pipeline output mode of the ISP(RGB/YUV422/YUV420/YUV444).
ds1_format_base_plane	SET/GET	Select the DS pipeline output mode of the ISP(RGB/YUV422/YUV420/YUV444).
dma_reader_output	SET/GET	Set DMA READER output from Full resolution pipe or from downscaler.
image_resize_enable	SET/GET	Enables or disables corresponding crop or down-scaler.
image_resize_height	SET/GET	Set the height of the image selected by IMAGE_CROP.
image_resize_width	SET/GET	Set the width of the image selected by IMAGE_CROP.
image_resize_type	SET/GET	Sets the type of resize corresponding crop or down-scaler.
image_crop_xoffset	SET/GET	Set the x-offset of the image selected by IMAGE_CROP.
image_crop_yoffset	SET/GET	Set the y-offset of the image selected by IMAGE_CROP.

5.5 TSCENE_MODES

The TSCENE_MODES API commands are used to control the color of image, such as hue, saturation, contrast and sharpening.

Name	Type	Description
`color_mode`	SET/GET	Select the color mode of the ISP: BLACK_AND_WHITE or NORMAL.
`hue_theta`	SET/GET	Control the hue.
`saturation_strength`	SET/GET	Control the exact saturation strength.
`brightness_strength`	SET/GET	Control the exact brightness value.
`contrast_strength`	SET/GET	Control the exact contrast value.
`sharpening_strength`	SET/GET	Control the exact sharpening value.

5.6 TGENERAL

The TGENERAL API commands are used to control the current active context, the current release only supports one context.

Name	Type	Description
`general_context_number`	GET	Get total context numbers.
`general_active_context`	SET/GET	Get/Set current active API context.

5.7 TREGISTERS

The TREGISTERS API commands are used to control the registers manually, it supports sensor, lens and ISP registers access.

Name	Type	Description
`register_address`	SET/GET	Get/Set the registers address.
`register_size`	SET/GET	Get/Set the size of the register in bits (8/16/32).
`register_source`	SET/GET	Get/Set the register source(Sensor/Lens/ISP).
`register_value`	SET/GET	Get/Set the register value.

5.8 TSENSOR

The TSENSOR API commands are used to control the sensor mode via predefined preset, it also supports query sensor mode via command `sensor_info_preset`.

Name	Type	Description
`sensor_supported_presets`	GET	Get the number of sensor supported preset modes.
`sensor_streaming`	SET/GET	Get/Set sensor streaming status(ON/OFF).
`sensor_preset`	SET/GET	Get/Set current sensor preset.
`sensor_wdr_mode`	GET	Get the wdr_mode(Linear/FS_Linear) of current sensor mode.
`sensor_width`	GET	Get the width of current sensor mode.

Name	Type	Description
sensor_height	GET	Get the height of current sensor mode.
sensor_fps	GET	Get the FPS of current sensor mode.
sensor_exposure	GET	Get the number of exposure (1/2/3/4) of current sensor mode.
sensor_info_preset	SET/GET	Get/Set query sensor mode.
sensor_info_wdr_mode	GET	Get the wdr_mode(Linear/FS_Linear) of queried sensor mode.
sensor_info_width	GET	Get the width of queried sensor mode.
sensor_info_height	GET	Get the height of queried sensor mode.
sensor_info_fps	GET	Get the FPS of queried sensor mode.
sensor_info_exposures	GET	Get the number of exposure (1/2/3/4) of queried sensor mode.
sensor_info_preset	SET/GET	Get/Set query sensor mode.
sensor_info_wdr_mode	GET	Get the wdr_mode(Linear/FS_Linear) of queried sensor mode.
sensor_info_width	GET	Get the width of queried sensor mode.
sensor_info_height	GET	Get the height of queried sensor mode.
sensor_info_fps	GET	Get the FPS of queried sensor mode.
sensor_info_exposures	GET	Get the number of exposure (1/2/3/4) of queried sensor mode.

5.9 TSTATUS

The TSTATUS API commands are used to show some important information in the driver, such as current total gain, current exposure, and so on.

Name	Type	Description
status_info_gain_log2	GET	Get current total gain in log2 format.
status_info_gain_ones	GET	Get current total gain.
status_info_exposure_log2	GET	Get current exposure value in log2 format.
status_info_awb_mix_light_contrast	GET	Get current AWB mix light contrast.
status_info_af_lens_pos	GET	Get current AF real lens position.
status_info_af_focus_value	GET	Get current AF focus sharpness value.

