# lib_camera: XCORE Camera Library

Publication Date: 2025/6/26
Document Number: XM-015339-UG v2.0.0

XMOS

IN THIS DOCUMENT

### Introduction

Lib Camera is a software library designed to interface camera sensors with xcore devices. It provides a comprehensive set of functionalities, including camera sensor configuration, Image Signal Processing, and image capture. This documentation offers an overview of the library's architecture, usage, and extensibility for custom applications.

With this guide, users can quickly get started capturing images and integrating the camera library with other XMOS libraries to build end-to-end computer vision solutions.

The documentation is organized as follows:

- *Overview*: Introduction to the library's purpose, supported hardware, key features, and high-level architecture.
- *Architecture*: Detailed explanation of the library's components, data flow, and concurrency model.
- *Getting Started*: Step-by-step instructions for setting up the library and running example applications.
- *Configuration*: Guidance on configuring the library for various hardware platforms and use cases.
- *Troubleshooting*: Tips and solutions for common issues encountered when using the library.
- *Contributing*: Information on contributing to the project, including bug reporting, feature requests, and development guidelines.
- *API Reference*: Comprehensive reference for the API functions and data structures.

# 1 Overview

This section provides an overview of the library, including its purpose, supported hardware, key features, and high-level architecture.

## 1.1 Purpose

The XMOS Camera Library is designed to provide a high-level interface for camera operations, including image capture, processing, and configuration. It abstracts the complexities of camera hardware and provides a simple API for developers to work with.

The library is intended to be connected to other XMOS libraries to perform more complex tasks, such as image classification, object tracking, and other computer vision applications. This document provides a series of examples or *Application Notes* to demonstrate how to use the library in conjunction with other XMOS libraries.

This documentation is intended for developers who either want to use the library in their applications or want to customise it to interface their own camera sensor with an XMOS device. A knowledge of camera communication protocols (MIPI, I2C) and image processing is recommended.

## 1.2 Supported Hardware

Lib Camera is designed to work with the following hardware:

- **Processor**: This library is designed to work with the XMOS xcore.ai processor family (*XU316*). It is important to mention that only the packages `265 pin FBGA` and

`128 pin TQFP` include a D-PHY receiver, which is required for MIPI-CSI2 camera connection.

- **Camera Sensors**: This library is designed to work with the IMX219 camera sensor. Other sensors may require modifications to the library. More information on how to add your own sensor can be found in the documentation. See *Configuration* for more information.
- **Camera Interface**: The library uses the `MIPI-CSI2` interface for camera connection and I2C for camera control. The library is designed to work with the D-PHY receiver on the xcore.ai processor family. It is important to note that the library does not support the parallel camera interface (DVP) or the USB camera interface.
- **Boards**: This library works directly with the xcore.ai Vision Development Kit (*XK-EVK-XU316-AIV*). It can be used with other boards that have a MIPI-CSI2 interface and I2C control, such as the xcore.ai Explorer Development Kit (*XK-EVK-XU316*), but some modifications may be required. More information on how to add your own board can be found in the documentation. See *Configuration* for more information.

## 1.3 Key features

- Image Signal Processing (ISP): The library includes ISP capabilities to enhance the image quality and transform raw image data into the requested format by the user. The included functions are demosaicing, downsampling, image scaling and cropping, image rotation, Auto Exposure (AE), and Auto White Balance (AWB).
- Asynchronous Capture Mode: The library supports an asynchronous still mode where the ISP thread handles camera operations in a non-blocking manner. When an external thread or application requests a frame, the ISP thread starts the camera, captures the frame, processes it, and delivers it to the application without halting its execution. This allows the application to continue performing other tasks, such as running an AI model, while the camera is capturing and processing the next frame. Subsequent frame requests can be made immediately after the first, enabling efficient and seamless integration of camera operations into the application workflow.
- Dynamic Region of Interest (ROI): The library supports dynamic ROI selection, allowing the user to select a specific region of the image for processing. This is useful for applications that require only a portion of the image to be processed, such as object detection or tracking.
- RAW8 and RGB888 data format. The library supports the following data formats: RAW8 (RGGB) and RGB888. Refer to Camera Data Formats for more information. Data type is in `int8_t` by default. This can be changed via the `demux` options at compile time, or by converting the data using the `camera_conversion` functions.
- Image rotations: The library supports image rotation in two ways: 90 degrees and 180 degrees. The 90-degree rotation is performed in software, while the 180-degree rotation is performed in hardware by the camera sensor. The library also supports horizontal and vertical flipping of the image.
- Image I/O and conversion operations: The library provides a set of functions to convert the image data from one format to another or write image data to a file. The library supports only binary files or *BMP* files. The conversion functions are designed to be efficient and fast, allowing for real-time image processing.

## 1.4 High-level Architecture

The library is structured into several key components, each responsible for a specific aspect of camera operation. The library itself is designed to only use two threads: the *MIPI Receiver* thread and the *ISP* thread.

The high-level architecture is shown in Fig. 1:

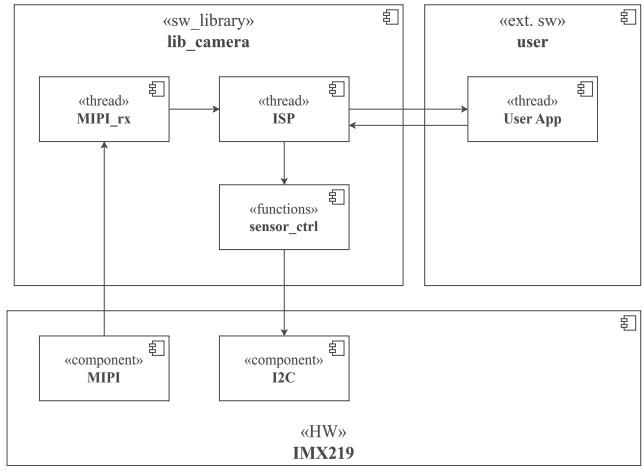


Fig. 1: High-level block diagram of the `lib_camera`.

Note that only the main components are shown in the diagram. Further details about the architecture and components are provided in the *Architecture* section.

- *MIPI Receiver Thread*: This thread is responsible for receiving data from the camera sensor over the MIPI-CSI2 interface. It handles the low-level details of the MIPI protocol and provides data packets to the ISP thread for processing.
- *ISP Thread*: This thread is responsible for handling both MIPI receiver request for incoming data and user requests to deliver the processed data. It also handles initialisation and configuration of the camera sensor.
- *Sensor Control*: Encapsulates a group of functions to control the camera sensor. It handles the I2C communication with the camera sensor and provides a high-level interface for configuring the sensor settings.
- *User Thread*: This thread is not part of the library but is provided as an example of how to use the library. It is responsible for setting the buffer where the image data will be stored and for processing the image data after it has been captured. Image size and properties are user centric, meaning that the user can set the image size and properties according to their needs. The library will then handle the conversion of the image data to the desired format.