5.10 TSELFTEST

The TSELFTEST API command is used to support a self-test feature in the driver.

Name	Type	Description
selftest_fw_revision	GET	GET firmware revision for self-test.

6 Calibration tables

6.1 Static calibrations

Static calibrations are the LUTs generated by the Arm Calibration Tool. They are mainly automatically calculated based on the RAW images captured from the sensor under tuning.

The short description of static calibration tables is given below. Please refer to the Arm Calibration Tool User Guide for details.

Name	Description
CALIBRATION_LIGHT_SRC	Calibration RG BG values for extra light sources. Note: CWF lighting should be set as the first entire.
CALIBRATION_RG_POS	The Red-Green position values.
CALIBRATION_BG_POS	The Blue-Green position values.
CALIBRATION_MESH_RGBG_WEIGHT	Calibration LUT for AWB weighting.
CALIBRATION_MESH_LS_WEIGHT	Calibration LUT for extra light source weighting set in CALIBRATION_LIGHT_SRC.
CALIBRATION_MESH_COLOR_TEMPERATURE	WB Calibration values for color temperature estimations.
CALIBRATION_WB_STRENGTH	White Balance gain adjuster strength value for a sky scene for the RG, BG channels.
CALIBRATION_SKY_LUX_TH	The lux threshold value for a sky scene
CALIBRATION_CT_RG_POS_CALC	LUT containing R:G calibration points of light sources used for calibration.
CALIBRATION_CT_BG_POS_CALC	LUT containing B:G calibration points of light sources used for calibration.
CALIBRATION_COLOR_TEMP	A table of values to set the temperature of particular light points set.
CALIBRATION_CT65POS	The position in color_temp closest to 1e6/6500 in CALIBRATION_COLOR_TEMP.
CALIBRATION_CT40POS	The position in color_temp closest to 1e6/4000 in CALIBRATION_COLOR_TEMP.
CALIBRATION_CT30POS	The position in color_temp closest to 1e6/3000 in CALIBRATION_COLOR_TEMP.
CALIBRATION_EVTOLUX_EV_LUT	Exposure values LUT corresponding to the lux values.
CALIBRATION_EVTOLUX_LUX_LUT	Lux values LUT corresponding to the EV values.
CALIBRATION_BLACK_LEVEL_R	Black level value of the Red (R) channel.
CALIBRATION_BLACK_LEVEL_GR	Black level value of the Green (GR) channel.
CALIBRATION_BLACK_LEVEL_GB	Black level value of the Green (GB) channel.
CALIBRATION_BLACK_LEVEL_B	Black level value of the Green (GR) channel.
CALIBRATION_STATIC_WB	The calibration values for static white.

Name	Description
CALIBRATION_MT_ABSOLUTE_LS_A_CCM	Calibration values for Color Correction Matrix under A lighting.
CALIBRATION_MT_ABSOLUTE_LS_D40_CCM	Calibration values for Color Correction Matrix under D40 lighting.
CALIBRATION_MT_ABSOLUTE_LS_D50_CCM	Calibration values for Color Correction Matrix under D50 lighting.
CALIBRATION_SHADING_LS_A_R	Shading value for red channel under A lighting conditions.
CALIBRATION_SHADING_LS_A_G	Shading value for green channel under A lighting conditions.
CALIBRATION_SHADING_LS_A_B	Shading value for blue channel under A lighting conditions.
CALIBRATION_SHADING_LS_TL84_R	Shading value for red channel under TL84 lighting conditions.
CALIBRATION_SHADING_LS_TL84_G	Shading value for green channel under TL84 lighting conditions.
CALIBRATION_SHADING_LS_TL84_B	Shading value for blue channel under TL84 lighting conditions.
CALIBRATION_SHADING_LS_D65_R	Shading value for red channel under D65 lighting conditions.
CALIBRATION_SHADING_LS_D65_G	Shading value for green channel under D65 lighting conditions.
CALIBRATION_SHADING_LS_D65_B	Shading value for blue channel under D65 lighting conditions.
CALIBRATION_AWB_WARMING_LS_A	Calibration values for auto white balance under A lighting conditions.
CALIBRATION_AWB_WARMING_LS_D50	Calibration values for auto white balance under D65 lighting conditions.
CALIBRATION_AWB_WARMING_LS_D75	Calibration values for auto white balance under D75 lighting conditions.
CALIBRATION_NOISE_PROFILE	The lookup table for noise profile calibrations.
CALIBRATION_DEMOSAIC	The lookup table for demosaic calibrations
CALIBRATION_GAMMA	The lookup table for gamma calibrations.
CALIBRATION_IRIDIX_ASYMMETRY	The lookup table for an Iridix parameter.
CALIBRATION_AWB_SCENE_PRESETS	The auto white balance scene type pre-sets for different foreseeable light scene conditions.
CALIBRATION_WDR_NP_LUT	Noise profile LUT used in WDR frame stitching.
CALIBRATION_CA_FILTER_MEM	CAC filter memory.
CALIBRATION_CA_CORRECTION_MEM	CAC mesh memory.
CALIBRATION_LUT3D_MEM	LUT3D memory.
CALIBRATION_DECOMPANDER0_MEM	The lookup table for Decompander0.
CALIBRATION_DECOMPANDER1_MEM	The lookup table for Decompander1.
CALIBRATION_SHADING_RADIAL_R	The lookup table for Radial Shading R Channel.
CALIBRATION_SHADING_RADIAL_G	The lookup table for Radial Shading G Channel.

Name	Description
CALIBRATION_SHADING_RADIAL_B	The lookup table for Radial Shading B Channel.

Table 8. Static calibrations

6.2 Dynamic calibrations

Dynamic calibrations are the LUTs manually generated by IQ engineers.

The short description of dynamic calibration tables is given below. Please refer to the Arm Calibration Tool User Guide for details.

Name	Description
CALIBRATION_STITCHING_LM_MED_NOISE_INTENSITY	Modulates motion intensity to be denoised. Only in 3:1 mode.
AWB_COLOUR_PREFERENCE	Calibration values for auto white balance color preference CCT.
CALIBRATION_AWB_MIX_LIGHT_PARAMETERS	AWB mix light parameter.
CALIBRATION_PF_RADIAL_LUT	Purple fringe radial weight.
CALIBRATION_PF_RADIAL_PARAMS	x center, y center, rm_off_centre_mult.
CALIBRATION_SINTER_RADIAL_LUT	The lookup table containing Radial Sinter values.
CALIBRATION_SINTER_RADIAL_PARAMS	The lookup table containing the values for Radial Sinter Parameters.
CALIBRATION_AWB_BG_MAX_GAIN	Maximum AWB bg gain according to total gain.
CALIBRATION_IRIDIX8_STRENGTH_DK_ENH_CONTROL	Iridix8 strength and dark enhancement control: [0] - dark_prc [1] - bright_prc [2] - min_dk: minimum dark enhancement [3] - max_dk: maximum dark enhancement [4] - pD_cut_min: minimum intensity cut for dark regions in which dk_enh will be applied [5] - pD_cut_max: maximum intensity cut for dark regions in which dk_enh will be applied [6] - dark contrast min [7] - dark contrast max [8] - min_str: iridix strength in percentage [9] - max_str: iridix strength in percentage: 50 = 1x gain. 100 = 2x gain [10] - dark_prc_gain_target: target in histogram (percentage) for dark_prc after iridix is applied [11] - contrast_min: clip factor of strength for LDR scenes.