## 1.5 Additional Resources

- MIPI CSI-2 specification: MIPI
- XMOS I2C library user Guide: XMOS I2C
- XMOS XC Programming Guide: XMOS XC Programming Guide

- XMOS C Programming Guide: XMOS C Programming Guide
- XMOS XS3A Architecture: XMOS XS3 Architecture
- Sony IMX219 datasheet: IMX219

# 2 Architecture

This section provides a detailed overview of the architecture of the library, including its components, data flow, and concurrency model. It is intended for developers who want to understand how the library works and how to customise it for their own applications.

## 2.1 Components

In this library, components refer to a high-level description of the main parts of the library. The library is composed of several components, each responsible for a specific task. A component could be a group of modules, a class, a function or a thread. The main components of the library are:

Table 1: lib_camera components

| Component | Description |
|---|---|
| Sensor | The sensor refers to the camera hardware and its drivers, located in the `src/sensors` directory. Each sensor is implemented as a class that inherits from a generic base class (`sensor_base.cpp`). For example, `sensor_imx219.cpp` provides support for the IMX219 sensor. A C++ wrapper (`sensor_wrapper.cpp`) allows these classes to be used from C code, as called by `cameras_isp`. |
| Sensor Control | Sensor control covers all I2C operations for configuring the sensor, such as reading or writing registers for pixel format, exposure time, data format, and clock configuration. |
| Sensor Region | The Sensor Region, defined in `camera.h`, is a macro specifying the maximum image size the sensor can capture. This affects the achievable frame rate and selectable region. The maximum region is set by the sensor's active pixel area (e.g., 3280x2464 for IMX219). Downsampling and binning reduce this size. The ISP will reject regions larger than allowed, but the maximum can be adjusted based on user needs and sensor capabilities. |
| MIPI D-PHY SHIM | The MIPI Software Interface Module (SHIM) consist of a MIPI D-PHY receiver and a demultiplexer, which translates MIPI Lanes data into xcore ports. Usually, the MIPI Shims receive two MIPI data lanes, which are translated into 4 xcore ports, active, valid, clock and data. |
| MIPI Receiver | The MIPI receiver is a dedicated software thread that interfaces with the MIPI SHIM ports. It monitors the active and valid signals, collects incoming data, and assembles it into 32-bit words. The receiver continues this process until a complete image line is buffered, at which point it sends and signals the ISP to start processing the line. |
| ISP | In this context, the ISP englobes all the image processing functions, from MIPI packets to a desired output image. It consists of line-by-line processes as well as after the end of frame (EOF) functions. They are located in `src/isp`. |
| User Thread | The user thread or consumer is the thread or application that specifies the image and consumes it. This thread needs to call functions from the ISP that will allow configuring the desired image and taking a picture. This code is also located in `src/isp`. This library provides examples of how the user thread or consumer would look. |

## 2.2 Image Processing Blocks

The library supports two types of ISP components: Line ISP and End-of-Frame (EOF) ISP. The primary difference between them lies in how they process image data.

Line ISP components process data on a line-by-line basis or in groups of lines. This means the component processes each line of the image as it is received from the camera sensor. This approach is typically used when converting from RAW8 input to a desired output format, such as RGB888 or YUV422. Processing line by line is preferred in this case to avoid storing the entire RAW image in memory.

End-of-Frame ISP components, on the other hand, process data after the entire image has been received and processed by the line ISP components. These components handle tasks that do not necessarily require storing the entire image, such as auto-exposure adjustments, or tasks that can be performed in-place in memory, such as int8 to uint8 conversion or gamma correction.

## 2.3 Image Processing Pipeline

The image processing pipeline consists of several stages, including demosaicing, downsampling, image scaling and cropping, image rotation, Auto Exposure (AE), and Auto White Balance (AWB). Each stage is implemented as a separate function in the library, allowing for easy customisation and extension. Fig. 2 illustrates the image processing pipeline:

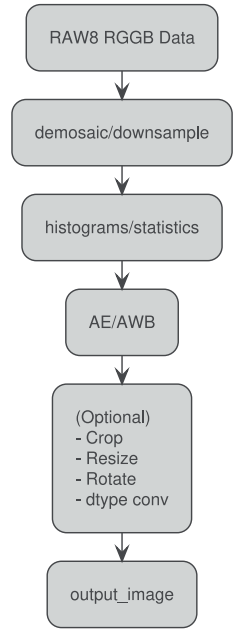


Fig. 2: ISP Pipeline Diagram

This diagram illustrates a basic image processing pipeline. It starts with RAW8 RGGB input data, followed by demosaicing and downsampling. Histograms and statistics are computed to support auto-exposure (AE) and auto white balance (AWB). Optional operations include cropping, resizing, rotating, and data type conversion. The final result is an output image.

## 2.4 Capture Sequence Diagram

Fig. 3 illustrates the interaction between the main components of the library during the image capture process. It shows how the MIPI receiver thread and the ISP thread work together to capture and process an image.

This diagram outlines the sequence of operations in the camera capture pipeline. The main thread starts the ISP and MIPI threads. The user thread prepares the image buffer, image metadata (*image_t*), and configuration (*image_cfg_t*). The ISP thread initialises

the camera sensor via I2C. The user then computes ISP coordinates and begins capture for each frame.

For every frame, the sensor is started, and MIPI sends line packets to the ISP. Based on the packet type (*FRAME_START*, *RAW8*, or *FRAME_END*), the ISP updates counters, processes expected lines, or performs post-processing (statistics, auto-exposure, auto white balance). Once the frame is complete, the sensor is stopped, and the captured image is retrieved.

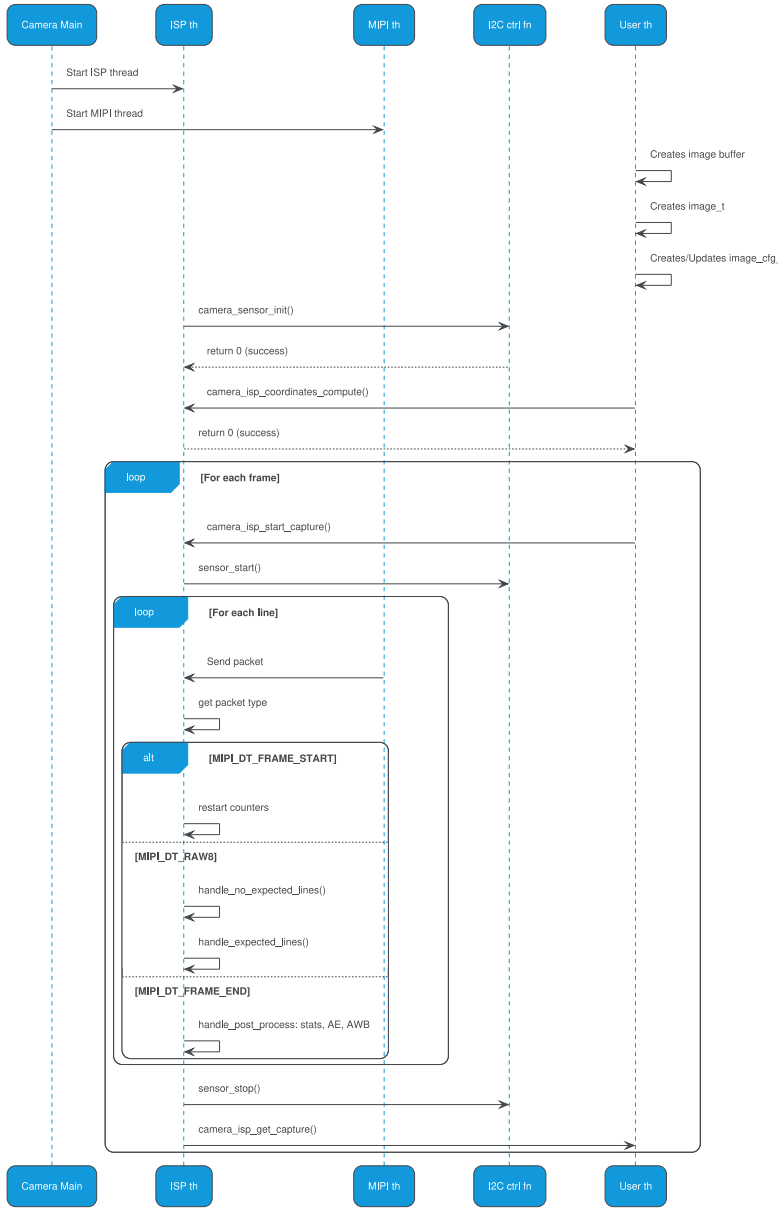


Fig. 3: Capture Sequence Diagram

# 3 Getting Started

This section provides a quick-start guide to help users get up and running with the library. It includes information on hardware and software requirements, installation instructions, and a minimal working example that captures a frame from the camera.

> **Note**
>
> To directly go to the instructions for building and running the example, refer to *Build and Run the Example* section.

## 3.1 RGB Capture Example

The following example demonstrates how to use the library to capture a frame from the camera and save it to a file. The example is located in the `examples` directory of the library source code.

The example consists of two main files:

- `mapfile.xc`: This file contains the entry point to the application (main). On one hand, it initialises the `camera_main` thread. On the other hand, it starts the `user_app` thread. `camera_main` is part of the camera library and is responsible for initialising the camera, configuring the camera settings, and starting the camera capture process. It also handles the MIPI receiver thread and the ISP thread. The `user_app` thread is responsible for processing the image data after it has been captured. Both communicate via the channel `c_cam`. For more information on XMOS channels, refer to the XMOS XC Programming Guide and the C version XMOS C Programming Guide.

```
int main(void)
{
  chan c_cam;

  // Parallel jobs
  par{
    on tile[1]: camera_main(c_cam);
    on tile[1]: user_app(c_cam);
  }
  return 0;
```

- `user_app.c`: This contains the user_app thread, which is responsible for setting the buffer where the image data will be stored and for processing the image data after it has been captured. Image size and properties are user centric, meaning that the user can set the image size and properties according to their needs. The library will then handle the conversion of the image data to the desired format.

```
    // Image and configuration
    const unsigned h = 200;
    const unsigned w = 200;
    const unsigned ch = 3;
    const unsigned img_size = h * w * ch;
    int8_t image_buffer[img_size] = { 0 };

    camera_cfg_t config = {
        .offset_x = 0,
        .offset_y = 0,
        .mode = MODE_RGB2,
    };
    image_cfg_t image = {
        .height = h,
        .width = w,
        .channels = ch,
        .size = h*w*ch,
        .ptr = &image_buffer[0],
        .config = &config
    };

    // set coords and send to ISP
    camera_isp_coordinates_compute(&image);
```

(continues on next page)

(continued from previous page)

```
camera_isp_start_capture(c_cam, &image);
sim_model_invoke(); // this is just some big delay to show that it is non-blocking
camera_isp_get_capture(c_cam);
```

First, the user must define the objects *camera_cfg_t* and *image_cfg_t*. The **camera_cfg_t** struct contains the camera configuration parameters, such as offsets to define the region of interest and the acquisition mode. The **image_cfg_t** struct contains the image data and properties, such as width, height, a pointer to the image buffer, and a pointer to the previously declared configuration. In the example, the user expects an image of 200x200x3 RGB int8 image.

Next, the user must call the *camera_isp_coordinates_compute()* function, which takes a pointer to **image_cfg_t**. This call computes the coordinates of the region of interest (ROI) based on the camera configuration (**camera_cfg_t**) and the capture mode. In this case, it will be captured in RGB format and downsampled by a factor of 2 and starting at position (0,0).

The function *camera_isp_coordinates_compute()* only needs to be called once if the user does not change the image size or properties. It needs to be called again each time image size, format or properties are changed.

Finally, the user must call the *camera_isp_start_capture()* function to start the camera capture process. When the user needs the frame, they can call the *camera_isp_get_capture()* function. This function will block until the frame is ready.

Once the frame is ready, the user can process the image data and save it to a file using the **save_image()** function.

The following sections provide detailed information about the requirements for running the example, as well as step-by-step instructions on how to build and execute it.

## 3.2 Hardware Requirements

The library is designed to work with the following hardware:

- xcore.ai Vision Development Kit (*XK-EVK-XU316-AIV*).
- 1x Micro USB cable.

## 3.3 Hardware Setup

- Plug the 1x Micro USB cable to both the host computer and the **DEBUG** Micro USB port.
- Face the xcore.ai Vision Development Kit horizontally, with the camera connector looking down.

Fig. 4 shows the hardware setup for the example:

## 3.4 Software Requirements

The following software is required to build and run the library:

- XTC tools: 15.3.1 XTC tools.
- Python: 3.10 or later Python.
- CMake: 3.21 or later CMAKE.

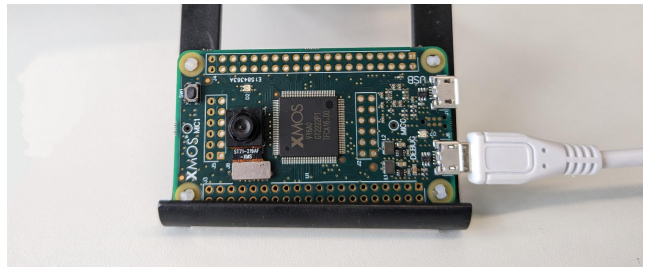

Fig. 4: Hardware Setup for the xcore.ai Vision Development Kit

## 3.5 Software Setup

Before building the example, ensure that the **XTC tools** are installed and properly activated in your development environment (XTC tools). This can be verified by running the following command in a terminal with the XTC tools sourced:

```
xcc --version
```

This command should display the version of the installed **XTC tools**. If the tools are not installed or activated, refer to the installation instructions in the documentation.

Once done, run the following commands from the root of the library:

```
# install python dependencies
pip install -r requirements.txt
# go to example folder
cd examples/capture_rgb
# build
cmake -G "Unix Makefiles" -B build
xmake -C build
```

If the build is successful, the message **[100%] Built target capture_rgb** and the usage report will be displayed:

Table 2: Memory Usage Per Tile

| Tile | Memory Used | Status |
|---|---|---|
| tile[0] | 4236 / 524288 | OKAY |
| tile[1] | 170692 / 524288 | OKAY |

Note that the memory usage is shown in bytes, and results could differ slightly. The maximum memory available for each tile is 524288 bytes. As we can see in this case, the memory usage is well within the limits for both tiles. Usage is about 0.81% of the total memory available for tile 0 and about 32% for tile 1.