Name	Description
	[12] - contrast_max: clip factor of strength for HDR scenes. [13] - max iridix gain [14] - print debug
CALIBRATION_CMOS_CONTROL	Parameters to control CMOS in manual mode.
CALIBRATION_STATUS_INFO	Firmware status information.
CALIBRATION_AE_BALANCED_CONTRAST_ADJUSTMENT	AE contrast adjustment parameters.
CALIBRATION_AUTO_LEVEL_CONTROL	Adaptive gamma controls: hist_target dark_prc min_gain max_gain enable/disable.
CALIBRATION_DP_SLOPE	The value of the Defect Pixel Slope.
CALIBRATION_DP_THRESHOLD	The value of the Defect Pixel Threshold.
CALIBRATION_STITCHING_LM_MOV_MULT	The gradient of the motion detection alpha ramp.
CALIBRATION_STITCHING_LM_NP	The value by which the noise profile is multiplied by to yield the expected noise amplitude.
CALIBRATION_STITCHING_MS_MOV_MULT	The gradient of the motion detection alpha ramp.
CALIBRATION_STITCHING_MS_NP	The value by which the noise profile is multiplied to yield the expected noise amplitude.
CALIBRATION_EVTOLUX_PROBABILITY_ENABLE	The value to enable/ disable the use of lux probability in AWB calculations.
CALIBRATION_AWB_AVG_COEF	The average coefficient value for Auto White Balance.
CALIBRATION_IRIDIX_AVG_COEF	The average coefficient value for Iridix.
CALIBRATION_IRIDIX_STRENGTH_MAXIMUM	The Maximum Strength of Iridix.
CALIBRATION_IRIDIX_MIN_MAX_STR	The minimum value of strength of Iridix to be applied, value taken from within the range of [0:255].
CALIBRATION_IRIDIX_EV_LIM_FULL_STR	The EV minimum value, in terms of EV_log2.
CALIBRATION_IRIDIX_EV_LIM_NO_STR	The EV maximum value, in terms of EV_log2.
CALIBRATION_AE_CORRECTION	Calibration values for the fine-tuning parameter, which alters the strength of ae_comp.
CALIBRATION_AE_EXPOSURE_CORRECTION	Calibration values for the fine-tuning parameter, which sets the exposure value nodes.
CALIBRATION_SINTER_STRENGTH	The lookup table for Sinter Strength calibrations.
CALIBRATION_SINTER_STRENGTH1	The lookup table for Sinter Strength1 calibrations.

Name	Description
CALIBRATION_SINTER_THRESH1	The lookup table for Sinter Thresh scale 1 calibrations.
CALIBRATION_SINTER_THRESH4	The lookup table for Sinter Thresh scale 4 calibrations.
CALIBRATION_SINTER_INTCONFIG	Modulates intensity - raw blending for sinter noise reduction.
CALIBRATION_SHARP_ALT_D	The lookup table for Directional sharpening calibrations.
CALIBRATION_SHARP_ALT_UD	The lookup table for Un-Directional sharpening calibrations.
CALIBRATION_SHARP_ALT_DU	The lookup table for Un-Directional sharpening calibrations.
CALIBRATION_DEMOSAIC_NP_OFFSET	The lookup table for Demosaic NP Offset.
CALIBRATION_MESH_SHADING_STRENGTH	The lookup table for Mesh Shading Strength.
CALIBRATION_SATURATION_STRENGTH	The lookup table for Saturation Strength.
CALIBRATION_CCM_ONE_GAIN_THRESHOLD	The threshold value of the Color Correction Matrix.
CALIBRATION_AE_CONTROL	Auto exposure control. [0] - AE convergence [1] - LDR AE target: this should match the 18% grey of the output gamma [2] - AE tail weight [3] - WDR mode only: Max percentage of clipped pixels for long exposure: WDR mode only: 256 = 100% clipped pixels [4] - WDR mode only: Time filter for exposure ratio [5] - control for clipping: bright percentage of pixels that should be below hi_target_prc [6] - control for clipping: highlights percentage (hi_target_prc): target for tail of histogram [7] - 1:0 enable | disable iridix global gain.
CALIBRATION_AE_CONTROL_HDR_TARGET	AE HDR target control. This target is modulated by total gain.
CALIBRATION_RGB2YUV_CONVERSION	YUV conversion matrix and offset.
CALIBRATION_AE_ZONE_WGHT_HOR	AE Zone weight horizontal.
CALIBRATION_AE_ZONE_WGHT_VER	AE Zone weight vertical.
CALIBRATION_AWB_ZONE_WGHT_HOR	AWB Zone weight horizontal.
CALIBRATION_AWB_ZONE_WGHT_VER	AWB Zone weight vertical.
CALIBRATION_SHARPEN_FR	The lookup table for Sharpen calibrations in Full Resolution.
CALIBRATION_SHARPEN_DS1	The lookup table for Sharpen calibrations in Downscale.
CALIBRATION_TEMPER_STRENGTH	The lookup table for Temper Strength.