### Build and Run the Example

Once the example is built, you can run it using the following command:

```
# run
xrun --xscope bin/capture_rgb.xe
# decode image
python decode.py
```

This will run the example, save the image to a binary file, and then the python script will decode the image and display it.

To go further, uncomment the following lines in the `user_app.c` and re-run the file:

```
    /* (Optional) try something out of bounds
    config.offset_x = 1.8;
    config.offset_y = 1.8;
    camera_isp_coordinates_compute(&image);
    camera_isp_coordinates_print(&image);
    */
```

This will try to apply an offset > 1.0 and will cause an error. The error will be displayed in the terminal, and the program will exit with an error code.

```
xrun: Program received signal ET_ECALL, Application exception.
0x00081ad6 in camera_isp_coordinates_compute (img_cfg=0xe2a2c) at ...camera_isp.c:216
```

The line that raised the error, corresponds to the following:

```
  xassert(cfg->x2 <= SENSOR_WIDTH && "x2");
  xassert(cfg->y2 <= SENSOR_HEIGHT && "y2");
```

As we can see, the error is raised because the offset is greater than the image size. Offset has to be a valid range in float from 0.0 to 1.0.

### 3.6 Related Documentation

The following application notes provide additional information about the library and how to use it with other XMOS libraries:

- AN02005: *Xmos Logo Detector*
- AN02010: *Face Identification on xcore.ai*
- AN02013: *FaceId And Keyword Spotting*

## 4 Configuration

This section provides guidance on configuring the xcore.ai camera library for a variety of hardware platforms and use cases. It covers the main configuration structures, logging options, default settings, and steps required to adapt the library for different boards and sensors.



### 4.1 Logging Options

The library provides a logging mechanism to help developers debug and monitor the camera's operation. The logging options are available via CMake options. The following options are available:

- -DDEBUG_PRINT_ENABLE_CAM_ISP=1 : This option enables debug prints for the ISP thread. It can be used to monitor the status of the ISP and its components.
- -DDEBUG_PRINT_ENABLE_CAM_MIPI=1 : This option enables debug prints for the MIPI thread. It can be used to monitor the status of the MIPI receiver and its components.
- -DCONFIG_APPLY_AE=1: This option enables the application of automatic exposure (AE) settings. It can be set to 0 to disable AE.
- -DCONFIG_APPLY_AWB=1: This option enables the application of automatic white balance (AWB) settings. It can be set to 0 to disable AWB.
- -DLIBXCORE_XASSERT_IS_ASSERT=1 : This option configures the library to use the standard `assert` C standard library instead of `lib_xcore/xassert` for runtime exceptions. Enabling this provides more detailed error reporting, as `assert` outputs information about the error location, while `xassert` does not.

## 4.2 Default Settings

The main default settings can be found in the `camera.h` file. The following settings are available:

```
// High-Level Sensor Configuration
#define SENSOR_WIDTH    800 ///< Sensor width in pixels
#define SENSOR_HEIGHT   800 ///< Sensor height in pixels

#define CONFIG_FLIP         FLIP_NONE ///< Flip mode: FLIP_NONE, FLIP_VERTICAL
#define CONFIG_BINNING      BINNING_ON  ///< Binning mode: BINNING_ON or BINNING_OFF
#define CONFIG_CENTRALISE   CENTRALISE_ON ///< Centralise mode: CENTRALISE_ON or CENTRALISE_OFF

#ifndef CONFIG_APPLY_AWB
#define CONFIG_APPLY_AWB    (1) ///< Apply White Balance: 1 to apply, 0 to skip
#endif

#ifndef CONFIG_APPLY_AE
#define CONFIG_APPLY_AE     (1) ///< Apply Auto Exposure: 1 to apply, 0 to skip
#endif
```

This chooses the default settings and options for the camera library. This configuration will impose some constraints that are suitable for most applications, but can be modified to suit specific needs.

## 4.3 Image Configuration

The primary structure used to configure these options is through *camera_cfg_t* for camera-specific settings and *image_cfg_t* for image-related settings. For more details on these structures, please refer to the API section *API Reference*.

Below is an example configuration:

```
camera_cfg_t config = {
    .offset_x = 0.2,
    .offset_y = 0.3,
    .mode = MODE_RGB2,
};

image_cfg_t image = {
    .height = 192,
    .width = 240,
    .channels = 3,
    .size = h * w * ch,
    .ptr = &image_buffer[0],
    .config = &config
};
```

In this example, the user has set the image size to 192x240 pixels with 3 channels (RGB). The *ptr* field points to the buffer where the image data will be stored. The *config* field points to the camera configuration structure, which contains additional settings such as

offsets and modes. In this case `MODE_RGB2` is selected, meaning that the image will take a region of 384x480 pixels from the sensor area to produce a 192x240 image. The offsets are set to 0.2 and 0.3, which means that the image will start from 20% and 30% of the sensor's maximum area.

## 4.4 Adding Support for the xcore.ai Evaluation Kit

The xcore.ai Explorer Development Kit (*XK-EVK-XU316*) is a development board that can be used with the xcore.ai camera library, it has a compatible FPC-24 connector and a MIPI D-PHY receiver. It can support cameras like the Raspberry Pi camera module v2.1 (IMX219) directly. The main difference is the board layout, in the xcore.ai Explorer Development Kit, I2C and MIPI lines collide in the same tile, so the user will need to adapt the hardware or software to make it work.

Regarding the **Hardware solution**, the user can route free pins on tile[1] to the I2C signals from tile [0] (SCL:X0D37:D16, SDA:X0D38:D17). This allows reuse of the same code as used for the xcore.ai Vision Development Kit. MIPI can be placed in both tiles, with the only restriction that MIPI and USB can't be placed on the same tile. The specific ports and pin assignments for the board can be found in the corresponding XN file and the board's manual. Fig. 5 illustrates how to achieve this:

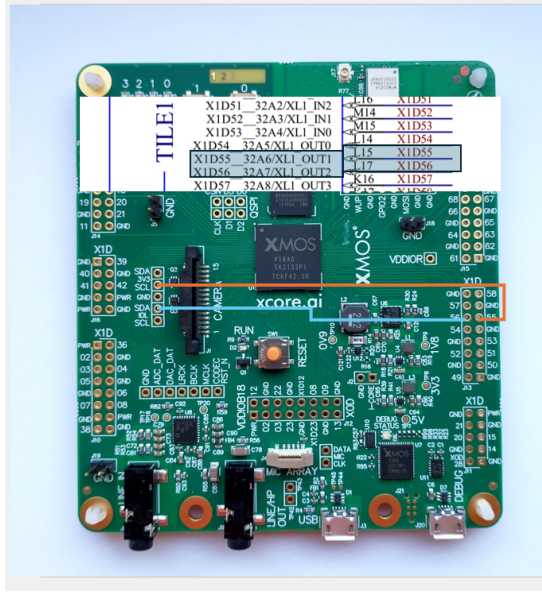


Fig. 5: xcore.ai Explorer Development Kit I2C connections

In this configuration, free pins on tile[1] (X1D56, X1D57) are used to connect the I2C signals from tile[0] (SDA, SCL respectively).

Regarding the **Software solution**, the user will need to adapt the code to work with the xcore.ai Explorer Development Kit. The following sections provide a guide on how to do this. The first change is to adapt the entry point of the program as follows:

```
#define  SCL_BIT_POS  0x1
#define  SCL_BIT_MASK 0xC
#define  SDA_BIT_POS  0x0
#define  SDA_BIT_MASK 0xC
```

(continues on next page)

(continued from previous page)

```
on tile[1]: camera_main(c_cam);
on tile[1]: user_app(c_cam);
```

```
#define CAMERA_MIPI_TILE 1
#define CAMERA_I2C_TILE  0

#define  SCL_BIT_POS  0x0
#define  SCL_BIT_MASK 0x0
#define  SDA_BIT_POS  0x0
#define  SDA_BIT_MASK 0x0

on tile[CAMERA_MIPI_TILE]: camera_main(c_cam, c_i2c);
on tile[CAMERA_MIPI_TILE]: user_app(c_cam);
on tile[CAMERA_I2C_TILE]: camera_sensor_control_rx(c_i2c);
```

The main difference is that in the xcore.ai Vision Development Kit the I2C and MIPI are on the same tile, so the sensor control object can be called directly as a function, while in the xcore.ai Explorer Development Kit, the I2C and MIPI ports are on different tiles, therefore the user will need to create a new thread (in the example `camera_sensor_control_rx`) to handle the I2C control, and both threads will communicate via a channel (in the example, `c_i2c`).

To enable this, the `camera_main` function needs to accept a second channel parameter, `c_i2c`. This channel is used for communication between the ISP thread and the I2C control thread, and should be passed to `camera_isp_thread`.

```
void camera_main(chanend_t c_cam, chanend_t c_i2c) {
 /*
 .....
 */
 PAR_JOBS(
     PJOB(camera_mipi_rx, (ctx.p_mipi_rxd, ctx.p_mipi_rxa, c_pkt.end_a, c_ctrl.end_a)),
     PJOB(camera_isp_thread,(c_pkt.end_b, c_ctrl.end_b, c_cam, c_i2c))
 );
```

The key distinction is that, on the xcore.ai Explorer Development Kit, I2C commands are not issued directly from the ISP thread. Instead, the ISP thread invokes `camera_sensor_control_tx`, which transmits the command over a channel to a dedicated thread (`camera_sensor_control_rx`) running on the tile with I2C access. This thread receives the command and performs the actual I2C transaction with the camera sensor. This separation enables inter-tile communication and proper handling of hardware constraints.

```
camera_sensor_init();
```

```
camera_sensor_control_tx(SENSOR_INIT, 0);
```

The `camera_sensor_control_rx` thread is responsible for receiving I2C control commands over the `c_i2c` channel (sent via `camera_sensor_control_tx`) and executing them on the sensor.

Below is a definition of what the `camera_sensor_control_rx` thread could look like:

```
void camera_sensor_control_rx(chanend_t c_i2c){
  uint32_t encoded_response;
  sensor_control_t cmd;
  uint8_t arg;
  int ret = 0;

  SELECT_RES(
    CASE_THEN(c_i2c, c_i2c_handler))
  {
    c_i2c_handler: {
      encoded_response = chan_in_word(c_i2c);
      cmd = (sensor_control_t)DECODE_CMD(encoded_response);
      arg = DECODE_ARG(encoded_response);
      switch (cmd)
      {
      case SENSOR_INIT:
        camera_sensor_init(); break;
      // ...
```

(continues on next page)

(continued from previous page)

```
      }
      assert((ret == 0) && "Could not perform I2C write");
    }
  }
}

void camera_sensor_control_tx(chanend_t c_i2c, sensor_control_t cmd, uint8_t arg) {
  uint32_t encoded_command = ENCODE_CTRL(cmd, arg);
  chan_out_word(c_i2c, encoded_command);
}
```

After implementing the above changes, the user will need to rebuild the application. The user can then run the example and verify that the camera is working correctly.

## 4.5 Adding Support for Other Boards

The xcore.ai camera library is designed to work with the supported hardware listed in the *Supported Hardware* section. However, it can be adapted to work with other boards that meet the minimum hardware requirements.

If the board meets these requirements, it may be necessary to adapt either the hardware or software configuration, depending on the board's architecture.

If both MIPI and I2C interfaces are available on the same tile, the code for the xcore.ai Vision Development Kit can generally be reused. Update the XN file and adjust the I2C port initialisation as needed, typically found under the `sensor` folder.

If MIPI and I2C are located on separate tiles, follow the approach described in the previous section for the xcore.ai Explorer Development Kit. This involves creating a dedicated thread for I2C control and establishing inter-tile communication via channels.

Additionally, ensure that the MIPI connector meets D-PHY specifications and that I2C lines are correctly routed to the appropriate pins for the custom board. Board-specific configuration may require further adjustments to the hardware description and initialisation code.

## 4.6 Adding a New Sensor

This section describes how to add a new sensor to the xcore.ai camera library. It covers both hardware and software aspects, including the necessary steps to ensure compatibility and functionality.

First, the user will need to see if their sensor is compatible with the xcore.ai Vision Development Kit.

The sensor has to:

- Support MIPI CSI2 protocol
- Be driven from a 3.3V source
- Have a compatible FPC-24 camera connector.

Fig. 6 shows the pinout of the 24-pin FPC connector on the xcore.ai Vision Development Kit:

Once a compatible sensor is available, the user will need to adapt the software.

By navigating to `sensors/api/SensorBase.hpp`, the user will find the `SensorBase` class which is intended to be derived from. It doesn't have anything to do with a particular sensor, it only provides an API to do basic I2C communication with the sensor. Inside

| NO | PIN NAME | NO | PIN NAME |
|---|---|---|---|
| 1 | NC | 16 | DGND |
| 2 | NC | 17 | MIPI-D2N |
| 3 | DOVDD1.8V | 18 | MIPI-D2P |
| 4 | DVDD1.2V | 19 | DGND |
| 5 | Sensor-PWDN | 20 | MIPI-D1N |
| 6 | DGND | 21 | MIPI-D1P |
| 7 | MCLK | 22 | DGND |
| 8 | DGND | 23 | MIPI-D0N |
| 9 | AGND | 24 | MIPI-D0P |
| 10 | AVDD2.8V | 25 | DGND |
| 11 | NC | 26 | MIPI-CLKN |
| 12 | DGND | 27 | MIPI-CLKP |
| 13 | MIPI-D3P | 28 | DGND |
| 14 | MIPI-D3N | 29 | SCL(1.8V) |
| 15 | DGND | 30 | SDA(1.8V) |

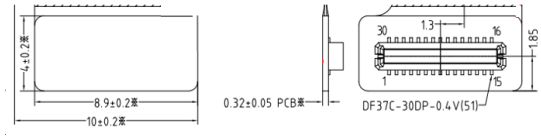


Fig. 6: FPC-24 connector pinout

`SensorBase` class users can also find some public virtual methods which will have to be implemented in the derived class.