Name	Description
CALIBRATION_SCALER_H_FILTER	Scaler H Filter.
CALIBRATION_SCALER_V_FILTER	Scaler V Filter.
CALIBRATION_SINTER_STRENGTH_MC_CONTRAST	Adjusts sinter global thresh according to contrast of scene calculated in iridix8.
CALIBRATION_EXPOSURE_RATIO_ADJUSTMENT	Adjusts the amount of clipped pixels for long exposure.
CALIBRATION_CNR_UV_DELTA12_SLOPE	The lookup table containing values for the Color Noise Reduction UV delta slope.
CALIBRATION_FS_MC_OFF	no description available.
CALIBRATION_SINTER_SAD	Calibration value to balance the contrast between edge detail and smoothness of flat regions.
CALIBRATION_CUSTOM_SETTINGS_CONTEXT	Custom ISP register software default values for ping/pong context.
CALIBRATION_CMOS_EXPOSURE_PARTITION_LUTS	Partition lookup tables to split exposure value.

Table 9. Dynamic calibrations

7 Control Tool

7.1 Overview

The ISP Driver is released with the implementation of the communication protocol which is used to establish the connection with the Arm Control Tool. The Linux version and Bare-Metal versions of the driver use different data channels to deliver packets from Arm Control Tool to the main driver code.

In both cases the implementation of the protocol is located under the `/app/app/control` folder. The files included there are optional and can be safely removed from the production version of the driver when they are not required anymore.

The communication channel serves the following purposes:

- Provide end-point connection to the Arm Control Tool back-end server
- Enables the access to the ISP registers through the character device.
- Enables the access to the API commands.
- Enables the access to the calibration look-up tables.

The common use case is to enable the protocol during the driver porting stage and to be able to control the internal driver state including the algorithms and the LUTs.

The Arm Control Tool is widely used during the IQ tuning session so it is essential to establish the connection with the back-end server.

7.2 Linux control channel

A control channel for Linux platform is implemented in the `main_firmware.c` file of the main V4L2 ISP device. The channel runs in a separate thread and waits for new request data packets on the `/dev/ac_isp` file.

Note: ***The Control Tool logic inside the ISP driver exposes the character device which is used by the Arm Control Tool server to establish the connection. Customers can remove it from the driver source code if it is not required.***

The following code starts the connection thread:

```
isp_fw_connections_thread = kthread_run(connection_thread, NULL,
"isp_connection");
```

The following function initializes the channel, processes requests and eventually destroys the channel.

```
static int connection_thread(void* foo)
{
  LOG(LOG_CRIT,"connection_thread start");

  acamera_connection_init();

  while (!kthread_should_stop())
  {
    acamera_connection_process();
  }

  acamera_connection_destroy();

  LOG(LOG_CRIT,"connection_thread stop");

  return 0;
}
```

7.3 Bare-Metal control channel

This version of the protocol is similar with the Linux version except:

- It works in the main application thread.
- The data channel is `cmd_queue` 1KB memory inside the ISP.
- The implementation pollss the `cmd_queue` memory constantly to verify if new requests are available.

Note: The Control Tool logic inside the ISP driver uses polling mechanism for communication with the Arm Control Tool server side. Customers can remove this logic to avoid any extra traffic caused by the protocol. But if this logic is removed then it will not be possible to connect to the Control Tool.

7.4 Protocol

The Communication protocol waits for requests and responds with replies. Any transaction (request or reply) has following format:

Offset	Field	Size	Description
0	Size	4	32-bits size of packet in bytes include the header.
4	ID	4	Unique 32-bits value which has to be copied in response. ID value of zero is reserved for asynchronous data packets sent from the server without a dedicated request from the client.
8	Type	2	Type of packet.

Offset	Field	Size	Description
10	Context	1	Application specific context ID.
11	Reserved	1	Reserved.
12	Data	N	Packet payload.

Note: All fields are written in the little-endian format.

The driver must respond to a command with the status and optionally the data.

Supported status values are:

Name	Value	Description
SUCCESS	0	Command completed successfully.
NOT_IMPLEMENTED	1	Requested command is not implemented in the driver.
NOT_SUPPORTED	2	Requested command is not supported in the current mode of operation.
NOT_PERMITTED	3	Not permitted.
NOT_EXISTS	4	Requested parameter does not exist.
FAIL	5	Command failed with some critical error.

For more information refer to the Arm Control Tool User Guide.