In order to implement a new sensor, the user will need to create a directory in `lib_camera/src/sensors`, implement a derived class with `initialize()`, `stream_start()`, `stream_stop()`, `set_exposure()`, `configure()` and `control()` methods.

After that's been done, the user will need to rebuild the application.

# 5 Troubleshooting

## 5.1 Common Issues

This section provides solutions to common issues that users may encounter when using the library.

1. **Camera Not Detected**: If the camera is not connected or not properly connected, the system will fail to detect it. In these cases, the board will blink red and the following error message will be displayed:

```
ERROR: IMX219 not connected
>> Verify that the sensor is properly connected"
```

To resolve this issue, ensure that the camera is properly connected to the board. If the camera is not detected, check the connection and try again.

2. **Invalid Image Configuration**: If the image configuration is invalid, the system will not be able to process the image correctly. This can happen if the image size is not supported by the camera or if the image format is not supported. In this case, the following error message will be displayed:

```
xrun: Program received signal ET_ECALL, Application exception.
0x00081a9a in camera_isp_coordinates_compute (img_cfg=0xd2280)
```

To resolve this issue, ensure that the image configuration is valid and supported by the camera. Check the camera documentation for supported image sizes and formats.

3. **Missing Timings**: It may occur that the camera is not able to meet the timings for a specific mode. This can be due to a variety of reasons, such as the camera running out of time while processing the current line before the next one arrives, the camera is configured faster than it can handle, either the PHY or the SHIM. If this happens, the camera will not be able to meet the required timing and the ISP will receive invalid packets. This will result in an error message similar to the following:

```
xrun: Program received signal ET_ECALL, Application exception.
0x00081c6a in handle_unknown_packet (data_type=<optimized out>)
at _camera_isp.c:77
```

4. **Incorrect Capture Sequence**: This issue may occur if ISP functions are called in the wrong order or too early. For example, calling `camera_isp_start_capture()` before computing the coordinates with `camera_isp_coordinates_compute()` can result in undefined behaviour, often leading to image corruption. Conversely, if `camera_isp_get_capture()` is called before `camera_isp_start_capture()`, the ISP will block while waiting for the image to be captured indefinitely.

5. **Incorrect Tile Placement**: Ensure that both the caller thread and the *camera_main* thread are running on the same tile (by default, tile 1). If they are placed on different tiles, code attempts to jump, branch, or return to an address that is not part of executable memory.

```
Frame time: 4805210 cycles
xrun: Program received signal ET_ILLEGAL_PC, Illegal program counter.
0x4c086f7e in ?? ()
```

6. **Image Corruption**: Image corruption can occur if the image buffer is modified before the image has been fully captured. If the image is modified while the ISP is processing it, or if the pointer is used during capture, this may result in undefined behavior and data corruption. To prevent this issue, ensure that the image is not modified or accessed until the ISP has completed processing.

# 6 Contributing

This section provides guidelines for contributing to the library. We welcome contributions from the community, whether it's bug fixes, new features, or improvements to the documentation.

The general approach to contributing to the library is to fork the repository, make your changes, and then submit a pull request. This allows us to review your changes and ensure they meet our quality standards before merging them into the main branch.

When doing so, it is recommended to add tests or examples to demonstrate the new functionality or bug fix. This will help us understand the changes and ensure they work as intended.

PRs will trigger a CI pipeline that will run the tests and check the code style. Make sure new tests are added to the pipeline modifying the correct CMakeLists.txt file.

The following section focuses on contributing to the ISP components.

## 6.1 Modifying or Adding an ISP component

The library supports two types of ISP components: Line ISP and End-of-Frame (EOF) ISP. The primary difference between them lies in how they process image data.

Line ISP components process data on a line-by-line basis or in groups of lines. This means the component processes each line of the image as it is received from the camera sensor. This approach is typically used when converting from RAW8 input to a desired output format, such as RGB888 or YUV422. Processing line by line is preferred in this case to avoid storing the entire RAW image in memory.

End-of-Frame ISP components, on the other hand, process data after the entire image has been received and processed by the line ISP components. These components handle tasks that do not necessarily require storing the entire image, such as auto-exposure adjustments, or tasks that can be performed in-place in memory, such as int8 to uint8 conversion or gamma correction.

For modifying components, the user should be aware of the following:

- camera_isp.c : This file contains both line process under the `handle_expected_lines()` function and EOF process under the `handle_post_process()` function.
- camera_isp.h : This file contains the different modes, and their corresponding maximum allowed regions for each mode.

Other files that may be of interest are:

- camera.h : Defines the absolute maximum sensor region. This will drive the maximum output image size.
- sensor_wrapper.h : Defines the sensor wrapper functions, which are used to communicate with the camera sensor and configure its settings. These wrapper functions are implemented in the `sensor_wrapper.c` file. This file serves as a C wrapper for the C++ implementation, specifically the `sensor_imx219.cpp` class, which is a subclass of the `sensor_base.cpp` class.

In order to add or modify an ISP component the user would have to submit their changes in the *isp* folder. As an example, let us suppose the user wants to improve the conversion from RAW8 to RGB888 (RGB1) with an improved algorithm, or wants to add a new alogrithm, like RAW8 to RGB565.

The user should implement the new algorithm in the ISP folder, ensuring it adheres to the library's coding standards and performance requirements. Additionally, the user should update the `camera_isp.c` file to integrate the new algorithm into the `handle_expected_lines()` function.

To validate the changes, the user should ensure that the existing tests for this component are updated or extended to cover the new algorithm. Submitting the PR will trigger the CI pipeline, which will run these tests. The pipeline will verify that the new algorithm produces results comparable to the Python and OpenCV implementations by checking metrics such as PSNR and SSIM. These metrics ensure the output quality is acceptable when compared to the OpenCV reference implementation. For more information on ISP tests, please check the *tests/isp* folder.

# 7 API Reference

This section contains the API reference for the Camera library.



## 7.1 Camera

The Camera module serves as the entry point for the camera library.

It defines global macros for main camera configuration and provides functions to configure and initialize the camera thread.

*group* **`Main camera functions`**

### Defines

**`SENSOR_WIDTH`**
Sensor width in pixels.

**`SENSOR_HEIGHT`**
Sensor height in pixels.

**`CONFIG_FLIP`**
Flip mode: FLIP_NONE, FLIP_VERTICAL.

**`CONFIG_BINNING`**
Binning mode: BINNING_ON or BINNING_OFF.

**`CONFIG_CENTRALISE`**
Centralise mode: CENTRALISE_ON or CENTRALISE_OFF.

**`CONFIG_APPLY_AWB`**
Apply White Balance: 1 to apply, 0 to skip.

**`CONFIG_APPLY_AE`**
Apply Auto Exposure: 1 to apply, 0 to skip.

### Functions

void **`camera_main`**(chanend_t c_camera)
Main entry point for the lib_camera module.
This function initializes and configures the MIPI interface, and starts both the MIPI RX and ISP processing threads.

**Parameters**

- **c_camera** – Channel endpoint for communication with the user application.

## 7.2 ISP

The ISP module offers an interface for configuring and controlling the camera's image processing pipeline.

It provides functions to set the output format, resolution, and various image processing parameters. This module includes both configuration and conversion functions essential for adapting the camera output to different requirements.

*group*
**Functions related to the Image Signal Processing (ISP) pipeline.**

**Defines**

**MODE_RAW_MAX_SIZE**

**MODE_RGB1_MAX_SIZE**

**MODE_RGB2_MAX_SIZE**

**MODE_RGB4_MAX_SIZE**

**MODE_YUV2_MAX_SIZE**

**Typedefs**

typedef unsigned **mipi_header_t**

**Enums**

enum **camera_mode_t**

Defines the camera output modes.
*Values:*

enumerator **MODE_RAW**

Unprocessed RAW sensor data.

enumerator **MODE_RGB1**

RGB mode with no downsample (demosaicing)

enumerator **MODE_RGB2**

RGB mode with x2 downsample.

enumerator **MODE_RGB4**

RGB mode with x4 downsample.

enumerator **MODE_YUV2**

YUV2 mode (YUV422)

struct **camera_cfg_t**

*#include <camera_isp.h>* Configuration structure for the camera.

**Public Members**

float **offset_x**

Horizontal offset in [0,1] range relative to the sensor area.

float **offset_y**

Vertical offset in [0,1] range relative to the sensor area.

*camera_mode_t* **mode**

Camera output mode.

unsigned **y2**

Region of interest (ROI) in MIPI coordinates.

unsigned **sensor_width**

Width of the MIPI region.

unsigned **sensor_height**

Height of the MIPI region.

struct **image_cfg_t**

*#include <camera_isp.h>* Image configuration structure.

**Public Members**

unsigned **height**

Height of the output image in pixels.

unsigned **width**

Width of the output image in pixels.

unsigned **channels**

Number of channels in the output image (ex: 1 for RAW, 3 for RGB)

unsigned **size**

Size of the output image in bytes.

int8_t ***ptr**

Pointer to the output image data.

*camera_cfg_t* ***config**

Pointer to the camera configuration structure.

*group*
**Functions related to the Image Signal Processing (ISP) pipeline.**

### Functions

void **camera_isp_prepare_capture**(chanend_t c_cam, *image_cfg_t* *image)

Captures frames until the AE is done or a maximum number of steps is reached. This function can be called before **camera_isp_start_capture** to ensure the image is well-exposed. It is optional and can be skipped if the user does not require auto-exposure or is willing to accept initial frames with incorrect exposure. Has to be called after **camera_isp_coordinates_compute**.

**Parameters**

- **c_cam** – camera channel
- **image** – image pointer and configuration

void **camera_isp_start_capture**(chanend_t c_cam, *image_cfg_t* *image)

Sends the camera configuration to the ISP thread and starts capture process. Capture process starts asynchronously, and the function returns immediately. This function should be called after **camera_isp_coordinates_compute**. To capture the image, the user should call **camera_isp_get_capture**.

**Parameters**

- **c_cam** – camera channel
- **image** – image pointer and configuration

void **camera_isp_get_capture**(chanend_t c_cam)

Receives an image from the ISP thread. This function blocks until the image is ready. This function should be called after **camera_isp_start_capture**. Image will be returned in the image structure passed to **camera_isp_start_capture**.

**Parameters**

- **c_cam** – camera channel

void **camera_isp_thread**(
streaming_chanend_t c_pkt, chanend_t c_ctrl, chanend_t c_cam,
)

Main thread function for the ISP. This function handles the interaction between the MIPI packet channel, control channel, and the camera channel. It processes incoming data and manages the ISP pipeline for image processing.

**Parameters**

- **c_pkt** – channel to receive mipi packets
- **c_ctrl** – channel to receive control messages from or to mipi
- **c_cam** – channel array between user and isp

void **camera_isp_coordinates_compute**(*image_cfg_t* *image_cfg)

compute MIPI coordinates, from user request to sensor dimensions.

**Parameters**

- **image_cfg** – pointer to the image configuration structure.

void **camera_isp_coordinates_print**(*image_cfg_t* *image_cfg)

prints the coordinates of the image_cfg

**Parameters**

- **image_cfg** – pointer to the image configuration structure.

void **camera_isp_raw8_to_raw8**(*image_cfg_t* *image, int8_t *data_in, unsigned ln)

Converts RAW8 lines from the sensor into an RAW8 image. Conversion: HxWx1 (RAW8) -> HxWx1 (RAW8)

**Parameters**

- **image** – structure containing image configuration and output pointer.
- **data_in** – pointer to the input RAW8 line data.
- **ln** – current sensor line number.

void **camera_isp_raw8_to_rgb1**(*image_cfg_t* *image, int8_t *data_in, unsigned ln)

Converts RAW8 lines into an RGB1 image. (image demosaicing) Conversion: HxWx1 (RAW8) -> HxWx3 (RGB)

**Parameters**

- **image** – structure containing image configuration and output RGB pointer.
- **data_in** – pointer to the input RAW8 data.
- **ln** – current sensor line number.

void **camera_isp_raw8_to_rgb2**(*image_cfg_t* *image, int8_t *data_in, unsigned ln)

Converts RAW8 lines into an RGB2 image. (downsampled by 2) Conversion: HxWx1 (RAW8) -> (H/2)x(W/2)x3 (RGB)

**Parameters**

- **image** – structure containing image configuration and output RGB pointer.
- **data_in** – pointer to the input RAW8 data.
- **ln** – current sensor line number.

void **camera_isp_raw8_to_rgb4**(*image_cfg_t* *image, int8_t *data_in, unsigned ln)

Converts RAW8 lines into an RGB4 image. (downsampled by 4) Conversion: HxWx1 (RAW8) -> (H/4)x(W/4)x3 (RGB)

**Parameters**

- **image** – structure containing image configuration and output RGB pointer.
- **data_in** – pointer to the input RAW8 data.
- **ln** – current sensor line number.

void **camera_isp_raw8_to_yuv2**(*image_cfg_t* *image, int8_t *data_in, unsigned ln)

Converts RAW8 lines into an YUV422 image. (downsampled by 2 horizontally)

**Parameters**

- **image** – structure containing image configuration and output YUV pointer.
- **data_in** – pointer to the input RAW8 data.
- **ln** – current sensor line number.

void **camera_isp_white_balance**(*image_cfg_t* *image)

Applies Static White Balancing to the image. This applies a gain to the R, G and B channels of the image. Image must be in RGB format. Image pointer is updated with the new image.

**Parameters**

- **image** – structure containing image configuration and output RGB pointer.

unsigned **camera_isp_auto_exposure**(*image_cfg_t* *image)

Computes camera gain to apply for a new auto exposure step given an image. Computes the histograms and statistics of the image and computes the new exposure value. It is based on false position method of histogram skewness. It works well in unimodal distributions, but it is not very robust in multimodal distributions.

**Parameters**

- **image** – structure containing image configuration and output RGB pointer.

**Returns**

uint8_t new exposure value in [1, 80] or 0 if the exposure is already adjusted.

*group* **Functions related to image conversion**

**Defines**

**GET_R**(rgb)

Get Red component from encoded XRGB uint32_t.

**GET_G**(rgb)

Get Green component from encoded XRGB uint32_t.

**GET_B**(rgb)

Get Blue component from encoded XRGB uint32_t.

**GET_Y**(yuv)

Get Y component from encoded XYUV uint32_t.

**GET_U**(yuv)

Get U component from encoded XYUV uint32_t.

**GET_V**(yuv)

Get V component from encoded XYUV uint32_t.

**Functions**

void **camera_rgb_to_greyscale4**(int8_t *gs_img, int8_t *img, unsigned n_pix)

Convert an RGB image to the greyscale one. The operation can be performed safely in-place on the same pointer. `n_pix` must be a multiple of 4.

**Parameters**

- **gs_img** – Greyscale image
- **img** – RGB image
- **n_pix** – Number of RGB pixels

void **camera_rgb_to_greyscale16**(int8_t *gs_img, int8_t *img, unsigned n_pix)

Convert an RGB image to the greyscale one. The operation can be performed safely in-place on the same pointer. `n_pix` must be a multiple of 16.

**Parameters**

- **gs_img** – Greyscale image
- **img** – RGB image
- **n_pix** – Number of RGB pixels

inline void **camera_rgb_to_greyscale**(
int8_t *gs_img, int8_t *img, unsigned n_pix,
)

Convert an RGB image to the greyscale one. The operation can be performed safely in-place on the same pointer. **n_pix** must be a multiple of 4.

**Parameters**

- **gs_img** – Greyscale image
- **img** – RGB image
- **n_pix** – Number of RGB pixels

int **camera_yuv_to_rgb**(int y, int u, int v)

Converts a YUV pixel to RGB.

**Parameters**

- **y** – Y component
- **u** – U component
- **v** – V component

**Returns**

int result of rgb conversion (need macros to decode output)

int **camera_rgb_to_yuv**(int r, int g, int b)

Converts a RGB pixel to YUV.

**Parameters**

- **r** – red component
- **g** – green component
- **b** – blue component

**Returns**

int result of yuv conversion (need macros to decode output)

void **camera_int8_to_uint8**(
uint8_t *output, int8_t *input, const unsigned length,
)

Convert an array of int8 to an array of uint8. Data can be updated in-place.

**Parameters**

- **output** – Array of uint8_t that will contain the output
- **input** – Array of int8_t that contains the input
- **length** – Length of the input and output arrays

void **camera_swap_dims**(
uint8_t *image_in, uint8_t *image_out, const unsigned height, const unsigned width, const unsigned channels,
)

Swaps image dimensions from [x][y][z] to [y][z][x].

**Parameters**

- **image_in** – Input image
- **image_out** – Output image
- **height** – Image height
- **width** – Image width

- **channels** – Number of channels

void **camera_rotate90**(
void *dst_img, void *src_img, const int16_t h, const int16_t w, const int16_t ch,
)

Rotates an RGB image 90 degrees clockwise, assumes both src and dst images are previously allocated, and input image pointer can't be reused for the output image. Input image dimensions are [height][width][channel], and only int8 or uint8 types are supported. Output image dimensions are [width][height][channel], and they are of the same type as the input image.

**Parameters**

- **dst_img** – Destination image
- **src_img** – Source image (only uint8 or int8 types are supported)
- **h** – Image height
- **w** – Image width
- **ch** – Number of channels

## 7.3 Sensors

This module provides a high-level class for the camera sensor. It includes set of methods that the library needs in order to configure and control the camera sensor.

*group* **Camera Sensor Base API**

Functions and classes to control camera sensors.

struct **i2c_line_t**

*#include <sensor_base.hpp>* I2C line structure.

struct **i2c_table_t**

*#include <sensor_base.hpp>* I2C table structure.

struct **i2c_config_t**

*#include <sensor_base.hpp>* I2C configuration structure.

class **SensorBase**

*#include <sensor_base.hpp>* Base class for camera sensors.
Subclassed by sensor::IMX219

**Public Functions**

**SensorBase**(*i2c_config_t* _conf)

Construct new *SensorBase*

> **Note**
>
> This will initialize I2C interface

**Parameters**

**_conf** – I2C master config to use for the sensor control

virtual int **initialize**()

Initialize sensor.

> **Note**
>
> This is a virtual function, and will have to be implemented in the derived class

virtual int **stream_start**()

Start data stream.

> **Note**
>
> This is a virtual function, and will have to be implemented in the derived class

virtual int **stream_stop**()

Stop data stream.

> **Note**
>
> This is a virtual function, and will have to be implemented in the derived class

virtual int **set_exposure**(uint32_t dBGain)

Set sensor exposure.

> **Note**
>
> This is a virtual function, and will have to be implemented in the derived class

**Parameters**

**dBGain** – Exposure gain in dB, can enable different types of camera gain

virtual int **configure**()

Set sensor resolution, binning mode, and RAW format.

> **Note**
>
> This is a virtual function, and will have to be implemented in the derived class

virtual void **control**(chanend_t c_control)

Control thread intry, will initialise and configure sensor inside.

> **Note**
>
> This is a virtual function, and will have to be implemented in the derived class

**Parameters**

**c_control** – Control channel

virtual int **set_test_pattern**(uint16_t pattern)

Set test pattern.

> **Note**
>
> This is a virtual function, and will have to be implemented in the derived class

**Parameters**

**pattern** – Test pattern to set

virtual int **check_sensor_is_connected**()

Check if sensor is connected.

> **Note**
>
> This is a virtual function, and will have to be implemented in the derived class

**Returns**

0 if succeeded, -1 if failed

## 7.4 I/O

The I/O module includes a set of functions to read and write image data from or to a host PC. It supports various image formats.

*group* **Functions related to file I/O operations for camera data**

**Functions**

void **camera_io_write_file**(char *filename, uint8_t *data, const size_t size)

Dumps data into a file, can be lossy if done over xscope.

**Parameters**

- **filename** – Name of the file
- **data** – Data to write
- **size** – Size of the data

void **camera_io_read_file**(char *filename, uint8_t *data, const size_t size)

Reads data into an array.

**Parameters**

- **filename** – Name of the file
- **data** – Data to read to
- **size** – Size of the file

void **camera_io_write_image_file**(
char *filename, uint8_t *image, const size_t height, const size_t width, const size_t channels,
)

Writes binary image file.

> **Note**
>
> Image has to be in [height][width][channel] format

**Parameters**

- **filename** – Name of the image
- **image** – Pointer to the image data
- **height** – Image height
- **width** – Image width
- **channels** – Number of channels

void **camera_io_write_bmp_file**(
char *filename, uint8_t *image, const size_t height, const size_t width, const size_t channels,
)

Writes bmp image file.

> **Note**
>
> Image has to be in [height][width][channel] format

**Parameters**

- **filename** – Name of the image
- **image** – Pointer to the image data
- **height** – Image height
- **width** – Image width
- **channels** – Number of channels

void **write_bmp_greyscale**(
char *filename, uint8_t *image, const size_t height, const size_t width,
)

Writes greyscale bmp image file.

> **Note**
>
> Image has to be in [height][width] format

**Parameters**

- **filename** – Name of the image
- **image** – Pointer to the image data
- **height** – Image height
- **width** – Image width